



Intel® Platform Innovation Framework for EFI PCI Platform Support Specification

Version 0.9
August 9, 2004

THIS SPECIFICATION IS PROVIDED "AS IS" WITH NO WARRANTIES WHATSOEVER, INCLUDING ANY WARRANTY OF MERCHANTABILITY, NONINFRINGEMENT, FITNESS FOR ANY PARTICULAR PURPOSE, OR ANY WARRANTY OTHERWISE ARISING OUT OF ANY PROPOSAL, SPECIFICATION OR SAMPLE. Except for a limited copyright license to copy this specification for internal use only, no license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted herein.

Intel disclaims all liability, including liability for infringement of any proprietary rights, relating to implementation of information in this specification. Intel does not warrant or represent that such implementation(s) will not infringe such rights.

Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them.

This document is an intermediate draft for comment only and is subject to change without notice. Readers should not design products based on this document.

Intel, the Intel logo, and Itanium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

* Other names and brands may be claimed as the property of others.

Copyright © 2002–2004, Intel Corporation.

Intel order number xxxxxx-001



Revision History

Revision	Revision History	Date
0.9	First public release.	8/9/04

1 Introduction	7
Overview	7
Conventions Used in This Document	7
Data Structure Descriptions	7
Protocol Descriptions	8
Procedure Descriptions	8
Pseudo-Code Conventions	9
Typographic Conventions	9
2 Design Discussion	11
Target Audience	11
PCI Platform Support Related Terms	11
PCI Platform Support Related Information	11
PCI Platform Protocol	12
PCI Platform Protocol Overview	12
Incompatible PCI Device Support Protocol	13
Incompatible PCI Device Support Protocol Overview	13
Usage Model for the Incompatible PCI Device Support Protocol	13
3 Code Definitions	15
Introduction	15
PCI Platform Protocol	15
EFI_PCI_PLATFORM_PROTOCOL	15
EFI_PCI_PLATFORM_PROTOCOL.PlatformNotify()	17
EFI_PCI_PLATFORM_PROTOCOL.PlatformPrepController()	19
EFI_PCI_PLATFORM_PROTOCOL.GetPlatformPolicy()	21
EFI_PCI_PLATFORM_PROTOCOL.GetPciRom()	24
Incompatible PCI Device Support Protocol	26
EFI_INCOMPATIBLE_PCI_DEVICE_SUPPORT_PROTOCOL	26
EFI_INCOMPATIBLE_PCI_DEVICE_SUPPORT_PROTOCOL.CheckDevice()	28
Tables	
Table 3-1. ACPI 2.0 QWORD Address Space Descriptor Usage	29
Table 3-2. ACPI 2.0 End Tag Usage	30

Overview

This specification defines the core code and services that are required for an implementation of the following protocols in the Intel® Platform Innovation Framework for EFI (hereafter referred to as the "Framework"):

- PCI Platform Protocol
- Incompatible PCI Device Support Protocol

The PCI Platform Protocol allows a PCI bus driver to obtain the platform policy and call a platform driver at various points in the enumeration phase. The Incompatible PCI Device Support Protocol allows a PCI bus driver to handle resource allocation for some PCI devices that do not comply with the *PCI Specification*.

This specification does the following:

- Describes the [basic components](#) of the PCI Platform Protocol
- Describes the [basic components](#) of the Incompatible PCI Device Support Protocol and how Framework-based firmware configures incompatible PCI devices
- Provides [code definitions](#) for the [PCI Platform Protocol](#), the [Incompatible PCI Device Support Protocol](#), and their related type definitions that are architecturally required by the *Intel® Platform Innovation Framework for EFI Architecture Specification*

Conventions Used in This Document

This document uses the typographic and illustrative conventions described below.

Data Structure Descriptions

Intel® processors based on 32-bit Intel® architecture (IA-32) are “little endian” machines. This distinction means that the low-order byte of a multibyte data item in memory is at the lowest address, while the high-order byte is at the highest address. Processors of the Intel® Itanium® processor family may be configured for both “little endian” and “big endian” operation. All implementations designed to conform to this specification will use “little endian” operation.

In some memory layout descriptions, certain fields are marked *reserved*. Software must initialize such fields to zero and ignore them when read. On an update operation, software must preserve any reserved field.

The data structures described in this document generally have the following format:

STRUCTURE NAME: The formal name of the data structure.

Summary: A brief description of the data structure.

Prototype: A “C-style” type declaration for the data structure.

Parameters: A brief description of each field in the data structure prototype.

Description:	A description of the functionality provided by the data structure, including any limitations and caveats of which the caller should be aware.
Related Definitions:	The type declarations and constants that are used only by this data structure.

Protocol Descriptions

The protocols described in this document generally have the following format:

Protocol Name:	The formal name of the protocol interface.
Summary:	A brief description of the protocol interface.
GUID:	The 128-bit Globally Unique Identifier (GUID) for the protocol interface.
Protocol Interface Structure:	A “C-style” data structure definition containing the procedures and data fields produced by this protocol interface.
Parameters:	A brief description of each field in the protocol interface structure.
Description:	A description of the functionality provided by the interface, including any limitations and caveats of which the caller should be aware.
Related Definitions:	The type declarations and constants that are used in the protocol interface structure or any of its procedures.

Procedure Descriptions

The procedures described in this document generally have the following format:

ProcedureName():	The formal name of the procedure.
Summary:	A brief description of the procedure.
Prototype:	A “C-style” procedure header defining the calling sequence.
Parameters:	A brief description of each field in the procedure prototype.
Description:	A description of the functionality provided by the interface, including any limitations and caveats of which the caller should be aware.
Related Definitions:	The type declarations and constants that are used only by this procedure.
Status Codes Returned:	A description of any codes returned by the interface. The procedure is required to implement any status codes listed in this table. Additional error codes may be returned, but they will not be tested by standard compliance tests, and any software that

uses the procedure cannot depend on any of the extended error codes that an implementation may provide.

Pseudo-Code Conventions

Pseudo code is presented to describe algorithms in a more concise form. None of the algorithms in this document are intended to be compiled directly. The code is presented at a level corresponding to the surrounding text.

In describing variables, a *list* is an unordered collection of homogeneous objects. A *queue* is an ordered list of homogeneous objects. Unless otherwise noted, the ordering is assumed to be First In First Out (FIFO).

Pseudo code is presented in a C-like format, using C conventions where appropriate. The coding style, particularly the indentation style, is used for readability and does not necessarily comply with an implementation of the *Extensible Firmware Interface Specification*.

Typographic Conventions

This document uses the typographic and illustrative conventions described below:

Plain text The normal text typeface is used for the vast majority of the descriptive text in a specification.

[Plain text \(blue\)](#) In the online help version of this specification, any [plain text](#) that is underlined and in blue indicates an active link to the cross-reference. Click on the word to follow the hyperlink. Note that these links are *not* active in the PDF of the specification.

Bold In text, a **Bold** typeface identifies a processor register name. In other instances, a **Bold** typeface can be used as a running head within a paragraph.

Italic In text, an *Italic* typeface can be used as emphasis to introduce a new term or to indicate a manual or specification name.

BOLD Monospace Computer code, example code segments, and all prototype code segments use a **BOLD Monospace** typeface with a dark red color. These code listings normally appear in one or more separate paragraphs, though words or segments can also be embedded in a normal text paragraph.

[Bold Monospace](#) In the online help version of this specification, words in a [Bold Monospace](#) typeface that is underlined and in blue indicate an active hyperlink to the code definition for that function or type definition. Click on the word to follow the hyperlink. Note that these links are *not* active in the PDF of the specification. Also, these inactive links in the PDF may instead have a [Bold Monospace](#) appearance that is underlined but in dark red. Again, these links are not active in the PDF of the specification.

Italic Monospace In code or in text, words in *Italic Monospace* indicate placeholder names for variable information that must be supplied (i.e., arguments).

`Plain Monospace` In code, words in a `Plain Monospace` typeface that is a dark red color but is not bold or italicized indicate pseudo code or example code. These code segments typically occur in one or more separate paragraphs.

`text text text`

In the PDF of this specification, text that is highlighted in yellow indicates that a change was made to that text since the previous revision of the PDF. The highlighting indicates only that a change was made since the previous version; it does not specify what changed. If text was deleted and thus cannot be highlighted, a note in red and highlighted in yellow (that looks like *Note: text text text.*) appears where the deletion occurred.

See the master Framework glossary in the Framework Interoperability and Component Specifications help system for definitions of terms and abbreviations that are used in this document or that might be useful in understanding the descriptions presented in this document.

See the master Framework references in the Interoperability and Component Specifications help system for a complete list of the additional documents and specifications that are required or suggested for interpreting the information presented in this document.

The Framework Interoperability and Component Specifications help system is available at the following URL:

<http://www.intel.com/technology/framework/spec.htm>

Target Audience

This document is intended for the following readers:

- BIOS developers, either those who create general-purpose BIOS and other firmware products or those who modify these products for use in Intel® architecture-based products.
- Operating system developers who will be adapting their shrink-wrapped operating system products to run on Intel architecture-based platforms.

Readers of this specification are assumed to have solid knowledge of the following documents:

- *EFI 1.10 Specification*
- *Intel® Platform Innovation Framework for EFI Architecture Specification*, version 0.9
- *IA-32 Intel® Architecture Software Developer's Manual*

See [Related Information from Intel Corporation](#) in the master Framework help system for the URLs for these documents.

PCI Platform Support Related Terms

The following terms are used throughout this document. See the following Framework specifications for additional definitions of PCI-related terms:

- [Intel® Platform Innovation Framework for EFI PCI Host Bridge Resource Allocation Protocol Specification](#)
- [Intel® Platform Innovation Framework for EFI Hot-Plug PCI Initialization Protocol Specification](#)

incompatible PCI device

A PCI device that does not fully comply with the [PCI Specification](#). Typically, this kind of device has a special requirement for Base Address Register (BAR) allocation. Some devices may want a special resource length or alignment, while others may want fixed I/O or memory locations.

PCI Platform Support Related Information

The following publications and sources of information may be useful to you or are referred to by this specification.

Specifications from Intel Corporation

- [Intel® Platform Innovation Framework for EFI PCI Host Bridge Resource Allocation Protocol Specification](#), version 0.9
- *IA-32 Intel® Architecture Software Developer's Manual*, volumes 1–3: See [Related Information from Intel Corporation](#) in the master Framework help system for the URL.

Industry Specifications

- *Advanced Configuration and Power Interface Specification* (hereafter referred to as the *ACPI Specification*), version 2.0: See [Industry Specifications](#) in the master Framework help system for the URL.

PCI Specifications

- *Conventional PCI Specification*, version 3.0: http://www.pcisig.com*
- *PCI-to-PCI Bridge Architecture Specification*, revision 1.2: http://www.pcisig.com*
- *PCI-to-PCI Bridges and CardBus Controllers on Windows 2000, Windows XP, and Windows Server 2003*:
http://www.microsoft.com/whdc/system/bus/PCI/pcibridge-cardbus.mspx*

PCI Platform Protocol

PCI Platform Protocol Overview

The [Intel® Platform Innovation Framework for EFI PCI Host Bridge Resource Allocation Protocol Specification](#) defines and describes the PCI Host Bridge Resource Allocation Protocol. The PCI Host Bridge Resource Allocation Protocol driver provides chipset-specific functionality that works across processor architectures and unique platform features. It does not address issues where an implementation varies across platforms.

In contrast, the [PCI Platform Protocol](#) that is defined here in this specification provides a set of protocol interfaces that allow the platform driver to do platform-specific actions. The purpose of the PCI Platform Protocol is to do the following:

- **Allow a PCI bus driver to obtain platform policy.** The platform can use this protocol to control whether the PCI bus driver reserves I/O ranges for ISA aliases and VGA aliases. The default policy for the PCI bus driver is to reserve I/O ranges for both ISA aliases and VGA aliases, which may result in a large amount of I/O space being unavailable for PCI devices. This protocol allows the platform driver to change this policy.
- **Call a platform driver at various points in the enumeration phase.** The platform driver can use these hooks to perform various platform-specific activities. Examples of such activities include but are not limited to the following:
 - [EFI_PCI_PLATFORM_PROTOCOL.PlatformPrepController\(\)](#) can be used to program the PCI subsystem vendor ID and device ID into onboard and chipset devices.
 - [PlatformPrepController\(\)](#) and [EFI_PCI_PLATFORM_PROTOCOL.PlatformNotify\(\)](#) can be used for implementing hardware workarounds.
 - [PlatformPrepController\(\)](#) can be used for preprogramming any backside registers that control the Base Address Register (BAR) window sizes.
 - [PlatformPrepController\(\)](#) can be used to set PCI or PCI-X* bus speeds for PCI bridges that support multiple bus speeds.
- Data hub records that are related to the PCI slot and embedded devices can be logged after PCI enumeration is complete.

- **Allow PCI option ROMs to be stored in local storage.** The platform can store PCI option ROMs in local storage (e.g., a firmware volume) and report their existence to the PCI bus driver using the [EFI_PCI_PLATFORM_PROTOCOL.GetPciRom\(\)](#) member function. Option ROMs for embedded PCI controllers are often stored in a platform-specific location. The same member function can be used to override the default PCI ROM on an add-in card with one from platform-specific storage.

A platform should implement this protocol if any of the functionality that is listed above is required.

See [Code Definitions](#) for the definition of [EFI_PCI_PLATFORM_PROTOCOL](#) and the member functions listed above. See the *Intel® Platform Innovation Framework for EFI PCI Host Bridge Resource Allocation Protocol Specification* for additional PCI-related design discussion.

Incompatible PCI Device Support Protocol

Incompatible PCI Device Support Protocol Overview

Some PCI devices do not fully comply with the [PCI Specification](#). For example, a PCI device may request that its I/O Base Address Register (BAR) be placed on a 0x200 boundary even though it is requesting an I/O with a length of 0x100. The [Incompatible PCI Device Support Protocol](#) allows a PCI bus driver to handle resource allocation for some PCI devices that do not comply with the *PCI Specification*.

In Framework-based firmware, the platform-specific PCI host bridge driver works with the generic, standard PCI bus driver to configure the entire PCI subsystem. Even though the exact configuration is up to individual incompatible devices, it is a platform choice to support those incompatible PCI devices. For example, one platform may not want to support those incompatible devices while another platform appears more tolerant of those devices.

See [Code Definitions](#) for the definition of the [EFI_INCOMPATIBLE_PCI_DEVICE_SUPPORT_PROTOCOL](#).

Usage Model for the Incompatible PCI Device Support Protocol

The following describes the usage model for the [Incompatible PCI Device Support Protocol](#):

1. The PCI bus driver locates [EFI_INCOMPATIBLE_PCI_DEVICE_SUPPORT_PROTOCOL](#). If the PCI bus driver cannot find this protocol, simply follow the regular PCI enumeration path. Otherwise, go to step 2.
2. For each PCI device that was detected, the PCI bus driver begins collecting the required PCI resources by probing the Base Address Register (BAR) for each device.
3. For each device, call [EFI_INCOMPATIBLE_PCI_DEVICE_SUPPORT_PROTOCOL.CheckDevice\(\)](#) to check whether this PCI device is an incompatible device. If this device is not an incompatible device, go to step 5.
4. Use the *Configuration* that is returned by [CheckDevice\(\)](#) to override or modify the original PCI resource requirements.
5. Follow the normal PCI enumeration process.

Introduction

This section contains the basic definitions of Framework protocols that provide PCI platform support. The following protocols are defined in this section:

- [EFI_PCI_PLATFORM_PROTOCOL](#)
- [EFI_INCOMPATIBLE_PCI_DEVICE_SUPPORT_PROTOCOL](#)

This section also contains the definitions for additional data types and structures that are subordinate to the structures in which they are called. The following types or structures can be found in "Related Definitions" of the parent function definition:

- [EFI_PCI_CHIPSET_EXECUTION_PHASE](#)
- [EFI_PCI_PLATFORM_POLICY](#)

PCI Platform Protocol

EFI_PCI_PLATFORM_PROTOCOL

Summary

This protocol provides the interface between the PCI bus driver/PCI Host Bridge Resource Allocation driver and a platform-specific driver to describe the unique features of a platform. This protocol is optional.

GUID

```
#define EFI_PCI_PLATFORM_PROTOCOL_GUID \
{ 0x7d75280, 0x27d4, 0x4d69, 0x90, 0xd0, 0x56, 0x43, 0xe2, 0x38,
  0xb3, 0x41);
```

Protocol Interface Structure

```
typedef struct _EFI_PCI_PLATFORM_PROTOCOL {
    EFI\_PCI\_PLATFORM\_PHASE\_NOTIFY PlatformNotify;
    EFI\_PCI\_PLATFORM\_PREPROCESS\_CONTROLLER PlatformPrepController;
    EFI\_PCI\_PLATFORM\_GET\_PLATFORM\_POLICY GetPlatformPolicy;
    EFI\_PCI\_PLATFORM\_GET\_PCI\_ROM GetPciRom;
} EFI\_PCI\_PLATFORM\_PROTOCOL;
```

Parameters

PlatformNotify

The notification from the PCI bus enumerator to the platform that it is about to enter a certain phase during the enumeration process. See the [PlatformNotify\(\)](#) function description.

PlatformPrepController

The notification from the PCI bus enumerator to the platform for each PCI controller at several predefined points during PCI controller initialization. See the [PlatformPrepController\(\)](#) function description.

GetPlatformPolicy

Retrieves the platform policy regarding enumeration. See the [GetPlatformPolicy\(\)](#) function description.

GetPciRom

Gets the PCI device's option ROM from a platform-specific location. See the [GetPciRom\(\)](#) function description.

Description

The **EFI_PCI_PLATFORM_PROTOCOL** is published by a platform-aware driver. This protocol is optional; see [PCI Platform Protocol Overview](#) in [Design Discussion](#) for scenarios in which this protocol is required. There cannot be more than one instance of this protocol in the system.

This protocol is installed on a separate handle during the DXE phase (before PCI enumeration). This handle may not have any other protocols installed on it. If the PCI bus driver detects the presence of this protocol during its **EFI_DRIVER_BINDING_PROTOCOL.Start()** function, it will use the PCI Platform Protocol to obtain information about the platform policy. The PCI bus driver will use this protocol to get the PCI device's option ROM from a platform-specific location in storage. It will also call the various member functions of this protocol at predefined points during PCI bus enumeration. The member functions can be used for performing any platform-specific initialization that is appropriate during the particular phase.

EFI_PCI_PLATFORM_PROTOCOL.PlatformNotify()

Summary

The platform driver receives notifications from the PCI bus enumerator at various phases during the enumeration, just like the PCI host bridge driver.

Prototype

```
typedef
EFI_STATUS
(EFIAPI * EFI_PCI_PLATFORM_PHASE_NOTIFY) (
    IN EFI\_PCI\_PLATFORM\_PROTOCOL                                *This,
    IN EFI\_HANDLE                                                HostBridge,
    IN EFI\_PCI\_HOST\_BRIDGE\_RESOURCE\_ALLOCATION\_PHASE              Phase,
    IN EFI\_PCI\_CHIPSET\_EXECUTION\_PHASE                            ChipsetPhase
);
```

Parameters

This

Pointer to the [EFI_PCI_PLATFORM_PROTOCOL](#) instance.

HostBridge

The handle of the host bridge controller. Type [EFI_HANDLE](#) is defined in [InstallProtocolInterface\(\)](#) in the *EFI 1.10 Specification*.

Phase

The phase of the PCI bus enumeration. Type [EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PHASE](#) is defined in [EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL.NotifyPhase\(\)](#) in the [Intel® Platform Innovation Framework for EFI PCI Host Bridge Resource Allocation Protocol Specification](#).

ChipsetPhase

Defines the execution phase of the PCI chipset driver. Type [EFI_PCI_CHIPSET_EXECUTION_PHASE](#) is defined in "Related Definitions" below.

Description

The [PlatformNotify\(\)](#) function can be used to notify the platform driver so that it can perform platform-specific actions. No specific actions are required.

Eight notification points are defined at this time. More synchronization points may be added as required in the future. The PCI bus driver calls the platform driver twice for every *Phase*—once before the PCI Host Bridge Resource Allocation Protocol driver is notified, and once after the PCI Host Bridge Resource Allocation Protocol driver has been notified.



This member function may not perform any error checking on the input parameters. It also does not return any error codes. If this member function detects any error condition, it needs to handle those errors on its own because there is no way to surface any errors to the caller.

Related Definitions

```

//*****
// EFI_PCI_CHIPSET_EXECUTION_PHASE
//*****
typedef enum {
    ChipsetEntry,
    ChipsetExit,
    MaximumChipsetPhase
} EFI_PCI_CHIPSET_EXECUTION_PHASE;
    
```

EFI_PCI_CHIPSET_EXECUTION_PHASE is used to call a platform protocol and execute platform-specific code. Following is a description of the fields in the above enumeration.

ChipsetEntry	The phase that indicates the entry point to the PCI Bus Notify phase. This platform hook is called before the PCI bus driver calls the <u>EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL</u> driver.
ChipsetExit	The phase that indicates the exit point to the Chipset Notify phase before returning to the PCI Bus Driver Notify phase. This platform hook is called after the PCI bus driver calls the <u>EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL</u> driver.

Status Codes Returned

EFI_SUCCESS	The function completed successfully.
-------------	--------------------------------------

EFI_PCI_PLATFORM_PROTOCOL.PlatformPrepController()

Summary

The platform driver receives notifications from the PCI bus enumerator at various phases during PCI controller initialization, just like the PCI host bridge driver.

Prototype

```
typedef
EFI_STATUS
(EFIAPI * EFI_PCI_PLATFORM_PREPROCESS_CONTROLLER) (
    IN EFI\_PCI\_PLATFORM\_PROTOCOL                *This,
    IN EFI\_HANDLE                                HostBridge,
    IN EFI\_HANDLE                                RootBridge,
    IN EFI\_PCI\_ROOT\_BRIDGE\_IO\_PROTOCOL\_PCI\_ADDRESS PciAddress,
    IN EFI\_PCI\_CONTROLLER\_RESOURCE\_ALLOCATION\_PHASE Phase,
    IN EFI\_PCI\_CHIPSET\_EXECUTION\_PHASE           ChipsetPhase
);
```

Parameters

This

Pointer to the [EFI_PCI_PLATFORM_PROTOCOL](#) instance.

HostBridge

The associated PCI host bridge handle. Type [EFI_HANDLE](#) is defined in [InstallProtocolInterface\(\)](#) in the *EFI 1.10 Specification*.

RootBridge

The associated PCI root bridge handle.

PciAddress

The address of the PCI device on the PCI bus. This address can be passed to the [EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL](#) functions to access the PCI configuration space of the device. See Table 12-1 in the *EFI 1.10 Specification* for the definition of [EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_PCI_ADDRESS](#).

Phase

The phase of the PCI controller enumeration. Type [EFI_PCI_CONTROLLER_RESOURCE_ALLOCATION_PHASE](#) is defined in [EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL.PreprocessController\(\)](#) in the [Intel® Platform Innovation Framework for EFI PCI Host Bridge Resource Allocation Protocol Specification](#).

ChipsetPhase

Defines the execution phase of the PCI chipset driver. Type [EFI_PCI_CHIPSET_EXECUTION_PHASE](#) is defined in [EFI_PCI_PLATFORM_PROTOCOL.PlatformNotify\(\)](#).

Description

The **PlatformPrepController()** function can be used to notify the platform driver so that it can perform platform-specific actions. No specific actions are required.

Several notification points are defined at this time. More synchronization points may be added as required in the future. The PCI bus driver calls the platform driver twice for every PCI controller—once before the PCI Host Bridge Resource Allocation Protocol driver is notified, and once after the PCI Host Bridge Resource Allocation Protocol driver has been notified.

This member function may not perform any error checking on the input parameters. It also does not return any error codes. If this member function detects any error condition, it needs to handle those errors on its own because there is no way to surface any errors to the caller.

Status Codes Returned

EFI_SUCCESS	The function completed successfully.
-------------	--------------------------------------

EFI_PCI_PLATFORM_PROTOCOL.GetPlatformPolicy()

Summary

The PCI bus driver and the PCI Host Bridge Resource Allocation Protocol driver can call this member function to retrieve platform policies regarding PCI enumeration.

Prototype

```
typedef
EFI_STATUS
(EFI_API * EFI_PCI_PLATFORM_GET_PLATFORM_POLICY) (
    IN EFI\_PCI\_PLATFORM\_PROTOCOL          *This,
    OUT EFI\_PCI\_PLATFORM\_POLICY          *PciPolicy,
);
```

Parameters

This

Pointer to the [EFI_PCI_PLATFORM_PROTOCOL](#) instance.

PciPolicy

The platform policy with respect to VGA and ISA aliasing. Type [EFI_PCI_PLATFORM_POLICY](#) is defined in "Related Definitions" below.

Description

The [GetPlatformPolicy\(\)](#) function retrieves the platform policy regarding PCI enumeration. The PCI bus driver and the PCI Host Bridge Resource Allocation Protocol driver can call this member function to retrieve the policy.

The [EFI_PCI_IO_PROTOCOL.Attributes\(\)](#) function allows a PCI device driver to ask for various legacy ranges. Because PCI device drivers run after PCI enumeration, a request for legacy allocation comes in after PCI enumeration. The only practical way to guarantee that such a request from a PCI device driver will be fulfilled is to preallocate these ranges during enumeration. The PCI bus enumerator does not know which legacy ranges may be requested and therefore must rely on [GetPlatformPolicy\(\)](#). The data that is returned by [GetPlatformPolicy\(\)](#) determines the supported attributes that are returned by the [EFI_PCI_IO_PROTOCOL.Attributes\(\)](#) function. See "[Related Definitions](#)" below for a description of the output parameter *PciPolicy*. For example, the platform can decide if it wishes to support devices that require ISA aliases using this parameter. Note that the [EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.GetAttributes\(\)](#) function returns the attributes that the root bridge hardware supports and does not depend upon preallocations.

Related Definitions

```
//*****
// EFI_PCI_PLATFORM_POLICY
//*****
typedef    UINT32    EFI_PCI_PLATFORM_POLICY;

#define    EFI_RESERVE_NONE_IO_ALIAS            0x0000
#define    EFI_RESERVE_ISA_IO_ALIAS            0x0001
#define    EFI_RESERVE_ISA_IO_NO_ALIAS        0x0002
#define    EFI_RESERVE_VGA_IO_ALIAS            0x0004
#define    EFI_RESERVE_VGA_IO_NO_ALIAS        0x0008
```

The legal combinations are listed in the table below.

<p>EFI_RESERVE_NONE_IO_ALIAS</p>	<p>Does not set aside either ISA or VGA I/O resources during PCI enumeration. By using this selection, the platform indicates that it does not want to support a PCI device that requires ISA or legacy VGA resources. If a PCI device driver asks for these resources, the request will be turned down.</p>
<p>EFI_RESERVE_ISA_IO_ALIAS EFI_RESERVE_VGA_IO_ALIAS</p>	<p>Sets aside the ISA I/O range and all the aliases during PCI enumeration. VGA I/O ranges and aliases are included in ISA alias ranges. In this scheme, 75 percent of the I/O space remains unused. By using this selection, the platform indicates that it wants to support PCI devices that require the following, at the cost of wasted I/O space:</p> <ul style="list-style-type: none"> • ISA range and its aliases • Legacy VGA range and its aliases <p>The PCI bus driver will not allocate I/O addresses out of the ISA I/O range and its aliases. The following are the ISA I/O ranges:</p> <ul style="list-style-type: none"> • n100–n3FF • n500–n7FF • n900–nBFF • nD00–nFFF <p>In this case, the PCI bus driver will ask the PCI host bridge driver for larger I/O ranges. The PCI host bridge driver is not aware of the ISA aliasing policy and merely attempts to allocate the requested ranges. The first device that requests the legacy VGA range will get all the legacy VGA range plus its aliased addresses forwarded to it. The first device that requests the legacy ISA range will get all the legacy ISA range plus its aliased addresses forwarded to it.</p>

<p>EFI_RESERVE_ISA_IO_NO_ALIAS EFI_RESERVE_VGA_IO_ALIAS</p>	<p>Sets aside the ISA I/O range (0x100–0x3FF) during PCI enumeration and the aliases of the VGA I/O ranges. By using this selection, the platform indicates that it will support VGA devices that require VGA ranges, including those that require VGA aliases. The platform further wants to support non-VGA devices that ask for the ISA range (0x100–3FF), but not if it also asks for the ISA aliases. The PCI bus driver will not allocate I/O addresses out of the legacy ISA I/O range (0x100–0x3FF) range or the aliases of the VGA I/O range. If a PCI device driver asks for the ISA I/O ranges, including aliases, the request will be turned down. The first device that requests the legacy VGA range will get all the legacy VGA range plus its aliased addresses forwarded to it. When the legacy VGA device asks for legacy VGA ranges and its aliases, all the upstream PCI-to-PCI bridges must be set up to perform 10-bit decode on legacy VGA ranges. To prevent two bridges from positively decoding the same address, all PCI-to-PCI bridges that are peers to this bridge will have to be set up to not decode ISA aliased ranges. In that case, all the devices behind the peer bridges can occupy only I/O addresses that are not ISA aliases. This is a limitation of PCI-to-PCI bridges and is described in the white paper PCI-to-PCI Bridges and Card Bus Controllers on Windows 2000, Windows XP, and Windows Server 2003. The PCI enumeration process must be cognizant of this restriction.</p>
<p>EFI_RESERVE_ISA_IO_NO_ALIAS EFI_RESERVE_VGA_IO_NO_ALIAS</p>	<p>Sets aside the ISA I/O range (0x100–0x3FF) during PCI enumeration. VGA I/O ranges are included in the ISA range. By using this selection, the platform indicates that it wants to support PCI devices that require the ISA range and legacy VGA range, but it does not want to support devices that require ISA alias ranges or VGA alias ranges. The PCI bus driver will not allocate I/O addresses out of the legacy ISA I/O range (0x100–0x3FF). If a PCI device driver asks for the ISA I/O ranges, including aliases, the request will be turned down. By using this selection, the platform indicates that it will support VGA devices that require VGA ranges, but it will not support VGA devices that require VGA aliases. To truly support 16-bit VGA decode, all the PCI-to-PCI bridges that are upstream to a VGA device, as well as upstream to the parent PCI root bridge, must support 16-bit VGA I/O decode. See the PCI-to-PCI Bridge Architecture Specification for information regarding the 16-bit VGA decode support. This requirement must hold true for every VGA device in the system. If any of these bridges does not support 16-bit VGA decode, it will positively decode all the aliases of the VGA I/O ranges and this selection must be treated like EFI_RESERVE_ISA_IO_NO_ALIAS EFI_RESERVE_VGA_IO_ALIAS.</p>

Status Codes Returned

<p>EFI_SUCCESS</p>	<p>The function completed successfully.</p>
<p>EFI_INVALID_PARAMETER</p>	<p><i>PciPolicy</i> is NULL.</p>

EFI_PCI_PLATFORM_PROTOCOL.GetPciRom()

Summary

Gets the PCI device's option ROM from a platform-specific location.

Prototype

```
typedef
EFI_STATUS
(EFI_API * EFI_PCI_PLATFORM_GET_PCI_ROM) (
    IN EFI_PCI_PLATFORM_PROTOCOL          *This,
    IN EFI_HANDLE                        PciHandle,
    OUT VOID                               **RomImage,
    OUT UINTN                              *RomSize
);
```

Parameters

This

Pointer to the EFI_PCI_PLATFORM_PROTOCOL instance.

PciHandle

The handle of the PCI device. Type EFI_HANDLE is defined in InstallProtocolInterface() in the *EFI 1.10 Specification*.

RomImage

If the call succeeds, the pointer to the pointer to the option ROM image. Otherwise, this field is undefined. The memory for *RomImage* is allocated by EFI_PCI_PLATFORM_PROTOCOL.GetPciRom() using the EFI Boot Service AllocatePool(). It is the caller's responsibility to free the memory using the EFI Boot Service FreePool(), when the caller is done with the option ROM.

RomSize

If the call succeeds, a pointer to the size of the option ROM size. Otherwise, this field is undefined.

Description

The GetPciRom() function gets the PCI device's option ROM from a platform-specific location. The option ROM will be loaded into memory. This member function is used to return an image that is packaged as a PCI 2.2 option ROM. The image may contain both legacy and EFI option ROMs. See the *EFI 1.10 Specification* for details. This member function can be used to return option ROM images for embedded controllers. Option ROMs for embedded controllers are typically stored in platform-specific storage, and this member function can retrieve it from that storage and return it to the PCI bus driver. The PCI bus driver will call this member function before scanning the ROM that is attached to any controller, which allows a platform to specify a ROM image that is different from the ROM image on a PCI card.

Status Codes Returned

EFI_SUCCESS	The option ROM was available for this device and loaded into memory.
EFI_NOT_FOUND	No option ROM was available for this device.
EFI_OUT_OF_RESOURCES	No memory was available to load the option ROM.
EFI_DEVICE_ERROR	An error occurred in getting the option ROM.

Incompatible PCI Device Support Protocol

EFI_INCOMPATIBLE_PCI_DEVICE_SUPPORT_PROTOCOL

Summary

Allows the PCI bus driver to support resource allocation for some PCI devices that do not comply with the [PCI Specification](#).

NOTE

This protocol is optional. Only those platforms that implement this protocol will have the capability to support incompatible PCI devices. The absence of this protocol can cause the PCI bus driver to configure these incompatible PCI devices incorrectly. As a result, these devices may not work properly.

GUID

```
#define EFI_INCOMPATIBLE_PCI_DEVICE_SUPPORT_PROTOCOL_GUID \
{0xeb23f55a, 0x7863, 0x4ac2, 0x8d, 0x3d, 0x95, 0x65, 0x35, 0xde, \
0x3, 0x75}
```

Protocol Interface Structure

```
typedef struct _EFI_INCOMPATIBLE_PCI_DEVICE_SUPPORT_PROTOCOL {
    EFI\_INCOMPATIBLE\_PCI\_DEVICE\_SUPPORT\_CHECK\_DEVICE CheckDevice;
} EFI\_INCOMPATIBLE\_PCI\_DEVICE\_SUPPORT\_PROTOCOL;
```

Parameters

CheckDevice

Returns a list of ACPI resource descriptors that detail any special resource configuration requirements if the specified device is a recognized incompatible PCI device. See the [CheckDevice\(\)](#) function description.

Description

The [EFI_INCOMPATIBLE_PCI_DEVICE_SUPPORT_PROTOCOL](#) is used by the PCI bus driver to support resource allocation for some PCI devices that do not comply with the [PCI Specification](#). This protocol can find some incompatible PCI devices and report their special resource requirements to the PCI bus driver. The generic PCI bus driver does not have prior knowledge of any incompatible PCI devices. It interfaces with the [EFI_INCOMPATIBLE_PCI_DEVICE_SUPPORT_PROTOCOL](#) to find out if a device is incompatible and to obtain the special configuration requirements for a specific incompatible PCI device.

This protocol is optional, and only one instance of this protocol can be present in the system. If a platform supports this protocol, this protocol is produced by a Driver Execution Environment (DXE) driver and must be made available before the Boot Device Selection (BDS) phase. The PCI bus driver will look for the presence of this protocol before it begins PCI enumeration.

If this protocol exists in a platform, it indicates that the platform has the capability to support those incompatible PCI devices. However, final support for incompatible PCI devices still depends on the implementation of the PCI bus driver. The PCI bus driver may fully, partially, or not even support these incompatible devices.

During PCI bus enumeration, the PCI bus driver will probe the PCI Base Address Registers (BARs) for each PCI device—regardless of whether the PCI device is incompatible or not—to determine the resource requirements so that the PCI bus driver can invoke the proper PCI resources for them. Generally, this resource information includes the following:

- Resource type
- Resource length
- Alignment

However, some incompatible PCI devices may have special requirements. As a result, the length or the alignment that is derived through BAR probing may not be exactly the same as the actual resource requirement of the device. For example, there are some devices that request I/O resources at a length of 0x100 from their I/O BAR, but these incompatible devices will never work correctly if an odd I/O base address, such as 0x100, 0x300, or 0x500, is assigned to the BAR. Instead, these devices request an even base address, such as 0x200 or 0x400. The Incompatible PCI Device Support Protocol can then be used to obtain these special resource requirements for these incompatible PCI devices. In this way, the PCI bus driver will take special consideration for these devices during PCI resource allocation to ensure that they can work correctly.

This protocol may support the following incompatible PCI BAR types:

- I/O or memory length that is different from what the BAR reports
- I/O or memory alignment that is different from what the BAR reports
- Fixed I/O or memory base address

See the [Conventional PCI Specification 3.0](#) for the details of how a PCI BAR reports the resource length and the alignment that it requires.

EFI_INCOMPATIBLE_PCI_DEVICE_SUPPORT_PROTOCOL.CheckDevice()**Summary**

Returns a list of ACPI resource descriptors that detail the special resource configuration requirements for an incompatible PCI device.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_INCOMPATIBLE_PCI_DEVICE_SUPPORT_CHECK_DEVICE) (
    IN EFI\_INCOMPATIBLE\_PCI\_DEVICE\_SUPPORT\_PROTOCOL *This,
    IN UINTN VendorId,
    IN UINTN DeviceId,
    IN UINTN RevisionId,
    IN UINTN SubsystemVendorId,
    IN UINTN SubsystemDeviceId,
    OUT VOID **Configuration
);
```

Parameters*This*

Pointer to the [EFI_INCOMPATIBLE_PCI_DEVICE_SUPPORT_PROTOCOL](#) instance.

VendorId

A unique ID to identify the manufacturer of the PCI device. See the [Conventional PCI Specification 3.0](#) for details.

DeviceId

A unique ID to identify the particular PCI device. See the *Conventional PCI Specification 3.0* for details.

RevisionId

A PCI device-specific revision identifier. See the *Conventional PCI Specification 3.0* for details.

SubsystemVendorId

Specifies the subsystem vendor ID. See the *Conventional PCI Specification 3.0* for details.

SubsystemDeviceId

Specifies the subsystem device ID. See the *Conventional PCI Specification 3.0* for details.

Configuration

A list of ACPI resource descriptors that detail the configuration requirement. See Table 3-1 in the "Description" subsection below for the definition.

Description

The **CheckDevice()** function returns a list of ACPI resource descriptors that detail the special resource configuration requirements for an incompatible PCI device.

Prior to bus enumeration, the PCI bus driver will look for the presence of the **EFI_INCOMPATIBLE_PCI_DEVICE_SUPPORT_PROTOCOL**. Only one instance of this protocol can be present in the system. For each PCI device that the PCI bus driver discovers, the PCI bus driver calls this function with the device's vendor ID, device ID, revision ID, subsystem vendor ID, and subsystem device ID. If the *VendorId*, *DeviceId*, *RevisionId*, *SubsystemVendorId*, or *SubsystemDeviceId* value is set to **(UINTN)-1**, that field will be ignored. The ID values that are not **(UINTN)-1** will be used to identify the current device.

This function will only return **EFI_SUCCESS**. However, if the device is an incompatible PCI device, a list of ACPI resource descriptors will be returned in *Configuration*. Otherwise, **NULL** will be returned in *Configuration* instead. The PCI bus driver does not need to allocate memory for *Configuration*. However, it is the PCI bus driver's responsibility to free it. The PCI bus driver then can configure this device with the information that is derived from this list of resource nodes, rather than the result of BAR probing.

Only the following two resource descriptor types from the [ACPI Specification](#) may be used to describe the incompatible PCI device resource requirements:

- QWORD Address Space Descriptor (ACPI 2.0, section 6.4.3.5.1)
- End Tag (ACPI 2.0, section 6.4.2.8)

The QWORD Address Space Descriptor can describe memory, I/O, and bus number ranges for dynamic or fixed resources. The configuration of a PCI root bridge is described with one or more QWORD Address Space Descriptors, followed by an End Tag. Table 3-1 and Table 3-2 below contain these two descriptor types. See the *ACPI Specification* for details on the field values. Click on the links below to take you directly to each table:

- Table 3-1: [ACPI 2.0 QWORD Address Space Descriptor Usage](#)
- Table 3-2: [ACPI 2.0 End Tag Usage](#)

Table 3-1. ACPI 2.0 QWORD Address Space Descriptor Usage

Byte Offset	Byte Length	Data	Description
0x00	0x01	0x8A	QWORD Address Space Descriptor
0x01	0x02	0x2B	Length of this descriptor in bytes, not including the first two fields.
0x03	0x01		Resource type: 0: Memory range 1: I/O range Other values will be ignored.
0x04	0x01		General flags. Ignored.
0x05	0x01		Type-specific flags. Ignored.
0x06	0x08		Address Space Granularity. Ignored.

continued

Table 3-1. ACPI 2.0 QWORD Address Space Descriptor Usage (continued)

Byte Offset	Byte Length	Data	Description
0x0E	0x08		Address Range Minimum. Fixed resource base. If the device does not request a fixed base, it must be 0.
0x16	0x08		Address Range Maximum. Used to convey the alignment information. This value must be $2^n - 1$. If no special alignment is required for the BAR, it must be 0. Then the alignment will set to (length-1) , where the length is derived through the BAR probing.
0x1E	0x08		Address Translation Offset. Used to indicate the BAR Index from 0 to 5. Specially, (UINT64) - 1 in this field means all the PCI BARs on the device.
0x26	0x08		Address Range Length. Length of the requested resource. If the device has no special length request, it must be 0. Then the length that was obtained from BAR probing will be applied.

Table 3-2. ACPI 2.0 End Tag Usage

Byte Offset	Byte Length	Data	Description
0x00	0x01	0x79	End Tag.
0x01	0x01	0x00	Checksum. Set to 0 to indicate that checksum is to be ignored.

Status Codes Returned

EFI_SUCCESS	The function always returns EFI_SUCCESS .
-------------	--