Extensible Firmware Interface Specification, Version 1.10

Changes from Review Draft Version 0.95

Revision 1.0

December 1, 2002

Intel order number: xxxxxx-001

History

Draft Revision	Draft Revision History	Date
0.95 Errata	Initial draft of errata	10/4/02
0.95 Errata with feedback	Added industry feedback	11/1/02
1.0	Updated with the final changes to the 1.0 revision of the EFI 1.10 Specification. Updated page numbers to match those in the 1.0 revision, not draft 0.95. Updated the title from “Errata” to “Changes from Review Draft Version 0.95.”	12/1/01

Contents

Introduction

This document contains the additions, changes, clarifications, and corrections that were made for the official 1.0 revision of the *Extensible Firmware Interface Specification,* version 1.10 (hereafter referred to as the "EFI 1.10 Specification"). This document includes any changes that were made since draft 0.95 of the EFI 1.10 Specification and reflects the industry feedback given on the *Errata to EFI 1.10 Specification (Draft 0.95)*.

This document is broken into the following subsections:

- Global changes: Lists the general changes made to the specification as a whole.
- Changes by section: Lists the specific changes made in each section. Changes are broken into the following subsections:
  - — Additions and changes
  - — Clarifications and corrections

  In each subsection, changes are listed in order by page number. If a section is not listed, then no changes were made since draft 0.95 of the EFI 1.10 Specification.

This document is intended to be used along with the EFI 1.10 Specification, revision 1.0, that is available from the EFI Web Site at:
http://developer.intel.com/technology/efi.

The page numbers indicated in this document refer to the page numbers in the 1.0 revision of the EFI 1.10 Specification, not draft 0.95.

Global Changes

This section lists the global changes that were made in the EFI 1.10 Specification for the revision 1.0 release. These changes are not technical in nature.

Typographic Conventions

Two additional typographic conventions were added:

- Plain text (blue)
- **BOLD Monospace (underlined)**

Any plain text underlined and in blue indicates an active link to the cross-reference.

Words in a **BOLD Monospace** typeface that is underlined and in a dark red color indicate an active hyperlink to the definition for that function or type definition.

Active Hyperlinks

Due to management and file size considerations, the number of active links to other locations in the EFI Specification was reduced in the electronic version of the document.

Now, only the first occurrence of the reference on each page is an active link. Subsequent references on the same page are not actively linked to the definition and use the standard, nonunderlined typeface.

As indicated in the "Typographic Conventions" section above, active hyperlinks have the following appearance:

- Links to general text: Plain text (blue)
- Links to function or type definitions: **BOLD Monospace (underlined)**

All hyperlinks to Web pages remain active links and were not affected.

Section 2: Overview

Additions and Changes

Section 2.3.2: IA-32 Platforms (Page 2-9 and 2-10)

For the bulleted list that starts with, “For an operating system to use any EFI runtime services, it must:,” added the following bullets at the end of the list:

- ACPI Tables loaded at boot time must be contained in memory of type **`EfiACPIReclaimMemory`**.
- The system firmware must not request a virtual mapping for any memory descriptor of type **`EfiACPIReclaimMemory`** or **`EfiACPIMemoryNVS`**.
- EFI memory descriptors of type **`EfiACPIReclaimMemory`** and **`EfiACPIMemoryNVS`** must be aligned on a 4 KB boundary and must be a multiple of 4 KB in size.
- Any EFI memory descriptor that requests a virtual mapping via the **`EFI_MEMORY_DESCRIPTOR`** having the **`EFI_MEMORY_RUNTIME`** bit set must be aligned on a 4 KB boundary and must be a multiple of 4 KB in size.
- An ACPI Memory Op-region must inherit cacheability attributes from the EFI memory map. If the system memory map does not contain cacheability attributes, the ACPI Memory Op-region must inherit its cacheability attributes from the ACPI name space. If no cacheability attributes exist in the system memory map or the ACPI name space, then the region must be assumed to be noncacheable.
- ACPI tables loaded at runtime must be contained in memory of type **`EfiACPIMemoryNVS`** or **`EfiFirmareReserved`**. The cacheability attributes for ACPI tables loaded at runtime (via ACPI LoadTable) should be defined in the EFI memory map. If no information about the table location exists in the EFI memory map the table is assumed to be non-cached.

Section 2.3.3: Itanium®-Based Platforms (Page 2-11)

After the sentence, "If **SetVirtualAddressMap()** has been used, then runtime service calls are made in virtual mode," added the following bulleted list:

- ACPI Tables loaded at boot time must be contained in memory of type **EfiACPIReclaimMemory**.
- The system firmware must not request a virtual mapping for any memory descriptor of type **EfiACPIReclaimMemory** or **EfiACPIMemoryNVS**.
- EFI memory descriptors of type **EfiACPIReclaimMemory** and **EfiACPIMemoryNVS** must be aligned on an 8 KB boundary and must be a multiple of 8 KB in size.
- Any EFI memory descriptor that requests a virtual mapping via the **EFI_MEMORY_DESCRIPTOR** having the **EFI_MEMORY_RUNTIME** bit set must be aligned on a 8 KB boundary and must be a multiple of 8 KB in size.
- An ACPI Memory Op-region must inherit cacheability attributes from the EFI memory map. If the system memory map does not contain cacheability attributes the ACPI Memory Op-region must inherit its cacheability attributes from the ACPI name space. If no cacheability attributes exist in the system memory map or the ACPI name space, then the region must be assumed to be noncacheable.
- ACPI tables loaded at runtime must be contained in memory of type **EfiACPIMemoryNVS** or **EfiFirmareReserved**. The cacheability attributes for ACPI tables loaded at runtime (via ACPI LoadTable) should be defined in the EFI memory map. If no information about the table location exists in the EFI memory map the table is assumed to be non-cached.

Clarifications and Corrections

Section 2.5.3: Host Bus Controllers – Figure 2-10 (Page 2-22)

In the figure, changed "PCI_ROOT_BRIDGE_PROTOCOL" to "PCI_ROOT_BRIDGE_IO_PROTOCOL."

Section 2.5.6: Platform Components (Page 2-26)

Added the following paragraph between the first and second paragraph of this section:

> Since the platform firmware may choose to only connect the devices required to produce consoles and gain access to a boot device, the OS present device drivers cannot assume that an EFI driver for a device has been executed. The presence of an EFI driver in the system firmware or in an option ROM does not guarantee that the EFI driver will be loaded, executed, or allowed to manage any devices in a platform. All OS present device drivers must be able to handle devices that have been managed by an EFI driver and devices that have not been managed by an EFI driver.

Section 4: EFI System Table

Additions and Changes

Section 4.4: EFI Boot Services Table – Related Definitions (Page 4-7)

Changed the declaration of the field *PCHandleProtocol* FROM:

```
EFI_HANDLE_PROTOCOL                 PCHandleProtocol;
```

TO:

```
VOID                                *Reserved;
```

Section 4.4: EFI Boot Services Table – Parameters (Page 4-9)

Changed the definition of the field *PCHandleProtocol* FROM:

PCHandleProtocol	Reserved. Must be **NULL**.

TO:

Reserved	Reserved. Must be **NULL**.

Clarifications and Corrections

None.

Section 5: Services – Boot Services

Additions and Changes

Section 5.2: Memory Allocation Services (Page 5-18)

Added the following sentence to the end of the last paragraph on page 5-18:

> The system firmware must follow the processor-specific rules outlined in sections 2.3.2 and 2.3.3 in the layout of the EFI memory map to enable the OS to make the required virtual mappings.

Section 5.2: AllocatePages() – Parameters – MemoryType (Page 5-21)

Changed the description of the *MemoryType* parameter FROM:

> The type of memory to allocate. The only types allowed are **EfiLoaderCode, EfiLoaderData, EfiRuntimeServicesCode, EfiRuntimeServicesData, EfiBootServicesCode, EfiBootServicesData, EfiACPIReclaimMemory, EfiACPIMemoryNVS**, and **EfiReservedMemoryType**. Normal allocations (that is, allocations by any EFI application) are of type **EfiLoaderData**. See "Related Definitions", Table5-5, and Table 5-6.

TO:

> The type of memory to allocate. The type **EFI_MEMORY_TYPE** is defined in "Related Definitions" below. These memory types are also described in more detail in Table 5-5 and Table 5-6. Normal allocations (that is, allocations by any EFI application) are of type **EfiLoaderData**. *MemoryType* values in the range 0x80000000..0xFFFFFFFF are reserved for use by EFI OS loaders that are provided by operating system vendors. The only illegal memory type values are those in the range **EfiMaxMemoryType**..0x7FFFFFFF.

Section 5.2: AllocatePages() – Status Codes Returned (Page 5-23)

Changed the "Status Codes Returned" table FROM:

EFI_SUCCESS	The requested pages were allocated.
EFI_OUT_OF_RESOURCES	The pages could not be allocated.
EFI_INVALID_PARAMETER	One of the parameters has an invalid value.
EFI_NOT_FOUND	The requested pages could not be found.

TO:

EFI_SUCCESS	The requested pages were allocated.
EFI_OUT_OF_RESOURCES	The pages could not be allocated.
EFI_INVALID_PARAMETER	*Type* is not **AllocateAnyPages** or **AllocateMaxAddress** or **AllocateAddress**.
EFI_INVALID_PARAMETER	*MemoryType* is in the range **EfiMaxMemoryType**..0x7FFFFFFF.
EFI_NOT_FOUND	The requested pages could not be found.

Section 5.2: AllocatePool() – Parameters – PoolType (Page 5-29)

Changed the description of the parameter *PoolType* FROM:

> The type of pool to allocate. The only supported types are **EfiLoaderData, EfiBootServicesData, EfiRuntimeServicesData, EfiACPIReclaimMemory,** and **EfiACPIMemoryNVS**. Type **EFI_MEMORY_TYPE** is defined in the **AllocatePages()** function description.

TO:

> The type of pool to allocate. Type **EFI_MEMORY_TYPE** is defined in the **AllocatePages()** function description. *PoolType* values in the range 0x80000000..0xFFFFFFFF are reserved for use by EFI OS loaders that are provided by operating system vendors. The only illegal memory type values are those in the range **EfiMaxMemoryType**..0x7FFFFFFF.

Section 5.3.1: InstallProtocolInterface() – Parameters – InterfaceType (Page 5-36)

Changed the description of the *InterfaceType* parameter FROM:

InterfaceType	Indicates whether *Interface* is supplied in native or p-code form. This value indicates the original execution environment of the request. See "Related Definitions."

TO:

InterfaceType	Indicates whether *Interface* is supplied in native form. This value indicates the original execution environment of the request. See "Related Definitions."

Section 5.3.1: InstallProtocolInterface() – Related Definitions (Page 5-37)

Changed the definition of **EFI_INTERFACE_TYPE** FROM:

```
//*******************************************************
//EFI_INTERFACE_TYPE
//*******************************************************
typedef enum {
     EFI_NATIVE_INTERFACE,
     EFI_PCODE_INTERFACE
} EFI INTERFACE_TYPE;
```

TO:

```
//*******************************************************
//EFI_INTERFACE_TYPE
//*******************************************************
typedef enum {
     EFI_NATIVE_INTERFACE
} EFI INTERFACE_TYPE;
```

Section 5.4: StartImage() (Page 5-80)

Added the following section, "EFI 1.10 Extension," between the "Description" and "Status Codes Returned" sections:

EFI 1.10 Extension

To maintain compatibility with EFI drivers that are written to the *EFI 1.02 Specification*, **StartImage()** must monitor the handle database before and after each image is started. If any handles are created or modified when an image is started, then **ConnectController()** must be called for each of the newly created or modified handles before **StartImage()** returns.

Section 5.5: CalculateCrc32() – Prototype – Crc32 (Page 5-93)

Changed the function prototype for **CaclulateCrc32()** FROM:

```
typedef
EFI_STATUS
CalculateCrc32 (
  IN  UINT8   *Data,
  IN  UINTN   DataSize,
  OUT UINT32  *Crc32
  );
```

TO:

```
typedef
EFI_STATUS
CalculateCrc32 (
  IN  VOID    *Data,
  IN  UINTN   DataSize,
  OUT UINT32  *Crc32
  );
```

Clarifications and Corrections

Section 5.1: Table 5-3 (Page 5-4)

Changed the LoadImage() entry in Table 5-3 FROM:

LoadImage()	<=	TPL_CALLBACK

TO:

LoadImage()	<	TPL_CALLBACK
StartImage()	<	TPL_CALLBACK
UnloadImage()	<=	TPL_CALLBACK
Exit()	<=	TPL_CALLBACK
Time Services	<=	TPL_CALLBACK

Section 5.1: CheckEvent() – Description (Page 5-12)

Changed the second sentence of the first paragraph of the "Description" section FROM:

If *Event* is of type **EFI_NOTIFY_WAIT**, there are three possibilities:

TO:

Otherwise, there are three possibilities:

Section 5.1: CheckEvent() – Status Codes Returned (Page 5-12)

Changed the "Status Codes Returned" table FROM:

EFI_SUCCESS	The event is in the signaled state.
EFI_NOT_READY	The event is not in the signaled state.

TO:

EFI_SUCCESS	The event is in the signaled state.
EFI_NOT_READY	The event is not in the signaled state.
EFI_INVALID_PARAMETER	*Event* is of type **EVT_NOTIFY_SIGNAL**.

Section 5.2: Memory Allocation Tables – Table 5-5 – EfiMemoryMappedIoPortSpace (Page 5-19)

Changed the description of "EfiMemoryMappedIoPortSpace" in Table 5-5 FROM:

System memory mapped IO region that is used to translate memory cycles to IO cycles.

TO:

System memory-mapped IO region that is used to translate memory cycles to IO cycles by the processor.

Section 5.2: Memory Allocation Tables – Table 5-5 (Page 5-19)

Added the following note below Table 5-5:

NOTE

There is only one region of type ***EfiMemoryMappedIoPortSpace*** *defined in the architecture for Itanium®-based platforms. As a result, there should be one and only one region of type* ***EfiMemoryMappedIoPortSpace*** *in the EFI memory map of an Itanium-based platform.*

Section 5.2: GetMemoryMap() – Related Definitions (Page 5-26)

Changed the description of the *NumberOfPages* field of the **EFI_MEMORY_DESCRIPTOR** structure FROM:

Number of pages in the memory region.

TO:

Number of 4 KB pages in the memory region.

Section 5.3.1: InstallProtocolInterface() – Description (Page 5-37)

Added the following sentence to the end of the first paragraph in the "Description" section:

> The same GUID cannot be installed more than once onto the same handle. If the same GUID is installed more than once onto the same handle, then the results are not predictable.

Section 5.3.1: RegisterProtocolNotify() – Parameters – Event (Page 5-42)

Changed the description of the *Event* parameter FROM:

> Event that is to be signaled whenever a protocol interface is registered for *Protocol*. Type **EFI_EVENT** is defined in the **InstallProtocolInterface()** function description.

TO:

> Event that is to be signaled whenever a protocol interface is registered for *Protocol*. The type **EFI_EVENT** is defined in the **InstallProtocolInterface()** function description. The same **EFI_EVENT** may be used for multiple protocol notify registrations.

Section 5.3.1: OpenProtocol() – Description (Page 5-50)

Changed the eleventh paragraph in the "Description" section FROM:

> If *Attributes* is **BY_CHILD_CONTROLLER** and *AgentHandle* is identical to *ControllerHandle*, then **EFI_INVALID_PARAMETER** is returned.

TO:

> If *Attributes* is **BY_CHILD_CONTROLLER** and *Handle* is identical to *ControllerHandle*, then **EFI_INVALID_PARAMETER** is returned.

Section 5.3.1: OpenProtocol() – Status Codes Returned (Page 5-53)

Changed the fifth status code listed in the "Status Codes Returned" table on page 5-53 FROM:

EFI_INVALID_PARAMETER	*Attributes* is **BY_CHILD_CONTROLLER** and *AgentHandle* is identical to *ControllerHandle*.

TO:

EFI_INVALID_PARAMETER	*Attributes* is **BY_CHILD_CONTROLLER** and *Handle* is identical to *ControllerHandle*.

Section 5.3.1: OpenProtocolInformation() – Description (Page 5-60)

Changed the first sentence of the third paragraph in the "Description" section FROM:

> If the interface specified by *Protocol* is supported by the handle specified by *Handle*, then *EntryBuffer* is allocated with the boot service **AllocatePages()**, and *EntryCount* is set to the number of entries in *EntryBuffer*.

TO:

> If the interface specified by *Protocol* is supported by the handle specified by *Handle*, then *EntryBuffer* is allocated with the boot service **AllocatePool()**, and *EntryCount* is set to the number of entries in *EntryBuffer*.

Section 5.3.1: DisconnectController() – Description (Page 5-66)

Changed the sixth sentence of the first paragraph FROM:

> If *ChildHandle* is the only child of *ControllerHandle*, then the drivers specified by *DriverImageHandle* will be disconnected from *ControllerHandle*.

TO:

> If *ChildHandle* is the only child of *ControllerHandle*, then the driver specified by *DriverImageHandle* will be disconnected from *ControllerHandle*.

Section 5.3.1: LocateHandleBuffer() – Prototype (Page 5-70)

Changed the "Prototype" section FROM:

```
typedef
EFI_STATUS
LocateHandle (
  IN EFI_LOCATE_SEARCH_TYPE  SearchType,
  IN EFI_GUID                *Protocol   OPTIONAL,
  IN VOID                    *SearchKey  OPTIONAL,
  IN OUT UINTN               *NoHandles,
  OUT EFI_HANDLE             **Buffer
  );
```

TO:

```
typedef
EFI_STATUS
LocateHandleBuffer (
  IN EFI_LOCATE_SEARCH_TYPE  SearchType,
  IN EFI_GUID                *Protocol   OPTIONAL,
  IN VOID                    *SearchKey  OPTIONAL,
  IN OUT UINTN               *NoHandles,
  OUT EFI_HANDLE             **Buffer
  );
```

Section 5.3.1: LocateHandleBuffer() – Description (Page 5-71)

Changed the second paragraph on page 5-71 of the "Description" section FROM:

> If *Buffer* is **NULL**, then **EFI_INVALID_PAREMETER** is returned.

TO:

> If *Buffer* is **NULL**, then **EFI_INVALID_PARAMETER** is returned.

Section 5.3.1: InstallMultipleProtocolInterfaces() – Description (Page 5-74)

Made the following two changes in the "Description" section:

1. Changed the fifth sentence of the first paragraph FROM:

   > The pairs of arguments are removed from the variable argument list until a **NULL** protocol GUID value is found.

   TO:

   > The pairs of arguments are removed in order from the variable argument list until a **NULL** protocol GUID value is found.

2. Added the following to the end of the first paragraph:

   > The same GUID cannot be installed more than once onto the same handle. If the same GUID is installed more than once onto the same handle, then the results are not predictable.

Section 5.3.1: UninstallMultipleProtocolInterfaces() – Description (Page 5-75)

Changed the fifth sentence of the first paragraph in the "Description" section FROM:

> The pairs of arguments are removed from the variable argument list until a **NULL** protocol GUID value is found.

TO:

> The pairs of arguments are removed in order from the variable argument list until a **NULL** protocol GUID value is found.

Section 5.4: StartImage() – Parameters – ExitDataSize (Page 5-80)

Changed the description of the *ExitDataSize* parameter FROM:

> Pointer to the size, in bytes, of *ExitData*.

TO:

> Pointer to the size, in bytes, of *ExitData*. If *ExitData* is **NULL**, then this parameter is ignored and the contents of *ExitDataSize* are not modified.

Section 5.5: CopyMem() – Description (Page 5-89)

Added the following paragraph to the end of the "Description" section:

> The contents of the *Destination* buffer on exit from this service must match the contents of the *Source* buffer on entry to this service. Due to potential overlaps, the contents of the *Source* buffer may be modified by this service. The following rules can be used to guarantee the correct behavior:
>
> 1. If *Destination* and *Source* are identical, then no operation should be performed.
> 2. If *Destination* > *Source* **and** *Destination* < (*Source* + *Length*), then the data should be copied from the *Source* buffer to the *Destination* buffer starting from the end of the buffers and working toward the beginning of the buffers.
> 3. Otherwise, the data should be copied from the *Source* buffer to the *Destination* buffer starting from the beginning of the buffers and working toward the end of the buffers.

Section 5.5: CalculateCrc32() – Parameters – Crc32 (Page 5-93)

Changed the description of the parameter *Crc32* FROM:

> The 32 bit CRC that was computer for the data buffer specified by *Data* and *DataSize*.

TO:

> The 32-bit CRC that was computed for the data buffer specified by *Data* and *DataSize*.

Section 6: Services – Runtime Services

Additions and Changes

Section 6.3: Virtual Memory Services (Page 6-16)

Added the following sentence to the end of the first paragraph on page 6-16:

> The system firmware must follow the processor-specific rules outlined in sections 2.3.2 and 2.3.3 in the layout of the EFI memory map to enable the OS to make the required virtual mappings.

Clarifications and Corrections

Section 6.1: GetVariable() – Status Codes Returned (Page 6-4)

Changed the "Status Codes Returned" table FROM:

EFI_SUCCESS	The function completed successfully.
EFI_NOT_FOUND	The variable was not found.
EFI_BUFFER_TOO_SMALL	The *DataSize* is too small for the result. *DataSize* has been updated with the size needed to complete the request.
EFI_INVALID_PARAMETER	One of the parameters has an invalid value.
EFI_DEVICE_ERROR	The variable could not be retrieved due to a hardware error.

TO:

EFI_SUCCESS	The function completed successfully.
EFI_NOT_FOUND	The variable was not found.
EFI_BUFFER_TOO_SMALL	The *DataSize* is too small for the result. *DataSize* has been updated with the size needed to complete the request.
EFI_INVALID_PARAMETER	*VariableName* is **NULL**.
EFI_INVALID_PARAMETER	*VendorGuid* is **NULL**.
EFI_INVALID_PARAMETER	*DataSize* is **NULL**.
EFI_INVALID_PARAMETER	*Data* is **NULL**.
EFI_DEVICE_ERROR	The variable could not be retrieved due to a hardware error.

Section 6.1: GetNextVariableName() – Status Codes Returned (Page 6-6)

Changed the "Status Codes Returned" table FROM:

EFI_SUCCESS	The function completed successfully.
EFI_NOT_FOUND	The next variable was not found.
EFI_BUFFER_TOO_SMALL	The *VariableNameSize* is too small for the result. *VariableNameSize* has been updated with the size needed to complete the request.
EFI_INVALID_PARAMETER	One of the parameters has an invalid value.
EFI_DEVICE_ERROR	The variable name could not be retrieved due to a hardware error.

TO:

EFI_SUCCESS	The function completed successfully.
EFI_NOT_FOUND	The next variable was not found.
EFI_BUFFER_TOO_SMALL	The *VariableNameSize* is too small for the result. *VariableNameSize* has been updated with the size needed to complete the request.
EFI_INVALID_PARAMETER	*VariableNameSize* is **NULL**.
EFI_INVALID_PARAMETER	*VariableName* is **NULL**.
EFI_INVALID_PARAMETER	*VendorGuid* is **NULL**.
EFI_DEVICE_ERROR	The variable name could not be retrieved due to a hardware error.

Section 6.1: SetVariable() – Description (Page 6-8)

Changed the last sentence of the last bullet FROM:

Variables that have runtime access but that are not non-volatile are effective read-only data variables once **ExitBootServices()** is performed.

TO:

Variables that have runtime access but that are not nonvolatile are read-only data variables once **ExitBootServices()** is performed.

Section 6.1: SetVariable() – Status Codes Returned (Page 6-8)

Changed the "Status Codes Returned" table FROM:

EFI_SUCCESS	The firmware has successfully stored the variable and its data as defined by the Attributes.
EFI_INVALID_PARAMETER	An invalid combination of Attribute bits was supplied, or the *DataSize* exceeds the maximum allowed.
EFI_OUT_OF_RESOURCES	Not enough storage is available to hold the variable and its data.
EFI_DEVICE_ERROR	The variable could not be saved due to a hardware failure.

TO:

EFI_SUCCESS	The firmware has successfully stored the variable and its data as defined by the Attributes.
EFI_INVALID_PARAMETER	An invalid combination of Attribute bits was supplied, or the *DataSize* exceeds the maximum allowed.
EFI_INVALID_PARAMETER	*VariableName* is an empty Unicode string.
EFI_OUT_OF_RESOURCES	Not enough storage is available to hold the variable and its data.
EFI_DEVICE_ERROR	The variable could not be saved due to a hardware failure.

Section 6.2: GetTime() – Status Codes Returned (Page 6-12)

Changed the "Status Codes Returned" table FROM:

EFI_SUCCESS	The operation completed successfully.
EFI_INVALID_PARAMETER	One of the parameters has an invalid value.
EFI_DEVICE_ERROR	The time could not be retrieved due to a hardware error.

TO:

EFI_SUCCESS	The operation completed successfully.
EFI_INVALID_PARAMETER	*Time* is **NULL**.
EFI_DEVICE_ERROR	The time could not be retrieved due to a hardware error.

Section 6.2: GetWakeupTime() – Status Codes Returned (Page 6-14)

Changed the "Status Codes Returned" table FROM:

EFI_SUCCESS	The alarm settings were returned.
EFI_INVALID_PARAMETER	A time field is out of range.
EFI_DEVICE_ERROR	The wakeup time could not be retrieved due to a hardware error.
EFI_UNSUPPORTED	A wakeup timer is not supported on this platform.

TO:

EFI_SUCCESS	The alarm settings were returned.
EFI_INVALID_PARAMETER	*Enabled* is **NULL**.
EFI_INVALID_PARAMETER	*Pending* is **NULL**.
EFI_INVALID_PARAMETER	*Time* is **NULL**.
EFI_DEVICE_ERROR	The wakeup time could not be retrieved due to a hardware error.
EFI_UNSUPPORTED	A wakeup timer is not supported on this platform.

Section 6.3: SetVirtualAddressMap() – Description (Page 6-18)

Changed the contents of the fourth paragraph of the "Description" section FROM:

> In addition, all of the fields of the EFI Runtime Services Table and several fields of the EFI System Table must be converted from physical pointers to virtual pointers using the **ConvertPointer()** function. The EFI System Table fields include *FirmwareVendor*, *RuntimeServices*, and *ConfigurationTable*. Because fields of both the EFI Runtime Services Table and the EFI System Table are being modified, the 32-bit CRC for the EFI Runtime Services Table and the EFI System Table must be recomputed.

TO:

> In addition, all of the fields of the EFI Runtime Services Table except *SetVirtualAddressMap* and *ConvertPointer* must be converted from physical pointers to virtual pointers using the **ConvertPointer()** service. The **SetVirtualAddressMap()** and **ConvertPointer()** services are only callable in physical mode, so they do not need to be converted from physical pointers to virtual pointers. Several fields of the EFI System Table must be converted from physical pointers to virtual pointers using the **ConvertPointer()** service. These fields include *FirmwareVendor*, *RuntimeServices*, and *ConfigurationTable*. Because contents of both the EFI Runtime Services Table and the EFI System Table are modified by this service, the 32-bit CRC for the EFI Runtime Services Table and the EFI System Table must be recomputed.

Section 6.4: ResetSystem() – Description (Page 6-22)

Changed the third and fourth paragraphs FROM:

> Calling this interface with *ResetType* of **EfiResetWarm** causes a system-wide initialization. The processors are set to their initial state, and pending cycles are not corrupted. If the system does not support this reset type, then an **EfiColdReset** must be performed.
>
> Calling this interface with *ResetType* of **EfiResetShutdown** causes power to be removed from the system. If the system does not support this reset type, then an **EfiColdReset** must be performed. If an ACPI S5 method is available to remove power from the system, then this reset type should not be used.

TO:

> Calling this interface with *ResetType* of **EfiResetWarm** causes a system-wide initialization. The processors are set to their initial state, and pending cycles are not corrupted. If the system does not support this reset type, then an **EfiResetCold** must be performed.
>
> Calling this interface with *ResetType* of **EfiResetShutdown** causes the system to enter a power state equivalent to the ACPI G2/S5 or G3 states. If the system does not support this reset type, then when the system is rebooted, it should exhibit the **EfiResetCold** attributes. If the ACPI S5 state is supported on the system, then this reset type should not be used.

Section 6.4: GetNextHighMonotonicCount() – Status Codes Returned (Page 6-23)

Changed the "Status Codes Returned" table FROM:

EFI_SUCCESS	The next high monotonic count was returned.
EFI_DEVICE_ERROR	The device is not functioning properly.
EFI_INVALID_PARAMETER	One of the parameters has an invalid value.

TO:

EFI_SUCCESS	The next high monotonic count was returned.
EFI_DEVICE_ERROR	The device is not functioning properly.
EFI_INVALID_PARAMETER	*HighCount* is **NULL**.

Section 8: Protocols – Device Path Protocol

Additions and Changes

Section 8.3.2.2: PCCARD Device Path – Table 8-4 (Page 8-5)

Changed Table 8-4 FROM:

Table 8-4. PCCARD Device Path

Mnemonic	Byte Offset	Byte Length	Description
Type	0	1	Type 1 – Hardware Device Path
Sub-Type	1	1	Sub-Type 2 – PCCARD
Length	2	2	Length of this structure in bytes. Length is 5 bytes.
Socket Number	4	1	Socket Number (0 = First Socket)

TO:

Table 8-4. PCCARD Device Path

Mnemonic	Byte Offset	Byte Length	Description
Type	0	1	Type 1 – Hardware Device Path
Sub-Type	1	1	Sub-Type 2 – PCCARD
Length	2	2	Length of this structure in bytes. Length is 5 bytes.
Function Number	4	1	Function Number (0 = First Function)

Section 8.3.3: ACPI Device Path (Page 8-7)

Inserted the following paragraphs just before Table 8-8:

The ACPI Device Path node only supports numeric 32-bit values for the _HID and _UID values. The Expanded ACPI Device Path node supports both numeric and string values for the _HID, _UID, and _CID values. As a result, the ACPI Device Path node is smaller and should be used if possible to reduce the size of device paths that may potentially be stored in nonvolatile storage. If a string value is required for the _HID field, or a string value is required for the _UID field, or a _CID field is required, then the Expanded ACPI Device Path node must be used. If a string field of the Expanded ACPI Device Path node is present, then the corresponding numeric field is ignored.

The _HID and _CID fields in the ACPI Device Path node and Expanded ACPI Device Path node are stored as a 32-bit compressed EISA-type IDs. The following macro can be used to compute these EISA-type IDs from a Plug and Play Hardware ID. The Plug and Play Hardware IDs used to compute the _HID and _CID fields in the EFI device path nodes must match the Plug and Play Hardware IDs used to build the matching entries in the ACPI tables. The compressed EISA-type IDs produced by this macro differ from the compressed EISA-type IDs stored in ACPI tables. As a result, the compressed EISA-type IDs from the ACPI Device Path nodes can not be directly compared to the compressed EISA-type IDs from the ACPI table.

```
#define EFI_PNP_ID(ID)  (UINT32)(((ID) << 16) | 0x41D0)
#define EISA_PNP_ID(ID) EFI_PNP_ID(ID)
```

Section 8.3.3: ACPI Device Path – Table 8-8 (Page 8-8)

Changed Table 8-8 FROM:

Table 8-8. ACPI Device Path

Mnemonic	Byte Offset	Byte Length	Description
Type	0	1	Type 2 – ACPI Device Path.
Sub-Type	1	1	Sub-Type 1 ACPI Device Path.
Length	2	2	Length of this structure in bytes. Length is 12 bytes.
_HID	4	4	Device's PnP hardware ID stored in a numeric 32-bit compressed EISA-type ID or a String. This value must match the corresponding _HID in the ACPI name space.
_UID	8	4	Unique ID that is required by ACPI if two devices have the same _HID. This value must also match the corresponding _UID/_HID pair in the ACPI name space. Only the 32-bit numeric value type of _UID is supported; thus strings must not be used for the _UID in the ACPI name space.

TO:

Table 8-8. ACPI Device Path

Mnemonic	Byte Offset	Byte Length	Description
Type	0	1	Type 2 – ACPI Device Path.
Sub-Type	1	1	Sub-Type 1 ACPI Device Path.
Length	2	2	Length of this structure in bytes. Length is 12 bytes.
_HID	4	4	Device's PnP hardware ID stored in a numeric 32-bit compressed EISA-type ID. This value must match the corresponding _HID in the ACPI name space.
_UID	8	4	Unique ID that is required by ACPI if two devices have the same _HID. This value must also match the corresponding _UID/_HID pair in the ACPI name space. Only the 32-bit numeric value type of _UID is supported; thus strings must not be used for the _UID in the ACPI name space.

Section 8.3.3: ACPI Device Path – Table 8-9 (Page 8-8)

Changed Table 8-9 FROM:

Table 8-9. Expanded ACPI Device Path

Mnemonic	Byte Offset	Byte Length	Description
Type	0	1	Type 2 – ACPI Device Path.
Sub-Type	1	1	Sub-Type 2 Expanded ACPI Device Path.
Length	2	2	Length of this structure in bytes. Length is 16 bytes.
_HID	4	4	Device's PnP hardware ID stored in a numeric 32-bit compressed EISA-type ID or a String. This value must match the corresponding _HID in the ACPI name space.
_UID	8	4	Unique ID that is required by ACPI if two devices have the same _HID. This value must also match the corresponding _UID/_HID pair in the ACPI name space. Only the 32-bit numeric value type of _UID is supported; thus strings must not be used for the _UID in the ACPI name space.
_CID	12	4	Device's compatible PnP hardware ID stored in a numeric 32-bit compressed EISA-type ID or a String. This value must match at least one of the compatible device IDs returned by the corresponding _CID in the ACPI name space.

TO:

Table 8-9. Expanded ACPI Device Path

Mnemonic	Byte Offset	Byte Length	Description
Type	0	1	Type 2 – ACPI Device Path.
Sub-Type	1	1	Sub-Type 2 Expanded ACPI Device Path.
Length	2	2	Length of this structure in bytes. Minimum length is 19 bytes. The actual size will depend on the size of the _HIDSTR, _UIDSTR, and _CIDSTR fields.
_HID	4	4	Device's PnP hardware ID stored in a numeric 32-bit compressed EISA-type ID. This value must match the corresponding _HID in the ACPI name space.
_UID	8	4	Unique ID that is required by ACPI if two devices have the same _HID. This value must also match the corresponding _UID/_HID pair in the ACPI name space.
_CID	12	4	Device's compatible PnP hardware ID stored in a numeric 32-bit compressed EISA-type ID. This value must match at least one of the compatible device IDs returned by the corresponding _CID in the ACPI name space.
_HIDSTR	16	>=1	Device's PnP hardware ID stored as a null-terminated ASCII string. This value must match the corresponding _HID in the ACPI name space. If the length of this string not including the null-terminator is 0, then the _HID field is used. If the length of this null-terminated string is greater than 0, then this field supercedes the _HID field.
_UIDSTR	Varies	>=1	Unique ID that is required by ACPI if two devices have the same _HID. This value must also match the corresponding _UID/_HID pair in the ACPI name space. This value is stored as a null-terminated ASCII string. If the length of this string not including the null-terminator is 0, then the _UID field is used. If the length of this null-terminated string is greater than 0, then this field supercedes the _UID field. The Byte Offset of this field can be computed by adding 16 to the size of the _HIDSTR field.
_CIDSTR	Varies	>=1	Device's compatible PnP hardware ID stored as a null-terminated ASCII string. This value must match at least one of the compatible device IDs returned by the corresponding _CID in the ACPI name space. If the length of this string not including the null-terminator is 0, then the _CID field is used. If the length of this null-terminated string is greater than 0, then this field supercedes the _CID field. The Byte Offset of this field can be computed by adding 16 to the sum of the sizes of the _HIDSTR and _UIDSTR fields.

Section 8.3.4.11: InfiniBand Device Path – Table 8-20 (Page 8-13)

Changed Table 8-20 FROM:

Table 8-20. InfiniBand Device Path

Mnemonic	Byte Offset	Byte Length	Description
Type	0	1	Type 3 – Messaging Device Path
Sub-Type	1	1	Sub-Type 9 – InfiniBand
Length	2	2	Length of this structure in bytes
Reserved	4	4	Reserved
Node GUID[1]	8	8	64 bit node GUID[1] of the IOU
IOC GUID[1]	16	8	64 bit GUID[1] of the IOC
Device ID	24	8	64 bit persistent ID of the device
Note 1	The usage of the term GUID is per the Infiniband Specification. This is not the same as the `EFI_GUID` type defined in the EFI Specification.		

TO:

Table 8-20. InfiniBand Device Path

Mnemonic	Byte Offset	Byte Length	Description
Type	0	1	Type 3 – Messaging Device Path
Sub-Type	1	1	Sub-Type 9 – InfiniBand
Length	2	2	Length of this structure in bytes. Length is 48 bytes.
Resource Flags	4	4	Flags to help identify/manage InfiniBand device path elements: • Bit 0 – IOC/Service (0b = IOC, 1b = Service) • Bit 1 – Extend Boot Environment • Bit 2 – Console Protocol • Bit 3 – Storage Protocol • Bit 4 – Network Protocol All other bits are reserved.
PORT GID	8	16	128-bit Global Identifier for remote fabric port
IOC GUID/Service ID	24	8	64-bit unique identifier to remote IOC or server process. Interpretation of field specified by Resource Flags (bit 0)
Target Port ID	32	8	64-bit persistent ID of remote IOC port
Device ID	40	8	64-bit persistent ID of remote device
Note	The usage of the term GUID and GID are per the InfiniBand Specification. The term GUID is not the same as the `EFI_GUID` type defined in the EFI Specification.		

Clarifications and Corrections

None.

Section 9: Protocols – EFI Driver Model

Additions and Changes

Section 9.4: EFI_DRIVER_CONFIGURATION_PROTOCOL.SetOptions() – Parameters – ControllerHandle (Page 9-35)

Changed the description of *ControllerHandle* FROM:

> The handle of the controller to set options on.

TO:

> The handle of the controller to set options on. If *ControllerHandle* is a valid **EFI_HANDLE** that is being managed by this driver, then the user will be allowed to set options for the controller specified by *ControllerHandle*. If this parameter is **NULL**, then the options will be set for all the controllers that this driver is currently managing. If *ControllerHandle* is **NULL**, then setting options for a child controller is not supported, so *ChildHandle* must also be **NULL**.

Clarifications and Corrections

None.

Section 10: Protocols – Console Support

Additions and Changes

Section 10.7: PUGA_FW_SERVICE_DISPATCH.DispatchService() – Prototype (Page 10-39)

Changed the "Prototype" section FROM:

```
typedef
UGA_STATUS
(UGA_FW_CALL_TYPE_API *PUGA_FW_SERVICE_DISPATCH) (
  IN  PUGA_DEVICE            pDevice,
  IN OUT PUGA_IO_REQUEST     pIoRequest
  );
```

TO:

```
typedef
UGA_STATUS
(EFIAPI *PUGA_FW_SERVICE_DISPATCH) (
  IN  PUGA_DEVICE            pDevice,
  IN OUT PUGA_IO_REQUEST     pIoRequest
  );
```

Section 10.7: PUGA_FW_SERVICE_DISPATCH.DispatchService() – Related Definitions (Page 10-39)

Inserted the following statement at the beginning of the "Related Definitions" section:

```
typedef UINT32  UGA_STATUS;
```

Clarifications and Corrections

Section 10.5: EFI_UGA_DRAW_PROTOCOL.Blt() – Prototype (Page 10-31)

Changed the "Prototype" section FROM:

```
typedef
EFI_STATUS
(EFIAPI *EFI_UGA_DRAW_PROTOCOL_BLT) (
  IN EFI_UGA_DRAW_PROTOCOL     *This,
  IN OUT EFI_UGA_PIXEL          *BltBuffer,  OPTIONAL
  IN UINTN                     SourceX,
  IN UINTN                     SourceY,
  IN UINTN                     DestinationX,
  IN UINTN                     DestinationY,
  IN UINTN                     Width,
  INUINTN                      Height,
  IN UINTN                      Delta       OPTIONAL
  );
```

TO:

```
typedef
EFI_STATUS
(EFIAPI *EFI_UGA_DRAW_PROTOCOL_BLT) (
  IN EFI_UGA_DRAW_PROTOCOL     *This,
  IN OUT EFI_UGA_PIXEL          *BltBuffer,  OPTIONAL
  IN  EFI_UGA_BLT_OPERATION    BltOperation,
  IN UINTN                     SourceX,
  IN UINTN                     SourceY,
  IN UINTN                     DestinationX,
  IN UINTN                     DestinationY,
  IN UINTN                     Width,
  IN UINTN                     Height,
  IN UINTN                      Delta       OPTIONAL
  );
```

Section 11: Protocols – Bootable Image Support

Additions and Changes

Section 11.3: EFI_FILE_IO_INTERFACE.OpenVolume() – Status Codes Returned (Page 11-19)

Added the following return code to the "Status Codes Returned" section:

EFI_MEDIA_CHANGED	The device has a different medium in it or the medium is no longer supported. Any existing file handles for this volume are no longer valid. To access the files on the new medium, the volume must be reopened with **`OpenVolume()`**.

Section 11.3: EFI_FILE.Open() – Description (Page 11-23)

Changed the second paragraph of the "Description" section FROM:

> If **`EFI_FILE_MODE_CREATE`** is set, then the file is created in the directory. If the final location of *`FileName`* does not refer to a directory then the operation fails. If the file does not exist in the directory, then a new file is created. If the file already exists in the directory, then the existing file is deleted, and the new file is created.

TO:

> If **`EFI_FILE_MODE_CREATE`** is set, then the file is created in the directory. If the final location of *`FileName`* does not refer to a directory, then the operation fails. If the file does not exist in the directory, then a new file is created. If the file already exists in the directory, then the existing file is opened.

Clarifications and Corrections

Section 11.7: UNICODE_COLLATION Protocol – Protocol Interface Structure (Page 11-50)

Changed the :Protocol Interface Structure" section FROM:

```
typedef struct {
   EFI_UNICODE_COLLATION_STRICOLL      StriColl;
   EFI_UNICODE_COLLATION _METAIMATCH   MetaiMatch;
   EFI_UNICODE_STRLWR                  StrLwr;
   EFI_UNICODE_STRUPR                  StrUpr;
   EFI_UNICODE_FATTOSTR                FatToStr;
   EFI_UNICODE_STRTOFAT                StrToFat;
   CHAR8                               *SupportedLanguages;
} UNICODE_COLLATION_INTERFACE;
```

TO:

```
typedef struct {
   EFI_UNICODE_COLLATION_STRICOLL      StriColl;
   EFI_UNICODE_COLLATION_METAIMATCH    MetaiMatch;
   EFI_UNICODE_COLLATION_STRLWR        StrLwr;
   EFI_UNICODE_COLLATION_STRUPR        StrUpr;
   EFI_UNICODE_COLLATION_FATTOSTR      FatToStr;
   EFI_UNICODE_COLLATION_STRTOFAT      StrToFat;
   CHAR8                               *SupportedLanguages;
} UNICODE_COLLATION_INTERFACE;
```

Section 11.7: UNICODE_COLLATION.MetaiMatch() – Prototype (Page 11-55)

Changed the "Prototype" section FROM:

```
BOOLEAN
     (EFIAPI *EFI_UNICODE_COLLATION_STRICOLL) (
     IN UNICODE_COLLATION_INTERFACE   *This,
     IN CHAR16                        *String,
     IN CHAR16                        *Pattern
);
```

TO:

```
BOOLEAN
(EFIAPI *EFI_UNICODE_COLLATION_METAIMATCH) (
     IN UNICODE_COLLATION_INTERFACE   *This,
     IN CHAR16                        *String,
     IN CHAR16                        *Pattern
);
```

Section 12: Protocols – PCI Bus Support

Additions and Changes

None.

Clarifications and Corrections

Section 12.1: EFI PCI Root Bridge I/O Support – (Page 12-1)

Added the following paragraph at the end of Section 12.1:

> All the services described in this chapter that generate PCI transactions follow the ordering rules defined in the *PCI Specification*. If the processor is performing a combination of PCI transactions and system memory transactions, then there is no guarantee that the system memory transactions will be strongly ordered with respect to the PCI transactions. If strong ordering is required, then processor-specific mechanisms may be required to guarantee strong ordering. For example, Itanium-based systems may require the use of memory fences to guarantee ordering.

Section 12.2: EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL – Related Definitions (Page 12-11)

Added the following **#define** at the end of the list of **EFI_PCI_ATTRIBUTE**s:

```
#define EFI_PCI_ATTRIBUTE_DUAL_ADDRESS_CYCLE       0x8000
```

Section 12.2: EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL – Related Definitions (Page 12-12)

Added the following explanation after the **EFI_PCI_IO_ATTRIBUTE_MEMORY_DISABLE** field description:

> **EFI_PCI_ATTRIBUTE_DUAL_ADDRESS_CYCLE**
>
> This bit may only be used in the *Attributes* parameter to **AllocateBuffer()**. If this bit is set, then the PCI controller that is requesting a buffer through **AllocateBuffer()** is capable of producing PCI Dual Address Cycles, so it is able to access a 64-bit address space. If this bit is not set, then the PCI controller that is requesting a buffer through **AllocateBuffer()** is not capable of producing PCI Dual Address Cycles, so it is only able to access a 32-bit address space.

Section 12.2: EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL – Related Definitions – EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_OPERATION (Page 12-13)

Changed the following section FROM:

```
typedef enum {
  EfiPciOperationBusMasterRead,
  EfiPciOperationBusMasterWrite,
  EfiPciOperationBusMasterCommonBuffer,
  EfiPciOperationMaximum
} EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_OPERATION;
```

EfiPciOperationBusMasterRead

A read operation from system memory by a bus master.

EfiPciOperationBusMasterWrite

A write operation to system memory by a bus master.

EfiPciOperationBusMasterCommonBuffer

Provides both read and write access to system memory by both the CPU and a bus master. The buffer is coherent from both the CPUs and the bus masters point of view.

TO:

```
typedef enum {
  EfiPciOperationBusMasterRead,
  EfiPciOperationBusMasterWrite,
  EfiPciOperationBusMasterCommonBuffer,
  EfiPciOperationBusMasterRead64,
  EfiPciOperationBusMasterWrite64,
  EfiPciOperationBusMasterCommonBuffer64,
  EfiPciOperationMaximum
} EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_OPERATION;
```

EfiPciOperationBusMasterRead

A read operation from system memory by a bus master that is not capable of producing PCI dual address cycles.

EfiPciOperationBusMasterWrite

A write operation to system memory by a bus master that is not capable of producing PCI dual address cycles.

EfiPciOperationBusMasterCommonBuffer

Provides both read and write access to system memory by both the CPU and a bus master that is not capable of producing PCI dual address cycles. The buffer is coherent from both the processor's and the bus master's point of view.

EfiPciOperationBusMasterRead64

A read operation from system memory by a bus master that is capable of producing PCI dual address cycles.

EfiPciOperationBusMasterWrite64

A write operation to system memory by a bus master that is capable of producing PCI dual address cycles.

EfiPciOperationBusMasterCommonBuffer64

Provides both read and write access to system memory by both the processor and a bus master that is capable of producing PCI dual address cycles. The buffer is coherent from both the processor's and the bus master's point of view.

Section 12.2: EFI PCI Root Bridge I/O Protocol – Description – (Page 12-15)

Changed the DMA pseudo code FROM:

DMA Bus Master Read Operation

- Call **Map()** for **EfiPciOperationBusMasterRead**
- Program DMA Bus Master with the *DeviceAddress* returned by **Map()**
- Start the DMA Bus Master
- Wait for DMA Bus Master to complete the read operation
- Call **Unmap()**

DMA Bus Master Write Operation

- Call **Map()** for **EfiPciOperationBusMasterWrite**
- Program DMA Bus Master with the *DeviceAddress* returned by **Map()**
- Start the DMA Bus Master
- Wait for DMA Bus Master to complete the write operation
- Call **Unmap()**

DMA Bus Master Common Buffer Operation

- Call **AllocateBuffer()** to allocate a common buffer
- Call **Map()** for **EfiPciOperationBusMasterCommonBuffer**
- Program DMA Bus Master with the *DeviceAddress* returned by **Map()**
- The common buffer can now be accessed equally by the CPU and the DMA bus master
- Call **Unmap()**
- Call **FreeBuffer()**

TO:

DMA Bus Master Read Operation

- Call **Map()** for **EfiPciOperationBusMasterRead** or **EfiPciOperationBusMasterRead64**.
- Program the DMA Bus Master with the *DeviceAddress* returned by **Map()**.
- Start the DMA Bus Master.
- Wait for DMA Bus Master to complete the read operation.
- Call **Unmap()**.

DMA Bus Master Write Operation

- Call **Map()** for **EfiPciOperationBusMasterWrite** or **EfiPciOperationBusMasterRead64**.
- Program the DMA Bus Master with the *DeviceAddress* returned by **Map()**.
- Start the DMA Bus Master.
- Wait for DMA Bus Master to complete the write operation.
- Perform a PCI controller specific read transaction to flush all PCI write buffers (See *PCI Specification* Section 3.2.5.2) .
- Call **Flush()**.
- Call **Unmap()**.

DMA Bus Master Common Buffer Operation

- Call **AllocateBuffer()** to allocate a common buffer.
- Call **Map()** for **EfiPciOperationBusMasterCommonBuffer** or **EfiPciOperationBusMasterCommonBuffer64**.
- Program the DMA Bus Master with the *DeviceAddress* returned by **Map()**.
- The common buffer can now be accessed equally by the processor and the DMA bus master.
- Call **Unmap()**.
- Call **FreeBuffer()**.

Section 12.2: EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.PollMem() – Description (Page 12-17)

Added the following paragraph to the end of the "Description" section:

All the PCI transactions generated by this function are guaranteed to be completed before this function returns. However, if the memory mapped I/O region being accessed by this function has the **EFI_PCI_ATTRIBUTE_MEMORY_CACHED** attribute set, then the transactions will follow the ordering rules defined by the processor architecture.

Section 12.2: EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.PollIo() – Description (Page 12-19)

Added the following paragraph to the end of the "Description" section:

> All the PCI transactions generated by this function are guaranteed to be completed before this function returns.

Section 12.2: EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.MemRead() – Description (Page 12-21)

Added the following paragraph to the end of the "Description" section:

> All the PCI read transactions generated by this function are guaranteed to be completed before this function returns. All the PCI write transactions generated by this function will follow the write ordering and completion rules defined in the *PCI Specification*. However, if the memory-mapped I/O region being accessed by this function has the **EFI_PCI_ATTRIBUTE_MEMORY_CACHED** attribute set, then the transactions will follow the ordering rules defined by the processor architecture.

Section 12.2: EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.IoRead() – Description (Page 12-23)

Added the following paragraph to the end of the "Description" section:

> All the PCI transactions generated by this function are guaranteed to be completed before this function returns.

Section 12.2: EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.PciRead() – Description (Page 12-25)

Added the following paragraph to the end of the "Description" section:

> All the PCI transactions generated by this function are guaranteed to be completed before this function returns.

Section 12.2: EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.CopyMem() – Prototype (Page 12-26)

Changed the "Prototype" section FROM:

```
typedef
EFI_STATUS
(EFIAPI *EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_COPY_MEM) (
  IN     EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL        *This,
  IN     EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_WIDTH  Width,
  IN     UINT64                                 Destddress,
  IN     UINT64                                 Srcddress,
  IN     UINTN                                  Count
  );
```

TO:

```
typedef
EFI_STATUS
(EFIAPI *EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_COPY_MEM) (
  IN    EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL         *This,
  IN    EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_WIDTH   Width,
  IN    UINT64                                  DestAddress,
  IN    UINT64                                  SrcAddress,
  IN    UINTN                                   Count
  );
```

Section 12.2: EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.CopyMem() – Description (Page 12-27)

Added the following to the end of the "Description" section:

The contents of the *DestAddress* buffer on exit from this service must match the contents of the *SrcAddress* buffer on entry to this service. Due to potential overlaps, the contents of the *SrcAddress* buffer may be modified by this service. The following rules can be used to guarantee the correct behavior:

1. If *DestAddress* > *SrcAddress* **and** *DestAddress* < (*SrcAddress* + *Width* size * *Count*), then the data should be copied from the *SrcAddress* buffer to the *DestAddress* buffer starting from the end of buffers and working toward the beginning of the buffers.
2. Otherwise, the data should be copied from the *SrcAddress* buffer to the *DestAddress* buffer starting from the beginning of the buffers and working toward the end of the buffers.

All the PCI transactions generated by this function are guaranteed to be completed before this function returns. All the PCI write transactions generated by this function will follow the write ordering and completion rules defined in the *PCI Specification*. However, if the memory-mapped I/O region being accessed by this function has the **EFI_PCI_ATTRIBUTE_MEMORY_CACHED** attribute set, then the transactions will follow the ordering rules defined by the processor architecture.

Section 12.2: EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.Map() – Parameters (Page 12-28)

Changed the description of the *DeviceAddress* parameter FROM:

DeviceAddress	The resulting map address for the bus master PCI controller to use to access the system memory's *HostAddress*. Type **EFI_PHYSICAL_ADDRESS** is defined in Chapter 5.

TO:

DeviceAddress	The resulting map address for the bus master PCI controller to use to access the system memory's *HostAddress*. Type **EFI_PHYSICAL_ADDRESS** is defined in Chapter 5. This address can not be used by the processor to access the contents of the buffer specified by *HostAddress*.

Section 12.2: EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.Map() – Description (Page 12-29)

Changed the "Description" section FROM:

The **Map()** function provides the PCI controller specific addresses needed to access system memory. This function is used to map system memory for PCI bus master DMA accesses.

All PCI bus master accesses must be performed through their mapped addresses and such mappings must be freed with **Unmap()** when complete. If the bus master access is a single read or single write data transfer, then **EfiPciOperationBusMasterRead** or **EfiPciOperation-BusMasterWrite** is used and the range is unmapped to complete the operation. If performing an **EfiPciOperationBusMasterRead** operation, all the data must be present in system memory before **Map()** is performed. Similarly, if performing an **EfiPciOperation-BusMasterWrite**, the data cannot be properly accessed in system memory until **Unmap()** is performed.

Bus master operations that require both read and write access or require multiple host device interactions within the same mapped region must use **EfiPciOperation-BusMasterCommonBuffer**. However, only memory allocated via the **AllocateBuffer()** interface can be mapped for this type of operation.

In all mapping requests the resulting *NumberOfBytes* actually mapped may be less than the requested amount. In this case, the DMA operation will have to be broken up into smaller chunks. The **Map()** function will map as much of the DMA operation as it can at one time. The caller may have to loop on **Map()** and **Unmap()** in order to complete a large DMA transfer.

TO:

The **Map()** function provides the PCI controller specific addresses needed to access system memory. This function is used to map system memory for PCI bus master DMA accesses.

All PCI bus master accesses must be performed through their mapped addresses and such mappings must be freed with **Unmap()** when complete. If the bus master access is a single read or single write data transfer, then **EfiPciOperationBusMasterRead**, **EfiPciOperationBusMasterRead64**, **EfiPciOperationBusMasterWrite**, or **EfiPciOperationBusMasterWrite64** is used and the range is unmapped to complete the operation. If performing an **EfiPciOperationBusMasterRead** or **EfiPciOperationBusMasterRead64** operation, all the data must be present in system memory before **Map()** is performed. Similarly, if performing an **EfiPciOperation-BusMasterWrite** or **EfiPciOperationBusMasterWrite64** the data cannot be properly accessed in system memory until **Unmap()** is performed.

Bus master operations that require both read and write access or require multiple host device interactions within the same mapped region must use **EfiPciOperation-BusMasterCommonBuffer** or **EfiPciOperationBusMasterCommonBuffer64**. However, only memory allocated via the **AllocateBuffer()** interface can be mapped for this type of operation.

In all mapping requests the resulting *NumberOfBytes* actually mapped may be less than the requested amount. In this case, the DMA operation will have to be broken up into smaller chunks. The **Map()** function will map as much of the DMA operation as it can at one time. The caller may have to loop on **Map()** and **Unmap()** in order to complete a large DMA transfer.

Section 12.2: EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.UnMap() – Description (Page 12-30)

Changed the “Description” section FROM:

The **Unmap()** function completes the **Map()** operation and releases any corresponding resources. If the operation was an **EfiPciOperationBusMasterWrite**, the data is committed to the target system memory. Any resources used for the mapping are freed.

TO:

The **Unmap()** function completes the **Map()** operation and releases any corresponding resources. If the operation was an **EfiPciOperationBusMasterWrite** or **EfiPciOperationBusMasterWrite64**, the data is committed to the target system memory. Any resources used for the mapping are freed.

Section 12.2: EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.AllocateBuffer() (Page 12-31)

Changed the entire API definition to the following:

EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.AllocateBuffer()

Summary

Allocates pages that are suitable for an **EfiPciOperationBusMasterCommonBuffer** or **EfiPciOperationBusMasterCommonBuffer64** mapping.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_ALLOCATE_BUFFER) (
  IN     EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL  *This,
  IN     EFI_ALLOCATE_TYPE                Type,
  IN     EFI_MEMORY_TYPE                  MemoryType,
  IN     UINTN                            Pages,
  OUT    VOID                             **HostAddress,
  IN     UINT64                           Attributes
  );
```

Parameters

This	A pointer to the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL**. Type **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL** is defined in Section 12.2.1.
Type	This parameter is not used and must be ignored.
MemoryType	The type of memory to allocate, **EfiBootServicesData** or **EfiRuntimeServicesData**. Type **EFI_MEMORY_TYPE** is defined in Chapter 5.
Pages	The number of pages to allocate.
HostAddress	A pointer to store the base system memory address of the allocated range.
Attributes	The requested bit mask of attributes for the allocated range. Only the attributes **EFI_PCI_ATTRIBUTE_MEMORY_WRITE_COMBINE**, **EFI_PCI_ATTRIBUTE_MEMORY_CACHED**, and **EFI_PCI_ATTRIBUTE_DUAL_ADDRESS_CYCLE** may be used with this function. If any other bits are set, then **EFI_UNSUPPORTED** is returned. This function may choose to ignore this bit mask. The **EFI_PCI_ATTRIBUTE_MEMORY_WRITE_COMBINE**, and

EFI_PCI_ATTRIBUTE_MEMORY_CACHED attributes provide a hint to the implementation that may improve the performance of the calling driver. The implementation may choose any default for the memory attributes including write combining, cached, both, or neither as long as the allocated buffer can be seen equally by both the processor and the PCI bus master.

Description

The **AllocateBuffer()** function allocates pages that are suitable for an **EfiPciOperationBusMasterCommonBuffer** or **EfiPciOperationBusMasterCommonBuffer64** mapping. This means that the buffer allocated by this function must support simultaneous access by both the CPU and a PCI Bus Master. The device address that the PCI Bus Master uses to access the buffer can be retrieved with a call to **Map()**.

If the **EFI_PCI_ATTRIBUTE_DUAL_ADDRESS_CYCLE** bit of *Attributes* is set, then when the buffer allocated by this function is mapped with a call to **Map()**, the device address that is returned by **Map()** must be within the 64-bit device address space of the PCI Bus Master.

If the **EFI_PCI_ATTRIBUTE_DUAL_ADDRESS_CYCLE** bit of *Attributes* is clear, then when the buffer allocated by this function is mapped with a call to **Map()**, the device address that is returned by **Map()** must be within the 32-bit device address space of the PCI Bus Master.

If the memory allocation specified by *MemoryType* and *Pages* can not be satisfied, then **EFI_OUT_OF_RESOURCES** is returned.

Status Codes Returned

EFI_SUCCESS	The requested memory pages were allocated.
EFI_INVALID_PARAMETER	*MemoryType* is invalid.
EFI_INVALID_PARAMETER	*HostAddress* is **NULL**.
EFI_UNSUPPORTED	*Attributes* is unsupported. The only legal attribute bits are **MEMORY_WRITE_COMBINE**, **MEMORY_CACHED**, and **DUAL_ADDRESS_CYCLE**.
EFI_OUT_OF_RESOURCES	The memory pages could not be allocated.

Section 12.2: EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.Flush() (Page 12-34)

Changed the entire API definition to the following:

EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.Flush()

Summary

Flushes all PCI posted write transactions from a PCI host bridge to system memory.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_FLUSH) (
  IN  EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL  *This
  );
```

Parameters

This	A pointer to the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL**. Type **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL** is defined in Section 12.2.1.

Description

The **Flush()** function flushes any PCI posted write transactions from a PCI host bridge to system memory. Posted write transactions are generated by PCI bus masters when they perform write transactions to target addresses in system memory.

This function does not flush posted write transactions from any PCI bridges. A PCI controller specific action must be taken to guarantee that the posted write transactions have been flushed from the PCI controller and from all the PCI bridges into the PCI host bridge. This is typically done with a PCI read transaction from the PCI controller prior to calling **Flush()**.

If the PCI controller specific action required to flush the PCI posted write transactions has been performed, and this function returns **EFI_SUCCESS**, then the PCI bus master's view and the processor's view of system memory are guaranteed to be coherent. If the PCI posted write transactions cannot be flushed from the PCI host bridge, then the PCI bus master and processor are not guaranteed to have a coherent view of system memory, and **EFI_DEVICE_ERROR** is returned.

Status Codes Returned

EFI_SUCCESS	The PCI posted write transactions were flushed from the PCI host bridge to system memory.
EFI_DEVICE_ERROR	The PCI posted write transactions were not flushed from the PCI host bridge due to a hardware error.

Section 12.3.2: PCI Bus Drivers (Page 12-49)

In the paragraph after Figure 12-9, changed two instances of "PCI Host Bridge I/O Protocol" to "PCI Root Bridge I/O Protocol."

Section 12.3.2: PCI Bus Drivers - Figure 12-10 (Page 12-49)

In the figure, changed "PCI_ROOT_BRIDGE_PROTOCOL" to "PCI_ROOT_BRIDGE_IO_PROTOCOL."

Section 12.3.2.1: Driver Binding Protocol for PCI Bus Drivers (Page 12-50)

In the first two paragraphs, changed two instances of "PCI Host Bridge I/O Protocol" to "PCI Root Bridge I/O Protocol."

Section 12.4: EFI PCI I/O Protocol (Pages 12-55 to 12-93)

Changed eight instances of "PCI Host Bridge I/O Protocol" to "PCI Root Bridge I/O Protocol."

Section 12.4: EFI PCI I/O Protocol (Page 12-55)

Changed the second bullet in the list FROM:

- A device driver model that abstracts the I/O addresses, Memory addresses, and PCI Configuration addresses from the PCI device driver. Instead, BAR (Base Address Register) relative addressing is used for I/O and Memory accesses, and device relative addressing is used for PCI Configuration accesses.

TO:

- A device driver model that abstracts the I/O addresses, Memory addresses, and PCI Configuration addresses from the PCI device driver. Instead, BAR (Base Address Register) relative addressing is used for I/O and Memory accesses, and device relative addressing is used for PCI Configuration accesses. The BAR relative addressing is specified in the PCI I/O services as a BAR index. A PCI controller may contain a combination of 32-bit and 64-bit BARs. The BAR index represents the logical BAR number in the standard PCI configuration header starting from the first BAR. The BAR index does not represent an offset into the standard PCI Configuration Header because those offsets will vary depending on the combination and order of 32-bit and 64-bit BARs.

Section 12.4: EFI PCI I/O Protocol – Related Definitions (Page 12-60)

Added the following **`#define`** at the end of the list of **`EFI_PCI_IO_ATTRIBUTE`**s:

```
#define EFI_PCI_IO_ATTRIBUTE_DUAL_ADDRESS_CYCLE      0x8000
```

Section 12.4: EFI PCI I/O Protocol – Related Definitions (Page 12-62)

Added the following to the after the description of **EFI_PCI_IO_ATTRIBUTE_EMBEDDED_ROM**:

EFI_PCI_IO_ATTRIBUTE_DUAL_ADDRESS_CYCLE

If this bit is set,, then the PCI controller is capable of producing PCI Dual Address Cycles, so it is able to access a 64-bit address space. If this bit is not set, then the PCI controller is not capable of producing PCI Dual Address Cycles, so it is only able to access a 32-bit address space.

Section 12.4: EFI PCI I/O Protocol – Description – DMA Bus Master Write Operation (Page 12-64)

Changed the "DMA Bus Master Write Operation" section FROM:

- Call **Map()** for **EfiPciOperationBusMasterWrite**
- Program DMA Bus Master with the *DeviceAddress* returned by **Map()**
- Start the DMA Bus Master
- Wait for DMA Bus Master to complete the write operation
- Call **Unmap()**

TO:

- Call **Map()** for **EfiPciOperationBusMasterWrite**.
- Program the DMA Bus Master with the *DeviceAddress* returned by **Map()**.
- Start the DMA Bus Master.
- Wait for DMA Bus Master to complete the write operation.
- Perform a PCI controller specific read transaction to flush all PCI write buffers (See *PCI Specification* Section 3.2.5.2).
- Call **Flush()**.
- Call **Unmap()**.

Section 12.4: EFI_PCI_IO_PROTOCOL.PollMem() – Description (Page 12-66)

Added the following paragraph to the end of the "Description" section:

All the PCI transactions generated by this function are guaranteed to be completed before this function returns. However, if the memory mapped I/O region being accessed by this function has the **EFI_PCI_ATTRIBUTE_MEMORY_CACHED** attribute set, then the transactions will follow the ordering rules defined by the processor architecture.

Section 12.4: EFI_PCI_IO_PROTOCOL.PollIo() – Description (Page 12-68)

Added the following paragraph to the end of the "Description" section:

All the PCI transactions generated by this function are guaranteed to be completed before this function returns.

Section 12.4: EFI_PCI_IO_PROTOCOL.MemRead() – Description (Page 12-70)

Added the following paragraph to the end of the "Description" section:

> All the PCI transactions generated by this function are guaranteed to be completed before this function returns. All the PCI write transactions generated by this function will follow the write ordering and completion rules defined in the *PCI Specification*. However, if the memory-mapped I/O region being accessed by this function has the **EFI_PCI_ATTRIBUTE_MEMORY_CACHED** attribute set, then the transactions will follow the ordering rules defined by the processor architecture.

Section 12.4: EFI_PCI_IO_PROTOCOL.IoRead() – Description (Page 12-72)

Added the following paragraph to the end of the "Description" section:

> All the PCI transactions generated by this function are guaranteed to be completed before this function returns.

Section 12.4: EFI_PCI_IO_PROTOCOL.PciRead() – Description (Page 12-74)

Added the following paragraph to the end of the "Description" section:

> All the PCI transactions generated by this function are guaranteed to be completed before this function returns.

Section 12.4: EFI_PCI_IO_PROTOCOL.CopyMem() – Description (Page 12-76 and 12-77)

Added the following to the end of the "Description" section:

> The contents of the *DestOffset* buffer on exit from this service must match the contents of the *SrcOffset* buffer on entry to this service. Due to potential overlaps, the contents of the *SrcOffset* buffer may be modified by this service. The following rules can be used to guarantee the correct behavior:
>
> 1. If *DestOffset* > *SrcOffset* **and** *DestOffset* < (*SrcOffset* + *Width* size * *Count*), then the data should be copied from the *SrcOffset* buffer to the *DestOffset* buffer starting from the end of buffers and working toward the beginning of the buffers.
> 2. Otherwise, the data should be copied from the *SrcOffset* buffer to the *DestOffset* buffer starting from the beginning of the buffers and working toward the end of the buffers.
>
> All the PCI transactions generated by this function are guaranteed to be completed before this function returns. All the PCI write transactions generated by this function will follow the write ordering and completion rules defined in the *PCI Specification*. However, if the memory mapped I/O region being accessed by this function has the

EFI_PCI_ATTRIBUTE_MEMORY_CACHED attribute set, then the transactions will follow the ordering rules defined by the processor architecture.

Section 12.4: EFI_PCI_IO_PROTOCOL.Map() – Parameters – (Page 12-78)

Changed the description of the *DeviceAddress* parameter FROM:

DeviceAddress	The resulting map address for the bus master PCI controller to use to access the hosts *HostAddress*. Type **EFI_PHYSICAL_ADDRESS** is defined in Chapter 5.

TO:

DeviceAddress	The resulting map address for the bus master PCI controller to use to access the hosts *HostAddress*. Type **EFI_PHYSICAL_ADDRESS** is defined in Chapter 5. This address can not be used by the processor to access the contents of the buffer specified by *HostAddress*.

Section 12.4: EFI_PCI_IO_PROTOCOL.AllocateBuffer() (Page 12-81)

Changed the entire API definition to the following:

EFI_PCI_IO_PROTOCOL.AllocateBuffer()

Summary

Allocates pages that are suitable for an **EfiPciIoOperationBusMasterCommonBuffer** mapping.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PCI_IO_PROTOCOL_ALLOCATE_BUFFER) (
  IN     EFI_PCI_IO_PROTOCOL    *This,
  IN     EFI_ALLOCATE_TYPE      Type,
  IN     EFI_MEMORY_TYPE        MemoryType,
  IN     UINTN                  Pages,
  OUT    VOID                   **HostAddress,
  IN     UINT64                 Attributes
  );
```

Parameters

This	A pointer to the **EFI_PCI_IO_PROTOCOL** instance. Type **EFI_PCI_IO_PROTOCOL** is defined in Section 12.4.
Type	This parameter is not used and must be ignored.

MemoryType	The type of memory to allocate, **EfiBootServicesData** or **EfiRuntimeServicesData**. Type **EFI_MEMORY_TYPE** is defined in Chapter 5.
Pages	The number of pages to allocate.
HostAddress	A pointer to store the base system memory address of the allocated range.
Attributes	The requested bit mask of attributes for the allocated range. Only the attributes **EFI_PCI_ATTRIBUTE_MEMORY_WRITE_COMBINE**, and **EFI_PCI_ATTRIBUTE_MEMORY_CACHED** may be used with this function. If any other bits are set, then **EFI_UNSUPPORTED** is returned. This function may choose to ignore this bit mask. The **EFI_PCI_ATTRIBUTE_MEMORY_WRITE_COMBINE**, and **EFI_PCI_ATTRIBUTE_MEMORY_CACHED** attributes provide a hint to the implementation that may improve the performance of the calling driver. The implementation may choose any default for the memory attributes including write combining, cached, both, or neither as long as the allocated buffer can be seen equally by both the processor and the PCI bus master.

Description

The **AllocateBuffer()** function allocates pages that are suitable for an **EfiPciIoOperationBusMasterCommonBuffer** mapping. This means that the buffer allocated by this function must support simultaneous access by both the processor and a PCI Bus Master. The device address that the PCI Bus Master uses to access the buffer can be retrieved with a call to **Map()**.

If the current attributes of the PCI controller has the **EFI_PCI_IO_ATTRIBUTE_DUAL_ADDRESS_CYCLE** bit set, then when the buffer allocated by this function is mapped with a call to **Map()**, the device address that is returned by **Map()** must be within the 64-bit device address space of the PCI Bus Master. The attributes for a PCI controller can be managed by calling **Attributes()**.

If the current attributes for the PCI controller has the **EFI_PCI_IO_ATTRIBUTE_DUAL_ADDRESS_CYCLE** bit clear, then when the buffer allocated by this function is mapped with a call to **Map()**, the device address that is returned by **Map()** must be within the 32-bit device address space of the PCI Bus Master. The attributes for a PCI controller can be managed by calling **Attributes()**.

If the memory allocation specified by *MemoryType* and *Pages* cannot be satisfied, then **EFI_OUT_OF_RESOURCES** is returned.

Status Codes Returned

EFI_SUCCESS	The requested memory pages were allocated.
EFI_INVALID_PARAMETER	*MemoryType* is invalid.
EFI_INVALID_PARAMETER	*HostAddress* is **NULL**.
EFI_UNSUPPORTED	*Attributes* is unsupported. The only legal attribute bits are **MEMORY_WRITE_COMBINE** and **MEMORY_CACHED**.
EFI_OUT_OF_RESOURCES	The memory pages could not be allocated.

Section 12.4: EFI_PCI_IO_PROTOCOL.Flush() (Page 12-84)

Changed the entire API definition to the following:

EFI_PCI_IO_PROTOCOL.Flush()

Summary

Flushes all PCI posted write transactions from a PCI host bridge to system memory.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PCI_IO_PROTOCOL_FLUSH) (
  IN  EFI_PCI_IO_PROTOCOL  *This
  );
```

Parameters

This	A pointer to the **EFI_PCI_IO_PROTOCOL** instance. Type **EFI_PCI_IO_PROTOCOL** is defined in Section 12.4.

Description

The **Flush()** function flushes any PCI posted write transactions from a PCI host bridge to system memory. Posted write transactions are generated by PCI bus masters when they perform write transactions to target addresses in system memory.

This function does not flush posted write transactions from any PCI bridges. A PCI controller specific action must be taken to guarantee that the posted write transactions have been flushed from the PCI controller and from all the PCI bridges into the PCI host bridge. This is typically done with a PCI read transaction from the PCI controller prior to calling **Flush()**.

If the PCI controller specific action required to flush the PCI posted write transactions has been performed, and this function returns **EFI_SUCCESS**, then the PCI bus master's view and the processor's view of system memory are guaranteed to be coherent. If the PCI posted write transactions cannot be flushed from the PCI host bridge, then the PCI bus master and processor are not guaranteed to have a coherent view of system memory, and **EFI_DEVICE_ERROR** is returned.

Status Codes Returned

EFI_SUCCESS	The PCI posted write transactions were flushed from the PCI host bridge to system memory.
EFI_DEVICE_ERROR	The PCI posted write transactions were not flushed from the PCI host bridge due to a hardware error.

Section 12.4: EFI_PCI_IO_PROTOCOL.Attributes() – Description (Page 12-88)

Added the following paragraph to the end of the "Description" section:

> This function will also return **EFI_UNSUPPORTED** if more than one PCI controller on the same PCI root bridge has already successfully requested one of the ISA addressing attributes. For example, if one PCI VGA controller had already requested the **VGA_IO** and **VGA_MEMORY** attributes, then a second PCI VGA controller on the same root bridge cannot succeed in requesting those same attributes. This restriction applies to the ISA-, VGA-, and IDE-related attributes.

Section 12.4: EFI_PCI_IO_PROTOCOL.GetBarAttributes() – Status Codes Returned (Page 12-91)

Changed the "Status Codes Returned" table FROM:

EFI_SUCCESS	If *Supports* is not **NULL**, then the attributes that the PCI controller supports are returned in *Supports*. If *Resources* is not **NULL**, then the ACPI 2.0 resource descriptors that the PCI controller is currently using are returned in *Resources*.
EFI_OUT_OF_RESOURCES	There are not enough resources available to allocate *Resources*.
EFI_INVALID_PARAMETER	Both *Supports* and *Attributes* are **NULL**.

TO:

EFI_SUCCESS	If *Supports* is not **NULL**, then the attributes that the PCI controller supports are returned in *Supports*. If *Resources* is not **NULL**, then the ACPI 2.0 resource descriptors that the PCI controller is currently using are returned in *Resources*.
EFI_OUT_OF_RESOURCES	There are not enough resources available to allocate *Resources*.
EFI_UNSUPPORTED	*BarIndex* not valid for this PCI controller.
EFI_INVALID_PARAMETER	Both *Supports* and *Attributes* are **NULL**.

Section 13: Protocols – SCSI Bus Support

Additions and Changes

Section 13.1: EFI_SCSI_PASS_THRU Protocol – GUID (Page 13-1)

Changed the "GUID" section FROM:

```
#define EFI_SCSI_PASS_THRU_PROTOCOL_GUID \
{ 0xb881c8f5,0x4c81,0x11d4,0x80,0xb1,0x00,0xc0,0x4f,0x60,0xdf,0xee }
```

TO:

```
#define EFI_SCSI_PASS_THRU_PROTOCOL_GUID \
{ 0xa59e8fcf,0xbda0,0x43bb,0x90,0xb1,0xd3,0x73,0x2e,0xca,0xa8,0x77 }
```

Section 13.1: EFI_SCSI_PASS_THRU_PROTOCOL.PassThru() – Related Definitions – TransferLength (Page 13-6)

Changed the description of *TransferLength* FROM:

> On input, the size, in bytes, of *DataBuffer*. On output, the number of bytes transferred between the SCSI controller and the SCSI device. If *TransferLength* is larger than the SCSI controller can handle, then the SCSI controller will transfer its maximum amount, and will update *TransferLength* with the number of bytes actually transferred.

TO:

> On input, the size, in bytes, of *DataBuffer*. On output, the number of bytes transferred between the SCSI controller and the SCSI device. If *TransferLength* is larger than the SCSI controller can handle, no data will be transferred, *TransferLength* will be updated to contain the number of bytes that the SCSI controller is able to transfer, and **EFI_BAD_BUFFER_SIZE** will be returned.

Section 13.1: EFI_SCSI_PASS_THRU_PROTOCOL.PassThru() – Description (Page 13-8)

Changed the seventh paragraph FROM:

> If the data buffer described by *DataBuffer* and *TransferLength* is too big to be transferred in a single command, then **EFI_WARN_BUFFER_TOO_SMALL** is returned. The number of bytes actually transferred is returned in *TransferLength.*

TO:

> If the data buffer described by *DataBuffer* and *TransferLength* is too big to be transferred in a single command, then no data is transferred and **EFI_BAD_BUFFER_SIZE** is returned. The number of bytes that can be transferred in a single command are returned in *TransferLength.*

Section 13.1: EFI_SCSI_PASS_THRU_PROTOCOL.PassThru() – Status Codes Returned (Page 13-9)

Changed the second return code FROM:

EFI_WARN_BUFFER_TOO_SMALL	The SCSI Request Packet was executed, but the entire *DataBuffer* could not be transferred. The actual number of bytes transferred is returned in *TransferLength*. See *HostAdapterStatus*, *TargetStatus*, *SenseDataLength*, and *SenseData* in that order for additional status information.

TO:

EFI_BAD_BUFFER_SIZE	The SCSI Request Packet was not executed. The number of bytes that could be transferred is returned in *TransferLength*. See *HostAdapterStatus*, *TargetStatus*, *SenseDataLength*, and *SenseData* in that order for additional status information.

Clarifications and Corrections

None.

Section 14: Protocols – USB Support

Additions and Changes

None.

Clarifications and Corrections

Section 14.1: EFI_USB_HC_PROTOCOL.SetState() – Description (Page 14-8)

Made the following two changes in the "Description" section:

1. Changed the second sentence of the first paragraph in the "Description" section FROM:

   There are three states defined states for the USB host controller.

   TO:

   There are three states defined for the USB host controller.

2. Changed the second sentence of the second paragraph in the "Description" section FROM:

   If a device error occurs while attempting the place the USB host controller into the state specified by *State*, then **EFI_DEVICE_ERROR** is returned.

   TO:

   If a device error occurs while attempting to place the USB host controller into the state specified by *State*, then **EFI_DEVICE_ERROR** is returned.

Section 14.1: EFI_USB_HC_PROTOCOL.ControlTransfer() – Description (Page 14-11)

Made the following two changes in the "Description" section:

1. Changed the first sentence of the fourth paragraph FROM:

   EFI_INVALID_PARAMETER is returned is one of the following conditions are satisfied:

   TO:

   EFI_INVALID_PARAMETER is returned if one of the following conditions is satisfied:

2. In the numbered list in the fourth paragraph of the "Description" section, changed item #4 FROM:

   MaxiumPacketLenth is not valid. If *IsSlowDevice* is **TRUE**, then *MaximumPacketLength* must be 1, 2, 4 or 8. If *IsSlowDevice* is **FALSE**, then *MaximumPacketLength* must be 1, 2, 4, 8, 16, 32, or 64.

   TO:

   MaximumPacketLength is not valid. If *IsSlowDevice* is **TRUE**, then *MaximumPacketLength* must be 8. If *IsSlowDevice* is **FALSE**, then *MaximumPacketLength* must be 8, 16, 32, or 64.

Section 14.1: EFI_USB_HC_PROTOCOL.BulkTransfer() – Description (Page 14-14)

Made the following two changes in the "Description" section:

1. Changed the first sentence of the fifth paragraph of the "Description" section FROM:

   EFI_INVALID_PARAMETER is returned if one of the following conditions are satisfied:

   TO:

   EFI_INVALID_PARAMETER is returned if one of the following conditions is satisfied:

2. In the numbered list in the fifth paragraph of the "Description" section, changed item #3 FROM:

   MaxiumPacketLenth is not valid. The legal value of this parameter is: 1, 2, 4, 8, 16, 32, or 64.

   TO:

   MaximumPacketLength is not valid. The legal value of this parameter is 8, 16, 32, or 64.

Section 14.1: EFI_USB_HC_PROTOCOL.AsyncInterruptTransfer() – Description (Page 14-18)

Changed the first sentence of the fifth paragraph in the "Description" section FROM:

> **EFI_INVALID_PARAMETER** is returned is one of the following conditions are satisfied:

TO:

> **EFI_INVALID_PARAMETER** is returned if one of the following conditions is satisfied:

Section 14.1: EFI_USB_HC_PROTOCOL.SyncInterruptTransfer() – Description (Page 14-20)

Made the following two changes in the "Description" section:

1. Changed the first sentence of the fourth paragraph in the "Description" section FROM:

   > **EFI_INVALID_PARAMETER** is returned is one of the following conditions are satisfied:

   TO:

   > **EFI_INVALID_PARAMETER** is returned if one of the following conditions is satisfied:

2. In the numbered list in the fourth paragraph of the "Description" section, changed item #4 FROM:

   > *MaxiumPacketLenth* is not valid. The legal value of this parameter is: for the full-speed device, it should be 1, 2, 4, 8, 16, 32, or 64; for the slow device, it is limited to 1, 2, 4, or 8.

   TO:

   > *MaximumPacketLength* is not valid. The legal value of this parameter is: for the full-speed device, it should be 8, 16, 32, or 64; for the slow device, it is limited to 8.

Section 14.1: EFI_USB_HC_PROTOCOL.IsochronousTransfer() – Description (Page 14-22)

Changed the first sentence of the third paragraph of the “Description” section FROM:

> **EFI_INVALID_PARAMETER** is returned is one of the following conditions are satisfied:

TO:

> **EFI_INVALID_PARAMETER** is returned if one of the following conditions is satisfied:

Section 14.1: EFI_USB_HC_PROTOCOL.AsyncIsochronousTransfer() – Description (Page 14-24)

Changed the first sentence of the third paragraph on the “Description” section FROM:

> **EFI_INVALID_PARAMETER** is returned is one of the following conditions are satisfied:

TO:

> **EFI_INVALID_PARAMETER** is returned if one of the following conditions is satisfied:

Section 15: Protocols – Network Support

Additions and Changes

None.

Clarifications and Corrections

Section 15.3: EFI_PXE_BASE_CODE Protocol – Related Definitions – StationIp (Page 15-34)

Added the following sentence to the end of the *StationIp* field description:

> This field must be set to a valid IP address by either **Dhcp()** or **SetStationIp()** before the **Discover()**, **Mtftp()**, **UdpRead()**, **UdpWrite()**, or **Arp()** functions are called.

Section 15.3: EFI_PXE_BASE_CODE Protocol – Related Definitions – SubnetMask (Page 15-34)

Added the following sentence to the end of the *SubnetMask* field description:

> This field must be set to a valid subnet mask by either **Dhcp()** or **SetStationIp()** before the **Discover()**, **Mtftp()**, **UdpRead()**, **UdpWrite()**, or **Arp()** functions are called.

Section 15.3: EFI_PXE_BASE_CODE.Mtftp() – Parameters – BufferSize (Page 15-50)

Replaced the following *BufferSize* parameter description:

> For read-file and write-file operations, this is the size of the buffer specified by *BufferPtr*. For read-file operations, if **BufferSize* is smaller than the size of the file being read, then this field will return the required size. For get-file-size operations, this field returns the size of the requested file.

WITH:

> For get-file-size operations, **BufferSize* returns the size of the requested file. For read-file and write-file operations, this parameter is set to the size of the buffer specified by the *BufferPtr* parameter. For read-file operations, if **EFI_BUFFER_TOO_SMALL** is returned, **BufferSize* returns the size of the requested file.

Section 15.3: EFI_PXE_BASE_CODE.Mtftp() – Description (Page 15-52)

Added the following at the end of the second paragraph in the "Description" section:

> Applications using the **PxeBc.Mtftp()** services should use the get-file-size operations to determine the size of the downloaded file prior to using the read-file operations—especially when downloading large (greater than 64 MB) files—instead of making two calls to the read-file operation. Following this recommendation will save time if the file is larger than expected and the TFTP server does not support TFTP option extensions. Without TFTP option extension support, the client has to download the entire file, counting and discarding the received packets, to determine the file size.

Section 15.3: EFI_PXE_BASE_CODE.SetParameters() – Parameters – NewSendGUID (Page 15-61)

Added the following sentence to the end of the *NewSendGUID* parameter description:

> If *NewSendGUID* is **TRUE** and there is no SystemGUID, then **EFI_INVALID_PARAMETER** is returned.

Section 16: Protocols – Debugger Support

Additions and Changes

None.

Clarifications and Corrections

Section 16.2.1: EFI_DEBUG_SUPPORT_PROTOCOL – Protocol Interface Structure (Page 16-3)

Changed the first line of the "Protocol Interface Structure" section FROM:

```
typedef struct _EFI_DEBUG_SUPPORT_PROTOCOL {
```

TO:

```
typedef struct {
```

Section 16.2.1: EFI_DEBUG_SUPPORT_PROTOCOL.RegisterPeriodicCallback() – Related Definitions – EFI_SYSTEM_CONTEXT_EBC (Page 16-6)

Changed the **EFI_SYSTEM_CONTEXT_EBC** structure FROM:

```
typedef struct {
  UINT64              R0, R1, R2, R3, R4, R5, R6, R7;
  UINT64              Flags;
  UINT64              ControlFlags;
  UINTN               Ip;
} EFI_SYSTEM_CONTEXT_EBC;
```

TO:

```
typedef struct {
  UINT64              R0, R1, R2, R3, R4, R5, R6, R7;
  UINT64              Flags;
  UINT64              ControlFlags;
  UINT64              Ip;
} EFI_SYSTEM_CONTEXT_EBC;
```

Section 16.2.1: EFI_DEBUG_SUPPORT_PROTOCOL.RegisterPeriodicCallback() – Related Definitions – EFI_SYSTEM_CONTEXT_IA32 (Page 16-7)

Changed the **EFI_SYSTEM_CONTEXT_IA32** structure FROM:

```
typedef struct {
  UINT32              ExceptionData;  // ExceptionData is
                                      // additional data pushed
                                      // on the stack by some
                                      // types of IA32 exceptions
  EFI_FXSAVE_STATE    FxSaveState;
  UINT32              Dr0, Dr1, Dr2, Dr3, Dr6, Dr7;
  UINT32              Cr0, Cr1, Cr2, Cr3, Cr4;
  UINT32              Eflags;
  UINT32              Ldtr, Tr;
  UINT64              Gdtr, Idtr;
  UINT32              Eip;
  UINT32              Gs, Fs, Es, Ds, Cs, Ss;
  UINT32              Edi, Esi, Ebp, Esp, Ebx, Edx, Ecx, Eax;
} EFI_SYSTEM_CONTEXT_IA32;
```

TO:

```
typedef struct {
  UINT32              ExceptionData;  // ExceptionData is
                                      // additional data pushed
                                      // on the stack by some
                                      // types of IA32 exceptions
  EFI_FXSAVE_STATE    FxSaveState;
  UINT32              Dr0, Dr1, Dr2, Dr3, Dr6, Dr7;
  UINT32              Cr0, Cr1 /* Reserved */, Cr2, Cr3, Cr4;
  UINT32              Eflags;
  UINT32              Ldtr, Tr;
  UINT32              Gdtr[2], Idtr[2];
  UINT32              Eip;
  UINT32              Gs, Fs, Es, Ds, Cs, Ss;
  UINT32              Edi, Esi, Ebp, Esp, Ebx, Edx, Ecx, Eax;
} EFI_SYSTEM_CONTEXT_IA32;
```

Section 16.3.1: EFI_DEBUGPORT_PROTOCOL – Protocol Interface Structure (Page 16-15)

Changed the first line of the "Protocol Interface Structure" section FROM:

```
typedef struct _EFI_DEBUG_PORT_PROTOCOL {
```

TO:

```
typedef struct {
```

Section 16.3.3: EFI Debugport Variable (Page 16-21)

Changed the **#define** statements between the second and third paragraphs FROM:

```
#define EFI_DEBUGPORT_ENV_VAR_GUID EFI_DEBUGPORT_PROTOCOL_GUID
```

TO:

```
#define EFI_DEBUGPORT_VARIABLE_NAME L"DEBUGPORT"
#define EFI_DEBUGPORT_VARIABLE_GUID EFI_DEBUGPORT_PROTOCOL_GUID
```

Section 16.4.3: EFI Image Info (Page 16-24)

Changed the GUID after the second paragraph FROM:

```
#define EFI_DEBUG_IMAGE_INFO_TABLE_GUID    \
{ 0x49152e77,0x1ada,0x4764,0xb7,0xa2,0x7a,0xfe,0xfe,0xd9,0x5e,0x8b }
```

TO:

```
#define EFI_DEBUG_IMAGE_INFO_TABLE_GUID    \
{ 0x49152E77,0x1ADA,0x4764,0xB7,0xA2,0x7A,0xFE,0xFE,0xD9,0x5E,0x8B }
```

Section 16.4.3: EFI Image Info (Page 16-25)

Made the following changes on page 16-25:

1. Replaced the three instances of **UPDATE_STATUS_UPDATE_IN_PROGRESS** with **EFI_DEBUG_IMAGE_INFO_UPDATE_IN_PROGRESS**.

2. Replaced the two instances of **UPDATE_STATUS_TABLE_MODIFIED** with **EFI_DEBUG_IMAGE_INFO_TABLE_MODIFIED**.

3. Changed the first line of the **EFI_DEBUG_IMAGE_INFO_TABLE_HEADER** structure FROM:

```
typedef struct _EFI_DEBUG_IMAGE_INFO_TABLE_HEADER {
```

TO:

```
typedef struct {
```

4. Changed the **EFI_DEBUG_IMAGE_INFO_NORMAL** structure FROM:

```
typedef struct _EFI_DEBUG_IMAGE_INFO_NORMAL {
  UINT32                        ImageInfoType;
  EFI_LOADED_IMAGE              *LoadedImageProtocolInstance;
  EFI_HANDLE                    ImageHandle;
} EFI_DEBUG_IMAGE_INFO_NORMAL;
```

TO:

```
typedef struct {
  UINT32                        ImageInfoType;
  EFI_LOADED_IMAGE_PROTOCOL     *LoadedImageProtocolInstance;
  EFI_HANDLE                    ImageHandle;
} EFI_DEBUG_IMAGE_INFO_NORMAL;
```

Section 18: Protocols – Device I/O Protocol

Additions and Changes

None.

Clarifications and Corrections

Section 18: First text paragraph (Page 18-1)

Added the following paragraph after the first paragraph of the chapter:

> The services defined in this chapter have been superceded by the services described in Chapter 12. Both the PCI Root Bridge I/O Protocol and the PCI I/O Protocol provide a more complete set of services for managing PCI devices. The PCI Root Bridge I/O Protocol and PCI I/O Protocol are not defined in the *EFI 1.02 Specification*. If an EFI image is required to be compliant with the *EFI 1.02 Specification*, then the Device I/O Protocol is the only option for these types of I/O services. If an EFI image is required to be compliant with the *EFI 1.10 Specification*, then the PCI Root Bridge I/O Protocol or the PCI I/O Protocol must be used for these types of I/O services.

Section 19: EFI Byte Code Virtual Machine

Additions and Changes

None.

Clarifications and Corrections

Section 19.2: Memory Ordering (Page 19-3)

Inserted the following section between Section 19.1.6 and what is now Section 19.3, "Virtual Machine Registers." This insertion bumped all the remaining section numbers:

Section 19.2: Memory Ordering

The term memory ordering refers to the order in which a processor issues reads (loads) and writes (stores) out onto the bus to system memory. The EBC Virtual Machine enforces strong memory ordering, where reads and writes are issued on the system bus in the order they occur in the instruction stream under all circumstances.

Appendix E: 32/64-Bit UNDI Specification

Additions and Changes

None.

Clarifications and Corrections

Section E.4.7: Initialize (Page E-56)

Added the following sentence to the end of the third paragraph:

> Control will not be returned to the caller and the **COMMAND_COMPLETE** status flag will not be set until the NIC is ready to transmit.

Section E.4.8: Reset (Page E-60)

Added the following sentence to the end of the second paragraph:

> Control will not be returned to the caller and the **COMMAND_COMPLETE** status flag will not be set until the NIC is ready to transmit.

Section E.4.11: Receive Filters (Page E-66)

Added the following sentence to the end of the first paragraph:

> Control will not be returned to the caller and the **COMMAND_COMPLETE** status flag will not be set until the NIC is ready to receive.

Appendix H: Compression Source Code

Additions and Changes

Appendix H: Compression Source Code – AllocateMemory() (Page H-8, Line 10)

Changed line ten on page H-8 FROM:

```
mText = malloc (WNDSIZ * 2 + MAXMATCH);
```

TO:

```
UINT32      i;


mText = malloc (WNDSIZ * 2 + MAXMATCH);
for (i = 0; i < WNDSIZ * 2 + MAXMATCH; i ++) {
  mText[i] = 0;
}
```

Clarifications and Corrections

None.

Glossary

Additions and Changes

None.

Clarifications and Corrections

Glossary (Page Glossary-11)

Removed the entry for “PCI Host Bridge I/O Protocol.”