PCI Firmware Specification
Revision 3.0

June 20, 2005

Revision	Revision History	Date
1.0	Original issue distributed by Intel Corporation	9/28/1992
2.0	Updated to be in synch with PCI Bus Specification, Rev. 2.0	7/20/1993
2.1	Added functions for PCI IRQ routing; clarifications.	8/26/1994
3.0	Updated revision evolving the specification to include all aspects of PCI and system firmware.	6/20/2005

PCI-SIG® 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 PCI-SIG make a commitment to update the information contained herein.

Contact the PCI-SIG office to obtain the latest revision of the specification.

Questions regarding this specification or membership in PCI-SIG may be forwarded to:

Membership Services
www.pcisig.com
E-mail: administration@pcisig.com
Phone: 503-619-0569
Fax: 503-644-6708

Technical Support
techsupp@pcisig.com

DISCLAIMER

Contents

Figures

Tables

1

1. Introduction

This document describes the hardware independent firmware interface for managing PCI, PCI-X, and PCI Express™ systems in a host computer.

1.1. Scope

This document is developed based on the *PCI BIOS Specification, Revision 2.1*. It continues to provide the PCI BIOS support on the PC-compatible systems.

In addition, this document also provides the descriptions or references of the following:

- PCI related firmware services for DIG64-compliant systems that are compliant with *Developer's Interface Guide for 64-bit Intel Architecture-based Servers (DIG64)*, Version 2.1 (DIG64 2.1)[1].
- Advanced Configuration and Power Interface (ACPI) services for supporting PCI, PCI-X, and PCI Express devices and systems: ACPI is being used on both PC-compatible and DIG64-compliant systems.
- Extensible Firmware Interface (EFI) services for supporting PCI, PCI-X, and PCI Express devices and systems: EFI is designed to be processor and platform architecture independent and is the operating system boot interface for DIG64-compliant systems.
- Requirements and services for supporting PCI Expansion ROMs: The format, contents, and code entry points for PCI Expansion ROMs, along with system firmware services and execution environment, are described in this document. This information was documented in prior versions of the Conventional *PCI Local Bus Specification* (*Revision 2.3* and earlier), and is now maintained solely in this document.

1.2. Reference Documents

The following documents are a part of this specification to the extent specified herein:

Advanced Configuration and Power Interface, Revision 3.0 (ACPI 3.0), September, 2004, http://www.acpi.info

Extensible Firmware Interface Specification, Version 1.10, January 7, 2003, http://developer.intel.com/technology/efi/

EFI 1.10 Driver Writer's Guide, Draft 0.9, July 20, 2004, http://developer.intel.com/technology/efi/

---

[1] *Developer's Interface Guide for 64-bit Intel Architecture-based Servers (DIG64)*, Version 2.1, January 2002, http://www.dig64.org.

Developer's Interface Guide for 64-bit Intel Architecture-based Servers (DIG64), Version. 2.1 (DIG64 2.1), January 2002, http://www.dig64.org

PCI Local Bus Specification, Revision 3.0 (PCI 3.0)

PCI Express Base Specification, Revision 1.0a

PCI Express Card Electromechanical Specification, Revision 1.0

PCI Express to PCI/PCI-X Bridge Specification, Revision 1.0

PCI Express Mini Card Electromechanical Specification, Revision 1.0

PCI Local Bus Specification, Revision 2.3

PCI-X Addendum to the PCI Local Bus Specification, Revision 2.0

PCI Hot-Plug Specification, Revision 1.1

PCI Standard Hot-Plug Controller and Subsystem Specification, Revision 1.0 (SHPC 1.0)

PCI Power Management Interface Specification, Revision 1.1

Guidelines for 64-bit Global Identifier (EUI-64) Registration Authority

1.3. Terms and Acronyms

Term	Definition
Active State Power Management (ASPM)	An autonomous hardware based active state mechanism, Active State Power Management defined in PCI Express.
Advanced Error Reporting (AER)	A capability for advanced error control and reporting defined in PCI Express.
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 I/O 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.
Message Signaled Interrupt (MSI)	An optional feature that enables a device to request service by writing a system-specified DW of data to a system-specified address using a Memory Write semantic Request.
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.

Term	Definition
Operating System Hot plug (OSHP)	An ACPI control method to transfer control of Hot-Plug to the operating system. On some systems, this method is defined for Ports that are Hot-Plug capable and being controlled by ACPI firmware.
PCI	An acronym for Peripheral Component Interconnect.
Segment Group Number	A number in the range 0…65535 that uniquely selects a PCI Segment Group. PCI Segment Group is purely a software concept managed by system firmware and used by the operation system. It is a logical collection of PCI buses (or bus segments). There is no tie to any physical entities. It is a way to logically group the PCI bus segments and PCI Express Hierarchies. PCI Segment Group concept enables support for more than 256 buses in a system by allowing the reuse of the PCI bus numbers. Within each PCI Segment Group, the bus numbers for the PCI buses must be unique. PCI buses in different PCI Segment Group are permitted to have the same bus number. A PCI Segment Group contains one or more PCI host bridges. There is at least one PCI Segment Group, PCI Segment Group 0, in a system.
Standard Hot-Plug Controller (SHPC)	A PCI Hot-Plug Controller compliant with SHPC 1.0.
Vendor ID	A predefined field in configuration space that (along with Device ID) uniquely identifies the device.

2

2. Traditional PCI BIOS

2.1. 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 to operate in the following modes of the X86 architecture. The modes are: real-mode, 16:16 protected mode (also known as 286 protected 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-GB address space).

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

2.2. Assumptions and Constraints

2.2.1. ROM BIOS Location

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

2.2.2. Calling Conventions

The PCI BIOS functions use the X86 CPU's registers to pass arguments and return status. The caller must use the appropriate sub-function 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 (i.e., 16-bit or 32-bit) as the code segment.

The PCI BIOS provides a 16-bit real and protected mode interface and a 32-bit protected 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 protected 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.[2] The INT 1Ah entry point supports 16-bit code only. Protected mode callers of this interface must set the CS selector base to 0F000h.

The protected mode interface supports 32-bit protected 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 (refer to Section 2.3). 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 I/O space. Implementers of the PCI BIOS must assume that CS is execute-only and DS is read-only.

2.2.3. Interrupt Support

To support PC-compatible BIOS, the system must support the INTx routing. MSI or MSI-X may be supported for the operating system use after booted from BIOS.

2.3. BIOS32 Service Directory[3]

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.

2.3.1. 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 2-1.

---

[2] 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] 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.

Table 2-1. Data Structure Fields for the BIOS32 Service Directory

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.

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.

2.3.2. 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 1 K of stack space. The calling environment must also allow access to I/O 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.

2.4. 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 I/O space and provide at least 1 K of stack space. Platform BIOS writers must assume that CS is execute-only and DS is read-only.

2.5. Host Interface

2.5.1. 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.

2.5.2. 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 if and only if 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 if and only if 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 do not 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.

A BIOS that complies with Version 3.00 of the specification will return with 0300h in the [BX] register.

The value returned in [AL] identifies what specific hardware characteristics the platform supports in relation to accessing configuration space and generating PCI Special Cycles (see Figure 2-1). The PCI Specification defines two hardware 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 hardware 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.

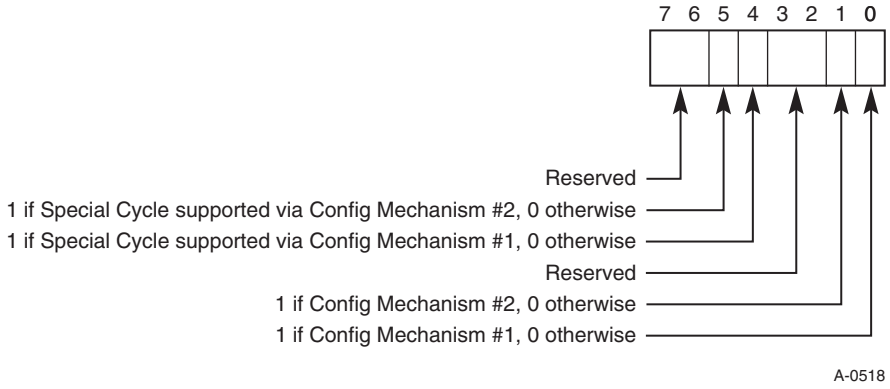

Figure 2-1: Layout of Value Returned in [AL]

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.

If the [BX] register indicates that the BIOS is compliant with version 3.00 (or later), then the [CH] register will have the following meaning:

CH = Level of BIOS support:
- bit 0 set = Functions 06h through 0Dh inclusively are implemented for register access below 256.
- bit 1 set = Functions 06h through 0Dh inclusively are implemented for register access in the range 256 - 4096 for POST only.
- bit 2 set = Function 0Eh has been implemented.
- bit 3 set = Function 0Fh has been implemented.
- bit 4 set = Function 02h has been implemented.
- bit 5 set = Function 03h has been implemented.
- bit 6 set = BIOS supports Option ROM Configuration Code execution.
- bit 7 set = BIOS supports calling the Option ROM with DMTF CLP style configuration data.

Note that version 3.00 of this specification requires that bit 0 and bit 1 be set. This may not be the case in future versions of the specification.

2.5.3. 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 bypassed in favor of the direct I/O mechanisms described in the PCI Specification.

2.5.4. Find PCI Class Code

This function returns the location of PCI devices that have a specific Class Code. Given a Class Code and an 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 3 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".

2.6. PCI Support Functions

The following functions provide support for several PCI specific operations.

2.6.1. 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.

2.6.2. Get PCI Interrupt Routing Expansions

Description:

Logical input parameters:

RouteBuffer Pointer to buffer data structure

This routine returns the PCI interrupt routing Expansions 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 2-2.

Table 2-2: Layout of IRQ Routing Table Entry

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 (for OEM use)

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 Expansions 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 indicates that the PCI interrupt pin has no connection to the interrupt controller.[4]

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 00h) are OEM specific.[5] For end user ease-of-use, slots in the system should be clearly labeled (e.g., solder mask, back panel, etc.).

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

[5]For example, a system with 4 ISA slots and 3 PCI slots arranged as 3-ISA, 3-PCI, and 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.

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

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

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.

IMPLEMENTATION NOTE

Defining the DataBuffer in C

The code example above is syntactically correct for C language in all processor modes. But the storage size of the BYTE FAR * offset varies for each mode as follows:

- Real Mode: 2 WORD (Segment:Offset)
- PM16: 2 WORD (Selector:Offset)
- PM32: 3 WORD (Selector:32bitOffset)

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.

ENTRY:

[AH]	PCI_FUNCTION_ID
[AL]	GET_IRQ_ROUTING_EXPANSIONS
[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 64 K. For information on 32-bit code, refer to Section 2.4.
[ES]	Segment or Selector for *RouteBuffer* parameter.
[DI] for 16-bit code [EDI] for 32-bit code	Offset for *RouteBuffer* parameter.

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.

2.6.3. Set PCI Hardware Interrupt

Description:

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

This function is optional in this version of the specification. The caller is responsible for checking the return code to determine if the call is supported.

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 (i.e., 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 (i.e., they have the same link field in the Get Routing Expansions 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 5 bits) and Function (lower 3 bits) numbers.
[DS]	Segment or Selector for BIOS data. For 16-bit code, the real-mode segment or the Protected Mode selector must resolve to physical address 0F0000h and have a limit of 64 K. For 32-bit code, see Section 2.4.

EXIT:

[AH] Return Code:

SUCCESSFUL

SET_FAILED

FUNC_NOT_SUPPORTED

[CF] Completion Status, set = error, cleared = success.

2.7. Accessing Configuration Space

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 ES descriptors must be setup to encompass the physical address reported by the PCI Express Base Address Register (BAR) region. The ES descriptor must have a base of zero with the limit of 4 GB to encompass the 256-MB PCI Express Base Address Register (BAR) region. The calling environment must allow access to I/O space and provide at least 1 K of stack space. Platform BIOS writers must assume that CS is execute-only, DS is read-only, and ES is read/write.

The underlying assumptions being made are the following:

- If PCI BIOS functions support access to extended configuration registers, the PCI Express memory mapped configuration base address must be programmed with an address region that exists below 4 GB of memory.
- The caller has the responsibility to appropriately parse the data returned from accessing non-existing registers of a PCI Device. The caller has to ensure that Register accesses beyond 256 bytes be invoked only on devices that have the support for extended configuration space. For example: if the caller accesses DWORD Register 0FFC (4092) of a regular PCI device, the data returned cannot be predicted.
- PCI BIOS functions do not comprehend the concept of PCI Segment Groups (for definition, refer to Section 4.3.2.3) and, hence, can only support access to the devices in the default PCI Segment Group, namely, PCI Segment Group 0.

2.7.1. Access Rules for PCI Express I/O and Memory Mapped Accesses

Firmware should use the PCI I/O Index/Data mechanism to access configuration space registers 0-255. Only accesses to configuration space registers 256 and beyond should use the PCI Express memory-mapped access mechanism.

2.7.2. INT1Ah Access Calls in Real Mode

The real mode invocations of INT1Ah for reading the PCI Express Extended Configuration space are intended to be used for handling the internal BIOS/system firmware code calls during POST (if needed). After POST, it is recommended that the PCI Express Extended Configuration space be accessed directly without the use of Int 1Ah. The ACPI MCFG table describes the location of the PCI Express configuration space, and this table will be present in a PCI 3.0 compliant firmware implementation. If INT1Ah is invoked after POST for reading the PCI Express extended configuration space, the firmware may return FUNC_NOT_SUPPORTED.

2.7.3. 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...4095) [bits 11:0]
	To Read Register number greater than 255 [bit15=1]
	To Read Register number less than or equal to 255 [bit15=0]

EXIT:

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

Input Register Requirements:

- If register DI has a register number less than or equal to 255 and does not have bit15=1, the BIOS will read the configuration space and return the value read with return code "SUCCESS".
- If register DI has a register number greater than 255 and does not have bit15=1, the BIOS will not try to read the configuration space and returns "BAD_REGISTER_NUMBER".
- If register DI has a register number greater than 255 with bit15=1 and the BIOS implements the extensions for accessing PCI Express Extended Configuration space, then the BIOS will read the configuration space and return the value read with return code "SUCCESS".
- If register DI has a register number greater than 255 with bit15=1 and the BIOS does not implement the extensions for accessing PCI Express Extended Configuration space, then the BIOS will return error code of "FUNC_NOT_SUPPORTED".

2.7.4. 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,...4094) [bits 11:0]
	To Read Register number greater than 255 [bit15=1]
	To Read Register number less than or equal to 254 [bit15=0]

EXIT:

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

Input Register Requirements:

- If register DI has a register number less than or equal to 254 and does not have bit15=1, the BIOS will read the configuration space and return the value read with return code "SUCCESS".
- If register DI has a register number greater than 255 and does not have bit15=1, the BIOS will not try to read the configuration space and returns "BAD_REGISTER_NUMBER".
- If register DI has a register number greater than 255 with bit15=1 and the BIOS implements the extensions for accessing PCI Express Extended Configuration space, then the BIOS will read the configuration space and return the value read with return code "SUCCESS".
- If register DI has a register number greater than 255 with bit15=1 and the BIOS does not implement the extensions for accessing PCI Express Extended Configuration space, then the BIOS will return error code of "FUNC_NOT_SUPPORTED".

2.7.5. 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,...4092) [bits 11:0]
	To Read Register number greater than 255 [bit15=1]
	To Read Register number less than or equal to 252 [bit15=0]

EXIT:

[ECX]	DWORD Read
[AH]	Return Code:
	SUCCESSFUL
	BAD_REGISTER_NUMBER
	FUNC_NOT_SUPPORTED
[CF]	Completion Status, set = error, reset = success.

Input Register Requirements:

- If register DI has a register number less than or equal to 252 and does not have bit15=1, the BIOS will read the configuration space and return the value read with return code "SUCCESS".
- If register DI has a register number greater than 255 and does not have bit15=1, the BIOS will not try to read the configuration space and returns "BAD_REGISTER_NUMBER".
- If register DI has a register number greater than 255 with bit15=1 and BIOS implements the extensions for accessing PCI Express Extended Configuration space, then the BIOS will read the configuration space and return the value read with return code "SUCCESS".
- If register DI has a register number greater than 255 with bit15=1 and the BIOS does not implement the extensions for accessing PCI Express Extended Configuration space, then the BIOS will return error code of "FUNC_NOT_SUPPORTED".

2.7.6. 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...4095) [bits 11:0]
	To Write Register number greater than 255 [bit15=1]
	To Write Register number less than or equal to 255 [bit15=0]
[CL]	Byte Value to Write

EXIT:

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

Input Register Requirements:

- If register DI has a register number less than or equal to 255 and does not have bit15=1, the BIOS will read the configuration space and return the value read with return code "SUCCESS".
- If register DI has a register number greater than 255 and does not have bit15=1, the BIOS will not try to read the configuration space and returns "BAD_REGISTER_NUMBER".
- If register DI has a register number greater than 255 with bit15=1 and the BIOS implements the extensions for accessing PCI Express Extended Configuration space, then the BIOS will read the configuration space and return the value read with return code "SUCCESS".
- If register DI has a register number greater than 255 with bit15=1 and the BIOS does not implement the extensions for accessing PCI Express Extended Configuration space, then the BIOS will return error code of "FUNC_NOT_SUPPORTED".

2.7.7. 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,...4094) [bits 11:0]
	To Write Register number greater than 255 [bit15=1]
	To Write Register number less than or equal to 254 [bit15=0]
[CX]	Word Value to Write

EXIT:

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

Input Register Requirements:

- If register DI has a register number less than or equal to 254 and does not have bit15=1, the BIOS will read the configuration space and return the value read with return code "SUCCESS".
- If register DI has a register number greater than 255 and does not have bit15=1, the BIOS will not try to read the configuration space and returns "BAD_REGISTER_NUMBER".
- If register DI has a register number greater than 255 with bit15=1 and the BIOS implements the extensions for accessing PCI Express Extended Configuration space, then the BIOS will read the configuration space and return the value read with return code "SUCCESS".
- If register DI has a register number greater than 255 with bit15=1 and the BIOS does not implement the extensions for accessing PCI Express Extended Configuration space, then the BIOS will return error code of "FUNC_NOT_SUPPORTED".

2.7.8. 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,...4092) [bits 11:0]
	To Write Register number greater than 255 [bit15=1]
	To Write Register number less than or equal to 252 [bit15=0]
[ECX]	DWORD Value to Write

EXIT:

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

Input Register Requirements:

- If register DI has a register number less than or equal to 252 and does not have bit15=1, the BIOS will read the configuration space and return the value read with return code "SUCCESS".
- If register DI has a register number greater than 255 and does not have bit15=1, the BIOS will not try to read the configuration space and returns "BAD_REGISTER_NUMBER".
- If register DI has a register number greater than 255 with bit15=1 and the BIOS implements the extensions for accessing PCI Express Extended Configuration space, then the BIOS will read the configuration space and return the value read with return code "SUCCESS".
- If register DI has a register number greater than 255 with bit15=1 and the BIOS does not implement the extensions for accessing PCI Express Extended Configuration space, then the BIOS will return error code of "FUNC_NOT_SUPPORTED".

2.8. Function List

Table 2-3: Function List

Function	AH	AL	Implementation	Notes
PCI_FUNCTION_ID	B1h			
PCI_BIOS_PRESENT		01h		
FIND_PCI_DEVICE		02h	Optional	1
FIND_PCI_CLASS_CODE		03h	Optional	1
GENERATE_SPECIAL_CYCLE		06h	Optional	1
READ_CONFIG_BYTE		08h	Optional	1
READ_CONFIG_WORD		09h	Optional	1
READ_CONFIG_DWORD		0Ah	Optional	1
WRITE_CONFIG_BYTE		0Bh	Optional	1
WRITE_CONFIG_WORD		0Ch	Optional	1
WRITE_CONFIG_DWORD		0Dh	Optional	1
GET_IRQ_ROUTING_EXPANSIONS		0Eh	Optional	2
SET_PCI_HW_INT		0Fh	Optional	2

Notes:

1. If the "PCI BIOS PRESENT (B101h)" function indicates "Presence", then all functions (06h through 0Dh inclusively) must be implemented for register accesses below 256 and must be present during POST and at run-time (after POST completes). For register accesses above 255, these functions must only be present during POST. There is not a requirement to implement the above 255 register access functions after POST completes; however, if implemented, they must not change the processor mode when invoked after POST (option ROMs might be executing in v86 mode).
2. Implementation of these INT1Ah sub-functions is optional for compliance with this version of the specification.

2.9. Return Code List

Table 2-4: 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

3

3. EFI PCI Services

EFI stands for Extensible Firmware Interface. The *EFI Specification, Version 1.10* or later (http://developer.intel.com/technology/efi/) describes an interface between the operating system and the platform firmware. The interface is in the form of data tables that contain platform-related information and boot and run-time services calls that are available to the operating system and its loader. Together, these provide a standard environment for booting an operating system. EFI is required on DIG64-compliant systems.

The following sections provide an overview of the EFI Services relevant to PCI (including Conventional PCI, PCI-X, and PCI Express). For details, refer to the EFI Specification. EFI is processor-agnostic.

3.1. EFI Driver Model

The EFI Driver Model is designed to support the execution of drivers that run in the pre-boot environment present on systems that implement the EFI firmware. These drivers may manage and control hardware buses and devices on the platform, or they may provide some software derived platform specific services.

The EFI Driver Model is designed to extend the EFI Specification in a way that supports device drivers and bus drivers. It contains information required by EFI driver writers to design and implement any combination of bus drivers and device drivers that a platform may need to boot an EFI-compliant operating system.

Applying the EFI Driver Model to PCI, the EFI Specification defines the PCI Root Bridge Protocol and the PCI Driver Model and describes how to write PCI bus drivers and PCI devices drivers in the EFI environment. For details, refer to the EFI Specification.

3.1.1. PCI Root Bridge Protocol

A PCI Root Bridge is represented in EFI as a device handle that contains a Device Path Protocol instance and a PCI Root Bridge Protocol instance.

PCI Root Bridge Protocol provides an I/O abstraction for a PCI Root Bridge that the host bus can perform. This protocol is used by a PCI Bus Driver to perform PCI Memory, PCI I/O, and PCI Configuration cycles on a PCI Bus. It also provides services to perform different types of bus mastering DMA on a PCI bus.

PCI Root Bridge Protocol abstracts device specific code from the system memory map. This allows system designers to make changes to the system memory map without impacting platform independent code that is consuming basic system resources. An example of such system memory

map changes is a system that provides non-identity memory mapped I/O (MMIO) mapping between the host processor view and the PCI device view.

3.1.2. PCI Driver Model

The PCI Driver Model is designed to extend the EFI Driver Model in a way that supports PCI Bus Drivers and PCI Device Drivers. This applies to Conventional PCI, PCI-X, and PCI Express.

PCI Bus Drivers manage PCI buses present in a system. The PCI Bus Driver creates child device handles that must contain a Device Path Protocol instance and a PCI I/O Protocol instance. The PCI I/O Protocol is used by the PCI Device Driver to access memory and I/O on a PCI controller.

PCI Device Drivers manage PCI controllers present on PCI buses. The PCI Device Drivers produce an I/O abstraction that may be used to boot an EFI compliant operating system.

3.2. PCI-X Mode 2 and PCI Express

The PCI-X Mode 2 and PCI Express provide a software programming model that is software compatible with the Conventional PCI.

EFI PCI I/O Protocol supports up to 4 GB of configuration space; therefore, it covers the PCI-X Mode 2 and PCI Express Extended Configuration space of 4 KB in size.

EFI uses a single timer interrupt in pre-boot, EFI device drivers are polled so INTx, MSI, or MSI-X is not used by EFI.

To identify a function (e.g., Conventional PCI, PCI-X vs. PCI Express), the EFI driver uses Device ID, Vendor ID, and Capability Pointer in the compatibility configuration space.

3.3. EFI Byte Code

The EFI Specification defines a virtual machine that provides a platform and CPU independent mechanism for loading and executing EFI device drivers. The instruction set of the virtual machine is called EFI Byte Code, or EBC.

For details of the EBC Virtual Machine, refer to the EFI Specification.

3.4. Universal Graphics Adapter

Universal Graphics Adapter (UGA) is defined in the EFI Specification to remove the hardware requirement to support legacy VGA and INT 10h BIOS. UGA also provides a software abstraction to draw on video screen with EFI UGA Draw Protocol and UGA I/O Protocol. UGA ROM uses the EBC format and the UGA driver follows the EFI Driver Model.

For EFI UGA Draw and I/O Protocol, refer to EFI Specification.

Note: A graphics adapter will still require a performance driver for high speed operation in the operating system.

Note: VGA hardware can support EFI UGA Protocols.

3.5. Device State at Firmware/Operating System Handoff

System firmware is only required to configure the boot and console devices. This section specifies the state of the PCI subsystem at firmware handoff and provides guidance to the operating system on how to determine if a particular component was configured by firmware. PCI subsystem refers to components that are compliant to the PCI, PCI-X, or PCI-Express Specifications. In this section, "PCI Specifications" refers to the PCI, PCI-X, or PCI Express Specification.

Firmware owns the PCI subsystem prior to handing control off to the operating system. The handoff point is the return from EFI ExitBootServices(). After the operating system loader calls ExitBootServices(), the operating system owns the PCI subsystem.

Firmware may provide pre-boot user interaction to allow the system operator to specify the desired boot and console devices.

Firmware shall configure the entire path to the console (both input and output) and boot devices. This includes, the chipset, bridges, and multi-function devices. The device configuration is required to load the operating system loader, display boot up messages, and allow operator interaction with the boot process.

Optionally, firmware may configure all devices and bridges in the system. Firmware is not required to configure devices other than boot and console devices.

Since not all devices may be configured prior to the operating system handoff, the operating system needs to know whether a specific BAR register has been configured by firmware. The operating system makes the determination by checking the I/O Enable, and Memory Enable bits in the device's command register, and Expansion ROM BAR enable bits. If the enable bit is set, then the corresponding resource register has been configured.

Note: The operating system does not use the state of the Bus Master Enable bit to determine the validity of the BARs. If the BAR ranges are enabled, the device must respond to those addresses. The device may not be able to master a transaction, but enabled BARs shall be configured correctly by firmware.

The operating system is required to configure PCI subsystems:

- During hotplug
- For devices that take too long to come out of reset
- PCI-to-PCI bridges that are at levels below what firmware is designed to configure

Firmware must configure all Host Bridges in the systems, even if they are not connected to a console or boot device. Firmware must configure Host Bridges in order to allow operating systems to use the devices below the Host Bridges. This is because the Host Bridges programming model is not defined by the PCI Specifications. "Configured" in this context means that:

- Memory and I/O resources are assigned and configured.
- Includes both the resources consumed by the Host Bridge and the resources passed through to the secondary bus.
- The bridge is enabled to receive and forward transactions.
- The bridge is operating in "safe" mode. Safe mode includes:
  - Enabling resources such as: I/O Port, Memory addresses, VGA routing, bus number, etc.
  - Enabling detection of parity and system errors
  - Programming cacheline, latency timer, and other registers as required by the PCI Specifications.

Firmware must report Host Bridges in the ACPI name space. Each Host Bridge object must contain the following objects:

- _HID and _CID
- _CRS to determine all resources consumed and produced (passed through to the secondary bus) by the host bridge. Firmware allocates resources (Memory Addresses, I/O Port, etc.) to Host Bridges. The _CRS descriptor informs the operating system of the resources it may use for configuring devices below the Host Bridge.
  - _TRA, _TTP, and _TRS translation offsets to inform the operating system of the mapping between the primary bus and the secondary bus.
- _PRT and the interrupt descriptor to determine interrupt routing.
- _BBN to obtain a bus number.
- _UID to match with EFI device path.
- _SEG if it has a non-zero PCI Segment Group number.
- _STA if hot plug is supported.
- _MAT if hot plug is supported.

Firmware is required to configure all PCI-to-PCI Bridges in the hierarchy leading to boot and console devices. Firmware may optionally configure all PCI-to-PCI Bridges in the system. When configuring a PCI-to-PCI Bridge, Firmware must set it to safe mode. This includes:

- Programming and enabling resources such as: I/O Port, Memory addresses, VGA routing, bus number, etc.
- Enabling detection of parity and system errors
- If applicable, program cache_line, latency timer, and other registers as required by the PCI Specifications.

- If applicable[6], disable Discard SERR# Enable. The Discard Timer SERR# Enable bit in the Bridge Control Register controls whether the timer waiting for the completion of a delayed transaction generates an SERR (value=1) or simply discards the transaction (value=0) on a time-out. The value of 1 is generally required when peer-to-peer transactions are allowed to and from cards under that bridge. Allowing peer-to-peer transactions is operating system policy and may not be supported on all platforms. Therefore, the bit should be set to 0 when firmware hands off to the operating system, and any changes to the setting to support peer-to-peer under the PCI-to-PCI Bridges should be made by the operating system.

The operating system may provide software to configure PCI-to-PCI bridges for optimum performance.

The slot power state for unoccupied slots shall be as required by the Standard Hot Plug Controller (SHPC) Specification. All slots with MRL closed must be enabled and their Power Indicators must be turned on. All slots with an open MRL must be disabled and their Power Indicators must be turned off. Refer to Section 3.5.1.3 (Initializing the Secondary Bus) of SHPC 1.0).

Firmware must deassert RST# for all occupied PCI slots below the Host Bridge. Firmware must observe required wait times such as Trhfa (RST# High to First configuration Access) after taking a bus out of reset. Firmware only needs to delay once for all PCI bus controllers before handing control to the operating system. This allows the operating system to successively walk PCI buses without having to successively delay (post reset quiesce period, 1 second) for each bus.

PCI-to-PCI Bridges have RST# asserted by default for the secondary bus. Firmware is not required to deassert RST# on secondary buses that are not used for boot and console devices.

EFI drivers and applications shall not change BAR assignments. The PCI BARs (Base Address Registers) and the configuration of any PCI-to-PCI bridge controllers belong to the firmware component that configured the PCI Bus prior to the execution of the device driver.

The operating system must not assume that all devices have been configured. Per Section 2.5.6 of EFI (rev 1.10): the presence of an EFI driver in the system firmware or in an option ROM does not guarantee that the EFI driver will be loaded, executed, or allowed to manage any devices in a platform. In addition, EFI drivers are not involved during PCI hot plug.

Note: The operating system does not have to walk all buses during boot. The kernel can automatically configure devices on request; i.e., an event can cause a scan of I/O on demand.

The operating system can determine if the device's BARs (Base Address Registers) have been configured by firmware by checking the I/O Enable, and Memory Enable bits in the device's command register, and Expansion ROM BAR enable bit. If the enable bit is set, then the corresponding resource has been configured. If the enable bit is not set, the operating system cannot assume that the associated BAR register contains valid information.

The address that a processor uses to access a device is not necessarily the same as the address stored in the device's BAR. The translation (_TRA, _TTP, and _TRS) information is not available to the operating system until after the operating system brings up the ACPI interpreter. The operating system must wait until after the ACPI interpreter is up to determine the address at the processor side associated with the BAR registers configured by firmware. Before re-enabling a resource, the operating system must reprogram the BAR register using a value that falls within the range reported

[6] For example, does not apply to PCI Express.

in the _CRS descriptor of the parent Host Bridge. The operating system must ensure that the address range is not used by any other device below that Host Bridge.

Operating systems must not configure devices with resources outside what is reported by the Host Bridge _CRS. Firmware reports the address ranges that are routed to that particular Host Bridge. There is no guarantee that devices under that bridge will respond to other address ranges.

The Expansion ROM BAR (at 0x30) is normally not enabled at the firmware handoff to the operating system and the operating system must assume that the BAR content is invalid. For some devices, when the Expansion ROM BARs are enabled, the device's other BARs are disabled. PCI specifies that a device may share decoders between the Expansion ROM BAR and other BARs, and that device independent software must not access the other BARs when the Expansion ROM BAR is enabled. Firmware may leave the Expansion ROM BARs enabled if it happens to know that the device does not share address decoders. This could be firmware based on the Device ID or firmware that is shipped in the card itself. The device independent operating system software must disable the Expansion ROM when accessing the device via the other BARs. If the operating system wants to use the Expansion ROM, it must "take turns" enabling the Expansion ROM BAR, using the ROM, then disabling the BAR again before resuming access to the card via its other BARs. In other words, the operating system shall not assume that the card has dual decoders. The operating system is not prohibited from accessing all the card's resources if it knows that the card has dual decoders and that the Expansion ROM BAR content is correct.

4

4. PCI Services in ACPI

The Advanced Configuration and Power Interface (ACPI) Specification describes a set of common firmware interfaces that enable robust operating system-directed platform device configuration and power management of both individual devices and the entire system. ACPI uses tables to describe system information, features, and methods for controlling those features.

The following sections provide an overview of ACPI interfaces that are relevant to platforms that support PCI hierarchies consisting of Conventional PCI, PCI-X, and PCI Express. For further details, refer to the latest ACPI Specification available at http://www.acpi.info.

4.1. Enhanced Configuration Access Method Base Address

On PC-compatible systems, the enhanced configuration access mechanism allows PCI configuration space to be accessed using memory primitives rather than I/O-based primitives (CF8/CFC mechanism). For PCI Express and PCI-X Mode 2 hierarchies on these systems, the memory mapped configuration access mechanism is the only way to access the extended configuration space (offsets 256-4095).

This section defines an ACPI-based mechanisms to communicate the memory mapped configuration space base address(es) used for Enhanced Configuration Access Mechanism (defined in PCI-X and PCI Express Local Bus Specifications) to the operating system:

- MCFG: An ACPI table-based mechanism that is used to communicate the memory mapped configuration space base addresses corresponding to the (non-hot removable) PCI Segment Groups and/or base address ranges within a PCI Segment Group available to the system at boot. The memory mapped configuration space address ranges described exclusively through the table mechanism are considered to be non-relocatable and non-hot removable for the current boot.
- _CBA: An ACPI method that is used to report the Enhanced Configuration Access base address for any PCI Segment Groups and/or base address ranges within a PCI Segment group. This allows run-time update for the hot added PCI components.

The MCFG table is only used to communicate the base addresses corresponding to the non-hot removable PCI Segment Groups available to the system at boot. The _CBA method enables the system to describe the base address of the memory mapped configuration space for hot plug capable PCI Segment Groups.

4.1.1. Background

The PCI Express and PCI-X Specifications define the Enhanced Configuration Access Mechanism for the PC-compatible systems, which allows access to the configuration registers via memory mapped address space. The base address of this memory mapped configuration space is platform specific and is communicated to the operating system via system firmware.

In a hierarchy that supports the enhanced configuration access mechanism, the first 256 bytes (offsets 0-255) of PCI 2.3 compatible configuration space can be accessed by either the PCI2.3 configuration mechanism (CF8/CFC) or using the enhanced configuration mechanism. The extended register configuration space (region from offset 256-4095) can be accessed only via the enhanced configuration mechanism for PCI Express devices and Mode 2 PCI-X devices on the PC-compatible systems.

Table 4-1: Memory Address PCI Express Configuration Space

Memory Address	PCI	PCI-X Mode 1	PCI-X Mode 2	PCI Express
A[(20+n):20] Bus[n:0], where n=0 to7	Applies	Applies	Applies	Applies
A[19:15] Device[4:0]	Applies	Applies	Applies	Applies
A[14:12] Function[2:0]	Applies	Applies	Applies	Applies
A[11:8] Extended Register[3:0]	N/A	N/A	Applies	Applies
A[7:0] Register[7:0]	Applies	Applies	Applies	Applies

The 256-MB window of memory mapped configuration space (assuming maximum addressable 4 KB per function, eight functions per device, 32 devices per bus) defined by PCI-X and PCI Express is capable of describing the entire 256 bus PCI Segment Group as shown in Figure 4-1.

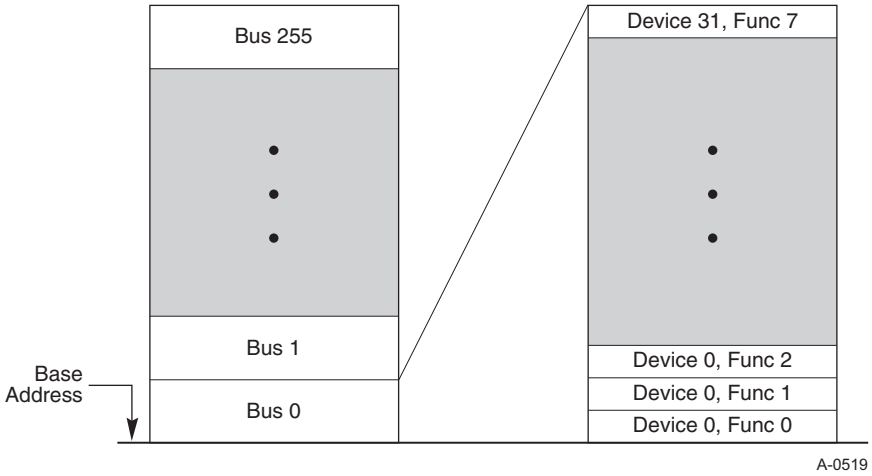

Figure 4-1: 256-MB Region for Enhanced Configuration Space Access Mechanism

The Enhanced Configuration Access Mechanism can be applied to access heterogeneous hierarchies consisting of PCI/PCI-X/PCI Express. In this case, the extended configuration space up to a limit of 4 KB per device-function is accessible for PCI Express devices and PCI-X Mode 2 devices. For

non-Mode 2 PCI-X and all PCI devices, only the first 256 bytes of configuration space are accessible.

Note that a given chipset implementation may choose to implement less than 256 MB for the memory mapped configuration space. Further, implementations with multiple host bridges or mixed hierarchies (for example, a multi-chip implementation with PCI Express as well as PCI-X at the root level) are allowed to implement the memory mapped configuration space in a discontinuous fashion; that is, the 256 PCI buses could be distributed across multiple host bridges in a non-overlapping fashion.

Note that in a multiple host bridge hierarchy, there is no requirement for the host bridges to program the buses within a PCI Segment Group or Groups in a contiguous fashion.

4.1.2. MCFG Table Description

The MCFG table is an ACPI table that is used to communicate the base addresses corresponding to the non-hot removable PCI Segment Groups range within a PCI Segment Group available to the operating system at boot. This is required for the PC-compatible systems.

The MCFG table is only used to communicate the base addresses corresponding to the PCI Segment Groups available to the system at boot. This table directly refers to PCI Segment Groups defined in the system via the _SEG object in the ACPI name space for the applicable host bridge device. For systems containing only a single PCI Segment Group, the default PCI Segment Group number, namely, PCI Segment Group 0, is implied. In such a case, the default PCI Segment Group need not be represented in the ACPI Name Space (i.e., no _SEG method is required in such a hierarchy).

The size of the memory mapped configuration region is indicated by the start and end bus number fields in the Memory mapped Enhanced configuration space base address allocation structure as shown in Table 4-3. 0-255 is the range of allowed bus numbers supported for a given PCI Segment Group.

Table 4-2 provides a description of the MCFG table.

Table 4-2: MCFG Table to Support Enhanced Configuration Space Access

Field	Byte Length	Byte Offset	Description
Header			
Signature	4	0	"MCFG". Signature for the Memory mapped configuration space base address Description Table. (refer to Note 1)
Length	4	4	Length, in bytes, of the entire MCFG Description table including the memory mapped configuration space base address allocation structures.
Revision	1	8	1
Checksum	1	9	Entire table must sum to zero
OEMID	6	10	OEM ID
OEM Table ID	8	16	For the MCFG Description Table, the table ID is the manufacture model ID
OEM Revision	4	24	OEM revision of MCFG table for supplied OEM Table ID
Creator ID	4	28	Vendor ID of utility that created the table
Creator Revision	4	32	Revision of utility that created the table
Reserved	8	36	Reserved
Configuration space base address allocation structure [n]	---	44	A list of the memory mapped configuration base address allocation structures. This list will contain one entry corresponding to each PCI Segment Group present in the platform. The structure of this entry is defined in Table 4-3.

Notes regarding Table 4-2:

1. A table signature "MCFG" is reserved for this purpose and the header for the table is shown in Table 4-2. Based on the signature and table revision, the operating system can then interpret the implementation-specific data within the table. The Table Revision for revision 1.0 of the MCFG table is set to 1.

2. If the operating system does not natively comprehend reserving the MMCFG region, the MMCFG region must be reserved by firmware. The address range reported in the MCFG table or by _CBA method (see Section 4.1.3) must be reserved by declaring a motherboard resource. For most systems, the motherboard resource would appear at the root of the ACPI namespace (under _SB) in a node with a _HID of EISAID (PNP0C02), and the resources in this case should not be claimed in the root PCI bus's _CRS. The resources can optionally be returned in Int15 E820 or EFIGetMemoryMap as reserved memory but must always be reported through ACPI as a motherboard resource.

3. This table must not include the memory mapped configuration base addresses for hot pluggable PCI Segment Groups. Such PCI Segment Groups must be described by using the _CBA method (refer to Section 4.1.3) in the corresponding ACPI name space object.

The structure in Table 4-3 describes the association between the PCI Segment Group and the corresponding memory mapped configuration base address. This table describes the details of this structure.

Table 4-3: Memory Mapped Enhanced Configuration Space Base Address Allocation Structure

Field	Byte Length	Byte Offset	Description
Base Address	8	0	Processor-relative Base Address for the Enhanced Configuration Access Mechanism
PCI Segment Group Number	2	8	PCI Segment Group Number. Default is 0. For all other PCI Segment Groups, this field value should correspond to the value returned by _SEG object in ACPI name space for the applicable host bridge device.
Start Bus Number	1	10	Start PCI Bus number decoded by the host bridge
End Bus Number	1	11	End PCI Bus number decoded by the host bridge
Reserved	4	12	Reserved

The MCFG table format allows for more than one memory mapped base address entry provided each entry (memory mapped configuration space base address allocation structure) corresponds to a unique PCI Segment Group consisting of 256 PCI buses. Multiple entries corresponding to a single PCI Segment Group is not allowed.

- The PCI Segment Group Number field denotes the PCI Segment Group corresponding to the base address field in the structure. For systems supporting multiple PCI Segment Groups, this field should correspond to the value returned by _SEG object in ACPI name space for the applicable host bridge device. If the system only contains a single (default) PCI Segment Group, namely, PCI Segment Group 0, no corresponding _SEG object is required.
- The base address field provides the 64-bit physical address of the base of the memory mapped configuration space associated with the PCI Segment Group. It is the responsibility of the provider of the table to ensure that the base address reported is consistent with the requirements for the hardware implementation. For PCI-X and PCI Express platforms utilizing the enhanced configuration access method, the base address of the memory mapped configuration space always corresponds to bus number 0 (regardless of the start bus number decoded by the host bridge) and further must comply with alignment requirements of the corresponding local bus specification. The unsupported upper bits of the physical address must be set to 0.

4.1.3. The _CBA Method

Some systems may support hot plug of host bridges that introduce either a range of buses within an existing PCI Segment Group or introduce a new PCI Segment Group. For example, each I/O chip in a multi-chip PCI Express root complex implementation could start a new PCI Segment Group. The base address of the memory mapped configuration space for such a hot pluggable PCI Segment Group or a range of buses within a PCI Segment Group is described using an ACPI control

method, _CBA, that is under the host bridge devices that are part of the PCI Segment Group. This applies to PC-compatible systems only.

The _CBA (Memory mapped Configuration Base Address) control method is an optional ACPI object that returns the 64-bit memory mapped configuration base address for the hot plug capable host bridge. The base address returned by _CBA is processor-relative address. The _CBA control method evaluates to an Integer.

This control method appears under a host bridge object. When the _CBA method appears under an active host bridge object, the operating system evaluates this structure to identify the memory mapped configuration base address corresponding to the PCI Segment Group for the bus number range specified in _CRS method. An ACPI name space object that contains the _CBA method must also contain a corresponding _SEG method.

For a host bridge that includes _CBA, the _CBA and _BBN control methods have to be executed first to enable PCI_Config_OpRegion access for devices below the bridge. As a result, the _CBA and BBN methods must not include PCI_Config opregions that refer to devices below the host bridge.

A set of hot pluggable host bridges could have _CBA under each of the host bridge devices, where each host bridge device is typically described in the ACPI name space with PNP0A08 for _HID and PNP0A03 for _CID. In this case, the memory mapped configuration base address (always corresponds to bus number 0) for the PCI Segment Group of the host bridge is provided by _CBA and the bus range covered by the base address is indicated by the corresponding bus range specified in _CRS.

If rebalancing of resources on a host bridge is supported via _PRS, _SRS, it is the responsibility of the operating system to reevaluate _CBA every time _CRS is evaluated.

Memory mapped configuration base addresses for non-hot pluggable host bridges must be described using MCFG table.

_CBA Control Method

Arguments:

None

Result Code:

Memory mapped configuration base address for the PCI-compatible host bridge returned as an integer

Note: Starting with ACPI2.0, integers are 64-bit entities.

Example ASL for _CBA usage:

```
Scope(\_SB) {
      …
Device(PCI1) {                      // Root PCI Bus
      Name(_HID, EISAID("PNP0A03")) // Need _HID for root device
      Name(_SEG, 1)                 // PCI Segment Group 1
      Method (_CRS, ResourceTemplate()
      {
      WORDBusNumber(ResourceProducer,MinFixed,MaxFixed,PosDecode,0x0,0x
      0,0x0,0xFF,0x100,,,)          // Bus range 0-255
      Method(_CBA, 0) {
Return (0xE000000000000000) // Bits 63:0 of the base address

      } // end of _CBA method
      } // end PCI1
}     // end scope SB
```

4.1.4. System Software Implication of MCFG and _CBA

The base address returned by MCFG table for a given PCI Segment group is always with respect to bus 0 as specified in the PCI Express Base Specification (and the PCI-X Specification). It is the responsibility of system software to calculate the start and end of the supported memory mapped configuration address range based on the start and end bus numbers specified in the MCFG entry. System software must make no assumptions about the memory range corresponding to the base address up to the start of the memory mapped configuration space (as specified by start bus number).

The base address returned by _CBA for a given PCI Segment Group is always with respect to bus 0 as specified in the PCI Express Base Specification (and the PCI-X Specification). It is the responsibility of system software to calculate the start and end of the supported memory mapped configuration address range for the host bridge based on the bus range supported by the host bridge as identified by _CRS.

IMPLEMENTATION NOTE

Multiple Host Bridges

A platform may have multiple PCI Express or PCI-X host bridges. The base address for the MMCONFIG space for these host bridges may need to be allocated at different locations. In such cases, using MCFG table and _CBA method as defined in this section means that each of these host bridges must be in its own PCI Segment Group.

If this platform also needs to support legacy operating systems or x86 BIOS Option ROMs, the CF8/CFCh access mechanism (which is PCI Segment Group unaware and only can support up to 256 bus numbers) must also be supported. There may be a number of implementation choices to make these two work together; for example, the bus numbers in each of the PCI Segment Group can be made to not overlap. This would make the CF8/CFCh access still appear to be Segment Group unaware and support up to 256 buses while using the MCFG/_CBA definition to describe host bridges MMCONFIG base addresses that are allocated at different locations.

Note that in this arrangement, even though the PCI Segment Group concept is used, the total number of PCI buses is still limited to 256 due to the CF8/CFCh limitation.

4.1.5. Plug-and-Play ID Defined for Enhanced Configuration Space Access Capable Devices

Currently, a Plug-and-Play ID (PNP ID) of PNP0A03 is used to indicate host-PCI bridge devices in ACPI name space. This PNP ID is used to describe both PCI/PCI-X hierarchies.

A new PNP ID of PNP0A08 is defined to indicate PCI Express as well as PCI-X Mode 2 host bridges to the operating system. The unique ID for these host bridges will allow the host to distinguish a PCI Express, PCI-X Mode 2 hierarchy in a mixed host-bridge environment and also allow the operating system to tune the driver loading process to the capabilities of the underlying I/O hierarchy.

To retain compatibility with older operating systems that do not recognize the new PNP ID, when PNP0A08 is used to describe a device in the namespace, it is also required to include PNP0A03 as the compatible ID (_CID).

Example ASL for PNP0A08 usage:

```
Device(PCI1) {                        // Root PCI Bus
      Name(_HID, EISAID("PNP0A08")) // Indicates PCI Express/PCI-X
      Mode 2 host hierarchy
      Name(_CID, EISAID("PNP0A03")) // To support legacy OS that does
                                    // not understand the new HID
      }
```

Notes regarding use of PNPID and PNP0A08 to indicate PCI Express/PCI-X Mode 2 host bridge hierarchy:

1. PNP0A08 only indicates support for extended configuration space, that is, each device function supports 4 K of configuration space. The support for memory mapped configuration access is indicated by MCFG/_CBA. For example, host bridges with extended configuration space support on DIG64-compliant systems, would indicate the host bridge capabilities using PNP0A08 but will not include MCFG table or _CBA methods.
2. For PCI-X Mode 2 host bridges, PNP ID of PNP0A08 only indicates that the host bridge is capable of supporting extended configuration space. OSPM must call _DSM to identify if the extended configuration capabilities are currently enabled.

4.2. Mechanism for Controlling System Wake From PCI Express

An errant PCI Express device may prevent the system from going to sleep by continuously asserting the PCI Express wake signal. The PCI Express wake registers, defined in ACPI FADT PM register, allows OSPM to disable the wake signal to allow the system to enter a sleep state.

Complete details are contained in the ACPI 3.0 Specification. The ACPI 3.0 Specification is located at http://www.acpi.info.

4.3. PCI Root Bridge Description

4.3.1. Identification

PCI Host bridge devices in ACPI name space are identified by Plug-and-Play ID (PNP ID). PNP0A03 is used to indicate PCI/PCI-X host bridge hierarchies.

A PNP ID of PNP0A08 is used to represent PCI Express and PCI-X Mode 2 host bridge hierarchies to indicate support for extended configuration space.

For further details, refer to Section 4.1.4.

4.3.2. Resource Description

4.3.2.1. Resource Setting

Host bridges resources programming is communicated to the operating system using ACPI methods _CRS, _SRS, and _PRS. _CRS indicates the current resource setting for the host bridge. This includes I/O space, memory space, and bus range assigned to the bridge by platform firmware.

A non-configurable device only specifies _CRS. However, if they are configurable, devices include _PRS to indicate the possible resource setting and _SRS to allow OSPM to specify a new resource allocation for the device.

4.3.2.2. Boot Bus Number

In multiple host bridge systems, _BBN method is used to provide PCI_Config Operation Region access for a specific bus.

4.3.2.3. PCI Segment Group

PCI Segment Group concept enables support for more than 256 buses in a system by allowing the reuse of the PCI bus numbers. The _SEG method is used to uniquely identify PCI Segment Groups.

Complete details are located in the ACPI 3.0 Specification. The ACPI 3.0 Specification is located at http://www.acpi.info.

4.4. PCI Interrupt Routing

The routing of PCI INTx interrupts to interrupt controllers cannot be deduced through the PCI register space. ACPI object, _PRT, is required under all PCI host bridges and is used to indicate the routing of the PCI INTx interrupts to the interrupt controllers. If the Root Ports are described in the ACPI namespace and have an associated _PRT, operating systems must evaluate and use routing as described in the _PRT.

MSI and MSI-X are standardized by the PCI-X and the PCI Express Base Specifications. ACPI is not involved in the use of MSI or MSI-X.

4.5. _OSC – A Mechanism for Exposing PCI Express Capabilities Supported by an Operating System

_OSC is an object that is used by OSPM to query the capabilities of a device (as defined in ACPI) and to communicate to the platform the feature support or capabilities provided by a device's driver. This object is a child object of the device in the ACPI namespace. Device specific objects are evaluated after _OSC invocation. For a complete description of _OSC method, refer to the ACPI 3.0 Specification. The ACPI 3.0 Specification can be found at http://www.acpi.info/.

4.5.1. _OSC Interface for PCI Host Bridge Devices

The _OSC interface defined in this section applies only to "Host Bridge" ACPI devices that originate PCI, PCI-X, or PCI Express hierarchies. These ACPI devices must have a _HID of (or a _CID including either EISAID("PNP0A03") or EISAID("PNP0A08")). For a host bridge device that originates a PCI Express hierarchy, the _OSC interface defined in this section is required. For a host bridge device that originates a PCI/PCI-X bus hierarchy, inclusion of an _OSC object is optional.

The _OSC interface for a PCI/PCI-X/PCI Express hierarchy is identified by the Universal Unique IDentifier (UUID) 33db4d5b-1ff7-401c-9657-7441c03dd766. A revision ID of 1 encompasses fields defined in this section of this revision of this specification comprised of three DWORDs, including the first DWORD described by the generic ACPI definition of _OSC.

The first DWORD in the _OSC Capabilities Buffer contains bits that are generic to _OSC. These include status and error information.

The second DWORD in the _OSC Capabilities Buffer is the Support Field. Bits defined in the Support Field provide information regarding operating system supported features. Contents in the Support Field are passed one way; the operating system will disregard any changes to this field when returned.

The third DWORD in the _OSC Capabilities Buffer is the Control Field. Bits defined in the Control Field are used to submit requests by the operating system for control/handling of the associated feature, typically (but not excluded to) those features that utilize native interrupts or events handled by an operating system-level driver. If any bits in the Control Field are returned cleared (masked to zero) by the _OSC control method, the respective feature is designated unsupported by the platform and must not be enabled by the operating system. Some of these features may be controlled by platform firmware prior to operating system boot or during runtime for a legacy operating system, while others may be disabled/inoperative until native operating system support is available.

If the _OSC control method is absent from the scope of a host bridge device, then the operating system must not enable or attempt to use any features defined in this section for the hierarchy originated by the host bridge. Doing so could contend with platform firmware operations or produce undesired results. It is recommended that a machine with multiple host bridge devices

should report the same capabilities for all host bridges and also negotiate control of the features described in the Control Field in the same way for all host bridges.

Table 4-4: Interpretation of the _OSC Support Field

Support Field Bit Offset	Interpretation
0	**Extended PCI Config operation regions supported** The operating system sets this bit to 1 if it supports ASL accesses through PCI Config operation regions to extended configuration space (offsets greater than 0xFF). Otherwise, the operating system sets this bit to 0.
1	**Active State Power Management supported** The operating system sets this bit to 1 if it natively supports configuration of Active State Power Management registers in PCI Express devices. Otherwise, the operating system sets this bit to 0.
2	**Clock Power Management Capability supported** The operating system sets this bit to 1 if it supports the Clock Power Management Capability and will enable this feature during a native hot plug insertion event if supported by the newly added device. Otherwise, the operating system sets this bit to 0. Note: The Clock Power Management Capability is defined in an errata to the *PCI Express Base Specification, Revision 1.0.*
3	**PCI Segment Groups supported** The operating system sets this bit to 1 if it supports PCI Segment Groups as defined by the _SEG object and access to the configuration space of devices in PCI Segment Groups as described by this specification. Otherwise, the operating system sets this bit to 0.
4	**MSI supported** The operating system sets this bit to 1 if it supports configuration of devices to generate message-signaled interrupts, either through the MSI Capability or the MSI-X Capability. Otherwise, the operating system sets this bit to 0.
5-31	Reserved

Table 4-5: Interpretation of the _OSC Control Field, Passed in via Arg3

Control Field Bit Offset	Interpretation
0	**PCI Express Native Hot Plug control** The operating system sets this bit to 1 to request control over PCI Express native hot plug. If the operating system successfully receives control of this feature, it must track and update the status of hot plug slots and handle hot plug events as described in the PCI Express Base Specification.
1	**SHPC Native Hot Plug control** The operating system sets this bit to 1 to request control over PCI/PCI-X Standard Hot-Plug Controller (SHPC) hot plug. If the operating system successfully receives control of this feature, it must track and update the status of hot plug slots and handle hot plug events as described in the SHPC Specification.
2	**PCI Express Native Power Management Events control** The operating system sets this bit to 1 to request control over PCI Express native power management event interrupts (PMEs). If the operating system successfully receives control of this feature, it must handle power management events as described in the PCI Express Base Specification.
3	**PCI Express Advanced Error Reporting control** The operating system sets this bit to 1 to request control over PCI Express Advanced Error Reporting. If the operating system successfully receives control of this feature, it must handle error reporting through the Advanced Error Reporting Capability as described in the PCI Express Base Specification; further, the operating system retains control of AER across power transitions for S1, S2, S3 system power states.
4	**PCI Express Capability Structure control** The operating system sets this bit to 1 to request control over the PCI Express Capability structures (standard and extended) defined in the *PCI Express Base Specification, Revision 1.1.* These capability structures are the PCI Express Capability, the Virtual Channel Extended Capability, the Power Budgeting Extended Capability, the Advanced Error Reporting Extended Capability, and the Serial Number Extended Capability. If the operating system successfully receives control of this feature, it is responsible for configuring the registers in all PCI Express Capabilities in a manner that complies with the PCI Express Base Specification. Additionally, the operating system is responsible for saving and restoring all PCI Express Capability register settings across power transitions to and from S1, S2, S3 system power states when register context may have been lost.
5-31	Reserved

Table 4-6: Interpretation of the _OSC Control Field, Returned Value

Control Field Bit Offset	Interpretation
0	**PCI Express Native Hot Plug control** The firmware sets this bit to 1 to grant control over PCI Express native hot plug interrupts. If firmware allows the operating system control of this feature, then in the context of the _OSC method, it must ensure that all hot plug events are routed to device interrupts as described in the PCI Express Base Specification. Additionally, after control is transferred to the operating system, firmware must not update the state of hot plug slots, including the state of the indicators and power controller. If control of this feature was requested and denied or was not requested, firmware returns this bit set to 0.
1	**SHPC Native Hot Plug control** The firmware sets this bit to 1 to grant control over control over PCI/PCI-X Standard Hot-Plug Controller (SHPC) hot plug. If firmware allows the operating system control of this feature, then in the context of the _OSC method, it must ensure that all hot plug events are routed to device interrupts as described in the SHPC 1.0. Additionally, after control is transferred to the operating system, firmware must not update the state of hot plug slots, including the state of the indicators and power controller. If control of this feature was requested and denied or was not requested, firmware returns this bit set to 0.
2	**PCI Express Native Power Management Events control** The firmware sets this bit to 1 to grant control over control over PCI Express native power management event interrupts (PMEs). If firmware allows the operating system control of this feature, then in the context of the _OSC method, it must ensure that all PMEs are routed to root port interrupts as described in the PCI Express Base Specification. Additionally, after control is transferred to the operating system, firmware must not update the PME Status field in the Root Status register or the PME Interrupt Enable field in the Root Control register. If control of this feature was requested and denied or was not requested, firmware returns this bit set to 0.
3	**PCI Express Advanced Error Reporting control** The firmware sets this bit to 1 to grant control over PCI Express Advanced Error Reporting. If firmware allows the operating system control of this feature, then in the context of the _OSC method, it must ensure that error messages are routed to device interrupts as described in the PCI Express Base Specification. Additionally, after control is transferred to the operating system, firmware must not modify the Advanced Error Reporting Capability. If control of this feature was requested and denied or was not requested, firmware returns this bit set to 0.
4	**PCI Express Capability Structure control** The firmware sets this bit to 1 to grant control over the PCI Express Capability. If the firmware does not grant control of this feature, firmware must handle configuration of the PCI Express Capability Structure. If firmware grants the operating system control of this feature, any firmware configuration of the PCI Express Capability may be overwritten by an operating system configuration, depending on operating system policy.
5-31	Reserved

4.5.2. Rules for Evaluating _OSC

This section defines when and how the operating system must evaluate _OSC as well as restrictions on firmware implementations.

4.5.2.1. Query Flag

If the Query Support Flag (Capabilities DWORD 1, bit 0) is set by the operating system when evaluating _OSC, no hardware settings are permitted to be changed by firmware in the context of the _OSC call. It is strongly recommended that the operating system evaluate _OSC with the Query Support Flag set until _OSC returns the Capabilities Masked bit clear, to negotiate the set of features to be granted to the operating system for native support. A platform may require a specific combination of features to be supported natively by an operating system before granting native control of a given feature.

4.5.2.2. Evaluation Conditions

The operating system must evaluate _OSC under the following conditions:

- During initialization of any driver that provides native support for features described in the section above. These features may be supported by one or many drivers, but should only be evaluated by the main bus driver for that hierarchy. Secondary drivers must coordinate with the bus driver to install support for these features. Drivers may not relinquish control of features previously obtained; i.e., bits set in Capabilities DWORD3 after the negotiation process must be set on all subsequent negotiation attempts.
- When a Notify(<device>, 8) is delivered to the PCI Host Bridge device.
- Upon resume from S4. Platform firmware will handle context restoration when resuming from S1-S3.

4.5.2.3. Sequence of _OSC Calls

The following rules govern sequences of calls to _OSC that are issued to the same host bridge and occur within the same boot.

- The operating system is permitted to evaluate _OSC an arbitrary number of times.
- If the operating system declares support of a feature in the Status Field in one call to _OSC, then it must preserve the set state of that bit (declaring support for that feature) in all subsequent calls.
- If the operating system is granted control of a feature in the Control Field in one call to _OSC, then it must preserve the set state of that bit (requesting that feature) in all subsequent calls.
- Firmware may not reject control of any feature it has previously granted control to.

There is no mechanism for the operating system to relinquish control of a feature previously requested and granted.

4.5.2.4. Dependencies Between _OSC Control Bits

Because handling of hot-plug events, power management events, and advanced error reporting all require the modification of PCI Express Capability registers, the operating system is required to claim control over the PCI Express Capability (bit 4 of the Control field) in conjunction with claiming control over PCI Express Native Hot Plug, PCI Express Native Power Management Events, or PCI Express Advanced Error Reporting (bits 0, 2, and 3 of the Control field). If the operating system attempts to claim control of any of these features without also claiming control over the PCI Express Capability, the firmware is required to refuse control of the feature being illegally claimed and mask the corresponding bit.

4.5.3. ASL Example

A sample _OSC implementation for a mobile system incorporating a PCI Express hierarchy is shown below:

```
Device(PCI0) // Root PCI bus
{
   Name(_HID,EISAID("PNP0A08"))  // PCI Express Root Bridge
   Name(_CID,EISAID("PNP0A03"))  // Compatible PCI Root Bridge

   Name(SUPP,0) // PCI _OSC Support Field value
   Name(CTRL,0) // PCI _OSC Control Field value

   Method(_OSC,4)
   {  // Check for proper UUID
      If(LEqual(Arg0,ToUUID("33DB4D5B-1FF7-401C-9657-7441C03DD766")))
      {
         // Create DWORD-addressable fields from the Capabilities Buffer
         CreateDWordField(Arg3,0,CDW1)
         CreateDWordField(Arg3,4,CDW2)
         CreateDWordField(Arg3,8,CDW3)

         // Save Capabilities DWORD2 & 3
         Store(CDW2,SUPP)
         Store(CDW3,CTRL)

         // Only allow native hot plug control if the OS supports:
         //  * ASPM
         //  * Clock PM
         //  * MSI/MSI-X
         If(LNotEqual(And(SUPP, 0x16), 0x16))
         {
            And(CTRL,0x1E,CTRL) // Mask bit 0 (and undefined bits)
         }

         // Always allow native PME, AER (no dependencies)

         // Never allow SHPC (no SHPC controller in this system)
         And(CTRL,0x1D,CTRL)
```

```
        If(Not(And(CDW1,1)))    // Query flag clear?
        {  // Disable GPEs for features granted native control.
           If(And(CTRL,0x01))  // Hot plug control granted?
           {
              Store(0,HPCE)// clear the hot plug SCI enable bit
              Store(1,HPCS)// clear the hot plug SCI status bit
           }
           If(And(CTRL,0x04))  // PME control granted?
           {
              Store(0,PMCE)// clear the PME SCI enable bit
              Store(1,PMCS)// clear the PME SCI status bit
           }
           If(And(CTRL,0x10))  // OS restoring PCI Express cap structure?
           {  // Set status to not restore PCI Express cap structure
              // upon resume from S3
              Store(1,S3CR)
           }
        }

        If(LNotEqual(Arg1,One))
        {  // Unknown revision
           Or(CDW1,0x08,CDW1)
        }

        If(LNotEqual(CDW3,CTRL))
        {  // Capabilities bits were masked
           Or(CDW1,0x10,CDW1)
        }
        // Update DWORD3 in the buffer
        Store(CTRL,CDW3)
        Return(Arg3)
     } Else {
        Or(CDW1,4,CDW1)     // Unrecognized UUID
        Return(Arg3)
     }
  }  // End _OSC

  // End PCI0
```

4.6. _DSM Definitions for PCI

_DSM (Device Specific Method) is defined in the ACPI 3.0 Specification. This object is a control method that enables devices to provide device specific control functions that are consumed by the device driver. Table 4-7 below lists the UUID, revision, and function definitions.

Table 4-7: _DSM Definitions for PCI

UUID	Revision	Function	Description
E5C937D0-3553-4d7a-9117-EA4D19C3434D	1	1	PCI Express Slot Information
	1	2	PCI Express Slot Number
	1	3	Vendor-specific Token ID
	1	4	PCI Bus Capabilities

4.6.1. _DSM for PCI Express Slot Information

This section describes how the PCI Express slot information is exposed through the _DSM ACPI method. In the future, this information may be reported through hardware mechanisms that the operating system has direct access to; if and when that is available, the information provided through the hardware mechanism overrides the information provided through the mechanism defined in this section. The operating system should not use the mechanism defined in this section.

Note: The SMBIOS Specification defines the Type 9 entry structure and enumerations that allows reporting slot information for PCI/PCI-X/PCI Express slots. However, the SMBIOS access mechanism does not comprehend platforms with dynamic characteristics that would render statically reported SMBIOS data invalid. _DSM mechanism is used on such platforms to report the equivalent of the SMBIOS Type 9 slot information. The UUID in _DSM in this context is {E5C937D0-3553-4d7a-9117-EA4D19C3434D}, the revision is 1, and the function is 1.

Note: Function 0 is a generic Query function that is supported by _DSMs with any UUID and Revision ID. The definition of function 0 is generic to _DSM and specified in the *ACPI Specification, Version 3.0*.

Location:

This object will be placed under the virtual PCI-to-PCI bridge object representing the PCI Express root port or switch port that generates the slot on the motherboard.

Arguments:

Arg0: UUID: E5C937D0-3553-4d7a-9117-EA4D19C3434D

Arg1: Revision ID: 1

Arg2: Function Index: 1

Arg3: Empty Package

Return:

ACPI Buffer type; the definition of the return is a package of two items and the description is as follows:

Package item 1:

Type: Integer

Purpose: status of operation

Description:

0: Failure

1: Success

Package item 2:

Type: Package

Purpose: PCI Express Slot Information

Description: A package of three integers:

Integer 1:

Bit Position	
0	Supports x1
1	Supports x2
2	Supports x4
3	Supports x8
4	Supports x12
5	Supports x16
Others	Reserved

This integer indicates the link width capabilities of the slot. More than one bit can be set in this field to indicate the link widths supported by the slot. For example, a x8 slot that is capable of supporting x4, x2, and x1 would be indicated by setting bits 0, 1, and 2 in this field. This field indicates the downshift capabilities to management software. For maximum and negotiated link width, refer to the PCI Express Base Specification.

Integer 2:

Value	Meaning
0h	Unknown
1h	PCI Express Card Slot
2h	PCI Express Server I/O Module Slot
3h	PCI Express ExpressCard* Slot
4h	PCI Express Mini Card Slot
5h	PCI Express Wireless Form Factor Slot
Others	Reserved

This field indicates the type of the slot.

Integer 3:

Bit Position	
0	SMBus signal
1	WAKE# signal
Others	Reserved

This field indicates the supported signal(s) of the slot. More than one bit can be set in this field to indicate the signals supported by the slot.

4.6.2. _DSM for PCI Express Slot Number

This section describes how PCI Express slot number information is exposed through the _DSM ACPI method. It provides a method to break the monolithic slot numbers that are already provided both in ACPI firmware (through the _SUN method) and in PCI Express hardware (through the PCI-to-PCI bridge Chassis ID and the PCI Express Slot Number) into tokens that each represent a discrete hardware element. This allows the slot numbers to be displayed to the user in a more meaningful way.

This method only indicates the system-specific mechanism for interpreting the sub-fields of slot numbers. It does not mandate the exact format in which this data must be displayed to the user.

This method may exist in any device in the _SB scope in the ACPI namespace. The information returned from this method applies to any PCI device underneath the device containing the method, unless it is over-ridden by another instance of this method further down in the namespace tree.

The UUID in _DSM in this context is {E5C937D0-3553-4d7a-9117-EA4D19C3434D}, the revision is 1, and the function is 2.

Note: Function 0 is a generic Query function that is supported by _DSMs with any UUID and Revision ID. The definition of function 0 is generic to _DSM and specified in the *ACPI Specification, Version 3.0*.

Arguments:

Arg0: UUID: E5C937D0-3553-4d7a-9117-EA4D19C3434D

Arg1: Revision ID: 1

Arg2: Function Index: 2

Arg3: Empty Package

Return:

A Package of token packages. The returned package must contain at lease one token package.

Each sub-package represents a slot number token, which is a bit range within one of the monolithic slot numbers that has a distinct meaning. For example, bits 4-7 of the PCI Express Slot Number could represent the I/O Cabinet the slot sits in, while bits 0-3 would be the slot number within the cabinet. Each of these fields would be represented by a different token.

Each token is a package of four items:

Source ID – This field indicates the source data of the bit range that this token represents.

Token ID – This field indicates the type of element the token represents. Token IDs 0-0x7F are reserved by this specification. Token IDs 0x80-0xFF are vendor-specific.

Start bit – This field indicates the first bit number of the token in the source data.

End bit – This field indicates the last bit number of the token in the source data.

Token IDs:

0 – Chassis

1 – Cabinet

2 – I/O Tray

3 – Module

4 – Slot

5-0x7F – Reserved

0x80-0xFF – Vendor-specific

Source IDs:

0 – The _SUN method in the ACPI namespace

1 – The data in the Chassis ID register of the PCI-to-PCI bridge Slot Numbering Capability

2 – The data in the Physical Slot Number field of the Slot Capabilities register in the PCI Express Capability

Example:

```
    Method (_DSM, 3) {                    // Assume GUID and revision match
        Return (    Package(3){           // PCI Express Slot Parsing
                     Package(4){
                        1,1,2,7          // bit 7:2 in the PPB Chassis ID
                     },                 // Token represents a “Cabinet”
                     Package(4){
                        1,2,0,1          // bit 1:0 in the PPB Chassis ID
                     },                 // Token represents an “I/O Tray”
                     Package(4){
                        2,4,0,7          // bit 7:0 in the PCI Express Physical
                                         // Slot Number
                     }
                })
    }
```

If the PPB Chassis ID of a device in the hierarchy under the location of this method contained the value 01001110b and the PCI Express Physical Slot Number contained 17Ah, the device's slot number could be displayed as “Cabinet 0x13, I/O Tray 2, Slot 0x7A”.

4.6.3. _DSM for Vendor-specific Token ID Strings

For vendor-specific token IDs, the tokens themselves do not provide enough information to display slot number information to the user. This section provides a method for the BIOS to return a human readable description in multiple languages of the hardware element represented by a vendor-specific token ID. In the example usage of the _DSM for PCI Express Slot Number in Section 4.6.2, if a vendor-specific token ID is used, the resulting string displayed to the user would contain a string returned from this method instead of a standard string like “Cabinet” or “Slot”.

This method returns a package of packages. Each sub-package consists of a Language identifier and corresponding unicode string for a given locale. Specifying a language identifier allows OSPM to easily determine if support for displaying the Unicode string is available. OSPM can use this information to determine whether or not to display the device string, or which string is appropriate for a user's preferred locale. It is assumed that OSPM will always support the primary English locale to accommodate English embedded in a non-English string, such as a brand name. If OSPM does not support the specific sub-language ID, it may choose to use the primary language ID for displaying device text.

The language IDs returned by this method indicate that the corresponding string follows the format specified in RFC 3066. Strings are returned in Unicode (UTF-16).

In addition to supporting the existing strings in RFC 3066, the following aliases are also supported:

RFC String	Supported Alias String
zh-Hans	zh-chs
zh-Hant	zh-cht

This method must exist in the same context as the _DSM for PCI Express Slot Number. The UUID in _DSM in this context is {E5C937D0-3553-4d7a-9117-EA4D19C3434D}, the revision is 1, and the function is 3.

Note: Function 0 is a generic Query function that is supported by _DSMs with any UUID and Revision ID. The definition of function 0 is generic to _DSM and specified in the *ACPI Specification, Version 3.0*.

Arguments:

Arg0: UUID: E5C937D0-3553-4d7a-9117-EA4D19C3434D

Arg1: Revision ID: 1

Arg2: Function Index: 3

Arg3: A package containing one element. The element is the Token ID to return the string for. The Token ID must be an integer in the range 0x80-0xFF.

Return:

A package of packages.

Each sub-package contains two elements:

The language ID of the string

The Unicode string representing the token ID

4.6.4. _DSM for PCI Bus Capabilities

This section defines a mechanism for a PCI root bus to report its capabilities and operating mode to software. PCI buses that are created by PCI to PCI bridges already report this information through hardware registers in the configuration space header, PCI-X capability and/or SHPC capability. Since root buses are exposed through ACPI, no such hardware registers exist, and this information must be described in a different manner.

The _DSM method, defined in ACPI 3.0, is used to provide the information about the capabilities of a root bus. The UUID used for the bus capabilities information and the structure of the buffer returned is described below.

Note: SMBIOS has type 9 entry reporting slot information, but the interface is static with no dynamic update capability. The entry has not been updated to reflect the most current technologies. The _DSM mechanism replaces the SMBIOS interface.

4.6.4.1. Bus Capabilities Structure

This section defines the concept of Bus Capabilities Structure. This structure describes the different attributes of a root bus. The Bus Capabilities Structure is bus-type specific and is returned by the _DSM defined here. A root bus can only have a single Bus Capability Structure.

The Bus Capabilities Structure must be aligned to a 4-byte boundary to ensure that future structure definitions do not cause any alignment issues for the consumer of these structures.

4.6.4.1.1. Bus Types

Table 4-8 describes the different bus types that can be used in the type field of the Bus Capabilities Structures. The type field is 2 bytes in size.

Table 4-8: Bus Types

Type	Description
0x0	Reserved
0x1	PCI
2+	Reserved

4.6.4.1.2. Bus Capabilities Structure Definitions

The Bus Capabilities Structures, for different bus types, are discussed in the following tables.

Table 4-9: PCI Bus Capability Structure

Field	Byte Length	Byte Offset	Description
Type	2	0	PCI (Type 0x1)
Length	2	2	The length of the structure, in bytes, including the type, starting from offset 0. This field is used to record the size of the entire structure. This field must return 16.
Attributes	1	4	The bit corresponding to each attribute of the bus should be set: 01h: 64-bit Device – This bit should be set if the secondary interface of the bus is 64 bits wide. 02h: PCI-X Mode 1 ECC Capable – This bit should be set if the bus is capable of supporting ECC in PCI-X Mode 1. 04h: Device ID Messaging Capable – This bit should be set if the bridge is capable of forwarding Device ID Message transactions.

Field	Byte Length	Byte Offset	Description
Current Speed/Mode	1	5	Defined encodings are as follows: 00000000b – 33 MHz Conventional Mode 00000001b – 66 MHz Conventional Mode 00000010b – 66 MHz PCI-X with Parity (Mode 1) 00000011b – 100 MHz PCI-X with Parity (Mode 1) 00000100b – 133 MHz PCI-X with Parity (Mode 1) 00000101b – 66 MHz PCI-X with ECC (Mode 1) 00000110b – 100 MHz PCI-X with ECC (Mode 1) 00000111b – 133 MHz PCI-X with ECC (Mode 1) 00001000b – 66 MHz PCI-X 266 (Mode 2) 00001001b – 100 MHz PCI-X 266 (Mode 2) 00001010b – 133 MHz PCI-X 266 (Mode 2) 00001011b – 66 MHz PCI-X 533 (Mode 2) 00001100b – 100 MHz PCI-X 533 (Mode 2) 00001101b – 133 MHz PCI-X 533 (Mode 2) 00001110b to 11111110b – Reserved
Supported Speeds/Modes	2	6	The bit corresponding to each supported bus speed/mode should be set: Bit[5:0] 000001b: Conventional PCI 33 MHz 000010b: Conventional PCI 66 MHz 000100b: PCI-X 66 MHz 001000b: PCI-X 133 MHz 010000b: PCI-X 266 MHz 100000b: PCI-X 533 MHz Bit[15:6] Reserved
Voltage	1	8	0: 3.3 V 1: 5 V 2+: Reserved
Reserved	7	9	0

4.6.4.1.3. _DSM for Bus Capabilities

This section describes how the Bus Capabilities structure is exposed through the _DSM ACPI method.

The _DSM method must appear in the context of a PCI root bus, identified by a _HID or _CID of PNP0A03. The UUID for reporting a Bus Capabilities structure in a _DSM in this context is {E5C937D0-3553-4d7a-9117-EA4D19C3434D}, the revision is 1, and the function is 4.

Note: Function 0 is a generic Query function that is supported by _DSMs with any UUID and Revision ID.

Location:

This object will be placed under the object representing the PCI or PCI-X root bus in the ACPI namespace. This object should have a _HID or _CID of PNP0A03.

Arguments:

Arg0: UUID: E5C937D0-3553-4d7a-9117-EA4D19C3434D

Arg1: Revision ID: 1

Arg2: Function Index: 4

Arg3: Empty Package

Return:

ACPI Buffer type; the definition of the return a package of two items and the description is as follows.

Package item 1:

Type: Integer

Purpose: status of operation

Description:

0: Failure

1: Success

Package item 2:

Type: Buffer

Purpose: Capabilities structure

Description: The buffer layout will be as follows:

Field	Byte Length	Byte Offset	Value	Description
Version	2	0	1	Version of the return buffer
Status	2	2	Zero or Ones	Status return of this method: Zero indicates success, Ones indicate failure.
Length	4	4	Computed	The length of the entire buffer, in bytes, including the version, starting from offset 0. This field is used to record the size of the entire buffer.
Bus Capabilities Structure	Dependent on capability type and size of structure	8	Computed	The bus capabilities as defined in the Bus Capabilities Structure Definitions

4.7. Generic ACPI PCI Slot Description

To describe a PCI slot, either for manageability purposes or for supporting the ACPI-based PCI hot plug, the ACPI name space must list eight device objects corresponding to the possible eight functions on the device in the slot. For each of the device object, _ADR is used to identify the PCI address (device number and function number) for each of the functions. Note that each slot can only support one device.

The following is an example name space for a slot:

```
// Device definitions for Slot 1
Device (S1F0) {    // Slot 1, Func#0
    Name(_ADR,0x00020000)
    Name(_SUN,0x00000001)
}
Device (S1F1) {    // Slot 1, Func#1
    Name(_ADR,0x00020001)
    Name(_SUN,0x00000001)
}
Device (S1F2) {    // Slot 1, Func#2
    Name(_ADR,0x00020002)
    Name(_SUN,0x00000001)
}
Device (S1F3) {    // Slot 1, Func#3
    Name(_ADR,0x00020003)
    Name(_SUN,0x00000001)
}
Device (S1F4) {    // Slot 1, Func#4
    Name(_ADR,0x00020004)
    Name(_SUN,0x00000001)
}
Device (S1F5) {    // Slot 1, Func#5
    Name(_ADR,0x00020005)
    Name(_SUN,0x00000001)
}
Device (S1F6) {    // Slot 1, Func#6
    Name(_ADR,0x00020006)
    Name(_SUN,0x00000001)
}
Device (S1F7) {    // Slot 1, Func#7
    Name(_ADR,0x00020007)
    Name(_SUN,0x00000001)
}
```

4.8. The OSHP Control Method

Some systems that include Standard Hot-plug Controls (SHPC) that are released before ACPI-compliant operating systems with native SHPC support are available, use ACPI firmware to operate the SHPC. Firmware control of the SHPC must be disabled if an operating system with native support is used. Platforms that provide ACPI firmware to operate the SHPC must also provide a control method to transfer control to the operating system. This method is called OSHP (Operating System Hot Plug) and is provided for each SHPC that is controlled by ACPI firmware. Operating systems with native SHPC support must execute the OSHP method, if present, for each SHPC

before accessing the SHPC's registers and when returning from a hibernated state. If an SHPC's OSHP method is executed multiple times, and the switch to operating system control has already been achieved, the method must return successfully without doing anything. After the OSHP method is executed, the firmware must not access the SHPC registers. If any signals such as the System Interrupt or PME# have been redirected for servicing by the firmware, they must be restored appropriately for operating system control.

The following is an example of a namespace entry for an SHPC that is managed by firmware:

```
Device(PPB1){
...
Method(OSHP, 0) {
// Disable firmware access to SHPC and restore
// the normal System Interrupt and Wakeup Signal
// connection. (See the Implementation Note below.)
}
...
}
```

IMPLEMENTATION NOTE

Controlling the SHPC Using ACPI

1. When using ACPI to control the SHPC, the following should be considered: Firmware should redirect the System Interrupt and the Wakeup signal to a GPE so that ACPI can service the interrupts instead of the operating system. An appropriate _Lxx GPE handler should be provided. When an operating system with native SHPC support executes the OSHP method, the firmware restores the normal System Interrupt and Wakeup signal connection so the interrupts can be serviced by the operating system. In a PCI-to-PCI bridge implementation, access to the SHPC registers is recommended to be done through the DWORD Select/DWORD Data pair in Configuration Space and not the memory Base Address register. This is because versions of ACPI prior to 2.0 do not allow memory access through a Base Address register. In a Host Bridge implementation, the Host Bridge Register Block can be accessed directly by declaring a memory space operation region to encompass it.
2. A platform may implement both OSHP and _OSC. However, an operating system that comprehends _OSC will preferentially call _OSC over OSHP.

4.9. Hot Plug Parameters

4.9.1. _HPP

This optional object evaluates to the cache-line size, latency timer, SERR enable, and PERR enable values to be used when configuring a PCI device inserted into a hot-plug slot or for performing configuration of PCI devices not configured by the BIOS at system boot. The object is placed under a PCI bus where this behavior is desired, such as a bus with hot-plug slots. _HPP provided settings apply to all child buses until another _HPP object is encountered.

4.9.2. _HPX

This optional object provides settings that apply to all child buses until another such object is encountered. The _HPX method supersedes the _HPP method; operating systems will preferentially call _HPX over _HPP if both objects exist. Refer to the *ACPI Specification, Version 3.0* for details.

4.9.3. Device State During Hot Plug

PCI Hot Plug support is done through ACPI mechanism or native support. In the case of using ACPI, _PS3 method brings the device to its D3 state. It is device dependent as to whether this is $D3_{hot}$ or $D3_{cold}$. The PCI Bus Power Management Specification defines the context preserved by hardware upon resume from the D3 state, either $D3_{hot}$ or $D3_{cold}$. After system software has brought the device out of the D3 state by using the _PS0 method, firmware with knowledge of the PCI Bus Power Management support in the device can determine what context was preserved upon resume from D3.

4.9.4. Slot Power State After Device Removal

PCI Hot Plug support is done through ACPI mechanism or native support. In the case using ACPI, _EJ0 is used to online remove a PCI device. At _EJ0 completion, the PCI device must be isolated for physical removal. _EJ0 is recommended to remove the power from the slot if the platform supports.

5

5. PCI Expansion ROMs

The conventional PCI Local Bus Specification provides a mechanism where devices can provide Expansion ROM code that can be executed for device-specific initialization and possibly a system boot function.

This section describes the format, contents, and code entry points for these Expansion ROMs. In addition, this section also describes the services and execution environment provided to those Expansion ROMs by Version 3.00 compliant system firmware.

The information in the Expansion ROMs is laid out to be compatible with existing Intel x86 Expansion ROM headers for ISA add-in cards, but it will also support other machine architectures.

PCI Expansion ROM code is never executed in place. It is always copied from the ROM device to RAM and executed (initialized) from RAM. After initialization the code is moved, by the Expansion ROM, to its final execution location in RAM, assuming the Expansion ROM code is PCI Specification 3.0 compliant. This enables dynamic sizing of the code (for initialization and run time) and provides speed improvements when executing run-time code.

5.1. PCI Expansion ROM Contents

PCI device Expansion ROMs may contain code (executable or interpretive) for multiple processor architectures. This may be implemented in a single physical ROM which can contain as many code images as desired for different system and processor architectures (see Figure 5-1). Each image must start on a 512-byte boundary and must contain the PCI Expansion ROM header. The starting point of each image depends on the size of previous images. The last image in a ROM has a special encoding in the header to identify it as the last image.

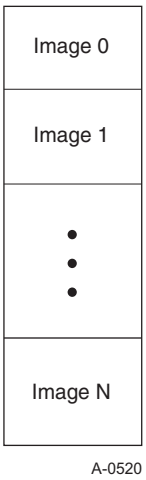

Figure 5-1: PCI Expansion ROM Structure

5.1.1. PCI Expansion ROM Header Format

The information required in each ROM image is split into two different areas. One area, the ROM header, is required to be located at the beginning of the ROM image. The second area, the PCI Data Structure, must be located in the first 64 KB of the image. The format for the PCI Expansion ROM Header is given below. The offset is a hexadecimal number from the beginning of the image, and the length of each field is given in bytes.

Offset	Length	Value	Description
0h	1	55h	ROM Signature, byte 1
1h	1	AAh	ROM Signature, byte 2
2h-17h	16h	xx	Reserved (processor architecture unique data)
18h-19h	2	xx	Pointer to PCI Data Structure

ROM Signature — The ROM Signature is a two-byte field containing a 55h in the first byte and AAh in the second byte. This signature must be the first two bytes of the ROM address space for each image of the ROM.

Pointer to PCI Data Structure — The Pointer to the PCI Data Structure is a two-byte pointer in little-endian format that points to the PCI Data Structure. The reference point for this pointer is the beginning of the ROM image.

5.1.2. PCI Data Structure Format

The PCI Data Structure must be located within the first 64 KB of the ROM image and must be DWORD aligned. The PCI Data Structure contains the following information:

Offset	Length	Description
0	4	Signature, the string “PCIR"
4	2	Vendor Identification
6	2	Device Identification
8	2	Device List Pointer
A	2	PCI Data Structure Length
C	1	PCI Data Structure Revision
D	3	Class Code
10	2	Image Length
12	2	Revision Level of the Vendor’s ROM
14	1	Code Type
15	1	Last Image Indicator
16	2	Maximum Run-time Image Length
18	2	Pointer to Configuration Utility Code Header
1A	2	Pointer to DMTF CLP Entry Point

Signature — These four bytes provide a unique signature for the PCI Data Structure. The string “PCIR” is the signature with “P” being at offset 0, “C” at offset 1, etc.

Vendor Identification — The Vendor Identification field is a 16-bit field with the same definition as the Vendor Identification field in the Configuration Space for this device.

Device Identification — The Device Identification field is a 16-bit field with the same definition as the Device Identification field in the Configuration Space for this device.

Device List Pointer — The Device List Pointer is a two-byte pointer in little-endian format that points to the list of Device IDs supported by this ROM. The beginning reference point (“offset zero”) for this pointer is the beginning of the PCI Data structure (the first byte of the Signature field). This field is only present in Revision 3.0 (and greater) PCI Data structures.

PCI Data Structure Length — The PCI Data Structure Length is a 16-bit field that defines the length of the data structure from the start of the data structure (the first byte of the Signature field). This field is in little-endian format and is in units of bytes.

PCI Data Structure Revision

The PCI Data Structure Revision field is an eight-bit field that identifies the data structure revision level. The revision level is 3 for the current specification.

Class Code

The Class Code field is a 24-bit field with the same fields and definition as the class code field in the Configuration Space for this device.

Image Length

The Image Length field is a two-byte field that represents the length of the image. This field is in little-endian format, and the value is in units of 512 bytes.

Revision Level

The Revision Level field is a two-byte field that contains the revision level of the Vendor's code in the ROM image.

Code Type

The Code Type field is a one-byte field that identifies the type of code contained in this section of the ROM. The code may be executable binary for a specific processor and system architecture or interpretive code. The following code types are assigned:

Type	Description
0	Intel x86, PC-AT compatible
1	Open Firmware standard for PCI[7]
2	Hewlett-Packard PA RISC
3	Extensible Firmware Interface (EFI)
4-FF	Reserved

Last Image Indicator

Bit 7 in this field tells whether or not this is the last image in the ROM. A value of 1 indicates "last image;" a value of 0 indicates that another image follows. Bits 0-6 are reserved.

Maximum Run-Time Image Length

The Image Length field is a two-byte field that represents the maximum length of the image after the initialization code has been executed. This field is in little-endian format, and the value is in units of 512 bytes. This field will be used to determine if the run-time image size is small enough to fit in the memory remaining in the system. This field is only present in Revision 3.0 and later of the PCI Data structure.

---

[7] Open Firmware is a processor architecture and system architecture independent standard for dealing with device specific option ROM code. Documentation for Open Firmware is available in the *IEEE 1275-1994 Standard for Boot (Initialization, Configuration) Firmware Core Requirements and Practices.* A related document, *PCI Bus Binding to IEEE 1275-1994*, specifies the application of Open Firmware to the PCI local bus, including PCI-specific requirements and practices. This document may be obtained using anonymous FTP to the machine *playground.sun.com* with the filename /pub/p1275/bindings/postscript/PCI.ps.

Pointer to Configuration Utility Code Header

This pointer is a two-byte pointer in little-endian format that points to the Expansion ROM's Configuration Utility Code Header table at the beginning of the configuration code block described in Section 5.2.1.24. The beginning reference point ("offset zero") for this pointer is the beginning of the Expansion ROM image. This field is only present in Revision 3.0 (and greater) PCI Data structures. A value of 0000 will be present in this field if the Expansion ROM does not support a Configuration Utility Code Header.

Pointer to DMTF CLP Entry Point

This pointer is a two-byte pointer in little-endian format that points to the execution entry point for the DMTP CLP code supported by this ROM. The beginning reference point ("offset zero") for this pointer is the beginning of the Expansion ROM image. This field is only present in Revision 3.0 (and greater) PCI Data structures. A value of 0000 will be present in this field if the Expansion ROM does not support a DMTF CLP code entry point as described in Section 5.2.1.25.

5.1.3. Device List Format

Revision 3.0 defines a Device List Pointer that points to the list of Device IDs supported by the Expansion ROM image. The beginning reference point ("offset zero") for this pointer is the beginning of the PCI Data structure (the first byte of the Signature field). If this field does not exist (i.e., its contents are zero), then the Expansion ROM only supports the one specific Device ID listed in the Device ID field in the PCI Data structure. However, if this field is non-zero then it must point to a Device List Table. The format of this table is shown in Table 5-1.

Table 5-1: Device List Table

Offset	Length	Description
0	2	Device ID 0 supported
2	2	Device ID 1 supported
4	2	Device ID N supported
N	2	Value of 0000h terminates list

5.2. Firmware Power-on Self Test (POST) Firmware

The description of the POST firmware (BIOS) in this section is only pertinent to the management of PCI Expansion ROMs. The POST Firmware goes through several steps to configure a PCI Expansion ROM. These steps are described below:

1. The POST Firmware examines the PCI bus topology to individually examine each PCI device. If the device has implemented an Expansion ROM Base Address register in Configuration Space, then the POST firmware proceeds to the next step.

   Note that individual functions on a multi-function PCI Device may each have an Expansion ROM Base register implementation.

   Note that the order in which PCI Devices are examined and initialized is not defined.

2. The POST firmware enables the Expansion ROM at an unused memory address.
3. The POST firmware checks the first two bytes in the Expansion ROM for the AA55h signature. If that signature is found, then there is a ROM present. Otherwise, no ROM is attached to the device, and the POST firmware proceeds to step 10.
4. If a ROM is attached, the POST firmware must search the ROM for an image that has the proper Code Type. The following steps can be used to search multiple images.

   If the pointer is valid (non-zero), examine the target location of the pointer. If the pointer is invalid, no further images can be located.

   The target location must contain the signature string "PCIR". If a valid signature is not present, no further images can be located.

   If the signature string is valid, examine the Code Type field to determine if it is correct for the expected type of execution environment. (Refer to the Code Type field entry for more information).

   If the Code Type field is not appropriate, then the BIOS needs to continue searching for Images as described in the next step. If the Code Type field is appropriate, the BIOS proceeds to step 5 below.

   The BIOS should then examine the "Last Image" field to determine if more images are present. If no further images are present, the BIOS should proceed to step 9.

   If more images are present, the Image Length field should be examined to determine the starting location of the next image. The Image Length is added to the starting address of the current image (not Image 0). Note that Image Length is in units of 512 bytes.

   The BIOS proceeds to step 3 above.

5. After the correct image has been selected, POST Firmware will verify that the Vendor ID and Device ID fields match the corresponding fields in the device.

6. If Vendor ID matches but the Device ID does not, the POST Firmware will examine the Device List Pointer. Assuming the Device List Pointer is not zero, the POST firmware will examine the Device List to find a match to the Device ID in the device. A value of 0000h indicates the end of the Device List.

   Note that only PCI Firmware 3.0 compliant Expansion ROMs support the Device List Pointer. POST Firmware should not examine the Device List Pointer field until it has confirmed that the PCI Data Structure Revision Level is 3 or greater.

7. After finding the proper image, the POST Firmware determines if there is sufficient space for the image to be copied into RAM in preparation for initialization. This is accomplished by examining the Maximum Run-Time Image Length field.

   Note that only PCI Firmware 3.0 compliant Expansion ROMs support the Maximum Run-Time Image Length field. POST Firmware should not examine the Maximum Run-Time Image Length field until it has confirmed that the PCI Data Structure Revision Level is 3.

   Note that if the image is a PCI 2.1 compliant Expansion ROM, the POST firmware must continue examining the images to determine if a PCI 3.0 compliant Expansion ROM exists for the same Vendor ID and Device ID. PCI 2.1 compliant Expansion ROMs must only be used by the POST firmware when a PCI 3.0 compliant Expansion ROM cannot be found.

8. The System firmware must confirm that the Option ROM image has a valid checksum before copying it into RAM. This checksum is calculated using Current Image Size as described in Section 5.2.1.4.

9. The POST firmware then copies the appropriate amount of data into RAM. The amount of data to copy is determined by examining the Image Length field in the PCI Data structure in the header.

10. The POST firmware then disables the Expansion ROM Base Address register.

Subsequent steps will vary depending on the value of the Code Type field.

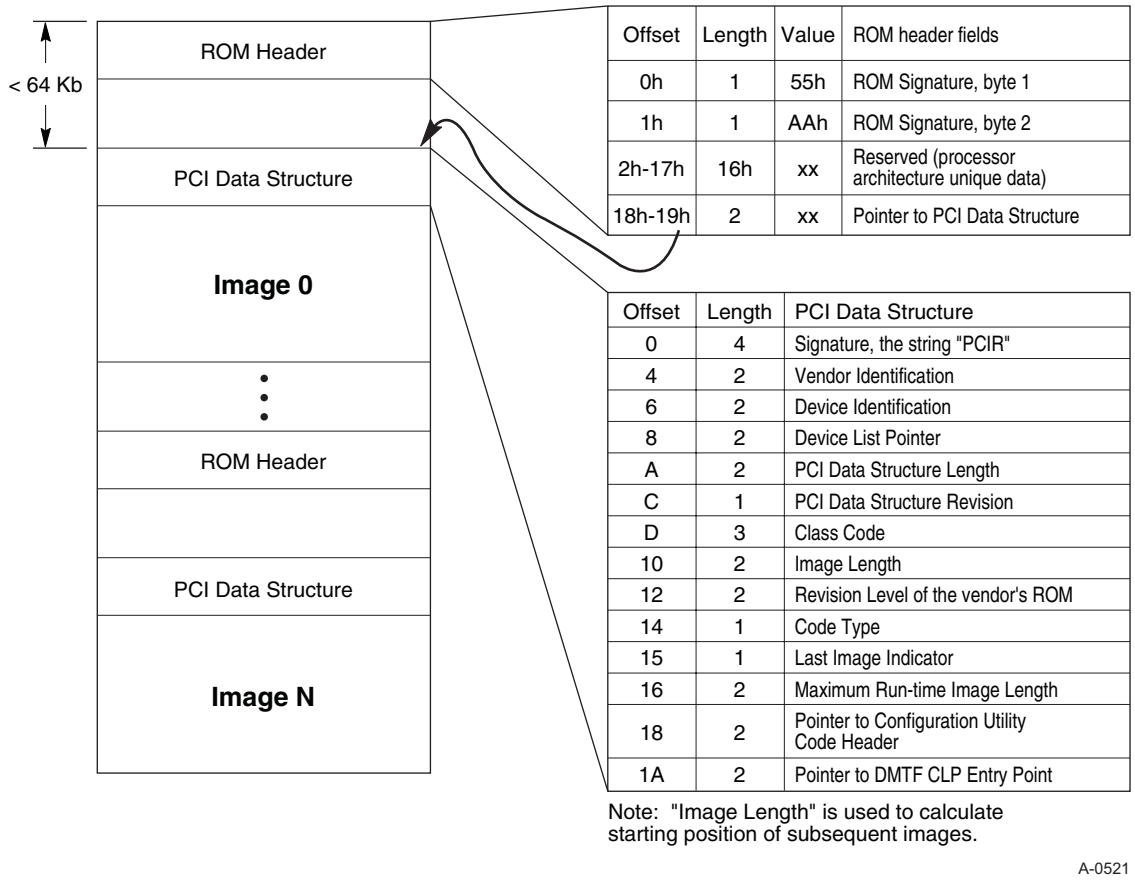

Figure 5-2: Image and Header Organization

5.2.1. PC-compatible Expansion ROMs (Code Type 0)

This section describes the additional steps performed by the POST Firmware when handling of ROM images that have a value of 0 in the Code Type field, indicating the image is Intel x86, PC-compatible.

In addition, this section also describes the new field for the Expansion ROM header for Code Type 0.

5.2.1.1. Expansion ROM Header Extensions

The standard header for PCI Expansion ROM images is expanded slightly for PC-compatibility (Code Type 0). Code Type 0 headers use offset 03h as the entry point for the Expansion ROM INIT function.

Offset	Length	Value	Description
0h	1	55h	ROM Signature byte 1
1h	1	AAh	ROM Signature byte 2
2h	1	Xx	Current Image Size in units of 512 bytes
3h	3	Xx	Entry point for INIT function. POST does a FAR CALL to this location.
6h-17h	12h	Xx	Reserved (application unique data)
18h-19h	2	Xx	Pointer to PCI Data Structure

5.2.1.2. POST Firmware Extensions

The POST firmware copies the entire Expansion ROM image into a RAM address, as described above. The RAM address can vary depending on whether the Expansion ROM identifies itself as being PCI Firmware 3.0 Specification compliant. (A value of 03h in the PCI Data Structure Revision field indicates compliance.)

If the Expansion ROM is compliant, then the following steps will be followed by the POST Firmware:

1. POST Firmware will place the Expansion ROM in RAM at an address that may not be the final run-time execution address. This address will be below the 1-MB address boundary.
2. The POST firmware will then call the INIT function entry point and supply the six arguments described in Table 5-2.
3. As part of the INIT function, the Expansion ROM code must place the final run-time Expansion ROM image into the run-time address provided by the POST firmware.

   Note that the Expansion ROM code must ensure that the device is quiescent until the interrupt service routine is in place at the final run-time location. Caution must be taken in chaining the interrupt service routine. It is possible that an interrupt may arrive while the interrupt service routine is being installed.
4. The POST Firmware will not write-protect the RAM containing the run-time Expansion ROM code at this step. In a PCI Firmware 3.0 compliant system, the write-protection step will be delayed until the Setup portion of the Expansion ROM code, if any, has been executed.
5. The POST firmware will erase the old image from the temporary RAM location.

Table 5-2: Arguments for a PCI 3.0 Compatible Expansion ROM

Argument Number	Register	Meaning
1	[AH]	Bus number
2	[AL]	Upper 5 bits are the Device number
3	[AL]	The lower 3 bits are the Function number
4	[BX]	Run-time address. This is the final run-time address where the ROM image will be located after Initialization. This address is in units of 16-bytes.
5	[CX]	Segment of the PMM Services Entry Point
6	[DX]	Offset of the PMM Services Entry Point

Table 5-3: Arguments for a Legacy Expansion ROM

Argument Number	Register	Meaning
1	[AH]	Bus number
2	[AL]	Upper 5 bits are the Device number
3	[AL]	The lower 3 bits are the Function number

If the Expansion ROM is not compliant with the PCI 3.0 Firmware Specification, then the following steps will be followed by the POST Firmware:

1. POST Firmware will place the Expansion ROM in RAM within the legacy compatibility address (usually 0C0000h to 0E0000h).
2. The POST firmware will then call the INIT function entry point and supply the three arguments described in Table 5-3.
3. The POST Firmware will then write-protect the RAM containing the run-time Expansion ROM code. The size of the Expansion ROM and amount of RAM to write-protect is determined by examining the Current Image Size field at offset 2 in the ROM header.

The POST firmware should be prepared to handle any mixture of Revision 3.0 compliant and non-complaint Expansion ROMs in the system.

5.2.1.3. Resizing of Expansion ROMs During INIT

When the Expansion ROM INIT function is called, it is possible for the Expansion ROM code to reduce the amount of space needed for the run-time code and run-time data. This permits the Expansion ROM to occupy the minimal amount of RAM at run-time.

For example, a device Expansion ROM may require 24 KB for its initialization and run-time code but only 8 KB for the run-time code. The image in the ROM will show a size of 24 KB (in the Image Length field in the PCI Data structure), so that the POST firmware copies the whole thing into RAM. Then when the INIT function is running, it can adjust the size down to 8 KB. This is accomplished by updating the Current Image Size field (at offset 2h) in the ROM Header.

When the INIT function completes, the POST firmware sees that the run-time size is 8 KB in this example and can copy the next Expansion firmware to the optimum location.

If the INIT function wants to completely remove itself from the Expansion ROM area, it does so by writing a zero to the Initialization Size field (the byte at offset 02h). In this case, no checksum has to be generated (since there is no length to checksum across).

5.2.1.3.1. Calculating a New Checksum at the End of INIT

The INIT function is responsible for guaranteeing that the checksum across the size of the image is correct. If the INIT function modifies the RAM area in any way, a new checksum must be calculated and stored in the image.

The checksum is validated by performing a byte-sum of the Expansion ROM across the entire address. The address range is calculated from the Current Image Size field in the ROM Header. The resulting byte-sum must be zero.

5.2.1.4. Image Structure and Length

A PC-compatible image has three lengths associated with it: a run-time length, an initialization length, and an image length. The image length is the total length of the image, and it must be greater than or equal to the initialization length. The image length is contained in the Image Length field in the PCI Data structure.

Once the image has been copied into RAM by the POST firmware, the Image Length field is no longer referenced by POST firmware and is no longer relevant. It does not need to be adjusted when the Expansion ROM size changes.

The initialization length specifies the amount of the image that contains both the initialization and run-time code. This is the amount of data that POST firmware will copy into RAM before executing the initialization routine. This length is calculated from the Current Image Size at offset 2h in the ROM header. Initialization length must be greater than or equal to run-time length.

The run-time length specifies the amount of the image that contains the run-time code. This is the amount of data the POST firmware will leave in RAM while the system is operating. Again this is calculated from the Current Image Size at offset 2h in the ROM header. The Current Image Size must be updated if the size of the Expansion ROM is changed during the INIT phase.

The PCI Data structure must be contained within the run-time portion of the image (if there is any), otherwise, it must be contained within the initialization portion

It is critical that the checksum be maintained at all times. When the POST firmware is examining the images in the Expansion ROM the checksum for the Image Length must be zero or the image will not be used. Later when the Expansion ROM is initialized and the Current Image Size is updated, the checksum must again be updated to maintain a valid image.

5.2.1.5. Memory Usage

In prior PCI Specifications, the Expansion ROM code had no method for reliably sharing the system resources with the BIOS, particularly memory. PCI Expansion ROMs attempted to allocate memory from the Extended BIOS Data Area (EBDA) or search for erased areas of low memory for

temporary use. Many PCI Expansion ROMs fail to implement robust algorithms for sharing memory, and unpredictable behavior may occur.

The Expansion ROM code must make a call to the POST Memory Manager (PMM) to reserve memory, either for temporary usage or for permanent usage. This BIOS function call may only be called during the Expansion ROM initialization phase (INIT phase).

IMPLEMENTATION NOTE

PMM Failure

The Expansion ROM code must be prepared to handle several PMM failure scenarios such as:

- The absence of PMM support in the BIOS
- The absence of PMM "Get Permanent Memory" support in the BIOS
- The PMM denying the request for the memory

In all cases, the 3.0 compliant Expansion ROM code must be prepared to gracefully handle these conditions.

5.2.1.6. Verification of BIOS Support

The Expansion ROM code must first verify that the system BIOS is compliant with version 3.00 (or later) of this specification. If the BIOS is compliant, the Expansion ROM code can call the PMM function REQUEST_SYSTEM_MEMORY to obtain memory for its own use. The Expansion ROM code also provides an argument to the PMM function to request either permanent memory or temporary memory from the BIOS. Refer to Section 5.2.1.20 for more details on the PMM functions.

Note that a 3.0 compliant Expansion ROM is not required to have backward compatibility and operate successfully with a PCI 2.1 BIOS in a system. It is a design choice whether the 3.0 Expansion ROM includes support for PCI 3.0 BIOSes only, or both the PCI 3.0 and PCI 2.1 BIOSes.

5.2.1.7. Permanent Memory

Permanent memory will continue to be assigned to the Expansion ROM even after the operating system has booted. This is accomplished by the BIOS updating the data for the Interrupt 15h, E820h function ("Get Memory Map"), to show that this memory allocated to the Expansion ROM is reserved, private, and unmovable.

In addition, the BIOS must support the ACPI memory tables that describe reserved regions of memory. Interrupt 15h, E820h function ("Get Memory Map"), and the ACPI memory tables must both contain entries for the memory reserved by Expansion ROMs.

The operating system must make the "Get Memory Map" call to the BIOS to understand the usage of this reserved area of memory. If the operating system fails to understand that this memory is reserved, then unpredictable behavior may occur.

5.2.1.8. Temporary Memory

If the Expansion ROM initialization code requests temporary memory via PMM, the BIOS will allocate memory that is usable only during the time of Expansion ROM code initialization. This is useful as a buffer for the Expansion ROM code to decompress the ROM for example. When the Expansion ROM initialization code finishes execution, and returns control to the BIOS, the memory will no longer be usable by the Expansion ROM. Any data left in this region may be erased or over-written.

5.2.1.9. Memory Locations

When the Expansion ROM initialization code requests memory, it can specify whether the memory should be located above or below the 1 MB (FFFFFh) boundary. Memory below the 1 MB boundary is very limited and Expansion ROMs should be frugal with this valuable system resource.

When the Expansion ROM initialization code requests permanent memory above 1 MB, it is assumed that the code is capable of accessing this space at a later time when the operating system may be running in protected mode.

Note that the Expansion ROM initialization code will be called from the BIOS in "big real mode,"[8] and the code will have data access to the full 32-bit address range of memory. This will permit access to any memory buffers allocated above the 1 MB address boundary.

5.2.1.10. Permanent Memory Size Limits

An Expansion ROM needs to limit the request for permanent allocation to 64 KB (total) for above 1 MB memory requests. The limit is 40 KB (total) for below 1 MB permanent memory requests. This total upper bound for permanent memory allocation can be tracked by the Firmware (System BIOS). The total amount of permanent memory available has to be shared between multiple Expansion ROMs. Before requesting memory, an Expansion ROM can query using PMM function 0 (pmmAllocate) call with the length "0" and appropriate flags to obtain the maximum permanent memory size available for allocation.

---

[8] In "Big Real Mode" (aka 32-bit Flat model) is an x86 CPU state where normal Real Mode has been enhanced to have all the segment limits set to 4 GB. The BIOS will accomplish this mode by entering Protected Mode, setting the segment limits to 4 GB, and returning to Real Mode (now "Big Real Mode"). In this mode, the processor will honor the segment register limits and they are unaffected by segment register loads. Refer to http://www.phoenix.com/resources/specs-pmm101.pdf for more information.

5.2.1.11. Multiple Requests for Memory

Expansion ROM initialization code may make multiple calls to the BIOS to request memory. Expansion ROM initialization code may request temporary memory or permanent memory any number of times. However, the total amount of permanent memory requested must not exceed the limits described in Section 5.2.1.10.

The Expansion ROM code must always check the status code being returned to determine if the requested memory was actually allocated to the Expansion ROM.

5.2.1.12. Protected Mode

If the Expansion ROM initialization code switches into Protected Mode, it must return control to the BIOS in Big Real Mode (a 32-bit flat model). In addition many BIOS services are not available in Protected Mode during Expansion ROM initialization. Refer to the 32-bit BIOS Service Directory (Section 2.3) for further details.

5.2.1.13. Run-Time Expansion ROM Size

The conventional PCI Local Bus Specification requires that the Expansion ROM code resize itself after initialization to reduce its use of the memory space. However it is not possible for the BIOS to know, prior to running the Expansion ROM initialization code, if the Expansion ROM will fit into the space remaining for Expansion ROM placement. A PCI 3.0 compliant Expansion ROM will have a new field in the PCI Data Structure header (Maximum Run-Time Image Length) to indicate the run-time space occupied by the Expansion ROM.

It may not possible for the Expansion ROM code to know the actual run-time size (which is affected by the presence or absence of other PCI devices in the system). This run-time size field describes the maximum run-time size that the Expansion ROM could occupy after the initialization code has been run.

The BIOS firmware will examine this field prior to calling the Expansion ROM initialization code. If there is insufficient space remaining to place the run-time Expansion ROM code, the BIOS may elect to not initialize the Expansion ROM.

5.2.1.14. Relocation of Expansion ROM Run-time Code

Prior revisions of the PCI Specification described the three arguments that are passed to the Expansion ROM initialization code. A PCI 3.0 compliant Expansion ROM will accept a fourth argument which describes the final run-time address of the Expansion ROM code.

A PCI 3.0 compliant Expansion ROM code must call the BIOS Function (Section 2.5.2) to determine if the BIOS is PCI 3.0 compliant and will provide this fourth argument.

This new argument permits the BIOS firmware to execute the Expansion ROM initialization code in a memory space other than its final run-time location. This is useful when there is insufficient space remaining in the compatibility region for the BIOS to copy the Expansion ROM initialization code, but there is sufficient space remaining for the run-time code to be placed correctly. The BIOS may

initialize the Expansion ROM code in a region other than where the final run-time code will reside. In all cases, this region will be below the 1 MB address boundary.

A 3.0 compliant BIOS will not provide a run-time address that partially overlaps the temporary INIT address. The two addresses (if different) will be entirely separate ranges of memory. However a 3.0 compliant BIOS may provide the same address for run-time as for the INIT address, similar to how a 2.1 compliant BIOS functions now. In either case, the Expansion ROM can examine the CS register and the fourth argument (run-time address) to determine if they are the same region.

Prior to returning control to the BIOS, the Expansion ROM code moves the run-time code, as described in the earlier section of the document. If the Expansion ROM code updates interrupt vectors to point to the run-time Expansion ROM code, those interrupt vectors must point to the final run-time location, not the temporary location where the BIOS may have temporarily placed the Expansion ROM code for initialization.

For example, if there is 16 K of available space remaining to place Expansion ROMs, and a 64-K Expansion ROM indicates (via the run-time size field) that its run-time size is 16 K then there is sufficient space for the run-time code to fit. However, the 64-K Expansion ROM initialization code must be executed first, but the 64 K of code cannot be placed in memory (there is only 16 K remaining), so it will fail to be initialized. However a PCI 3.0 compliant Expansion ROM can be initialized in another location and the run-time code then moved by the Expansion ROM code into its final address.

5.2.1.15. Expansion ROM Placement Address

Prior versions of the PCI Specifications have described the address where Expansion ROMs will be placed as typically being from 0C0000h up to 0E0000h. This version of the specification now expands that region to be from A0000h to FFFFFh inclusively. The PCI 3.0 compliant system firmware will place the PCI 3.0 compliant Expansion ROM code at any aligned address within this expanded range.

If an Expansion ROM not identified as being PCI 3.0 compliant, the system firmware will place the Expansion ROM within the legacy address range (0C0000h up to 0E0000h).

The system firmware must write-protect the region where the PCI Expansion ROM run-time code resides in the new range. System firmware must have a clear understanding of the systems capability to write-protect the various regions of RAM. The system firmware must write-protect the region where the PCI Expansion ROM run-time code resides in the new range. It is possible that a PCI 3.0 compliant Expansion ROM cannot be placed in the A0000h address region; for example, because the system does not support write-protecting this region.

The region from A0000h-BFFFFh is also the legacy frame buffer location for VGA device. The system firmware should exclude this range from the available Expansion ROMs space as long as VGA device is supported. For the system without supporting VGA device, the system firmware is responsible for determining if there are unacceptable risks in placing the Expansion ROMs in the region from A0000-BFFFFh.

5.2.1.16. VGA Expansion ROM

Prior versions of this specification have stated that VGA Expansion ROMs must be initialized into the 0C0000h address space. That requirement is lifted in this version of the specification. A PCI 3.0 compliant VGA Expansion ROM can be initialized at any location in the Expansion ROM placement address range.

Diagnostic firmware and POST firmware should not assume that if a valid PCI Expansion ROM exists at the 0C0000h address that it is a VGA Expansion ROM. Firmware should look at the Class Code field in the PCI Data structure to determine if the Expansion ROM belongs to a PCI VGA device.

The PCI 3.0 compliant BIOS may elect to always place the VGA Expansion ROM at the legacy 0C0000h address. The BIOS is responsible for determining if there unacceptable risks to placing the VGA Expansion ROM at non-legacy addresses.

5.2.1.17. Expansion ROM Placement Alignment

Traditional ISA legacy Expansion ROMs were placed in memory at addresses aligned on a 2-KB boundary. This tradition has carried through into PCI and it results in an average 1-KB gap between Expansion ROMs.

PCI 3.0 compliant Expansion ROMs may be placed at addresses evenly aligned on a 512-byte boundary. However, the BIOS must be aware of legacy operating systems and legacy applications that will continue to search for Expansion ROMs on the traditional 2-KB boundaries. The BIOS is responsible for determining if there are unacceptable risks in placing the Option ROMs at non-legacy addresses.

5.2.1.18. BIOS Boot Specification

All PCI 3.0 compliant Expansion ROMs must support the *BIOS Boot Specification* found at http://www.phoenix.com/resources/specs-bbs101.pdf. However this specification supercedes the *BIOS Boot Specification* for the definitions of the Expansion ROM placement address, the VGA Expansion ROM placement, and the Expansion ROM placement alignment.

5.2.1.19. Extended BIOS Data Area (EBDA) Usage

The Extended BIOS Data Area (EBDA) is a memory buffer allocated in the below 1 MB address region of memory. Historically, this region has been used by Expansion ROM code to create a permanent memory buffer for its own use. It is important that the EBDA memory buffer be used consistently and carefully as it is a shared system resource.

If an Expansion ROM requires memory below the 1 MB boundary, it should use the PMM functions to obtain this memory. The use of EBDA is discouraged due to the complexity of managing and accessing the EBDA memory.

If an Expansion ROM determines that the PMM services are not available, then it is permissible to use EBDA with the following guidelines in mind:

1. First the Expansion ROM must determine if the EBDA is present.

   Is 40:0E non-zero? Yes, then it is present. Go to step B.

   Allocate an EBDA.

   Read 40:13h size and subtract 1 K.

   Convert new size to segment and save in 40:0Eh.

   Set 40:0E offset 0 to 1 (1 K).

2. Make sure EBDA can handle the amount of memory needed.

   Obtain current EBDA size in 1 K.

   Add needed amount of memory (rounded up to nearest K).

   If the total memory is greater than 64 K, then an error has occurred. The memory request is too large.

3. Move existing EBDA down in memory.

   Disable Interrupts when moving EBDA

   Save current EBDA size as size of memory to move.

   Move current EBDA down to lower memory.

   Zero out old "new" region in upper memory.

   Adjust the 40:0E pointer to the new EBDA.

   Set 40:0E offset 0 to new size.

   Subtract the size allocated (in K) from the value in 40:13.

After following the above steps, the Expansion ROM may use the new memory (at the highest offset) in the EBDA buffer.

All code references to the EBDA buffer must be indirect references through the pointer at 40:0Eh. The Expansion ROM code must always read the pointer at 40:0Eh to determine the beginning segment of the EBDA buffer. This value will change as the EBDA buffer is expanded, while the offset within the buffer will remain constant.

The 8-bit location at 40:13h is the amount of memory installed below 1 MB. It is the highest usable memory address below 1 MB. Many BIOSes install the EBDA immediately below the highest usable memory address but not in all cases. The variable at 40:13h is not a reliable indicator of whether an EBDA has been allocated (use 40:0Eh instead), nor is it a reliable indicator of the size of the EBDA (use 40:0Eh offset 0 instead). Expansion ROM code should make no inferences about the EBDA from the value in 40:13h.

5.2.1.20. POST Memory Manager (PMM) Functions

This description of the new PMM functions is based on the existing PMM 1.01 Specification.[9]

Section 3.3.3 of that specification describes the input arguments for the various PMM calls. The flags field is now expanded to include a definition for bit 3 as shown in Table 5-4.

Table 5-4: Fields and Values for PMM Functions

Bits	Field	Value	Description
1…0	*Memory Type*	1..3	0 = Invalid 1 = Requesting convention memory block (0 to 1 MB) 2 = Requesting extended memory block (1 MB to 4 GB) 3 = Requesting either conventional or extended memory
2	*Alignment*	0..1	0 = No alignment 1 = Use alignment from length parameter
3	*Attribute*	0...1	0 = Temporary memory use during POST 1 = Permanent memory use
15 ..4	*Reserved*	0	Reserved for future use

5.2.1.21. Backward Compatibility of Option ROMs

It is intended that PCI 3.0 definitions of Option ROMs be compatible with the earlier PCI 2.1 definitions. It is possible for a single PCI 3.0 Option ROM image to function with PCI 3.0 compliant System Firmware and with PCI 2.1 compliant System Firmware. The PCI 3.0 fields and functions in the Option ROM will be ignored by PCI 2.1 compliant System Firmware. Any PCI 2.1 compliant System Firmware should have no difficulty locating the PCI 3.0 Option ROM and initializing it in a manner consistent with the PCI 2.1 Specification.

It is also possible to have two separate ROM images for the same PCI device: one for PCI 2.1 System Firmware and one for PCI 3.0 compliance. In this case, the PCI 2.1 Option ROM image must appear first in the sequence of images. PCI 3.0 System Firmware will first search for a PCI 3.0 Option ROM image and only use the PCI 2.1 Option ROM image if no PCI 3.0 Option ROM image is found.

[9] The *POST Memory Manager (PMM) Specification, Version 1.01*, can be found at http://www.pcisig.com/specifications/conventional/pci_firmware/.

5.2.1.22. Option ROM and IRET Handling

The Option ROMs must be capable of handling the level triggered PCI interrupt model where an interrupt can be shared between multiple devices. The Option ROMs must implement appropriate interrupt chaining mechanism and pass control to the next ISR (if present). There should be a default System BIOS interrupt handler for all hardware interrupts that executes the IRET and EOI (acknowledge). Typically most of the System BIOSes already have a default interrupt handler installed for handling spurious interrupts.

Note: The Option ROM always copies the current address (original Interrupt Vector Address) into its local buffer and proceeds to install its own Interrupt Vector address. On an ISR invocation, if an interrupt vector address has been registered by the Option ROM, it is supposed to invoke the original Interrupt Vector address. The Option ROM that is the first to install in the interrupt ISR chain will be the last Option ROM invoked during an ISR execution prior to the control being passed to the default System BIOS handler.

The interrupt can be left masked in the "Interrupt controller" prior to an Option ROM hooking an interrupt. Option ROM can unmask the interrupt once the ISR is installed. The default System BIOS interrupt handler must not mask the interrupt once it has been unmasked by the Option ROM. Option ROMs in the ISR chain should not mask the level triggered shared interrupt in the "Interrupt controller". System BIOS default interrupt handler should be capable of handling regular & spurious interrupts by appropriately issuing an EOI to Master/Slave Interrupt controller if IRQ is greater than or equal to 8.

In order to maintain backward compatibility with current system ROM implementations, the PCI option ROMs should check for PCI 3.0 compliant system BIOS via the PCI BIOS Present function. If a 3.0 compliant BIOS is detected, then the option ROM's interrupt handler should chain to the next interrupt handler as stated in this section and not perform an EOI. If a PCI 3.0 compatible system BIOS is not present, then the option ROM should just EOI and IRET.

5.2.1.23. Stack Size Requirement by Expansion ROM

The system firmware will provide a minimum 4-K size of stack to POST Expansion ROM. It is recommended that system firmware, operating system, and applications provide a minimum 4-K stack when calling to Expansion ROM at run time.

5.2.1.24. Configuration Code for Expansion ROMs

Certain Expansion ROMs contain configuration code or utility code that may be executed by the end user during Expansion ROM initialization. Since prior versions of this specification did not provide any mechanism for executing this code, the Expansion ROMs code writers implemented a wide variety of methods to invoke this configuration code.

In many cases the Expansion ROM code would place a prompt on the screen, such as "Press F2 to enter the XXXX Configuration utility", and snoop the keystrokes looking for an invocation. This caused issues to arise in many cases. There was no guarantee that the invocation keystroke was not already assigned to another purpose by the system firmware. Assigning multiple functionality to a

single keystroke may cause a portion of the system to become inaccessible during Expansion ROM initialization. In addition, the Expansion ROM must wait for some arbitrary period of time to be certain that the user had not pressed the keystroke. This adds an unnecessary delay to the Expansion ROM initialization code and to the overall system POST time, particularly in the cases where no change in configuration is needed by the end user.

Beginning with this version of the specification, the Expansion ROM vendors can isolate the configuration code and the system firmware will execute the configuration code only when specifically requested by the user. The Expansion ROM header (defined in Section 5.1.2) now contains an optional pointer to the header for the configuration code. During POST when the system firmware is preparing the Expansion ROM for the INIT stage, the system firmware will make a copy of the configuration utility code (if present in the ROM image) and leave it in temporary memory for later execution.

At the completion of the INIT phase, the Expansion ROM will shrink itself to the final run-time size (without the configuration utility code), and the configuration code will remain in temporary memory. Note that the BIOS will not write-protect the run-time Expansion ROM image until the very end of POST. This permits the Expansion ROM Configuration Utility code to update the run-time Expansion ROM code with configuration parameters or any other data collected by the Configuration Utility code. The Expansion ROM run-time size must include the region where configuration parameters are stored, and the size must not increase when the Configuration Utility code updates the run-time code with the configuration parameters. The Configuration Utility code must also update the checksum for the run-time Expansion ROM code to keep the run-time image valid.

System firmware will be responsible for managing the location and execution of the configuration utilities for all the Expansion ROMs. However, the Expansion ROM can control some of the ways that the System firmware treats the configuration utility code. There are three possible scenarios.

- **Legacy:** In this scenario, the Expansion ROM image does not contain a pointer to the configuration utility code header. The Expansion ROM either has no configuration code, or it prefers to run the configuration code during the INIT phase as 2.1 compliant Expansion ROMs do today. The system firmware does not manage the configuration utility code in this scenario.
- **Hybrid:** In this scenario, the Expansion ROM has a valid configuration utility code header. However the Expansion ROM runs the configuration utility code during the INIT phase, as described in the legacy style above. In addition the system firmware has preserved a copy of the configuration utility code and may execute it at a later point in POST. The configuration utility code must be prepared to be executed twice in this scenario.
- **Delayed Execution:** In this scenario, the Expansion ROM has a valid configuration utility code header. Unlike the previous two scenarios, the configuration utility is not executed (by the Expansion ROM) during the INIT phase. The configuration utility code is only executed by the system firmware at a later point in POST.

The preferred method is the "delayed execution" style described above. This permits the system firmware to manage the execution of the Configuration Utility code in a consistent and organized manner.

Support for Configuration Utility Code management is optional for the system firmware. The Expansion ROM must call the "PCI BIOS Present" function, as described in Section 2.5.2, to

determine whether the system firmware has implemented the support for Expansion ROM Configuration Code.

5.2.1.24.1. Executing the Expansion ROM Configuration Code

The configuration code remains in temporary memory so long as it is possible that an end user may need to use the configuration code. The system firmware will make a far call into the configuration utility code, and the configuration utility will perform a far return to pass control back to system firmware when it has completed. The configuration code need not preserve any registers, and is expected to update the BH register as defined below. The system firmware will call the configuration code in "Big Real Mode", as defined in the footnote to Section 5.2.1.9. In addition, the Bus/Device/Function number and the far pointer to the PMM entry point will be provided as argument as shown below.

Arg.	Register	Meaning
1	[AH]	Bus number
2	[AL]	Upper 5 bits are the Device number
3	[AL]	Lower 3 bits are the Function number
4	[BL]	System Firmware state: Bit 0 set = BIOS is performing Console Redirection. Bit 1-7 = Reserved.
5	[BH]	Configuration utility status. Cleared to zero by system firmware prior to call.
6	[CX]	Segment of the PMM Services Entry Point.
7	[DX]	Offset of the PMM Services Entry Point.
8	[SI]	Segment of the run-time Expansion ROM code.
9	[DI]	Offset of the run-time Expansion ROM code.

Prior to returning control to the system firmware via a far return, the configuration code will update the BH register as follows:

Bit 0 set = Configuration utility is requesting a system reset

Bit 1-7 = Reserved

The system firmware provides the far pointer to the PMM Services Entry point. The PMM Services Entry pointer is described in Section 3.1.1 "PMM Structure", (offset 7) of the *POST Memory Manager Specification, Version 1.01*. By providing the PMM Services Entry pointer, it is no longer necessary for the Expansion ROM code to search for the PMM Structure, validate the PMM signature, and validate the PMM checksum.

The configuration utility code can assume that all basic devices have been initialized and the system firmware is providing input and output services. The configuration utility code must use the system firmware Interrupt services for input and output functions.

System firmware will have a list of all the configuration code from the various Expansion ROMs. System firmware will be responsible for creating a method for the end user to selectively execute the

configuration code. This may be done within a specific page of the ROM based Setup program. However, the exact method for selecting and executing the individual Option ROM configuration code is not defined by this specification.

5.2.1.24.2. Configuration Utility Behavior Under Console Redirection

When the system firmware is running Console Redirection, it will set bit 0 of the BL register prior when calling the configuration utility code. This places additional restrictions on the configuration utility code, as described below:

- The configuration code must not change the video mode at any point during its execution. Attempts to enter any graphical video mode, in particular, are destructive to Console Redirection session. The BIOS will already be in the preferred text mode when it calls the configuration utility code, and the it must not be changed.
- The only permitted input device is keyboard via Int 16h. Mouse and other types of input devices that cannot be easily redirected over the Console Redirection will not function.
- The configuration code must perform all output via Int 10h services. Writing output directly to a video buffer may cause data to not be redirected correctly in the Console Redirection session. Preferrably, the configuration code would use the Int 10h Write-TTY function for screen output and would minimize the use of cursor repositioning, color dependencies, and "screen-scraping" (screen clearing) features, which perform poorly under Console Redirection.

5.2.1.24.3. Configuration Utility Code Header

The configuration utility code must be separated from the standard Expansion ROM code and be in a contiguous, self-contained block of code that can be independently executed by system firmware. The configuration utility code header resides at the very beginning of this code block and has the following format:

Offset	Length	Description
0	1	Configuration Code Length
1	40	Configuration Code Identifier string
41	1	Revision of this header structure

Configuration Code Length — This byte indicates size of the Configuration code in units of 512 bytes.

Configuration Code Identifier string — The Configuration Code Identifier string is a null-terminated ASCII text string giving the name of the vendor's configuration utility. For example, it may contain "ABC SCSI configuration utility". This text will be displayed to the end user by the System Firmware. Note that the system firmware will display the text until a null is encountered or until 40 bytes have been displayed.

Header Revision	This field is the version of the structure of the Configuration Code Header. This version of the header have a revision field value of 1.

The Configuration Code header and associated code is included in the Image Length field for the PCI Data Structure ROM size.

5.2.1.25. DMTF Command Line Protocol (CLP) Support

The Option ROM may optionally provide an entry point that will support device configuration via the DMTF CLP standard.[10] This interface will follow an API defined in the DMTF CLP Specification. This interface is accessed with a FAR CALL where the system ROM will pass in a DMTF CLP compatible configuration message that will target the device or a child of that device. This interface may be called multiple times in order to completely configure a device during pre-boot.

The option ROM code should assume that the system firmware will call the entry point in Big Real Mode and with a minimum of 4 KB for the stack.

Input parameters are described below:

ES:EDI = pointer to Command Line Protocol string buffer (NULL Terminated)

EBX = Command Status Flags:

Bit 0: Privilege Bit:

0 = No Privileges

1 = Administrator Privileges

Bits 7-3: Reserved

Bits 15-8: Session ID

Bits 31-16: Process ID

The output parameters are described below:

ES:EDI = pointer to Command Line Protocol Return string buffer (NULL Terminated)

EBX = Return Status Flags

Bit 0: Success flag

0 = Command Accepted

1 = Command Failed

Bits 7-1: Reserved (always set to 0)

---

[10] The DMTF CLP specification is available from the DMTF group at http://www.dmtf.org/apps/org/workgroup/svrmgmt/ in the Server Management Working Group. The PCI 3.0 Firmware Specification defers to the DMTF specification for any differences in description or implementation.

Bits 15-8: Return Status Code

0 = Command Completed Successfully

1 = Command Accepted, In Progress

2 = Invalid Target Specified

3 = Target Busy

4 = Insufficient Privilege

5 = Insufficient Resources

6 = Other Failure

7 - 255 = Reserved for future use

Bits 31-16: Process ID

All other x86 registers are preserved.

Note that not all PCI 3.0 system firmware will support calling the DMTF CLP entry point. The support for this function can be determined by examining the results of the "PCI BIOS Present" call described in Section 2.5.2. This is useful for diagnostics and validation in determining if the DMTF CLP interface will be used. In addition an Expansion ROM may change its configuration behavior if it determines that the PCI 3.0 system firmware will not call the DMTF CLP API.

Similarly an Expansion ROM is not required to have a DMTF CLP entry point. If the entry point is not present, the "Pointer to the DMTF CLP entry point" field in the PCI Data Structure (see Section 5.2.1) should be null.

5.2.2. EFI Expansion ROM (Type 3)

Section 12.4.1 "PCI Option ROMs" of the *Extensible Firmware Interface Specification, Version 1.10* (http://developer.intel.com/technology/efi/) describes ways to store EFI images (e.g., EFI drivers) in PCI Expansion ROMs. The description in Section 5.2.1 of this document does not apply to EFI Expansion ROMs.

6

6. PCI Services Specific to DIG64-compliant Systems

For DIG64-compliant systems based, the operating systems and upper layers of software shall use the SAL_PCI_CONFIG_READ and SAL_PCI_CONFIG_WRITE procedures to access the PCI configuration space at runtime. DIG64-compliant systems do not support the ACPI MCFG table and _CBA method.

For definition of these procedures, refer to the *Itanium Processor Family System Abstraction Layer Specification* (http://developer.intel.com/design/Itanium/Downloads/245359.htm). The usage of the segment (PCI Segment Group) must match that of the _SEG control method defined in ACPI.

Note: The SAL_PCI_CONFIG_READ procedure completes when the configuration transaction completes and the data read is returned. The SAL_PCI_CONFIG_WRITE procedure guarantees the completion of the configuration write.