Intel® Platform Innovation Framework for EFI PCI Host Bridge Resource Allocation Protocol Specification

Draft for Review

Version 0.9

August 9, 2004

Intel order number xxxxxx-001

Revision History

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

Contents

Figures

Tables

1 Introduction

Overview

This specification defines the core code and services that are required for an implementation of the PCI Host Bridge Resource Allocation Protocol of the Intel® Platform Innovation Framework for EFI (hereafter referred to as the "Framework"). This protocol is used by a PCI bus driver to program the PCI host bridge and configure the root PCI buses. The registers inside the PCI host bridge that control root PCI bus configuration are not governed by the PCI specification and vary from chipset to chipset. The PCI Host Bridge Resource Allocation Protocol is therefore specific to a particular chipset.

This specification does the following:

- Describes the basic components of the PCI Host Bridge Resource Allocation Protocol
- Describes several sample PCI architectures and a sample implementation of the PCI Host Bridge Resource Allocation Protocol
- Provides code definitions for the PCI Host Bridge Resource Allocation Protocol and the PCI-host-bridge-related type definitions that are architecturally required by the *Intel® Platform Innovation Framework for EFI Architecture Specification*

The **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL** does not describe platform policies. The platform policies are described by the **EFI_PCI_PLATFORM_PROTOCOL** and are outside the scope of this specification. The **EFI_PCI_PLATFORM_PROTOCOL** is defined in the *Intel® Platform Innovation Framework for EFI PCI Platform Support 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

2 Design Discussion

Introduction

This section provides background and design information for the PCI Host Bridge Resource Allocation Protocol. A PCI bus driver, running in the EFI Boot Services environment, uses this protocol to program PCI host bridge hardware. This protocol abstracts a PCI host bridge. In particular, functions for programming a PCI host bridge are defined here although other bus types may be supported in a similar fashion as extensions to this specification.

This chapter discusses the following:

- PCI terms that are used in this document
- An overview of the PCI Host Bridge Resource Allocation Protocol
- Sample PCI architectures
- ISA aliasing considerations
- Programming of standard PCI configuration registers
- Sample implementation

PCI Terms Used in This Document

The following PCI terms are used throughout this document. See the Glossary in the master Framework help system for definitions of general Framework terms.

coherency domain

The address resources of a system as seen by a processor. It consists of both system memory and I/O space.

HB

Host bridge. See PCI host bridge.

MWI

Memory Write and Invalidate. See the *PCI Local Bus Specification*, revision 2.3, for more information; see Industry Specifications in the Framework master help system for the URL for the PCI SIG.

PCI bus

A generic term used to describe any PCI-like buses, including conventional PCI, PCI-X*, and PCI Express*. From a software standpoint, a PCI bus is collection of up to 32 physical PCI devices that share the same physical PCI bus.

PCI bus driver

Software that creates a handle for every PCI controller in the system and installs both the PCI I/O Protocol and the Device Path Protocol onto that handle. It may optionally perform PCI enumeration if resources have not already been allocated to all the PCI controllers. It also loads and starts any EFI drivers that are found in any PCI option ROMs that were discovered during PCI enumeration.

PCI configuration space

The configuration channel that is defined by PCI to configure PCI devices into the resource domain of the system. Each PCI device must produce a standard set of registers in the form of a PCI configuration header and can optionally produce device-specific registers. The registers are addressed via Type 0 or Type 1 PCI configuration cycles as described by the *PCI Specification.* The PCI configuration space can be shared across multiple PCI buses. On Intel® architecture-based systems, the PCI configuration space is accessed via I/O ports 0xCF8 and 0xCFC. The PCI Express configuration space is accessed via a memory-mapped aperture.

PCI controller

A hardware components that is discovered by a PCI bus driver and is managed by a PCI device driver. This document uses the terms "PCI function" and "PCI controller" equivalently.

PCI device

A collection of up to 8 PCI functions that share the same PCI configuration space. A PCI device is physically connected to a PCI bus.

PCI enumeration

The process of assigning resources to all the PCI controllers on a given PCI host bridge. This process includes the following:

- Assigning PCI bus numbers and PCI interrupts
- Allocating PCI I/O resources, PCI memory resources, and PCI prefetchable memory resources
- Setting miscellaneous PCI DMA values

Typically, PCI enumeration is to be performed only once during the boot process.

PCI function

A controller that provides some type of I/O services. It consumes some combination of PCI I/O, PCI memory, and PCI prefetchable memory regions and the PCI configuration space. The PCI function is the basic unit of configuration for PCI.

PCI host bridge

The software abstraction that produces one or more PCI root bridges. All the PCI buses that are produced by a host bus controller are part of the same coherency domain. A PCI host bridge is an abstraction and may be made up of multiple hardware devices. Most systems can be modeled as having one PCI host bridge. This software abstraction is necessary while dealing with PCI resource allocation because resources that are assigned to one PCI root bridge depend on another and all the "related" PCI root bridges must be considered together during resource allocation.

PCI root bridge

A PCI root bridge that produces a root PCI bus. It bridges a root PCI bus and a bus that is not a PCI bus (e.g., processor local bus, InfiniBand* fabric). A PCI host bridge may have one or more root PCI bridges. Configurations of a root PCI bridge within a host bridge can have dependencies upon other root PCI bridges within the same host bridge.

PCI segment

A collection of up to 256 PCI buses that share the same PCI configuration space. A PCI segment is defined in section 6.5.6 of the *ACPI 2.0 Specification* as the _SEG object. The **`SAL_PCI_CONFIG_READ`** and **`SAL_PCI_CONFIG_WRITE`** procedures that are defined in chapter 9 of the *Intel® Itanium® Processor Family System Abstraction Layer Specification* define how to access the PCI configuration space in a system that supports multiple PCI segments; see Related Information from Intel in the master Framework help system for the URL for this specification. If a system supports only a single PCI segment, the PCI segment number is defined to be zero. The existence of PCI segments enables the construction of systems with greater than 256 PCI buses.

PERR

Parity Error.

RB

Root bridge. See PCI root bridge.

root PCI bus

A PCI bus that is not a child of another PCI bus. For every root PCI bus, there is an object in the ACPI name space with a Plug and Play (PNP) ID of "PNP0A03." Typical desktop systems include only one root PCI bus.

SERR

System error.

PCI Host Bridge Resource Allocation Protocol

PCI Host Bridge Resource Allocation Protocol Overview

The PCI Host Bridge Resource Allocation Protocol is used by a PCI bus driver to program a PCI host bridge. The registers inside a PCI host bridge that control configuration of PCI root buses are not governed by the PCI specification and vary from chipset to chipset. The PCI Host Bridge Resource Allocation Protocol implementation is therefore specific to a particular chipset.

Each PCI host bridge is comprised of one or more PCI root bridges, and there are hardware registers associated with each PCI root bridge. These registers control the bus, I/O, and memory resources that are decoded by the PCI root bus that the PCI root bridge produces and all the PCI buses that are children of that PCI root bus.

The **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL** allows for future innovation of the chipsets. It abstracts the PCI bus driver from the chipset details. This design allows system designers to make changes to the host bridge hardware without impacting a platform-independent PCI bus driver.

See PCI Host Bridge Resource Allocation Protocol in Code Definitions for the definition of **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL**.

Host Bus Controllers

A platform can be viewed as the following:

- A set of processors
- A set of core chipset components that may produce one or more host buses

The figure below shows a platform with ***n*** processors (CPUs) and a set of core chipset components that produce ***m*** host bridges (HBs).

Most systems with one PCI host bus controller will contain a single instance of the **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL**. More complex systems may contain multiple instances of this protocol.

NOTE

There is no relationship between the number of chipset components in a platform and the number of **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL** *instances. This protocol is an abstraction from a software point of view. This protocol is attached to the device handle of a PCI host bus controller, which itself is composed of one or more PCI root bridges. A PCI root bridge is a chipset component(s) that produces a physical PCI bus whose parent is not another physical PCI bus.*

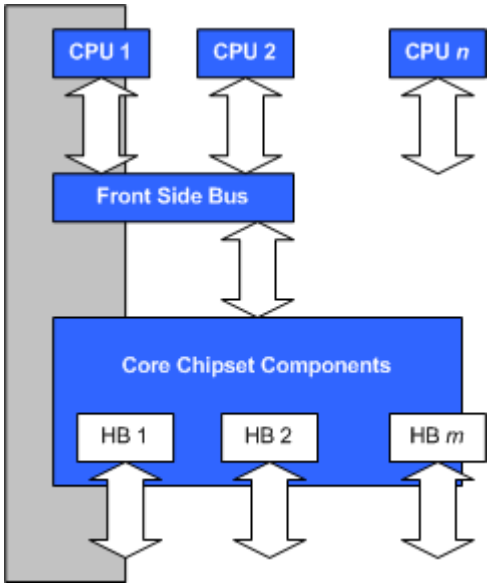

Figure 2-1. Host Bus Controllers

Producing the PCI Host Bridge Resource Allocation Protocol

EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL instances are produced by DXE drivers—most often by early DXE drivers.

The figure below shows how the **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL** is used to identify the associated PCI root bridges. After the steps in the figure are completed, the **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL** can then be queried to identify the device handles of the associated PCI root bridges. See the *EFI 1.10 Specification* for details of the PCI Root Bridge I/O Protocol.

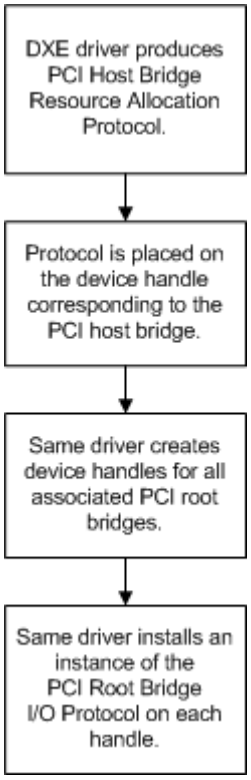

Figure 2-2. Producing the PCI Host Bridge Resource Allocation Protocol

Required PCI Protocols

The following protocols are mandatory if the system supports PCI devices or slots:

- **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL**
- **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL**

See the *EFI 1.10 Specification* for more information on the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL**.

Relationship with EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL

It is expected, although not necessary, that a chipset-aware driver will produce the following protocol instances:

- **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL**
- **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL**

Care has been taken to avoid overlap between the member functions of the two protocols. For example, **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL** does not describe the *SegmentNumber* or the final resource assignment for a root bridge, because these attributes are available using the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL**. Both protocols contain links to the associated instances of the other protocols, as follows:

- **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL**: Includes the handle of the PCI host bridge that is associated with the root bridge.
- **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL**: Provides a member function to retrieve the handles of the associated root bridges.

The definition of **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL** attempts to maintain compatibility with the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL** definition.

See the *EFI 1.10 Specification* for more information on the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL**.

Sample PCI Architectures

Sample PCI Architectures Overview

The PCI Host Bridge Resource Allocation Protocol is a protocol that is designed to provide a software abstraction for a wide variety of PCI architectures. This section provides examples of the following PCI architectures:

- Desktop system with 1 PCI root bridge
- Server system with 4 PCI root bridges
- Server system with 2 PCI segments
- Server system with 2 PCI host buses

This section is not intended to be an exhaustive list of the PCI architectures that the PCI Host Bridge Resource Allocation Protocol can support. Instead, it is intended to show the flexibility of this protocol to adapt to current and future platform designs.

Desktop System with 1 PCI Root Bridge

The figure below shows an example of a PCI host bus with one PCI root bridge. This PCI root bridge produces one PCI local bus that can contain PCI devices on the motherboard and/or PCI slots. This setup would be typical of a desktop system. In this system, the PCI root bridge needs minimal setup. Typically, the PCI root bridge will decode the following:

- The entire bus range on Segment 0
- The entire I/O space of the processor
- All the memory above the top of system memory

The firmware for this platform would produce the following:

- One instance of the PCI Host Bridge Resource Allocation Protocol
- One instance of PCI Root Bridge I/O Protocol

See the *EFI 1.10 Specification* for details of the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL**.

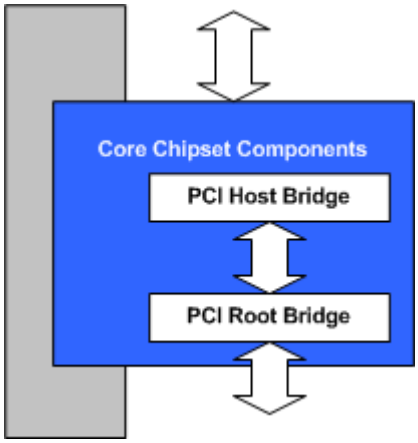

Figure 2-3. Desktop System with 1 PCI Root Bridge

Server System with 4 PCI Root Bridges

The figure below shows an example of a larger server with one PCI host Bus with four PCI root bridges (RBs). The PCI devices that are attached to the PCI root bridges are all part of the same coherency domain, which means they share the following:

- A common PCI I/O space
- A common PCI memory space
- A common PCI prefetchable memory space

As a result, each PCI root bridge must get resources out of a common pool. Each PCI root bridge produces one PCI local bus that can contain PCI devices on the motherboard or PCI slots. The firmware for this platform would produce the following:

- One instance of the PCI Host Bridge Resource Allocation Protocol
- Four instances of the PCI Root Bridge I/O Protocol

See the *EFI 1.10 Specification* for details of the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL**.

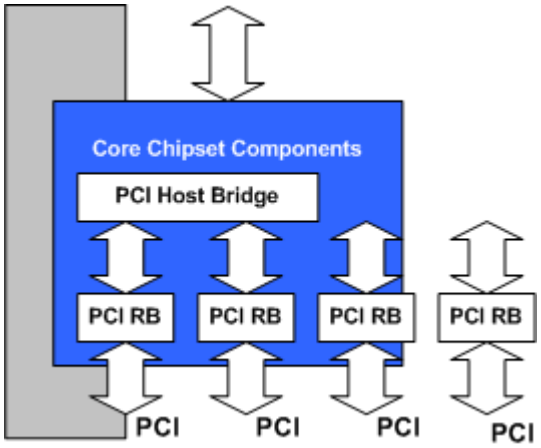

Figure 2-4. Server System with 4 PCI Root Bridges

Server System with 2 PCI Segments

The figure below shows an example of a server with one PCI host bus and two PCI root bridges (RBs). Each of these PCI root bridges is on a different PCI segment, which allows the system to have up to 512 PCI buses. A single PCI segment is limited to 256 PCI buses. These two segments do not share the same PCI configuration space, but they do share the following, which is why they can be described with a single PCI host bus:

- A common PCI I/O space
- A common PCI memory space
- A common PCI prefetchable memory space

The firmware for this platform would produce the following:

- One instance of the PCI Host Bridge Resource Allocation Protocol
- Two instances of the PCI Root Bridge I/O Protocol

See the *EFI 1.10 Specification* for details of the **`EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL`**.

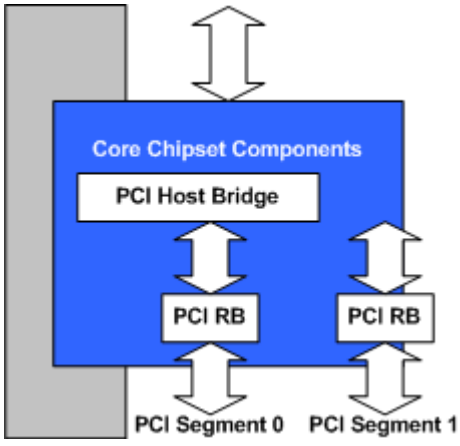

Figure 2-5. Server System with 2 PCI Segments

Server System with 2 PCI Host Buses

The figure below shows a server system with two PCI host buses and one PCI root bridge (RB) per PCI host bus. Like the figure in Server System with 2 PCI Segments, this system supports up to 512 PCI buses, but the following resources are not shared between the two PCI root bridges:

- PCI I/O space
- PCI memory space
- PCI prefetchable memory space

The firmware for this platform would produce the following:

- Two instances of the PCI Host Bridge Resource Allocation Protocol
- Two instances of the PCI Root Bridge I/O Protocol

See the *EFI 1.10 Specification* for details of the **`EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL`**.

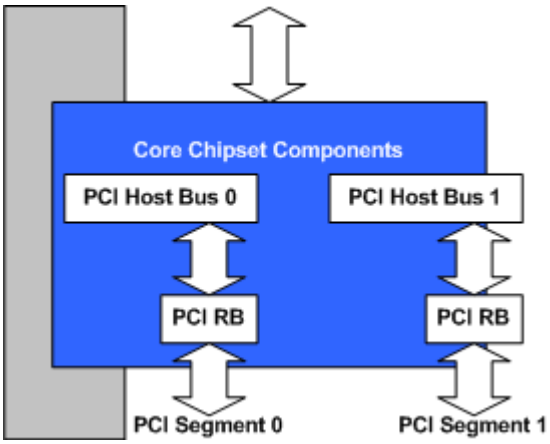

Figure 2-6. Server System with 2 PCI Host Buses

ISA Aliasing Considerations

The PCI host bridge driver will handle the ISA alias addresses based on the platform policy. The platform communicated the policy to the PCI host bridge driver using the **EFI_PCI_PLATFORM_PROTOCOL**. If the PCI host bridge driver cannot locate an instance of **EFI_PCI_PLATFORM_PROTOCOL**, it will not reserve the ISA alias addresses. The PCI bus driver is not aware of this policy and probes devices to gather resource requirements regardless of this policy. The **EFI_PCI_PLATFORM_PROTOCOL** is defined in the *Intel® Platform Innovation Framework for EFI PCI Platform Support Specification*.

NOTE

When it is started, a PCI device may request that the ISA alias ranges be forwarded to it through the **EFI_PCI_IO_PROTOCOL.Attributes()** *member function by setting the input parameter* ***Attributes*** *to* **EFI_PCI_IO_ATTRIBUTE_ISA_IO**. *If the ISA alias I/O addresses are not reserved during enumeration, such a request may fail because one or more PCI devices may be occupying aliased addresses.*

If the ISA alias I/O addresses are to be reserved during enumeration, the PCI host bridge driver is responsible for allocating four times the amount of the requested I/O. The PCI bus driver obtains the resources by calling one of the following member functions:

- **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL.GetProposedResources()**
- **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.Configuration()**

The PCI host bridge driver sets the _RNG bit to communicate the availability of the ISA alias range to the PCI bus driver. If the _RNG flag is set, the PCI bus enumerator is not allowed to allocate the ISA alias addresses to any PCI device. See Table 3-3 in the "Description" section of **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL** for the definition of the _RNG flag. In this case, a PCI device's request to turn on aliasing will succeed because one or more PCI devices may be occupying aliased addresses. The _RNG flag is the only aspect of the protocol interface structure that is affected by ISA aliasing.

Programming of Standard PCI Configuration Registers

This topic defines design guidelines for programming PCI configuration registers in the standard PCI header. It defines roles and responsibilities of various drivers.

Click on the following links to jump directly to the table listed:

- Table 2-1: Standard PCI Devices – Header Type 0
- Table 2-2: PCI-to-PCI Bridge – Header Type 1

Table 2-1. Standard PCI Devices – Header Type 0

PCI Configuration Register Bits	Programmed By
PCI command register – I/O, Memory, and Bus Master enable	PCI bus driver. This driver sets these values as requested by the device driver through the **EFI_PCI_IO_PROTOCOL** member functions.
PCI command register – SERR, PERR, MWI, Special Cycle Enable, Fast Back to Back Enable	Chipset/platform-specific code
PCI command register – VGA palette snoop	PCI device driver.
Cache line size	Chipset/platform code to match the processor's cache line size or some other value.
Latency timer	PCI bus driver. This driver programs this register to default values before it sends the **EfiPciBeforeResourceCollection** notification. For PCI devices, this value is 0x20. PCI-X* devices come out of reset with this register set to 0x40. The PCI bus driver does not change the setting. The PCI bus driver will also make sure that the default value for PCI devices is consistent with the **MIN_LAT** and **MAX_LAT** register values in the device's PCI configuration space. Chipset/platform code can overwrite this register during the **EfiPciBeforeResourceCollection** notification call. The new value may come from the end user using configuration options. The device driver may overwrite this value during its own **Start()** function.
BIST	PCI bus driver.
Base address registers	PCI bus driver.
Interrupt line	Not touched.
Subsystem vendor ID and Device ID	Chipset/platform code. Per the *PCI Specification,* these registers must get programmed before system software accesses the device. Some noncompliant or chipset devices may require that these registers be programmed during the preboot phase.

Table 2-2. PCI-to-PCI Bridge – Header Type 1

PCI Configuration Register Bits	Programmed By
PCI command register – I/O, Memory, Bus Master enable, VGA palette snoop	PCI bus driver. This driver sets these values as requested by the device driver through the **EFI_PCI_IO_PROTOCOL** member functions.
PCI command register – SERR, PERR, MWI, Fast Back to Back Enable, Special Cycle Enable	Chipset/platform-specific code.
Cache line size	Chipset/platform code to match the processor's cache line size or some other value.
Latency timer	PCI bus driver. This driver programs to default values before it sends the **EfiPciBeforeResourceCollection** notification. For PCI devices, this value is 0x20. PCI-X devices come out of reset with this register set to 0x40.The PCI bus driver does not change the setting. The PCI bus driver will also make sure that the default value for PCI devices is consistent with the **MIN_LAT** and **MAX_LAT** register values in the device's PCI configuration space. Chipset/platform code can overwrite this register during the **EfiPciBeforeResourceCollection** notification call. The new value may come from the end user using configuration options.
Base addresses registers, bus, I/O, and memory aperture registers	PCI bus driver.
Interrupt line	Not touched.
Bridge control register – ISA Enable, VGA Enable	PCI bus driver. This driver sets these values as requested by the device driver through the **EFI_PCI_IO_PROTOCOL** member functions.
Bridge control register – PERR Enable, SERR Enable, Fast Back to Back, Discard Timers	Chipset/platform-specific code.
Bridge control register – Secondary Bus Reset	PCI bus driver is permitted to reset the secondary bus during enumeration. The chipset/platform code may also reset the secondary bus during the **EfiPciBeforeChildBusEnumeration** notification.

Sample Implementation

Typically, the PCI bus driver will enumerate and allocate resources to all devices for a PCI host bridge. A sample algorithm for PCI enumeration is described below to clarify some of the finer points of the **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL**. Actual implementations may vary. Calls to **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL.PreprocessController()** are not included for the sake of clarity.

Unless noted otherwise, all functions that are listed below are member functions of the **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL**.

1. If the hardware supports dynamically changing the number of PCI root buses or changing the segment number that is associated with a PCI root bus, such changes must be completed before the next steps.
2. The chipset/platform driver(s) creates a device handle for the PCI host bridges in the system(s) and installs an instance of the **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL** on that handle.
3. The chipset/platform driver(s) creates a device handle for every PCI root bridge and installs the following on that handle:
   - An instance of **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL**
   - An instance of **EFI_DEVICE_PATH_PROTOCOL**

   It is expected that a single driver will handle a PCI host bridge, as well as all the associated PCI root bridges. The *ParentHandle* field of **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL** must be initialized with the handle for the PCI host bridge that contains an instance of the **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL**.

...Other initialization activities take place.

4. The **EFI_DRIVER_BINDING_PROTOCOL.Start()** function of the PCI bus driver is called and is passed the device handle of a PCI root bridge. The **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL** instance that is associated with the PCI root bridge can be found by using the *ParentHandle* field of **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL**. **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL** must be present in Framework-compliant systems.

...

5. Begin the PCI enumeration process. The order in which the various member functions are called cannot be changed. Between any two steps, there can be any amount of implementation-specific code as long as it does not call any member functions of **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL**. This requirement is necessary to keep the state machines in the PCI host bridge allocation driver and the PCI bus enumerator in sync.
6. Notify the host bridge driver that PCI enumeration is about to begin by calling **NotifyPhase (EfiPciHostBridgeBeginEnumeration)**. This member function must be the first one that gets called. PCI enumeration has two steps: bus enumeration and resource enumeration.

7. Notify the host bridge driver that bus enumeration is about to begin by calling **NotifyPhase (EfiPciHostBridgeBeginBusAllocation)**.
8. Do the following for every PCI root bridge handle:
   a. Call **StartBusEnumeration (This,RootBridgeHandle)**.
   b. Make sure each PCI root bridge handle supports the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL**.
   c. Allocate memory to hold resource requirements. These resources can be two resource trees: one to hold bus requirements and another to hold the I/O and memory requirements.
   d. Call **GetAllocAttributes()** to get the attributes of this PCI root bridge. This information is used to combine different types of memory resources in the next step.
   e. Scan all the devices in the specified bus range and on the specified segment. If it is a PCI-to-PCI bridge, update the bus numbers and program the bus number registers in the PCI-to-PCI bridge hardware. If it is an ordinary device, collect the resource request and add up all of these requests in multiple pools (e.g., I/O, 32-bit prefetchable memory). Combine different types of memory requests at an appropriate level based on the PCI root bridge attributes. Update the resource requirement information accordingly. On every PCI root bridge, reserve space to cover the largest expansion ROMs on that bus, which will allow the PCI bus driver to retrieve expansion ROMs from the PCI card or device without having to reprogram the PCI host bridge. Because the memory and I/O resource collection step does not call any member function of **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL**, it can be performed at a later time.
   f. Once the number of PCI buses under this PCI root bridge is known, call **SetBusNumbers()** with this information.
9. Notify the host bridge driver that the bus allocation phase is over by calling **NotifyPhase (EfiPciHostBridgeEndBusAllocation)**.
10. Notify the host bridge driver that resource allocation is about to begin by calling **NotifyPhase (EfiPciHostBridgeBeginResourceAllocation)**.
11. For every PCI root bridge handle, call **SubmitResources()**. The *Configuration* information is derived from the resource requirements that were computed in step 8 above.
12. Call **NotifyPhase (EfiPciHostBridgeAllocateResources)** to allocate the necessary resources. This call should not be made unless resource requirements for all the PCI root bridges have been submitted. If the call succeeds, go to next step. Otherwise, there are two options:
    a. Make do with the smaller ranges.
    b. Call **GetProposedResources()** to retrieve the proposed settings and examine the differences. Prioritize various requests and drop lower-priority requests. Call **NotifyPhase (EfiPciHostBridgeFreeResources)** to undo the previous allocation. Go back to step 11 with reduced requirements, which includes resubmitting requests for all the root bridges.
13. Call **NotifyPhase (EfiPciHostBridgeSetResources)** to program the hardware. At this point, the decode logic in this host bridge is fully set up.

14. Do the following for every root bridge handle:
    a. Obtain the resource range that is assigned to a PCI root bridge by calling the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.Configuration()** member function on that handle.
    b. From the resource range that is assigned to the PCI root bridge, assign resources to all the devices. Program the Base Address Registers (BARs) in all the PCI functions and decode registers in PCI-to-PCI bridges. If a PCI device has a PCI option ROM, copy the contents to a buffer in memory. It is possible to defer the BAR programming for a PCI controller until a connect request for the device is received.
    c. Create a device handle for each PCI device as required.
    d. Install an instance of **EFI_PCI_IO_PROTOCOL** and **EFI_DEVICE_PATH_PROTOCOL** on each of these handles.
15. Notify the host bridge driver that resource allocation is complete by calling **NotifyPhase (EfiPciHostBridgeEndResourceAllocation)**.
16. Deallocate any temporary buffers.

Looping on PCI root bridges is accomplished with the following algorithm:

```
RootBridgeHandle = NULL;
while (GetNextRootBridge(RootBridgeHandle) == EFI_SUCCESS) {
    . . .
}
```

3
Code Definitions

Introduction

This section contains the basic definitions of the PCI Host Bridge Resource Allocation Protocol. The following protocol is defined in this section:

- **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_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_HOST_BRIDGE_RESOURCE_ALLOCATION_PHASE**
- **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_ATTRIBUTES**
- **EFI_PCI_CONTROLLER_RESOURCE_ALLOCATION_PHASE**

PCI Host Bridge Resource Allocation Protocol

EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL

Summary

Provides the basic interfaces to abstract a PCI host bridge resource allocation. **This protocol is mandatory if the system includes PCI devices.**

GUID

```
#define EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL_GUID \
  {
0xCF8034BE,0x6768,0x4d8b,0xB7,0x39,0x7C,0xCE,0x68,0x3A,0x9F,0xBE
}
```

Protocol Interface Structure

```
typedef struct _EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL {
  EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL_NOTIFY_PHASE
                                          NotifyPhase;
  EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL_GET_NEXT_ROOT_BRIDGE
                                          GetNextRootBridge;
  EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL_GET_ATTRIBUTES
                                          GetAllocAttributes;
  EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL_START_BUS_ENUMERATION
                                          StartBusEnumeration;
  EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL_SET_BUS_NUMBERS
                                          SetBusNumbers;
  EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL_SUBMIT_RESOURCES
                                          SubmitResources;
  EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL_GET_PROPOSED_RESOURCES
                                          GetProposedResources;
  EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL_PREPROCESS_CONTROLLER
                                          PreprocessController;
} EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL;
```

Parameters

NotifyPhase

The notification from the PCI bus enumerator that it is about to enter a certain phase during the enumeration process. See the **NotifyPhase()** function description.

GetNextRootBridge

Retrieves the device handle for the next PCI root bridge that is produced by the host bridge to which this instance of the **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL** is attached. See the **GetNextRootBridge()** function description. See PCI Terms Used in This Document for a definition of a PCI root bridge.

GetAllocAttributes

Retrieves the allocation-related attributes of a PCI root bridge. See the **GetAllocAttributes()** function description.

StartBusEnumeration

Sets up a PCI root bridge for bus enumeration. See the **StartBusEnumeration()** function description.

SetBusNumbers

Sets up the PCI root bridge so that it decodes a specific range of bus numbers. See the **SetBusNumbers()** function description.

SubmitResources

Submits the resource requirements for the specified PCI root bridge. See the **SubmitResources()** function description.

GetProposedResources

Returns the proposed resource assignment for the specified PCI root bridges. See the **GetProposedResources()** function description.

PreprocessController

Provides hooks from the PCI bus driver to every PCI controller (device/function) at various stages of the PCI enumeration process that allow the host bridge driver to preinitialize individual PCI controllers before enumeration. See the **PreprocessController()** function description.

Description

The **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL** provides the basic resource allocation services to the PCI bus driver. There is one **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL** instance for each PCI host bridge in a system. The following will typically have only one PCI host bridge:

- Embedded systems
- Desktops
- Workstations
- Most servers

High-end servers may have multiple PCI host bridges. A PCI bus driver that wishes to manage a PCI bus in a system will have to retrieve the **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL** instance that is associated with the PCI bus to be managed. A device handle for a PCI host bridge will not contain an **EFI_DEVICE_PATH_PROTOCOL** instance because the PCI host bridge is a software abstraction and has no equivalent in the ACPI name space.

All applicable member functions use ACPI 2.0 resource descriptors to describe resources. Using ACPI resource descriptors does the following:

- Allows other types of resources to be described in the future because they are very generic in nature.
- Avoids multiple structure definitions for describing resources.
- Maintains compatibility with the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL** definition.

Only the following two resource descriptor types from the *ACPI Specification* may be used to describe the current resources that are allocated to a PCI root bridge:

- 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. Table 3-3 and Table 3-4 define how resource-specific flags are used. 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

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 2: Bus number range
0x04	0x01		General flags. Flags that are common to all resource types: • **Bits[7:4]:** Reserved (must be 0) • **Bit[3] _MAF:** Always returned as 1 while returning allocated requests to indicate that the specified max address is fixed. • **Bit[2] _MIF:** Always returned as 1 while returning allocated requests to indicate that the specified min address is fixed. • **Bit[1] _DEC:** Ignored. • **Bit[0]:** Ignored.

continued

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

Byte Offset	Byte Length	Data	Description
0x05	0x01		Type-specific flags. Ignored except as defined in Table 3-3 and Table 3-4 below.
0x06	0x08		Address Space Granularity. Used to differentiate between a 32-bit memory request and a 64-bit memory request. For a 32-bit memory request, this field should be set to 32. For a 64-bit memory request, this field should be set to 64. Ignored for I/O and bus resource requests. Ignored during **GetProposedResources()**.
0x0E	0x08		Address Range Minimum. Set to the base of the allocated address range (bus, I/O, memory) during **GetProposedResources()**. Ignored during **SubmitResources()**.
0x16	0x08		Address Range Maximum. Used to indicate alignment requirement during **SubmitResources()** and ignored during **GetProposedResources()**. This value must be 2^n-1. The address base must be a multiple of the granularity field. That is, if this field is 4k-1, the allocated address must be a multiple of 4 KB. **Note:** The interpretation of this field is different from the *ACPI Specification* and PCI Root Bridge I/O Protocol.
0x1E	0x08		Address Translation Offset. Used to indicate the allocation status during **GetProposedResources()** and ignored during **SubmitResources()**. Allocation status is defined in "Related Definitions" in **GetProposedResources()**. **Note:** The interpretation of this field is different from the *ACPI Specification* and PCI Root Bridge I/O Protocol.
0x26	0x08		Address Range Length. This field specifies the amount of resources that are requested or allocated in number of bytes.

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.

Table 3-3. I/O Resource Flag (Resource Type = 1) Usage

Bits	Meaning
Bits[7:1]	Ignored.
Bit[0]	_RNG. Ignored during an allocation request. Setting this bit while returning allocated resources means that the I/O allocation must be limited to the non-ISA I/O ranges. In that case, the PCI bus driver must allocate I/O addresses out of the non-ISA I/O ranges. The following are the non-ISA I/O ranges: • n100–n3FF • n500–n7FF • n900–nBFF • nD00–nFFF See ISA Aliasing Considerations for more details.

Table 3-4. Memory Resource Flag (Resource Type = 0) Usage

Bits	Meaning
Bits[7:3]	Ignored.
Bit[2:1]	_MEM. Memory attributes. **Value and Meaning:** 0 The memory is nonprefetchable. 1 Invalid. 2 Invalid. 3 The memory is prefetchable. **Note:** The interpretation of these bits is somewhat different from the *ACPI Specification.* According to the *ACPI Specification,* a value of 0 implies noncacheable memory and the value of 3 indicates prefetchable and cacheable memory.
Bit[0]	Ignored.

EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL. NotifyPhase()

Summary

These are the notifications from the PCI bus driver that it is about to enter a certain phase of the PCI enumeration process.

Prototype

```
typedef
EFI_STATUS
(EFIAPI
*EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL_NOTIFY_PHASE) (
  IN EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL   *This,
  IN EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PHASE      Phase
  );
```

Parameters

This

Pointer to the **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL** instance.

Phase

The phase during enumeration. Type **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PHASE** is defined in "Related Definitions" below.

Description

This member function can be used to notify the host bridge driver to perform specific actions, including any chipset-specific initialization, so that the chipset is ready to enter the next phase. Eight notification points are defined at this time. See "Related Definitions" below for definitions of various notification points and Sample Implementation in the Design Discussion chapter for usage.

More synchronization points may be added as required in the future.

Related Definitions

```
//*******************************************************
// EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PHASE
//*******************************************************
typedef enum {
  EfiPciHostBridgeBeginEnumeration,
  EfiPciHostBridgeBeginBusAllocation,
  EfiPciHostBridgeEndBusAllocation,
  EfiPciHostBridgeBeginResourceAllocation,
  EfiPciHostBridgeAllocateResources,
  EfiPciHostBridgeSetResources,
  EfiPciHostBridgeFreeResources,
  EfiPciHostBridgeEndResourceAllocation
} EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PHASE;
```

Following is a description of the fields in the above enumeration:

EfiPciHostBridgeBeginEnumeration	Resets the host bridge PCI apertures and internal data structures. The PCI enumerator should issue this notification before starting a fresh enumeration process. Enumeration cannot be restarted after sending any other notification such as **EfiPciHostBridgeBeginBusAllocation**.
EfiPciHostBridgeBeginBusAllocation	The bus allocation phase is about to begin. No specific action is required here. This notification can be used to perform any chipset-specific programming.
EfiPciHostBridgeEndBusAllocation	The bus allocation and bus programming phase is complete. No specific action is required here. This notification can be used to perform any chipset-specific programming.
EfiPciHostBridgeBeginResourceAllocation	The resource allocation phase is about to begin. No specific action is required here. This notification can be used to perform any chipset-specific programming.

continued

EfiPciHostBridgeAllocateResources	Allocates resources per previously submitted requests for all the PCI root bridges. These resource settings are returned on the next call to **GetProposedResources()**. Before calling **NotifyPhase()** with a *Phase* of **EfiPciHostBridgeAllocateResource**, the PCI bus enumerator is responsible for gathering I/O and memory requests for all the PCI root bridges and submitting these requests using **SubmitResources()**. This function pads the resource amount to suit the root bridge hardware, takes care of dependencies between the PCI root bridges, and calls the Global Coherency Domain (GCD) with the allocation request. In the case of padding, the allocated range could be bigger than what was requested. Note that the size of the allocated range could be smaller than what was requested. This scenario could happen due to an allocation failure, a host bridge hardware limitation, or any other reason. In that case, the call will return an **EFI_OUT_OF_RESOURCES** error. If the allocated windows are smaller than what was requested, the PCI bus enumerator may not be able to fit all the devices within the range. The PCI bus driver can call **GetProposedResouces()** to find out which of the resource types were partially allocated and the difference between the amount that was requested and the amount that was allocated. The PCI bus enumerator should readjust the requested sizes (by dropping certain PCI devices or PCI buses) to obtain a best fit. The PCI bus driver can call **NotifyPhase (EfiPciHostBridgeFreeResources)** to free up the original assignments and resubmit the adjusted resource requests with **SubmitResources()**.
EfiPciHostBridgeSetResources	Programs the host bridge hardware to decode previously allocated resources (proposed resources) for all the PCI root bridges. After the hardware is programmed, reassigning resources will not be supported. The bus settings are not affected.
EfiPciHostBridgeFreeResources	Deallocates resources that were previously allocated for all the PCI root bridges and resets the I/O and memory apertures to their initial state. The bus settings are not affected. If the request to allocate resources fails, the PCI enumerator can use this notification to deallocate previous resources, adjust the requests, and retry allocation.
EfiPciHostBridgeEndResourceAllocation	The resource allocation phase is completed. No specific action is required here. This notification can be used to perform any chipset-specific programming.

Status Codes Returned

EFI_SUCCESS	The notification was accepted without any errors.
EFI_INVALID_PARAMETER	The *Phase* is invalid.
EFI_NOT_READY	This phase cannot be entered at this time. For example, this error is valid for a *Phase* of **EfiPciHostBridgeAllocateResources** if **SubmitResources()** has not been called for one or more PCI root bridges before this call.
EFI_DEVICE_ERROR	Programming failed due to a hardware error. This error is valid for a *Phase* of **EfiPciHostBridgeSetResources**.
EFI_OUT_OF_RESOURCES	The request could not be completed due to a lack of resources. This error is valid for a *Phase* of **EfiPciHostBridgeAllocateResources** if the previously submitted resource requests cannot be fulfilled or were only partially fulfilled.

EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL. GetNextRootBridge()

Summary

Returns the device handle of the next PCI root bridge that is associated with this host bridge.

Prototype

```
typedef
EFI_STATUS
(EFIAPI
*EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL_GET_NEXT_ROOT_BRIDGE) (
  IN     EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL  *This,
  IN OUT EFI_HANDLE                                        *RootBridgeHandle
  );
```

Parameters

This

Pointer to the **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL** instance.

RootBridgeHandle

Returns the device handle of the next PCI root bridge. On input, it holds the *RootBridgeHandle* that was returned by the most recent call to **GetNextRootBridge()**. If *RootBridgeHandle* is **NULL** on input, the handle for the first PCI root bridge is returned. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification.*

Description

This function is called multiple times to retrieve the device handles of all the PCI root bridges that are associated with this PCI host bridge. Each PCI host bridge is associated with one or more PCI root bridges. On each call, the handle that was returned by the previous call is passed into the interface, and on output the interface returns the device handle of the next PCI root bridge. The caller can use the handle to obtain the instance of the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL** for that root bridge. When there are no more PCI root bridges to report, the interface returns **EFI_NOT_FOUND**. A PCI enumerator must enumerate the PCI root bridges in the order that they are returned by this function.

The search is initiated by passing in a **NULL** device handle as input. Some of the member functions of the **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL** operate on a PCI root bridge and expect the *RootBridgeHandle* as an input.

There is no requirement that this function return the root bridges in any specific relation with the EFI device paths of the root bridges.

This function can also be used to determine the number of PCI root bridges that were produced by this PCI host bridge. The host bridge hardware may provide mechanisms to change the number of

root bridges that it produces, but such changes must be completed before the **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL** is installed.

Status Codes Returned

EFI_SUCCESS	The requested attribute information was returned.
EFI_INVALID_PARAMETER	*RootBridgeHandle* is not an **EFI_HANDLE** that was returned on a previous call to **GetNextRootBridge()**.
EFI_NOT_FOUND	There are no more PCI root bridge device handles.

EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL. GetAllocAttributes()

Summary

Returns the allocation attributes of a PCI root bridge.

Prototype

```
typedef
EFI_STATUS
(EFIAPI * EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_GET_ATTRIBUTES) (
  IN  EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL  *This,
  IN  EFI_HANDLE                                        RootBridgeHandle,
  OUT UINT64                                            *Attributes
  );
```

Parameters

This

Pointer to the **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL** instance.

RootBridgeHandle

The device handle of the PCI root bridge in which the caller is interested. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification.*

Attributes

The pointer to attributes of the PCI root bridge. The permitted attribute values are defined in "Related Definitions" below.

Description

The function returns the allocation attributes of a specific PCI root bridge. The attributes can vary from one PCI root bridge to another. These attributes are different from the decode-related attributes that are returned by the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.GetAttributes()** member function. The *RootBridgeHandle* parameter is used to specify the instance of the PCI root bridge. The device handles of all the root bridges that are associated with this host bridge must be obtained by calling **GetNextRootBridge()**. The attributes are static in the sense that they do not change during or after the enumeration process. The hardware may provide mechanisms to change the attributes on the fly, but such changes must be completed before **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL** is installed. The permitted values of **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_ATTRIBUTES** are defined in "Related Definitions" below. The caller uses these attributes to combine multiple resource requests. For example, if the flag **EFI_PCI_HOST_BRIDGE_COMBINE_MEM_PMEM** is set, the PCI bus enumerator needs to include requests for the prefetchable memory in the nonprefetchable memory pool and not request any prefetchable memory.

Related Definitions

```
//*******************************************************
// EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_ATTRIBUTES
//*******************************************************

#define EFI_PCI_HOST_BRIDGE_COMBINE_MEM_PMEM       1
#define EFI_PCI_HOST_BRIDGE_MEM64_DECODE           2
```

Following is a description of the fields in the above definition:

EFI_PCI_HOST_BRIDGE_ COMBINE_MEM_PMEM	If this bit is set, then the PCI root bridge does not support separate windows for nonprefetchable and prefetchable memory. A PCI bus driver needs to include requests for prefetchable memory in the nonprefetchable memory pool.
EFI_PCI_HOST_BRIDGE_ MEM64_DECODE	If this bit is set, then the PCI root bridge supports 64-bit memory windows. If this bit is not set, the PCI bus driver needs to include requests for a 64-bit memory address in the corresponding 32-bit memory pool.

Status Codes Returned

EFI_SUCCESS	The requested attribute information was returned.
EFI_INVALID_PARAMETER	*RootBridgeHandle* is not a valid root bridge handle.
EFI_INVALID_PARAMETER	*Attributes* is **NULL**.

EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL. StartBusEnumeration()

Summary

Sets up the specified PCI root bridge for the bus enumeration process.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL_START_BUS_ENUMERATION) (
  IN  EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL  *This,
  IN  EFI_HANDLE                                        RootBridgeHandle,
  OUT VOID                                              **Configuration
  );
```

Parameters

This

Pointer to the **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL** instance.

RootBridgeHandle

The PCI root bridge to be set up. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification.*

Configuration

Pointer to the pointer to the PCI bus resource descriptor.

Description

This member function sets up the root bridge for bus enumeration and returns the PCI bus range over which the search should be performed in ACPI 2.0 resource descriptor format. The following table lists the fields in the ACPI 2.0 resource descriptor that are set for **StartBusEnumeration()**.

Table 3-5. ACPI 2.0 Resource Descriptor Field Values for StartBusEnumeration()

Field	Setting
Address Range Minimum	Set to the lowest bus number to be scanned.
Address Range Length	Set to the number of PCI buses that may be scanned. The highest bus number is computed by adding the length to the lowest bus number and subtracting 1.
Address Range Maximum	Ignored.
All other fields	Ignored.

Note: Click the links in the table above or see Table 3-1 in the "Description" section of the PCI Host Bridge Resource Allocation Protocol definition for a description of these ACPI resource descriptor fields.

This function cannot return resource descriptors for anything other than bus resources. This function can be used to prevent a PCI bus driver from scanning certain PCI buses to work around a chipset limitation. Because the size of ACPI resource descriptors is not fixed,

StartBusEnumeration() is responsible for allocating memory for the buffer *Configuration*.

The PCI segment is implicit and is identified by the *SegmentNumber* field in the instance of the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL** that is installed on the PCI root bridge handle *RootBridgeHandle*.

Status Codes Returned

EFI_SUCCESS	The PCI root bridge was set up and the bus range was returned in *Configuration*.
EFI_INVALID_PARAMETER	*RootBridgeHandle* is not a valid root bridge handle.
EFI_DEVICE_ERROR	Programming failed due to a hardware error.
EFI_OUT_OF_RESOURCES	The request could not be completed due to a lack of resources.

EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL. SetBusNumbers()

Summary

Programs the PCI root bridge hardware so that it decodes the specified PCI bus range.

Prototype

```
typedef
EFI_STATUS
(EFIAPI
*EFI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL_SET_BUS_NUMBERS) (
  IN EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL  *This,
  IN EFI_HANDLE                                        RootBridgeHandle,
  IN VOID                                              *Configuration
  );
```

Parameters

This

Pointer to the **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL** instance.

RootBridgeHandle

The PCI root bridge whose bus range is to be programmed. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification.*

Configuration

The pointer to the PCI bus resource descriptor.

Description

This member function programs the specified PCI root bridge to decode the bus range that is specified by the input parameter *Configuration*.

The bus range information is specified in terms of the ACPI 2.0 resource descriptor format. The following table lists the fields in the ACPI 2.0 resource descriptor that are set for **SetBusNumbers()**.

Table 3-6. ACPI 2.0 Resource Descriptor Field Values for SetBusNumbers()

Field	Setting
Address Range Minimum	Set to the lowest bus number to be decoded.
Address Range Length	Set to the number of PCI buses that should be decoded. The highest bus number is computed by adding the length to the lowest bus number and subtracting 1.
Address Range Maximum	Ignored.
All other fields	Ignored.

Note: Click the links in the table above or see Table 3-1 in the "Description" section of the PCI Host Bridge Resource Allocation Protocol definition for a description of these ACPI resource descriptor fields.

This call will return **EFI_INVALID_PARAMETER** without programming the hardware if either of the following are specified:

- Any descriptors other than bus type descriptors
- Any invalid descriptors

The bus range is typically a subset of what was returned during **StartBusEnumeration()**. If **SetBusNumbers()** is called with incorrect (but valid) parameters, it may cause system failure.

The PCI segment is implicit and is identified by the *SegmentNumber* field in the instance of the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL** that is installed on the PCI root bridge handle *RootBridgeHandle*. This call cannot alter the following:

- The *SegmentNumber* field in the corresponding instances of the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL**
- The segment number settings in the hardware

The caller is responsible for allocating and deallocating a buffer to hold *Configuration*. If the call returns **EFI_DEVICE_ERROR**, the PCI bus enumerator can optionally attempt another bus setting.

Status Codes Returned

EFI_SUCCESS	The bus range for the PCI root bridge was programmed.
EFI_INVALID_PARAMETER	*RootBridgeHandle* is not a valid root bridge handle.
EFI_INVALID_PARAMETER	*Configuration* is **NULL**.
EFI_INVALID_PARAMETER	*Configuration* does not point to a valid ACPI 2.0 resource descriptor.
EFI_INVALID_PARAMETER	*Configuration* does not include a valid ACPI 2.0 bus resource descriptor.
EFI_INVALID_PARAMETER	*Configuration* includes valid ACPI 2.0 resource descriptors other than bus descriptors.
EFI_INVALID_PARAMETER	*Configuration* contains one or more invalid ACPI resource descriptors.
EFI_INVALID_PARAMETER	"Address Range Minimum" is invalid for this root bridge.
EFI_INVALID_PARAMETER	"Address Range Length" is invalid for this root bridge.
EFI_DEVICE_ERROR	Programming failed due to a hardware error.

EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL. SubmitResources()

Summary

Submits the I/O and memory resource requirements for the specified PCI root bridge.

Prototype

```
typedef
EFI_STATUS
(EFIAPI * EFI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL_SUBMIT_RESOURCES) (
  IN EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL  *This,
  IN EFI_HANDLE                                        RootBridgeHandle,
  IN VOID                                              *Configuration
  );
```

Parameters

This

Pointer to the **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL** instance.

RootBridgeHandle

The PCI root bridge whose I/O and memory resource requirements are being submitted. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification.*

Configuration

The pointer to the PCI I/O and PCI memory resource descriptor.

Description

This function is used to submit all the I/O and memory resources that are required by the specified PCI root bridge. The input parameter *Configuration* is used to specify the following:

- The various types of resources that are required
- The associated lengths in terms of ACPI 2.0 resource descriptor format

The following table lists the fields in the ACPI 2.0 resource descriptor that are set for **SubmitResources()**.

Table 3-7. ACPI 2.0 Resource Descriptor Field Values for `SubmitResources()`

Field	Setting
Address Range Length	Set to the size of the aperture that is requested.
Address Space Granularity	Used to differentiate between a 32-bit memory request and a 64-bit memory request. For a 32-bit memory request, this field should be set to 32. For a 64-bit memory request, this field should be set to 64. All other values result in this function returning the error code of **`EFI_INVALID_PARAMETER`**.
Address Range Maximum	Used to specify the alignment requirement. If "Address Range Maximum" is of the form 2^n-1, this member function returns the error code **`EFI_INVALID_PARAMETER`**. The address base must be a multiple of the granularity field. That is, if this field is 4 KB-1, the allocated address must be a multiple of 4 KB.
Address Range Minimum	Ignored.
Address Translation Offset	Ignored.
All other fields	Ignored.

Note: Click the links in the table above or see Table 3-1 in the "Description" section of the PCI Host Bridge Resource Allocation Protocol definition for a description of these ACPI resource descriptor fields.

The caller must ask for appropriate alignment using the "Address Range Maximum" field. The caller is responsible for allocating and deallocating a buffer to hold *`Configuration`*.

It is considered an error if no resource requests are submitted for a PCI root bridge. If a PCI root bridge does not require any resources, a zero-length resource request must explicitly be submitted.

If the *`Configuration`* includes one or more invalid resource descriptors, all the resource descriptors are ignored and the function returns **`EFI_INVALID_PARAMETER`**.

Status Codes Returned

EFI_SUCCESS	The I/O and memory resource requests for a PCI root bridge were accepted.
EFI_INVALID_PARAMETER	*`RootBridgeHandle`* is not a valid root bridge handle.
EFI_INVALID_PARAMETER	*`Configuration`* is **`NULL`**.
EFI_INVALID_PARAMETER	*`Configuration`* does not point to a valid ACPI 2.0 resource descriptor.
EFI_INVALID_PARAMETER	*`Configuration`* includes requests for one or more resource types that are not supported by this PCI root bridge. This error will happen if the caller did not combine resources according to *`Attributes`* that were returned by **`GetAllocAttributes()`**.
EFI_INVALID_PARAMETER	"Address Range Maximum" is invalid.
EFI_INVALID_PARAMETER	"Address Range Length" is invalid for this PCI root bridge.
EFI_INVALID_PARAMETER	"Address Space Granularity" is invalid for this PCI root bridge.

EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL. GetProposedResources()

Summary

Returns the proposed resource settings for the specified PCI root bridge.

Prototype

```
typedef
EFI_STATUS
(EFIAPI * EFI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL_GET_PROPOSED_RESOURCES) (
  IN  EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL  *This,
  IN  EFI_HANDLE                                        RootBridgeHandle,
  OUT VOID                                              **Configuration
  );
```

Parameters

This

Pointer to the **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL** instance.

RootBridgeHandle

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

Configuration

The pointer to the pointer to the PCI I/O and memory resource descriptor.

Description

This member function returns the proposed resource settings for the specified PCI root bridge. The proposed resource settings are prepared when **NotifyPhase()** is called with a *Phase* of **EfiPciHostBridgeAllocateResources**. The output parameter *Configuration* specifies the following:

- The various types of resources, excluding bus resources, that are allocated
- The associated lengths in terms of ACPI 2.0 resource descriptor format

The following table lists the fields in the ACPI 2.0 resource descriptor that are set for **GetProposedResources()**.

Table 3-8. ACPI 2.0 Resource Descriptor Field Values for `GetProposedResources()`

Field	Setting
Address Range Length	Set to the size of the aperture that is requested.
Address Space Granularity	Ignored.
Address Range Minimum	Indicates the starting address of the allocated ranges.
Address Translation Offset	Indicates the allocation status. Allocation status is defined in "Related Definitions" below.
Address Range Maximum	Ignored.
All other fields	Ignored.

Note: Click the links in the table above or see Table 3-1 in the "Description" section of the PCI Host Bridge Resource Allocation Protocol definition for a description of these ACPI resource descriptor fields.

The callee is responsible for allocating a buffer to hold *`Configuration`* because the caller does not know the number of descriptors that are required. The caller is also responsible for deallocating the buffer.

If **`NotifyPhase()`** is called with a *`Phase`* of **`EfiPciHostBridgeAllocateResources`** and returns **`EFI_OUT_OF_RESOURCES`**, the PCI bus enumerator may use **`GetProposedResources()`** to retrieve the proposed settings. The **`EFI_OUT_OF_RESOURCES`** error status indicates that one or more requests could not be fulfilled or were partially fulfilled. Additional details of the allocation status for each type of resource can be retrieved from the "Address Translation Offset" field in the resource descriptor that was returned by this function; also see "Related Definitions" below for defined allocation status values. This error could happen for the following reasons:

- Allocation failure
- A limitation in the host bridge hardware
- Any other reason

If the allocated windows are smaller than what was requested, the PCI bus enumerator may not be able to fit all the devices within the range. In that case, the PCI bus enumerator may choose to readjust the requested sizes (by dropping certain devices or PCI buses) to obtain a best fit. The PCI bus driver calls **`NotifyPhase()`** with a *`Phase`* of **`EfiPciHostBridgeFreeResources`** to free the original assignments.

If this member function is able to only partially fulfill the requests for one or more resource types, the root bridges that are first in the list will get resources first. The ordering of the root bridges is determined by the output of **`GetNextRootBridge()`**. The handle to the first root bridge is obtained by calling **`GetNextRootBridge()`** with an input handle of **`NULL`**.

In the case of I/O resources, the PCI bus enumerator must check the _RNG flag. If this flag is set, the I/O ranges that are allocated to the devices must come from the non-ISA I/O subset.

For example, if this flag is set, the "Address Range Minimum" is 0x1000, and the "Address Range Length" is 0x1000, then the following I/O ranges can be allocated to PCI devices:

- 0x1000–0x10FF
- 0x1400–0x14FF
- 0x1800–0x18FF
- 0x1C00–0x1CFF

This call is made before **NotifyPhase()** is called with a *Phase* of **EfiPciHostBridgeSetResources**. After that time, the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.Configuration()** member function should be used to obtain the resources that were consumed by a particular PCI root bridge.

Related Definitions

```
//*******************************************************************
// EFI_RESOURCE_ALLOCATION_STATUS
//*******************************************************************
typedef  UINT64    EFI_RESOURCE_ALLOCATION_STATUS;

#define EFI_RESOURCE_SATISFIED                    0
#define EFI_RESOURCE_NOT_SATISFIED                (UINT64) -1
```

Following is a description of the fields in the above definition. All other values indicate that the request of this resource type could be partially fulfilled. The exact value indicates how much more space is still required to fulfill the requirement.

EFI_RESOURCE_SATISFIED	The request of this resource type could be fulfilled.
EFI_RESOURCE_NOT_SATISFIED	The request of this resource type could not be fulfilled for its absence in the host bridge resource pool.

Status Codes Returned

EFI_SUCCESS	The requested parameters were returned.
EFI_INVALID_PARAMETER	*RootBridgeHandle* is not a valid root bridge handle.
EFI_DEVICE_ERROR	Programming failed due to a hardware error.
EFI_OUT_OF_RESOURCES	The request could not be completed due to a lack of resources.

EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL. PreprocessController()

Summary

Provides the hooks from the PCI bus driver to every PCI controller (device/function) at various stages of the PCI enumeration process that allow the host bridge driver to preinitialize individual PCI controllers before enumeration.

Prototype

```
typedef
EFI_STATUS
(EFIAPI * EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL_PREPROCESS_CONTROLLER)(
  IN  EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL *This,
  IN  EFI_HANDLE                                       RootBridgeHandle,
  IN  EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_PCI_ADDRESS      PciAddress,
  IN  EFI_PCI_CONTROLLER_RESOURCE_ALLOCATION_PHASE     Phase
  );
```

Parameters

This

Pointer to the **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL** instance.

RootBridgeHandle

The associated PCI root bridge handle. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification.*

PciAddress

The address of the PCI device on the PCI bus. This address can be passed to the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL** member 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 device enumeration. Type **EFI_PCI_CONTROLLER_RESOURCE_ALLOCATION_PHASE** is defined in "Related Definitions" below.

Description

This function is called during the PCI enumeration process. No specific action is expected from this member function. It allows the host bridge driver to preinitialize individual PCI controllers before enumeration.

The parameter *RootBridgeHandle* can be used to locate the instance of the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL** that is installed on the root bridge that is the parent of the specific PCI function. The parameter *PciAddress* can be passed to the **Pci.Read()** and

Pci.Write() functions of the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL** instance to access the PCI configuration space of the specific PCI function.

This member function is invoked during PCI enumeration and before the PCI enumerator has created a handle for the PCI function. As a result, the **EFI_PCI_IO_PROTOCOL** cannot be used at this point.

Two notification points are defined at this time. See type **EFI_PCI_CONTROLLER_RESOURCE_ALLOCATION_PHASE** in "Related Definitions" below for definitions of these notification points and ISA Aliasing Considerations for usage. More synchronization points may be added as required in the future.

Related Definitions

```
//*******************************************************
// EFI_PCI_CONTROLLER_RESOURCE_ALLOCATION_PHASE
//*******************************************************
typedef enum {
  EfiPciBeforeChildBusEnumeration,
  EfiPciBeforeResourceCollection
} EFI_PCI_CONTROLLER_RESOURCE_ALLOCATION_PHASE;
```

Following is a description of the fields in the above enumeration:

EfiPciBeforeChildBusEnumeration	This notification is applicable only to PCI-to-PCI bridges and indicates that the PCI enumerator is about to begin enumerating the bus behind the PCI-to-PCI bridge. This notification is sent after the primary bus number, the secondary bus number, and the subordinate bus number registers in the PCI-to-PCI bridge are programmed to valid (but not necessary final) values. Programming of the bus number register allows the chipset code to scan devices on the bus that are immediately behind the PCI-to-PCI bridge. This notification can be used to reset the secondary PCI bus. Some PCI-to-PCI bridges can drive their secondary bus at various clock speeds (33 MHz or 66 MHz, for example) and support PCI-X* or conventional PCI mode. These bridges must be set up to operate at the correct speed and correct mode before the downstream devices and buses are enumerated. This notification can be used to perform that activity. The host bridge code cannot reprogram the bus numbers in the PCI-to-PCI bridge or reprogram any upstream devices during this notification. It can touch the downstream devices because the PCI enumerator has not found these devices. If there are multiple PCI-to-PCI bridges on the same PCI bus, the order in which the notification is sent to these bridges is implementation specific. On the other hand, it is guaranteed that a PCI-to-PCI bridge will see this notification before the downstream bridge receives this notification or its child devices receive the **EfiPciBeforeResourceCollection** notification.

continued

EfiPciBeforeResourceCollection	This notification is sent before the PCI enumerator probes the Base Address Register (BAR) registers for every valid PCI function. This notification can be used to program the backside registers that determine the BAR size or any other programming such as the master latency timer, cache line size, and PERR and SERR control. This notification is sent regardless of whether the function implements BAR or not. In the case of a multifunction device, this notification is sent for every function of the device. The order within the functions is not specified. The order in which this notification is sent to various devices/functions on the same bus is implementation specific.

Status Codes Returned

EFI_SUCCESS	The requested parameters were returned.
EFI_INVALID_PARAMETER	*RootBridgeHandle* is not a valid root bridge handle.
EFI_INVALID_PARAMETER	*Phase* is not a valid phase that is defined in **EFI_PCI_CONTROLLER_RESOURCE_ALLOCATION_PHASE**.
EFI_DEVICE_ERROR	Programming failed due to a hardware error. The PCI enumerator should not enumerate this device, including its child devices if it is a PCI-to-PCI bridge.