Intel® Platform Innovation Framework for EFI Hot-Plug PCI Initialization Protocol Specification

Draft for Review

Version 0.9

August 9, 2004

Revision History

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

Contents

1 Introduction

Overview

This specification defines the core code and services that are required for an implementation of the Hot-Plug PCI Initialization Protocol of the Intel® Platform Innovation Framework for EFI (hereafter referred to as the "Framework"). A PCI bus driver, running in the EFI Boot Services environment, uses this protocol to initialize the hot-plug subsystem. The same protocol may be used by other buses such as CardBus that support hot plugging. This specification does the following:

- Describes the basic components of the hot-plug PCI subsystem and the Hot-Plug PCI Initialization Protocol
- Provides code definitions for the Hot-Plug PCI Initialization Protocol and the hot-plug-PCI–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

2 Design Discussion

Hot-Plug PCI Initialization Protocol Introduction

This chapter describes the Hot-Plug PCI Initialization Protocol. A PCI bus driver, running in the EFI Boot Services environment, uses this protocol to initialize the hot-plug subsystem. This protocol is generic enough to include PCI-to-CardBus bridges and PCI Express* systems. This protocol abstracts the hot-plug controller initialization and resource padding. This protocol is required on platforms that support PCI Hot Plug* or PCI Express slots. For the purposes of initialization, a CardBus PC Card bus is treated in the same way. This protocol is not required on all other platforms.

This protocol is not intended to support hot plugging of PCI cards during the preboot stage. Separate components can be developed if such support is desired.

See Hot-Plug PCI Initialization Protocol in Code Definitions for the definition of **`EFI_PCI_HOT_PLUG_INIT_PROTOCOL`**.

Hot-Plug PCI Initialization Protocol Terms

The following terms are used throughout this document.

16-bit PC Card

Legacy cards that follow the *PC Card Standard* and operate in 16-bit mode.

CardBay PC Card

32-bit PC Cards that follow the high-performance serial *PC Card Standard.* After initialization, these devices appear as standard PCI devices.

CardBus bridge

A hardware controller that produces a CardBus bus. A CardBus bus can accept a CardBus PC Card as well as legacy 16-bit PC Cards. CardBus PC Cards appear just like PCI devices to the configuration software.

CardBus PC Card

32-bit PC Cards that follow the *PC Card Standard.*

HPB

Hot Plug Bus.

HPC

Hot Plug Controller. A generic term that refers to both a PHPC and a CardBus bridge.

HPRT

Hot Plug Resource Table.

JEITA

Japan Electronics and Information Technology Association.

legacy PHPC

PCI devices that can be identified by their class code but were defined prior to the *PCI Standard Hot-Plug Controller and Subsystem Specification*, revision 1.0. These devices have a base class of 0x6, subclass of 0x4, and programming interface of 0.

PC Card

Integrated circuit cards that follow the *PC Card Standard*. "PC Card" is a generic term that is used to refer to 16-bit PC Cards, 32-bit CardBus PC Cards, and high-performance CardBay PC Cards.

PC Card Standard

Refers to the set of specifications that are produced jointly by the PCMCIA and JEITA. This standard was defined to promote interchangeability among mobile computers.

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

A bus driver that initializes the PCI bus. As defined in the *EFI 1.10 Specification*, the PCI bus driver 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 as requested.

PCI configuration space

The configuration channel 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.

PCI controller

A hardware component that is discovered by a PCI bus driver and is managed by a PCI device driver. The terms "PCI function" and "PCI controller" are used equivalently in this document.

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 one another and all the "related" PCI root bridges must be considered together during resource allocation.

PCI root bridge

A PCI root bridge produces a root PCI bus. It bridges a root PCI bus and a bus that is not a PCI bus (for example, a processor local bus or 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; see Industry Specifications in the master Framework help system for the URL for the ACPI specification. 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.

PCI-to-CardBus bridges

A PCI device that produces a CardBus bus. The PCI-to-CardBus bridge has a type 2 PCI configuration header and has a class code of 0x070600.

PHPC

PCI Hot Plug* Controller. A hardware component that controls the power to one or more conventional PCI slots or PCI Express slots.

resource padding

Also known as resource overallocation. System resources are said to be overallocated if more resources are allocated to a PCI bus than are required. Resource padding allows a limited number of add-in cards to be hot added to a PCI bus without disturbing allocation to the rest of the buses.

root HPC

Root Hot Plug Controller. An HPC that gets reset only when the entire system is reset. Such HPCs can depend upon the system firmware to initialize them because system firmware is guaranteed to run after a system reset. An HPC that is embedded in the PCI root bridge is considered a root HPC bridge. Some platform chipsets include special-purpose PCI-to-PCI bridges. They appear like a PCI-to-PCI bridge to the configuration software, but their primary bus interface is not a PCI bus. Such a component can be considered a root HPC if it is not subordinate to an HPC. Legacy HPCs may be implemented as chipset devices that appear to be behind a special-purpose PCI-to-PCI bridge, but these HPCs are not reset when the secondary bus reset bit in the parent PCI-to-PCI bridge is set. Such HPCs are considered as root HPCs as well.

An HPC that is a child of a PCI-to-PCI bridge is not a root HPC. Such an HPC can be reset if the secondary bus reset bit in the PCI-to-PCI bridge is set by an operating system. Because the initialization code in the system firmware may not be executed during this case, such an HPC must initialize itself using hardware mechanisms, without any firmware intervention. An HPC that is a child of another HPC is not a root HPC. See section 3.5.1.3 in the *PCI Standard Hot-Plug Controller and Subsystem Specification*, revision 1.0, for details regarding this term.

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 ID of "PNP0A03." Typical desktop systems include only one root PCI bus.

SHPC

Standard (PCI) Hot Plug Controller. A PCI Hot Plug controller that conforms to the *PCI Standard Hot-Plug Controller and Subsystem Specification*, revision 1.0. This specification is published by the PCI Special Interest Group (PCI-SIG). An SHPC can either be embedded in a PCI root bridge or a PCI-to-PCI bridge.

Hot-Plug PCI Initialization Protocol Related Information

The following resources are referenced throughout this specification or may be useful to you:

- *Conventional PCI Specification*, revision 3.0: http://www.pcisig.com/*
- *PC Card Standard*, volumes 1, 7, and 8: http://www.pcmcia.org/*
- *PCI Express Base Specification*, revision 1.0a: http://www.pcisig.com/*
- *PCI Hot-Plug Specification*, revision 1.1: http://www.pcisig.com/*
- *PCI Standard Hot-Plug Controller and Subsystem Specification*, revision 1.0: http://www.pcisig.com/*

Requirements

Framework-based firmware must support platforms with PCI Hot Plug* slots and PCI Express* Hot Plug slots, as well as CardBus PC Card sockets. In both cases, the user is allowed to plug in new devices or remove existing devices during runtime. PCI Hot Plug slots are controlled by a PCI Hot Plug controller whereas CardBus sockets are controlled by a PCI-to-CardBus bridge. PCI Express Hot Plug slots are controlled by a PCI Express root port or a downstream port in a switch. The term "Hot Plug Controller" (HPC) in this document refers to all of these types of controllers. From the standpoint of initialization, all three are identical and have the same general requirements, as follows:

- The root HPCs may come up uninitialized after system reset. These HPCs must be initialized by the system firmware.
- Every HPC may require resource padding. The padding must be taken into account during PCI enumeration. This scenario is true for conventional PCI, PCI Express, and CardBus PC Cards because they all consume shared system resources (I/O, memory, and bus). These resources are produced by the root PCI bridge.

These general requirements place the following specific requirements on an implementation of the Framework:

- Framework-based firmware must handle root HPCs differently than other regular PCI devices. When a root HPC is initialized, the hot-plug slots or CardBus sockets are enabled and this process may uncover more PCI buses and devices. In that respect, root HPCs are somewhat like PCI bridges. The root HPC initialization process may involve detecting bus type and optimum bus speed. The initialization process may also detect faults and voltage mismatches. The initialization process may be specific to the controller and the platform. At the time of the root HPC initialization, the PCI bus may not be fully initialized and the standard PCI bus-specific protocols are not available. Framework-based firmware must provide an alternate infrastructure for the initialization code. In other words, the HPC initialization code should not be required to understand the bus numbering scheme and other chipset details.
- Framework-based firmware must support an unlimited number of HPCs in the system. Framework-based firmware must support various types of HPCs as long as they follow industry standards or conventions. A mix of various types of HPCs is allowed.
- Framework-based firmware must support legacy PCI Hot Plug Controllers (PHPCs; class code 0x6, subclass code 0x4) as well as Standard (PCI) Hot Plug Controllers (SHPCs). Other conventional PCI Hot Plug controllers are not supported.

- Framework-based firmware must be capable of supporting a PHPC that is a child of another PHPC. In that case, the *PCI Standard Hot-Plug Controller and Subsystem Specification* requires that the child PHPC must be initialized without firmware assistance because it is not a root PHPC.
- Framework-based firmware must be capable of supporting SHPCs on an add-in card. In that case, the *PCI Standard Hot-Plug Controller and Subsystem Specification* requires that such an SHPC must be initialized without firmware assistance because it is not a root PHPC. Framework-based firmware must also support plug-in CardBus bridges that follow the *CardBus Specification*, which is part of the *PC Card Standard*.
- As stated above, root HPCs may require firmware initialization. Framework-based firmware must be capable of supporting root HPCs that are initialized by hardware and do not require any firmware initialization.
- A Framework-based PCI bus enumerator must overallocate resources for PCI Hot Plug buses and CardBus sockets. The amount of overallocation may be platform specific.
- The root HPC initialization process may be time consuming. An SHPC can take as long as 15 seconds to enable power to a hot-plug bus without violating the PCI Special Interest Group (PCI-SIG*) requirements. Framework-based firmware should be able to initialize multiple HPCs in parallel to reduce boot time. In contrast, CardBus initialization is quick.
- Framework-based firmware should be able to handle when an HPC fails. Framework-based firmware should be able to handle an HPC that has been disabled.
- The PCI bus driver in Framework-based firmware is not required to assume anything that is not in one of the PCI-SIG specifications.
- The Framework should work in a PCI Express system. The *PCI Express Base Specification* defines a register interface that is different and simpler than the SHPC register interface. Hot-plug capability is built into 3GIO* switches, each of which appears as PCI-to-PCI bridges in the configuration space. 3GIO systems naturally contain HPCs behind HPCs.
- It must be possible to produce legacy Hot Plug Resource Tables (HPRTs) if necessary. HPRTs are described in the *PCI Standard Hot-Plug Controller and Subsystem Specification.*

Sample Implementation for a Platform Containing PCI Hot Plug* Slots

Typically, the PCI bus driver will enumerate and allocate resources to all devices for a PCI host bridge. A sample algorithm for PCI bus enumeration is described below to clarify some of the finer points of the **EFI_PCI_HOT_PLUG_INIT_PROTOCOL**. Actual implementations may vary although the relative ordering of events is critical. The activities related to PCI Hot Plug* are underlined. Please note that multiple passes of bus enumeration are required in a system containing PCI Hot Plug slots.

See the *Intel® Platform Innovation Framework for EFI PCI Host Bridge Resource Allocation Protocol Specification* for definitions of the **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL** and its member functions.

1. If the platform supports PCI Hot Plug, an instance of the **EFI_PCI_HOT_PLUG_INIT_PROTOCOL** is installed.
2. The PCI enumeration process begins.
3. Look for instances of the **EFI_PCI_HOT_PLUG_INIT_PROTOCOL**. If it is not found, all the hot-plug subsystem initialization steps can be skipped. If one exists, create a list of root Hot

Plug Controllers (HPCs) by calling `EFI_PCI_HOT_PLUG_INIT_PROTOCOL.GetRootHpcList()`.

4. Notify the host bridge driver that bus enumeration is about to begin by calling `EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL.NotifyPhase (EfiPciHostBridgeBeginBusAllocation)`.
5. For every PCI root bridge handle, do the following:
   a. Call `EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL.StartBusEnumeration (This,RootBridgeHandle)`.
   b. Make sure each PCI root bridge handle supports the `EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL`. See the *EFI 1.10 Specification* for the definition of the PCI Root Bridge I/O Protocol.
   c. Allocate memory to hold resource requirements. These can be two resource descriptors, one to hold bus requirements and another to hold the I/O and memory requirements.
   d. Call `EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL.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.
      1. Scan all the devices in the specified bus range and the specified segment, one bus at a time. If the device is a PCI-to-PCI bridge, update the bus numbers and program the bus number registers in the PCI-to-PCI bridge hardware. If the device path of a device matches that of a root HPC and it is not a PCI-to-CardBus bridge, it must be initialized by calling `EFI_PCI_HOT_PLUG_INIT_PROTOCOL.InitializeRootHpc()` before the bus it controls can be fully enumerated. The PCI bus enumerator determines the PCI address of the PCI Hot Plug Controller (PHPC) and passes it as an input to `InitializeRootHpc()`.
   e. Continue to scan devices on that root bridge and start the initialization of all root HPCs.
   f. Call `EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL.SetBusNumbers()` so that the HPCs under initialization are still accessible. `SetBusNumbers()` cannot affect the PCI addresses of the HPCs.
6. Wait until all the HPCs that were found on various root bridges in step 5 to complete initialization.
7. Go back to step 5 for another pass and rescan the PCI buses. For all the root HPCs and the nonroot HPCs, call `EFI_PCI_HOT_PLUG_INIT_PROTOCOL.GetResourcePadding()` to obtain the amount of overallocation and add that amount to the requests from the physical devices. Reprogram the bus numbers by taking into account the bus resource padding information. This action will require calling `EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL.SetBusNumbers()`. The rescan is not required if there is only one root bridge in the system.

....

Once the memory resources are allocated and a PCI-to-CardBus bridge is part of the `HpcList`, it will be initialized.

3 Code Definitions

Introduction

This section contains the basic definitions that are related to PCI Hot Plug*. The following protocol is defined in this section:

- **EFI_PCI_HOT_PLUG_INIT_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_HPC_LOCATION**
- **EFI_HPC_STATE**
- **EFI_HPC_PADDING_ATTRIBUTES**

Hot-Plug PCI Initialization Protocol

EFI_PCI_HOT_PLUG_INIT_PROTOCOL

Summary

This protocol provides the necessary functionality to initialize the Hot Plug Controllers (HPCs) and the buses that they control. This protocol also provides information regarding resource padding.

NOTE

This protocol is required only on platforms that support one or more PCI Hot Plug slots or CardBus sockets.*

GUID

```
#define EFI_PCI_HOT_PLUG_INIT_PROTOCOL_GUID \
  { 0xaa0e8bc1, 0xdabc, 0x46b0, 0xa8, 0x44, 0x37, 0xb8, 0x16,
0x9b, 0x2b, 0xea }
```

Protocol Interface Structure

```
typedef struct _EFI_PCI_HOT_PLUG_INIT_PROTOCOL {
  EFI_GET_ROOT_HPC_LIST                  GetRootHpcList;
  EFI_INITIALIZE_ROOT_HPC                InitializeRootHpc;
  EFI_GET_HOT_PLUG_PADDING               GetResourcePadding;
} EFI_PCI_HOT_PLUG_INIT_PROTOCOL;
```

Parameters

`GetRootHpcList`

Returns a list of root HPCs and the buses that they control. See the **`GetRootHpcList()`** function description.

`InitializeRootHpc`

Initializes the specified root HPC. See the **`InitializeRootHpc()`** function description.

`GetResourcePadding`

Returns the resource padding that is required by the HPC. See the **`GetResourcePadding()`** function description.

Description

The **`EFI_PCI_HOT_PLUG_INIT_PROTOCOL`** provides a mechanism for the PCI bus enumerator to properly initialize the HPCs and CardBus sockets that require initialization. The HPC initialization takes place before the PCI enumeration process is complete. There cannot be more than one instance of this protocol in a system. This protocol is installed on its own separate handle.

Because the system may include multiple HPCs, one instance of this protocol should represent all of them. The protocol functions use the device path of the HPC to identify the HPC. When the PCI bus enumerator finds a root HPC, it will call **`EFI_PCI_HOT_PLUG_INIT_PROTOCOL.InitializeRootHpc()`**. If **`InitializeRootHpc()`** is unable to initialize a root HPC, the PCI enumerator will ignore that root HPC and continue the enumeration process. If the HPC is not initialized, the devices that it controls may not be initialized, and no resource padding will be provided.

From the standpoint of the PCI bus enumerator, HPCs are divided into the following two classes:

- **Root HPC:** See Hot-Plug PCI Initialization Protocol Terms for the definition. These HPCs must be initialized by calling **`InitializeRootHpc()`** during the enumeration process. These HPCs will also require resource padding. The platform code must have *a priori* knowledge of these devices and must know how to initialize them. There may not be any way to access their PCI configuration space before the PCI enumerator programs all the upstream bridges and thus enables the path to these devices. The PCI bus enumerator is responsible for determining the PCI bus address of the HPC before it calls **`InitializeRootHpc()`**.
- **Nonroot HPC:** See Hot-Plug PCI Initialization Protocol Terms for the definition. These HPCs will not need explicit initialization during enumeration process. These HPCs will require resource padding. The platform code does not have to have *a priori* knowledge of these devices.

EFI_PCI_HOT_PLUG_INIT_PROTOCOL.GetRootHpcList()

Summary

Returns a list of root Hot Plug Controllers (HPCs) that require initialization during the boot process.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_GET_ROOT_HPC_LIST) (
  IN EFI_PCI_HOT_PLUG_INIT_PROTOCOL     *This,
  OUT UINTN                             *HpcCount,
  OUT EFI_HPC_LOCATION                  **HpcList
);
```

Parameters

This

Pointer to the **EFI_PCI_HOT_PLUG_INIT_PROTOCOL** instance.

HpcCount

The number of root HPCs that were returned.

HpcList

The list of root HPCs. *HpcCount* defines the number of elements in this list. Type **EFI_HPC_LOCATION** is defined in "Related Definitions" below.

Description

This procedure returns a list of root HPCs. The PCI bus driver must initialize these controllers during the boot process. The PCI bus driver may or may not be able to detect these HPCs. If the platform includes a PCI-to-CardBus bridge, it can be included in this list if it requires initialization. The *HpcList* must be self consistent. An HPC cannot control any of its parent buses. Only one HPC can control a PCI bus. Because this list includes only root HPCs, no HPC in the list can be a child of another HPC. This policy must be enforced by the **EFI_PCI_HOT_PLUG_INIT_PROTOCOL**. The PCI bus driver may not check for such invalid conditions.

The callee allocates the buffer *HpcList*.

Related Definitions

```
//*******************************************************
// EFI_HPC_LOCATION
//*******************************************************
typedef struct {
  EFI_DEVICE_PATH_PROTOCOL   *HpcDevicePath;
  EFI_DEVICE_PATH_PROTOCOL   *HpbDevicePath;
} EFI_HPC_LOCATION;
```

HpcDevicePath

The device path to the root HPC. An HPC cannot control its parent buses. The PCI bus driver requires this information so that it can pass the correct *HpcPciAddress* to the **InitializeRootHpc()** and **GetResourcePadding()** functions. Type **EFI_DEVICE_PATH** is defined in **LocateDevicePath()** in the *EFI 1.10 Specification.*

HpbDevicePath

The device path to the Hot Plug Bus (HPB) that is controlled by the root HPC. The PCI bus driver uses this information to check if a particular PCI bus has hot-plug slots. The device path of a PCI bus is the same as the device path of its parent. For Standard (PCI) Hot Plug Controllers (SHPCs) and PCI Express*, *HpbDevicePath* is the same as *HpcDevicePath*.

Status Codes Returned

EFI_SUCCESS	*HpcList* was returned.
EFI_OUT_OF_RESOURCES	*HpcList* was not returned due to insufficient resources.
EFI_INVALID_PARAMETER	*HpcCount* is **NULL**.
EFI_INVALID_PARAMETER	*HpcList* is **NULL**.

EFI_PCI_HOT_PLUG_INIT_PROTOCOL.InitializeRootHpc()

Summary

Initializes one root Hot Plug Controller (HPC). This process may causes initialization of its subordinate buses.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_INITIALIZE_ROOT_HPC) (
  IN  EFI_PCI_HOT_PLUG_INIT_PROTOCOL    *This,
  IN  EFI_DEVICE_PATH_PROTOCOL          *HpcDevicePath,
  IN  UINT64                            HpcPciAddress,
  IN  EFI_EVENT                         Event, OPTIONAL
  OUT EFI_HPC_STATE                     *HpcState
);
```

Parameters

This

Pointer to the **EFI_PCI_HOT_PLUG_INIT_PROTOCOL** instance.

HpcDevicePath

The device path to the HPC that is being initialized. Type **EFI_DEVICE_PATH** is defined in **LocateDevicePath()** in the *EFI 1.10 Specification.*

HpcPciAddress

The address of the HPC function on the PCI bus.

Event

The event that should be signaled when the HPC initialization is complete. Set to **NULL** if the caller wants to wait until the entire initialization process is complete. The event must be of type **EFI_EVENT_NOTIFY_SIGNAL**. Type **EFI_EVENT** is defined in **CreateEvent()** in the *Intel® Platform Innovation Framework for EFI Driver Execution Environment Core Interface Specification* (DXE CIS).

HpcState

The state of the HPC hardware. The type **EFI_HPC_STATE** is defined in "Related Definitions" below.

Description

This function initializes the specified HPC. At the end of initialization, the hot-plug slots or sockets (controlled by this HPC) are powered and are connected to the bus. All the necessary registers in the HPC are set up. For a Standard (PCI) Hot Plug Controller (SHPC), the registers that must be set up are defined in the *PCI Standard Hot Plug Controller and Subsystem Specification*. For others HPCs, they are specific to the HPC hardware. The initialization process may choose not to enable certain PCI Hot Plug* slots or sockets for any reason. The PCI Hot Plug slots or CardBus sockets that are left disabled at this stage are not available to the system. A PCI slot may be disabled due to a power fault, PCI bus type mismatch, or power budget constraints. The HPC initialization process can be time consuming. Powering up the slots that are controlled by SHPCs can take up to 15 seconds. In a system with multiple HPCs, it is desirable to perform these activities in parallel. Therefore, this procedure supports nonblocking execution mode.

If **`InitializeRootHpc()`** is called with a non-**`NULL`** event, HPC initialization is considered complete after the event is signaled. If **`InitializeRootHpc()`** is called with a non-**`NULL`** event, a return from **`InitializeRootHpc()`** with **`EFI_SUCCESS`** marks the completion of the HPC initialization.

The PCI bus enumerator will call this function for every root HPC that is returned by **`GetRootHpcList()`**.

The PCI bus enumerator must make sure that the registers that are required during HPC initialization are accessible before calling **`InitializeRootHpc()`**. The determination of whether the registers are accessible is based on the following rules:

- For HPCs (legacy HPCs, SHPCs inside a PCI-to-PCI bridge, and PCI Express* HPCs), the PCI configuration space of the HPC device must be accessible. In other words, all the upstream bridges including root bridges and special-purpose PCI-to-PCI bridges are programmed to forward PCI configuration cycles to the HPC.
- SHPCs inside a root bridge are accessible without any initialization of the PCI bus.
- PCI-to-CardBus bridges have their registers mapped into the memory space using a memory Base Address Register (BAR).

This function takes the device path of the HPC as an input. At the time of HPC initialization, the PCI bus enumeration is not complete. The PCI bus enumerator may not have created a handle for the HPC and the hot-plug initialization code cannot use the **`EFI_PCI_IO_PROTOCOL`** or **`EFI_DEVICE_PATH_PROTOCOL`** like other PCI device drivers. The device path uniquely identifies the HPC and also the PCI bus that it controls.

If the HPC is a PCI device, the hot-plug initialization code may need its address on the PCI bus (**`EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_PCI_ADDRESS`**; see Table 12-1 in the *EFI 1.10 Specification* for its definition) to access its registers. The PCI address of a regular PCI device is dynamic but is known to the PCI bus driver. Therefore, the PCI bus driver provides it through the input parameter *`HpcPciAddress`* to this function. Passing this address eliminates the need for **`InitializeRootHpc()`** to convert the device path into the PCI address. If the HPC is a function in a multifunction device, this address is the PCI address of that function. The HPC's configuration space must be accessible at the specified *`HpcPciAddress`* until the HPC initialization is complete. In other words, the PCI bus driver cannot renumber PCI buses that are upstream to the HPC while it is being initialized.

This member function can use the **`LocateDevicePath()`** function to locate the appropriate instance of the **`EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL`**.

If the *`Event`* is not **`NULL`**, this function will return control to the caller without completing the entire initialization. This function must perform some basic checks to make sure that it knows how to initialize the specified HPC before returning control. The *`Event`* is signaled when the initialization process completes, regardless of whether it results in a failure. The caller must check *`HpcState`* to get the initialization status after the event is signaled.

If *Event* is not **NULL**, it is possible that the *Event* may be signaled before this function returns. There are at least two cases where that may happen:

- A simple implementation of **EFI_PCI_HOT_PLUG_INIT_PROTOCOL** may force the caller to wait until the initialization is complete. In that case, the **InitializeRootHpc()** function may signal the event before it returns control back to the caller.
- The HPC may already have been initialized by the time **InitializeRootHpc()** is called. In that case, **InitializeRootHpc()** will signal *Event* and return control back to the caller.

HpcState returns the state of the HPC at the time when control returns. If *Event* is **NULL**, *HpcState* must indicate that the HPC has completed initialization. If *Event* is not **NULL**, *HpcState* can indicate that the HPC has not completed initialization when this function returns, but *HpcState* must be updated before *Event* is signaled.

The firmware may not wait until **InitializeRootHpc()** to start HPC initialization. The firmware may start the initialization earlier in the boot process and the initialization may be completely done by the time the PCI bus enumerator calls **InitializeRootHpc()**. An HPC can be initialized by hardware alone, and no firmware initialization may be needed. For such HPCs, this member function does not have to do any real work. In such cases, **InitializeRootHpc()** merely acts as a synchronization point.

Related Definitions

```
//*******************************************************
// EFI_HPC_STATE
//*******************************************************
// Describes current state of an HPC

typedef UINT16 EFI_HPC_STATE;

#define  EFI_HPC_STATE_INITIALIZED    0x01
#define  EFI_HPC_STATE_ENABLED        0x02
```

Following is a description of the possible states for **EFI_HPC_STATE**.

0	Not initialized.
EFI_HPC_STATE_INITIALIZED	The HPC initialization function was called and the HPC completed initialization, but it was not enabled for some reason. The HPC may be disabled in hardware, or it may be disabled due to user preferences, hardware failure, or other reasons. No resource padding is required.
EFI_HPC_STATE_INITIALIZED | EFI_HPC_ENABLED	The HPC initialization function was called, the HPC completed initialization, and it was enabled. Resource padding is required.

Status Codes Returned

EFI_SUCCESS	If *Event* is **NULL**, the specific HPC was successfully initialized. If *Event* is not **NULL**, *Event* will be signaled at a later time when initialization is complete.
EFI_UNSUPPORTED	This instance of **EFI_PCI_HOT_PLUG_INIT_PROTOCOL** does not support the specified HPC. If *Event* is not **NULL**, it will not be signaled.
EFI_OUT_OF_RESOURCES	Initialization failed due to insufficient resources. If *Event* is not **NULL**, it will not be signaled.
EFI_INVALID_PARAMETER	*HpcState* is **NULL**.

EFI_PCI_HOT_PLUG_INIT_PROTOCOL.GetResourcePadding()

Summary

Returns the resource padding that is required by the PCI bus that is controlled by the specified Hot Plug Controller (HPC).

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_GET_HOT_PLUG_PADDING) (
  IN  EFI_PCI_HOT_PLUG_INIT_PROTOCOL     *This,
  IN  EFI_DEVICE_PATH_PROTOCOL           *HpcDevicePath,
  IN  UINT64                             HpcPciAddress,
  OUT EFI_HPC_STATE                      *HpcState,
  OUT VOID                               **Padding,
  OUT EFI_HPC_PADDING_ATTRIBUTES         *Attributes
);
```

Parameters

This

Pointer to the **EFI_PCI_HOT_PLUG_INIT_PROTOCOL** instance.

HpcDevicePath

The device path to the HPC. Type **EFI_DEVICE_PATH** is defined in **LocateDevicePath()** in the *EFI 1.10 Specification.*

HpcPciAddress

The address of the HPC function on the PCI bus.

HpcState

The state of the HPC hardware. Type **EFI_HPC_STATE** is defined in **EFI_PCI_HOT_PLUG_INIT_PROTOCOL.InitializeRootHpc()**.

Padding

The amount of resource padding that is required by the PCI bus under the control of the specified HPC. Because the caller does not know the size of this buffer, this buffer is allocated by the callee and freed by the caller.

Attributes

Describes how padding is accounted for. The padding is returned in the form of ACPI 2.0 resource descriptors. The exact definition of each of the fields is the same as in **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL.SubmitResources()** in the *Intel® Platform Innovation Framework for EFI PCI Host Bridge Resource Allocation Protocol Specification*. Type **EFI_HPC_PADDING_ATTRIBUTES** is defined in "Related Definitions" below.

Description

This function returns the resource padding that is required by the PCI bus that is controlled by the specified HPC. This member function is called for all the root HPCs and nonroot HPCs that are detected by the PCI bus enumerator. This function will be called before PCI resource allocation is completed. This function must be called after all the root HPCs, with the possible exception of a PCI-to-CardBus bridge, have completed initialization. Waiting until initialization is completed allows the HPC driver to optimize the padding requirement. The calculation may take into account the number of empty and/or populated PCI Hot Plug* slots, the number of PCI-to-PCI bridges among the populated slots, and other factors. This information is available only after initialization is complete. PCI-to-CardBus bridges require memory resources before the initialization is started and therefore are considered an exception. The padding requirements are relatively constant for PCI-to-CardBus bridges and an estimated value must be returned.

If **`InitializeRootHpc()`** is called with a non-**`NULL`** event, HPC initialization is considered complete after the event is signaled. If **`InitializeRootHpc()`** is called with a non-**`NULL`** event, a return from **`InitializeRootHpc()`** with **`EFI_SUCCESS`** marks the completion of HPC initialization.

The input parameters *`HpcDevicePath`*, *`HpcPciAddress`*, and *`HpcState`* are described in **`EFI_PCI_HOT_PLUG_INIT_PROTOCOL.InitializeRootHpc()`**. The value of *`HpcPciAddress`* for the same root HPC may be different from what was passed to **`InitializeRootHpc()`**. The HPC's configuration space must be accessible at the specified *`HpcPciAddress`* until this function returns control.

The padding is returned in the form of ACPI 2.0 resource descriptors. The exact definition of each of the fields is the same as in the **`EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL.SubmitResources()`** function. See the *Intel® Platform Innovation Framework for EFI PCI Host Bridge Resource Allocation Protocol Specification* for the definition of this function.

The PCI bus driver is responsible for adding this resource request to the resource requests by the physical PCI devices. If *`Attributes`* is **`EfiPaddingPciBus`**, the padding takes effect at the PCI bus level. If *`Attributes`* is **`EfiPaddingPciRootBridge`**, the required padding takes effect at the root bridge level. For details, see the definition of **`EFI_HPC_PADDING_ATTRIBUTES`** in "Related Definitions" below.

Note that the padding request cannot ask for specific legacy resources such as COM port addresses. Legacy PC Card devices may require such resources. Supporting these resource requirements is outside the scope of this specification.

Related Definitions

```
//*******************************************************
// EFI_HPC_PADDING_ATTRIBUTES
//*******************************************************
// Describes how resource padding should be applied

typedef enum {
  EfiPaddingPciBus,
  EfiPaddingPciRootBridge
} EFI_HPC_PADDING_ATTRIBUTES;
```

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

EfiPaddingPciBus	Apply the padding at a PCI bus level. In other words, the resources that are allocated to the bus containing hot-plug slots are padded by the specified amount. If the hot-plug bus is behind a PCI-to-PCI bridge, the PCI-to-PCI bridge apertures will indicate the padding.
EfiPaddingPciRootBridge	Apply the padding at a PCI root bridge level. If a PCI root bridge includes more than one hot-plug bus, the resource padding requests for these buses are added together and the resources that are allocated to the root bridge are padded by the specified amount. This strategy may reduce the total amount of padding, but requires reprogramming of PCI-to-PCI bridges in a hot-add event. If the hot-plug bus is behind a PCI-to-PCI bridge, the PCI-to-PCI bridge apertures do not indicate the padding for that bus.

Status Codes Returned

EFI_SUCCESS	The resource padding was successfully returned.
EFI_UNSUPPORTED	This instance of the **EFI_PCI_HOT_PLUG_INIT_PROTOCOL** does not support the specified HPC.
EFI_NOT_READY	This function was called before HPC initialization is complete.
EFI_INVALID_PARAMETER	*HpcState* is **NULL**.
EFI_INVALID_PARAMETER	*Padding* is **NULL**.
EFI_INVALID_PARAMETER	*Attributes* is **NULL**.
EFI_OUT_OF_RESOURCES	ACPI 2.0 resource descriptors for *Padding* cannot be allocated due to insufficient resources.