



## PCI-SIG ENGINEERING CHANGE REQUEST

<b>TITLE:</b>	UEFI PCI Services Update
<b>DATE:</b>	8/4/09
<b>AFFECTED DOCUMENT:</b>	PCI Firmware Specification Ver 3.0
<b>SPONSOR:</b>	Dong Wei, HP

### Part I

#### 1. **1. Summary of the Functional Changes**

This is a request to update the UEFI PCI Services. No functional changes. In the case of the UGA reference, UGA has been obsolete by the UEFI Specification and is replaced by the new GOP.

Formatted: Bullets and Numbering

#### 2. **2. Benefits as a Result of the Changes**

Provide an update to refer to the current state of the UEFI specification to replace the reference to EFI Specification which is now obsolete.

Formatted: Bullets and Numbering

#### 3. **3. Assessment of the Impact**

Low. UGA was not widely implemented. GOP is expected by the UEFI compliant OSes.

Formatted: Bullets and Numbering

#### 4. **4. Analysis of the Hardware Implications**

None

Formatted: Bullets and Numbering

#### 5. **5. Analysis of the Software Implications**

Low, UGA was not widely implemented. GOP is expected by the UEFI compliant OSes.

Formatted: Bullets and Numbering

## Part II

### Detailed Description of the change

Update Chapter 3 with the following changes to reflect the lated state of UEFI:

#### 3. UEFI PCI Services

UEFI stands for Unified Extensible Firmware Interface. The UEFI Specification, Version 2.3 or later (<http://www.uefi.org>) describes an interface between the operating system and the platform firmware. The interface is in the form of data tables that contain platform-related information and boot and run-time services calls that are available to the operating system and its loader. Together, these provide a standard environment for booting an operating system.

**Deleted:** 1.10

**Deleted:** developer.intel.com/technology/efi/

**Deleted:** EFI is required on DIG64-compliant systems.

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

#### 3.1. UEFI Driver Model

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

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

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

##### 3.1.1. PCI Root Bridge Protocol

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

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

PCI Root Bridge Protocol abstracts device specific code from the system memory map. This allows system designers to make changes to the system memory map without impacting platform independent code that is consuming basic system resources. An example of such system memory map changes is a system that provides non-identity memory mapped I/O (MMIO) mapping between the host processor view and the PCI device view.

# Request Request Request Request Request Request Request Request

## 3.1.2. PCI Driver Model

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

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

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

## 3.2. PCI-X Mode 2 and PCI Express

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

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

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

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

## 3.3. EFI Byte Code

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

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

## 3.4. Graphics Output Protocol

Graphics Output Protocol (GOP) is defined in the UEFI Specification to remove the hardware requirement to support legacy VGA and INT 10h BIOS. GOP provides a software abstraction to draw on video screen.

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

Note: VGA hardware can support GOP.

## 3.5. Device State at Firmware/Operating System Handoff

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

**Deleted:** Universal Graphics Adapter

**Deleted:** Universal Graphics Adapter

**Deleted:** UGA

**Deleted:** UGA

**Deleted:** also

**Deleted:** with EFI UGA Draw Protocol and UGA I/O Protocol

**Deleted:** UGA ROM uses the EBC format and the UGA driver follows the EFI Driver Model.¶ For EFI UGA Draw and I/O Protocol, refer to EFI Specification.

**Deleted:** EFI UGA Protocols

## Request Request Request Request Request Request Request Request

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

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

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

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

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

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

The operating system is required to configure PCI subsystems:

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

## Request Request Request Request Request Request Request Request Request

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

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

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

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

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

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

## Request Request Request Request Request Request Request Request Request

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

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

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

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

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

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

The operating system must not assume that all devices have been configured. Per Section 2.5.6 of UEFI (rev 2.3): the presence of an UEFI driver in the system firmware or in an option ROM does not guarantee that the UEFI driver will be loaded, executed, or allowed to manage any devices in a platform. In addition, UEFI drivers are not involved during PCI hot plug.

Deleted: 1.10

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

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

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

---

<sup>1</sup> For example, does not apply to PCI Express.

## **Request Request Request Request Request Request Request Request**

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

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