Intel® Platform Innovation Framework for EFI System Management Mode Core Interface Specification (SMM CIS)

Version 0.9a

November 19, 2003

Intel order number xxxxxx-001

Revision History

Revision	Revision History	Date
0.9	First public release.	9/16/03
0.9a	Add additional bit fields to the RegionState in the SMRAM Descriptor data structure	11/19/03

Contents

Figures

Tables

1 Introduction

Overview

This specification defines the core code and services that are required for an implementation of the System Management Mode (SMM) phase of the Intel® Platform Innovation Framework for EFI (hereafter referred to as the "Framework"). This SMM Core Interface Specification (CIS) does the following:

- Describes the basic components of SMM
- Provides code definitions for services and functions that are architecturally required by the *Intel® Platform Innovation Framework for EFI Architecture Specification*
- Describes the interactions between SMM and other phases in the Framework
- Describes processor-specific details in SMM mode for IA-32 and Intel® Itanium® processors

See Organization of the SMM CIS for more information.

Rationale

Certain artifacts of the hardware and platform design require programmatic workarounds. This interface design aims to provide a clean mechanism for installing these modules. Possible candidates include the following:

- ACPI S3 reserve handler
- Enable/disable ACPI mode
- Power button support while not in ACPI mode
- Error logging for ECC/PERR/SERR in IA-32
- Protected flash writes on some IA-32 platforms
- Century rollover bug workaround

Organization of the SMM CIS

This SMM Core Interface Specification (CIS) is organized as listed in the table below. Because the SMM is just one component of a Framework-based firmware solution, there are a number of additional specifications that are referred to throughout this document.

- For references to other Framework specifications, click on the hyperlink in the page or navigate through the table of contents (TOC) in the left navigation pane to view the referenced specification.
- For references to non-Framework specifications, see References in the Interoperability and Component Specifications help system.

Table 1-1. Organization of the SMM CIS

Book	Description
Overview	Describes the major components of SMM, including the System Management System Table (SMST), System Management Library (SMLib) services, and SMM protocols.
System Management System Table (SMST)	Defines the data structure that provides access to the services that can be used while in SMM.
Services - SMM	Defines a set of standard functions that a conformant SMM implementation will publish for use by SMM drivers.
Services - SMM Library (SMLib)	Defines a set of worker routines that are usable by a large class of drivers in SMM.
SMM Protocols	Defines a series of protocols that abstract the loading of DXE drivers into SMM, manipulation of the System Management RAM (SMRAM) apertures, and generation of System Management Interrupts (SMIs).
SMM Child Dispatch Protocols	Defines a series of protocols that abstract installation of handlers for a chipset-specific SMM design.
Interactions with PEI, DXE, and BDS	Describes issues related to image verification and interactions between SMM and other Framework phases.
Appendix	Provides additional definitions of nonarchitectural SMM protocols and processor-specific information for IA-32 and Itanium® processors.

Conventions Used in This Document

This document uses the typographic and illustrative conventions described below.

Data Structure Descriptions

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

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

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

STRUCTURE NAME:	The formal name of the data structure.
Summary:	A brief description of the data structure.
Prototype:	A “C-style” type declaration for the data structure.
Parameters:	A brief description of each field in the data structure prototype.
Description:	A description of the functionality provided by the data structure, including any limitations and caveats of which the caller should be aware.
Related Definitions:	The type declarations and constants that are used only by this data structure.

Protocol Descriptions

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

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

Procedure Descriptions

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

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

Pseudo-Code Conventions

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

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

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

Typographic Conventions

This document uses the typographic and illustrative conventions described below:

Plain text	The normal text typeface is used for the vast majority of the descriptive text in a specification.
Plain text (blue)	In the online help version of this specification, any plain text that is underlined and in blue indicates an active link to the cross-reference. Click on the word to follow the hyperlink. Note that these links are *not* active in the PDF of the specification.
Bold	In text, a **Bold** typeface identifies a processor register name. In other instances, a **Bold** typeface can be used as a running head within a paragraph.
Italic	In text, an *Italic* typeface can be used as emphasis to introduce a new term or to indicate a manual or specification name.
`BOLD Monospace`	Computer code, example code segments, and all prototype code segments use a **`BOLD Monospace`** typeface with a dark red color. These code listings normally appear in one or more separate paragraphs, though words or segments can also be embedded in a normal text paragraph.
`Bold Monospace`	In the online help version of this specification, words in a **`Bold Monospace`** typeface that is underlined and in blue indicate an active hyperlink to the code definition for that function or type definition. Click on the word to follow the hyperlink. Note that these links are *not* active in the PDF of the specification. Also, these inactive links in the PDF may instead have a **`Bold Monospace`** appearance that is underlined but in dark red. Again, these links are not active in the PDF of the specification.
`Italic Monospace`	In code or in text, words in *`Italic Monospace`* indicate placeholder names for variable information that must be supplied (i.e., arguments).
`Plain Monospace`	In code, words in a `Plain Monospace` typeface that is a dark red color but is not bold or italicized indicate pseudo code or example code. These code segments typically occur in one or more separate paragraphs.

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

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

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

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

2 Overview

Definition of Terms

The following terms are used in the SMM Core Interface Specification (CIS). See Glossary in the master help system for additional definitions.

Communicate

Intermodule communication. Mechanism for posting data to an SMM handler. See `EFI_SMM_BASE_PROTOCOL.Communicate()`.

C-SEG

Compatibility Segment. SMRAM that is located at address 0xA0000 through 0xBFFFF, which is the location of the VGA frame buffer, below the 1 MB address.

GMCH

Graphics Memory Controller Hub.

HMAC

Hashed Message Authentication Code.

H-SEG

High Segment. SMRAM that is the same physical memory as C-SEG (i.e., frame buffer) but is remapped by the chipset to appear to the processor at address 0xFEFFA0000 through 0xFEFFBFFF.

ICH

I/O Controller Hub.

IP

Instruction pointer.

IPI

Interprocessor Interrupt. This interrupt is the means by which multiple processors in a system or a single processor can issue APIC-directed messages for communicating with self or other processors.

MCH

Memory Controller Hub.

MTRR

Memory Type Range Register.

PMI

Platform Management Interrupt. Maskable, level-activated interrupt on the Itanium® processor family.

RSM

Resume. On IA-32, processor instruction to exit from System Management Mode (SMM).

SBE

Single-Bit Error.

SMI

System Management Interrupt. Nonmaskable interrupt on IA-32 processors that evolves the process to SMM.

SMM

System Management Mode. A processor mode on IA-32 processors, in addition to the following modes: real, protected, and V86.

SMM handler

A DXE runtime driver that has relocated itself into SMRAM via the `EFI_SMM_BASE_PROTOCOL.Register()` function.

SMST

System Management System Table. Hand-off to handler.

T-SEG

Top Segment. It is physical memory that is reserved for SMRAM at the top of physical memory below 4 GB. The physical start and processor view of this memory are identical.

System Management Mode (SMM)

System Management Mode (SMM) on IA-32 processors is a mode of operation that is distinct from the flat-model, protected-mode operation of the Driver Execution Environment (DXE) and Pre-EFI Initialization (PEI) phases. SMM is defined to be a real-mode environment with 32-bit data access and is activated in response to an interrupt type or using the System Management Interrupt (SMI) pin. The interesting point about SMM is that it is an OS-transparent mode of operation and is a distinct operational mode. It can coexist within an OS runtime.

The Framework SMM design provides a mechanism to load DXE runtime drivers into SMM. The SMM infrastructure code will be loaded by an Boot Service driver and then does the following:

- Prepares an execution environment that relocates itself to the appropriate SMRAM location.
- Trampolines into flat-model protected mode.
- Supports receiving image loading requests from Boot Service agents. The SMM infrastructure code also supports receiving messages from both Boot Service and Runtime agents.

The implementation of the SMM phase is more dependent on the processor architecture than any other phase.

The figure below shows the Framework SMM architecture.

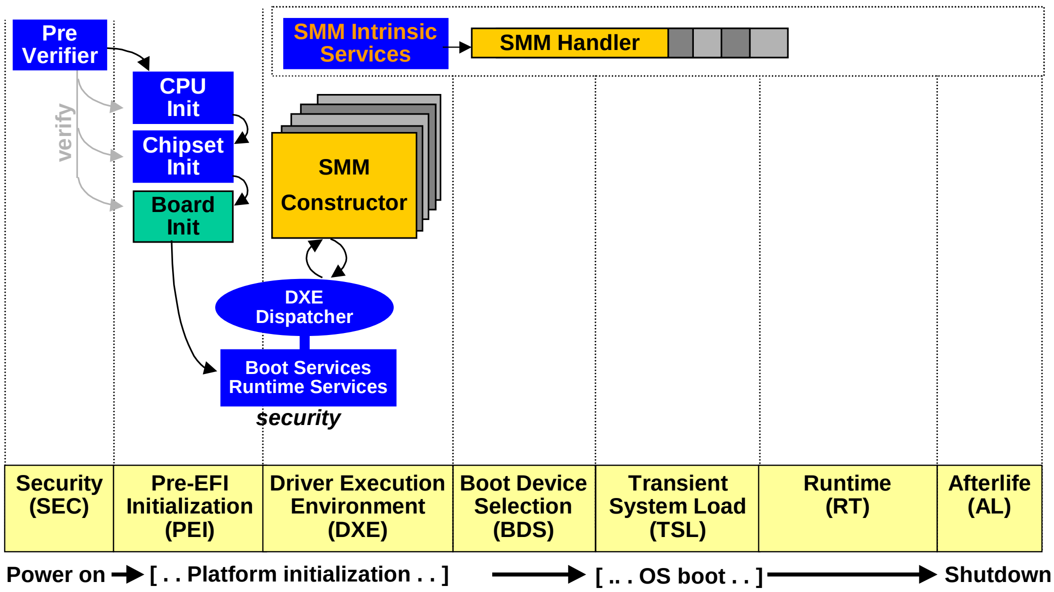

Figure 2-1. Framework SMM Architecture

SMM on the Itanium® Processor Family

Similarly, for the Itanium® processor family, there is a mode of firmware operation that is invoked by the Platform Management Interrupt (PMI). The firmware, in response to the PMI pin or interrupt type, will gain control in physical mode.

This physical mode of operation is not a unique processor mode as SMM is on IA-32, but for purposes of this description, "SMM" will be used to describe the operational regime for both IA-32 and Itanium processors. The characteristic that PMI-based firmware on Itanium processors and SMI-based firmware on IA-32 share is the OS-transparency.

System Management System Table (SMST)

The chief mechanism for passing information and enabling activity in the SMM handler is the System Management System Table (SMST).

This table provides access to the SMST-based services, called SMM Services, which drivers can use while executing within the SMM context. The address of the SMST can be ascertained from the `EFI_SMM_BASE_PROTOCOL.GetSmstLocation()` service.

SMM Services

SMM and Available Services

There are two types of services available during SMM:

- SMM Services
- SMM Library (SMLib) Services

NOTE

The SMM architecture does not support the execution of handlers written to the EFI Byte Code (EBC) specification.

SMM Services

The model of SMM in the Framework will have constraints similar to those of EFI runtime drivers. Specifically, the dispatch of drivers in SMM will not be able to use core protocol services. There will be SMST-based services, called SMM Services, that the drivers can access using an SMM equivalent of the EFI System Table, but the core protocol services will not necessarily be available during runtime.

Instead, the full collection of EFI Boot Services and EFI Runtime Services are available only during the driver load or "constructor" phase. This constructor visibility is useful in that the SMM driver can leverage the rich set of EFI services to do the following:

- Marshall interfaces to other EFI services.
- Discover EFI protocols that are published by peer SMM drivers during their constructor phases.

This design makes the EFI protocol database useful to these drivers while outside of SMM and during their initial load within SMM.

The SMST-based services that are available include the following:

- A minimal, blocking variant of the device I/O protocol
- A memory allocator from SMM memory

These services are exposed by entries in the System Management System Table (SMST).

SMM Library (SMLib) Services

Additional services in the SMM Library (SMLib) are exposed as conventional EFI protocols that are located during the constructor phase of the SMM driver in SMM. For example, the status code equivalent in SMM is simply an EFI protocol whose interface references an SMM-based driver's service. Other SMM drivers locate this SMM-based status code and can use it during runtime to emit error or progress information.

SMM Drivers

Loading Drivers into SMM

The model for loading drivers into SMM is that the DXE SMM runtime driver will have a dependency expression that includes at least the **`EFI_SMM_BASE_PROTOCOL`**. This dependency is necessary because the DXE runtime driver that is intended for SMM will use the **`EFI_SMM_BASE_PROTOCOL`** to reload itself into SMM and rerun its entry point in SMM. In addition, other SMM-loaded protocols can be placed in the dependency expression of a given SMM DXE runtime driver. The logic of the DXE Dispatcher—namely, checking if the GUIDs for the protocols are present in the protocol database—can then be used to determine if the driver can be loaded.

Once loaded into SMM, the DXE SMM runtime driver can use a very limited set of services. The driver can use EFI Boot Services while in its constructor entry point that runs in the boot service space and SMM. In this second entry point in SMM, the driver can do several things:

- Register an interface in the conventional protocol database to name the SMM-resident interfaces to future-loaded SMM drivers
- Register with the SMM infrastructure code for a callback in response to an SMI-pin activation or an SMI-based message from a boot service or runtime agent (i.e., outside-of-SMM code)

After this "constructor" phase in SMM, however, the environmental constraints are the same as other runtime drivers. Specifically, the SMM driver should not rely upon any other boot services because the operational mode of execution can migrate away from these services (the **`ExitBootServices()`** call is asynchronous to invoking the SMM infrastructure code). Several EFI Runtime Services can have the bulk of their processing migrated into SMM, and the runtime-visible portion would simply be a proxy that uses the **`EFI_SMM_BASE_PROTOCOL`** to "thunk" or call back into SMM to implement the services. Having a proxy allows for a model of sharing error-handling code, such as flash access services, with runtime code, such as the EFI Runtime Services **`GetVariable()`** or **`SetVariable()`**.

IA-32 SMM Drivers

The IA-32 runtime drivers are not callable from SMM because of the **`SetVirtualAddress()`** action that is taken upon the image. As such, code that needs to be shared between SMM and EFI runtime should migrate into SMM.

Itanium® Processor Family SMM Drivers

The runtime drivers for the Itanium® processor family are callable from a Platform Management Interrupt (PMI) because each is a variant of a position-independent code (PIC) runtime driver.

SMM Protocols

SMM Protocols

The system architecture of the SMM driver is broken into the following two pieces:

- SMM Base Protocol
- SMM Access Protocol

The SMM Base Protocol will be published by a processor driver and is responsible for the following:

- Initializing the processor state
- Registering the handlers

The SMM Access Protocol understands the particular enable and locking mechanisms that an IA-32 memory controller might support while executing in SMM. For the Itanium® processor family, the SMM Access Protocol is not needed because the PMI does not engender a unique processor execution mode. As a result, there is no possibility of the memory complex having any modal behavior.

The following topics show the SMM protocols that are published for IA-32 and Itanium®-based systems.

SMM Protocols for IA-32

The following figure shows the SMM protocols that are published for an IA-32 system.

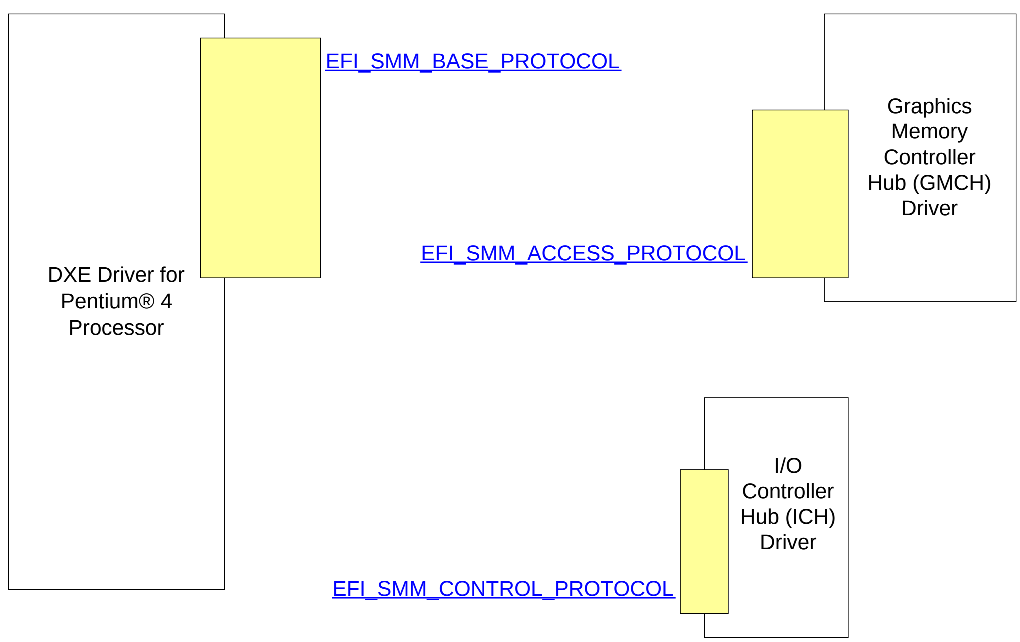

Figure 2-2. Published Protocols for IA-32 Systems

SMM Protocols for Itanium®-Based Systems

The following figure shows the SMM protocols that are published for Itanium®-based systems.

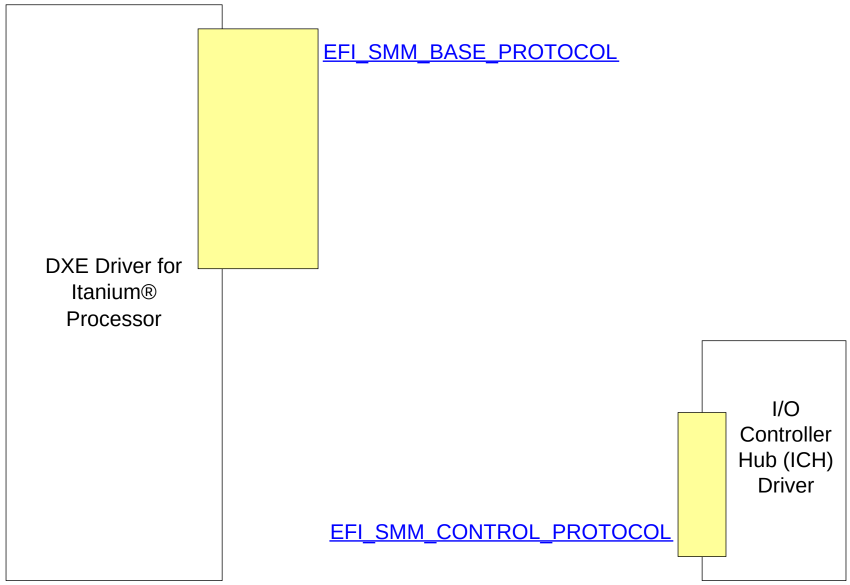

Figure 2-3. Published Protocols for Itanium-Based Systems

SMM Infrastructure Code and Dispatcher

SMM Infrastructure Code and Dispatcher

The SMM infrastructure code centers around the SMM Dispatcher. The SMM Dispatcher's job is to hand control to the SMM handlers in an orderly manner. The SMM infrastructure code also assists in SMM-to-SMM communication. The SMM handles are PE32+ images that have an image type of **EFI_IMAGE_SUBSYSTEM_EFI_RUNTIME_DRIVER**.

See System Management System Table (SMST) for more information on the mechanism for passing information and enabling activity in the SMM handler.

Initializing the SMM Phase

Initializing the SMM Phase

The SMM driver for the Framework is essentially a registration vehicle for dispatching drivers in response to the following:

- IA-32: System Management Interrupts (SMIs)
- Itanium® processor family: Platform Management Interrupts (PMIs)

Throughout this specification, the term *platform management* is synonymous with *system management* to avoid using "xMI" and "xMM" monikers.

The figure below shows the relationship of System Management RAM (SMRAM) to main memory in IA-32.

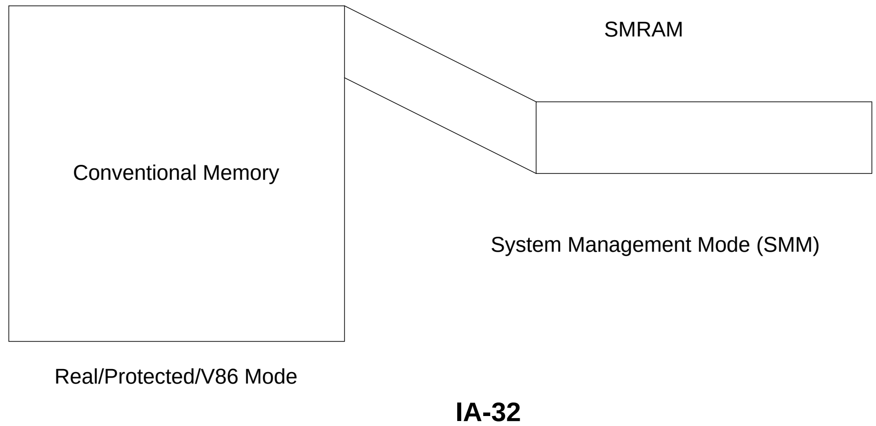

Figure 2-4. SMRAM Relationship to Main Memory

Processor Execution Mode

SMM is entered asynchronously to the main program flow. SMM was originally designed to be opaque to the operating system and provide a transparent power management facility.

In IA-32, SMM is a processor operating mode in the same fashion as V86, real mode, and protected mode. With power-management policy beyond the Advanced Power Management (APM) era, such as ACPI, the original intent of the processor mode became less important. However, in the interim period, additional uses of SMM have been introduced. These alternate uses of SMM that are initiated by preboot agents include the following:

- Workarounds for chipset errata
- Error logging
- Platform security

A System Management Interrupt (SMI) can be entered by activating the SMI logic pin on the baseboard or using the local APIC.

In Itanium® architecture, there is no distinguished processor mode for the manageability interruption. Instead, the processor supports a Platform Management Interrupt (PMI), which is a maskable interruption. PMI can also be entered using a message on the local Streamlined Advanced Programmable Interrupt Controller (SAPIC).

This architecture describes a mechanism for loading the modules of required code that embody the functionality mentioned above. The instantiation of the protocol that supports the loading of the handler images runs in normal boot-services memory. It is only the handler images that need to run in the System Management Random Access Memory (SMRAM). See SMM Protocols earlier in this Overview chapter for more information on SMM protocols.

Access to Platform Resources

As a policy decision, the execution of SMM handlers is logically precluded from accessing conventional memory resources. As such, there is no easy binding mechanism through a call or trap interface to leverage services in the preempted, non-SMM state.

However, there is a library of services, the SMM Services, that supports a subset of the core EFI services, such as memory allocation, the Device I/O Protocol, and others. The SMM driver execution mode has the same structure as the EFI baseline—namely a component that executes in boot services mode and that can possibly execute in runtime. The latter mechanism happens using an unregister event when **`ExitBootServices()`** is invoked.

3
System Management System Table (SMST)

Introduction

This section describes the System Management System Table (SMST). The SMST is a set of capabilities exported for use by all drivers that are loaded into System management RAM (SMRAM).

The SMST is akin to the Boot Services table in the EFI System Table in that it is a fixed set of services and data, by design, and does not admit to the extensibility of an EFI protocol interface. The SMST is provided by the Framework's SMM infrastructure component, which also manages the following:

- Dispatch of drivers in SMM
- Allocations of SMRAM
- Transitioning the Framework into and out of the respective system management mode of the processor

SMM Handler Entry Point

EFI_SMM_HANDLER_ENTRY_POINT

Summary

This function is the main entry point for an SMM handler dispatch or communicate-based callback. An SMM handler is a DXE runtime driver that has relocated itself into SMRAM via the **EFI_SMM_BASE_PROTOCOL.Register()** service.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMM_HANDLER_ENTRY_POINT) (
  IN EFI_HANDLE                 SmmImageHandle,
  IN OUT VOID                   *CommunicationBuffer OPTIONAL,
  IN OUT UINTN                  *SourceSize OPTIONAL
);
```

Parameters

SmmImageHandle

A unique value returned by the SMM infrastructure in response to registration for a communicate-based callback or dispatch. Type **EFI_HANDLE** is defined in the **InstallProtocolInterface()** function definition in the *EFI 1.10 Specification.*

CommunicationBuffer

An optional buffer that will be populated by the SMM infrastructure in response to a non-SMM agent (preboot or runtime) invoking the **EFI_SMM_BASE_PROTOCOL.Communicate()** service.

SourceSize

If *CommunicationBuffer* is non-**NULL**, this field indicates the size of the data payload in this buffer.

Description

This service is the prototype of how an SMM driver exposes a callback into its SMM-loaded image from the SMM infrastructure. The SMM infrastructure can call an image within SMM in response to the following two types of events:

- SMI dispatch
- **`EFI_SMM_BASE_PROTOCOL.Communicate()`**

The callback is negotiated with the SMM infrastructure via the service **`EFI_SMM_BASE_PROTOCOL.RegisterCallback()`**. This interface has the additional fields of a communication buffer so that a non-SMM agent can convey a message into the SMM-based agent. This communication is a stylized form of interprocess communication (IPC) that is mediated by the SMM infrastructure. This SMM infrastructure multiplexes the passing of the buffer from a possibly virtual-mode, runtime environment into the physical-mode, SMM environment.

EFI Table Header

EFI_TABLE_HEADER

Summary

Data structure that precedes all of the services in the System Management System Table (SMST).

Related Definitions

```
typedef struct {
  UINT64     Signature;
  UINT32     Revision;
  UINT32     HeaderSize;
  UINT32     CRC32;
  UINT32     Reserved;
} EFI_TABLE_HEADER;
```

Parameters

Signature

A 64-bit signature that identifies the type of table that follows.

Revision

The revision of the SMM CIS to which this table conforms. The upper 16 bits of this field contain the major revision value, and the lower 16 bits contain the minor revision value. The minor revision values are limited to the range of 00..99.

HeaderSize

The size in bytes of the entire table including the **EFI_TABLE_HEADER**.

CRC32

The 32-bit CRC for the entire table. This value is computed by setting this field to 0 and computing the 32-bit CRC for *HeaderSize* bytes. This value should be computed across all of the entries in the SMST. The SMM infrastructure code will compute this value.

Reserved

Reserved field that must be set to 0.

Description

The data type **EFI_TABLE_HEADER** is the data structure that precedes all of the standard EFI table types. It includes a signature that is unique for each table type, a revision of the table that may be updated as extensions are added to the EFI table types, and a 32-bit CRC so a consumer of an EFI table type can validate the contents of the EFI table.

System Management System Table (SMST)

EFI_SMM_SYSTEM_TABLE

Summary

The System Management System Table (SMST) is a table that contains a collection of common services for managing SMRAM allocation and providing basic I/O services. These services are intended for both preboot and runtime usage.

Related Definitions

```
#define SMM_SMST_SIGNATURE   EFI_SIGNATURE_32('S','M','S','T')
#define EFI_SMM_SYSTEM_TABLE_REVISION    (0<<16) | (0x09)

typedef struct _EFI_SMM_SYSTEM_TABLE {
  EFI_TABLE_HEADER                     Hdr;

  CHAR16                               *SmmFirmwareVendor;
  UINT32                               SmmFirmwareRevision;

  EFI_SMM_INSTALL_CONFIGURATION_TABLE SmmInstallConfigurationTable;

  //
  // I/O Services
  //

  EFI_GUID                             EfiSmmCpuIoGuid;
  EFI_SMM_CPU_IO_INTERFACE             SmmIo;

  //
  // Runtime memory service
  //
  EFI_SMM_ALLOCATE_POOL               SmmAllocatePool;
  EFI_SMM_FREE_POOL                   SmmFreePool;
  EFI_SMM_ALLOCATE_PAGES              SmmAllocatePages;
  EFI_SMM_FREE_PAGES                  SmmFreePages;

  //
  // CPU information records
  //

  UINTN                                CurrentlyExecutingCpu;
  UINTN                                NumberOfCpus;
  EFI_SMM_CPU_SAVE_STATE               *CpuSaveState;
  EFI_SMM_FLOATING_POINT_SAVE_STATE
                               *CpuOptionalFloatingPointState;
```

```
    //
    // Extensibility table
    //

    UINTN                                   NumberOfTableEntries;
    EFI_CONFIGURATION_TABLE                 *SmmConfigurationTable;

  } EFI_SMM_SYSTEM_TABLE;
```

Parameters

Hdr

The table header for the System Management System Table (SMST). This header contains the **SMM_SMST_SIGNATURE** and **EFI_SMM_SYSTEM_TABLE_REVISION** values along with the size of the **EFI_SMM_SYSTEM_TABLE** structure and a 32-bit CRC to verify that the contents of the SMST are valid.

SmmFirmwareVendor

A pointer to a **NULL**-terminated Unicode string containing the vendor name. It is permissible for this pointer to be **NULL**.

SmmFirmwareRevision

The particular revision of the firmware.

SmmInstallConfigurationTable

Adds, updates, or removes a configuration table entry from the SMST. See the **SmmInstallConfigurationTable()** function description.

EfiSmmCpuIoGuid

A GUID that designates the particular CPU I/O services. Type **EFI_SMM_CPU_IO_GUID** is defined in the **SmmIo()** function description. Type **EFI_GUID** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification.*

SmmIo

Provides the basic memory and I/O interfaces that are used to abstract accesses to devices. See the **SmmIo()** function description.

SmmAllocatePool

Allocates pool memory from SMRAM for IA-32 or runtime memory for the Itanium® processor family. See the **SmmAllocatePool()** function description.

SmmFreePool

Returns pool memory to the system. See the **SmmFreePool()** function description.

SmmAllocatePages

Allocates memory pages from the system. See the **SmmAllocatePages()** function description.

SmmFreePages

Frees memory pages for the system. See the **SmmFreePages()** function description.

CurrentlyExecutingCpu

A 1-relative number between 1 and the *NumberOfCpus* field. This field designates which processor is executing the SMM infrastructure. This number also serves as an index into the *CpuSaveState* and *CpuOptionalFloatingPointState* fields.

NumberOfCpus

The number of current operational processors in the platform.

CpuSaveState

A pointer to a catenation of the **EFI_SMM_CPU_SAVE_STATE** structure. The size of this entire table is *NumberOfCpus* * size of the **EFI_SMM_CPU_SAVE_STATE**. Type **EFI_SMM_CPU_SAVE_STATE** is defined in SMM CPU Information Records in Services - SMM.

CpuOptionalFloatingPointState

A pointer to a catenation of the **EFI_SMM_FLOATING_POINT_SAVE_STATE**. The size of this entire table is *NumberOfCpus* * size of the **EFI_SMM_FLOATING_POINT_SAVE_STATE**. These fields are populated **only** if there is **at least** one SMM driver that has registered for a callback with the *FloatingPointSave* field in **EFI_SMM_BASE_PROTOCOL.RegisterCallback()** set to **TRUE**. Type **EFI_SMM_FLOATING_POINT_SAVE_STATE** is defined in SMM CPU Information Records in Services - SMM.

NumberOfTableEntries

The number of EFI Configuration Tables in the buffer *SmmConfigurationTable*.

SmmConfigurationTable

A pointer to the EFI Configuration Tables. The number of entries in the table is *NumberOfTableEntries*.

Description

The table is similar to the EFI System Table, but it is flat. The only notable artifact from the EFI System Table is the ability to register additional tables prior to locking the System Management Random Access Memory (SMRAM) and exiting boot services.

The *CurrentlyExecutingCpu* parameter is a value that is less than or equal to the *NumberOfCpus* field. The *CpuSaveState* is a pointer to a contiguous run of **EFI_SMM_CPU_STATE** structures in SMRAM. The *CurrentlyExecutingCpu* can be used as an index to locate the respective save-state for which the given processor is executing, if so desired. The same indexing scheme is used for the *CpuOptionalFloatingPointState* structure.

The **EFI_SMM_CPU_STATE** is a data structure that contains the SMM save-state information for IA-32 and the record of saved data for Itanium processors. The data for each processor instance are linearly concatenated in SMRAM.

When a handler is executed, it is passed the **EFI_SMM_HANDLER_ENTRY_POINT**.

SMM Configuration Table

EFI_CONFIGURATION_TABLE

Summary

The *ConfigurationTable* field of the System Management System Table (SMST) points to a list of GUID/pointer pairs. Some GUIDs may be required for OS and firmware interoperability. Other GUIDs may be defined as required by different IBV, OEMs, IHVs, and OSVs.

Related Definitions

```
typedef struct{
  EFI_GUID                          VendorGuid;
  VOID                              *VendorTable;
} EFI_CONFIGURATION_TABLE;
```

Parameters

VendorGuid

The 128-bit GUID value that uniquely identifies the EFI Configuration Table. Type **EFI_GUID** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification.*

VendorTable

A pointer to the table associated with *VendorGuid*.

Description

The EFI Configuration Table is the *SmmConfigurationTable* field in the **EFI_SMM_SYSTEM_TABLE**. This table contains a set of GUID/pointer pairs. Each element of this table is described by this **EFI_CONFIGURATION_TABLE** structure. The number of types of configuration tables is expected to grow over time, which is why a GUID is used to identify the configuration table type. The EFI Configuration Table may contain at most once instance of each table type.

4
Services - SMM

Introduction

The expectation is that the SMM drivers can be built in the same framework as other DXE runtime drivers. A set of services is available to facilitate loading SMM drivers into SMRAM. The following topics describe these services.

SMM Install Configuration Table

SmmInstallConfigurationTable()

Summary

Adds, updates, or removes a configuration table entry from the System Management System Table (SMST).

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMM_INSTALL_CONFIGURATION_TABLE) (
  IN struct _EFI_SMM_SYSTEM_TABLE  *SystemTable,
  IN EFI_GUID                      *Guid,
  IN VOID                          *Table,
  IN UINTN                         TableSize
  )
```

Parameters

SystemTable

A pointer to the System Management System Table (SMST).

Guid

A pointer to the GUID for the entry to add, update, or remove.

Table

A pointer to the buffer of the table to add.

TableSize

The size of the table to install.

Description

The **SmmInstallConfigurationTable()** function is used to maintain the list of configuration tables that are stored in the SMST. The list is stored as an array of (GUID, Pointer) pairs. The list must be allocated from pool memory with *PoolType* set to **EfiRuntimeServicesData**.

If *Guid* is not a valid GUID, **EFI_INVALID_PARAMETER** is returned. If *Guid* is valid, there are four possibilities:

- If *Guid* is not present in the SMST and *Table* is not **NULL**, then the (*Guid*, *Table*) pair is added to the SMST. See Note below.
- If *Guid* is not present in the SMST and *Table* is **NULL**, then **EFI_NOT_FOUND** is returned.
- If *Guid* is present in the SMST and *Table* is not **NULL**, then the (*Guid*, *Table*) pair is updated with the new *Table* value.
- If *Guid* is present in the SMST and *Table* is **NULL**, then the entry associated with *Guid* is removed from the SMST.

If an add, modify, or remove operation is completed, then **EFI_SUCCESS** is returned.

NOTE

If there is not enough memory to perform an add operation, then **EFI_OUT_OF_RESOURCES** *is returned.*

For Itanium®-based systems, a possible candidate for installation here would be the System Abstraction Layer (SAL) System Table. The reason is that a power-button support handler in Itanium-based systems has to issue a **PAL_HALT_LIGHT** call.

Status Codes Returned

EFI_SUCCESS	The (*Guid*, *Table*) pair was added, updated, or removed.
EFI_INVALID_PARAMETER	*Guid* is not valid.
EFI_NOT_FOUND	An attempt was made to delete a nonexistent entry.
EFI_OUT_OF_RESOURCES	There is not enough memory available to complete the operation.

SMM I/O Services

SMM CPU I/O Overview

The interfaces provided in **EFI_SMM_CPU_IO_INTERFACE** are for performing basic operations to memory and I/O. The **EFI_SMM_CPU_IO_INTERFACE** can be thought of as the bus driver for the system. The system provides abstracted access to basic system resources to allow a driver to have a programmatic method to access these basic system resources.

The **EFI_SMM_CPU_IO_INTERFACE** allows for future innovation of the platform. It abstracts device-specific code from the system memory map. This abstraction allows system designers to greatly change the system memory map without impacting platform-independent code that is consuming basic system resources.

The device I/O services in the system are blocking and will be installed by the agent that abstracts the compatibility bus.

The SMM handler that supports the SMM device I/O services must be executed prior to any other handler installations. The DXE grammar mechanism should be used to enforce this requirement. If this temporal ordering is carried out, then the preamble initialization of the SMM processor I/O handler can populate the SMST using the **SmmInstallConfigurationTable()** mechanism and the GUID listed in **SmmIo()**.

SmmIo()

Summary

Provides the basic memory and I/O interfaces that are used to abstract accesses to devices.

GUID

```
#define EFI_SMM_CPU_IO_GUID \
  { 0x5f439a0b, 0x45d8, 0x4682, 0xa4, 0xf4, 0xf0, 0x57, 0x6b,
   0x51, 0x34, 0x41 }
```

Protocol Interface Structure

```
typedef struct _EFI_SMM_CPU_IO_INTERFACE {
  EFI_SMM_IO_ACCESS      Mem;
  EFI_SMM_IO_ACCESS      Io;
} EFI_SMM_CPU_IO_INTERFACE;
```

Parameters

Mem

Allows reads and writes to memory-mapped I/O space. See the **Mem()** function description. Type **EFI_SMM_IO_ACCESS** is defined in "Related Definitions" below.

Io

Allows reads and writes to I/O space. See the **Io()** function description. Type **EFI_SMM_IO_ACCESS** is defined in "Related Definitions" below.

Description

The **EFI_SMM_CPU_IO_INTERFACE** service provides the basic memory, I/O, and PCI interfaces that are used to abstract accesses to devices.

Related Definitions

```
//*******************************************************
// EFI_SMM_IO_ACCESS
//*******************************************************
typedef struct {
  EFI_SMM_CPU_IO    Read;
  EFI_SMM_CPU_IO    Write;
} EFI_SMM_IO_ACCESS;
```

Read

This service provides the various modalities of memory and I/O read.

Write

This service provides the various modalities of memory and I/O write.

EFI_SMM_CPU_IO_INTERFACE.Mem()

Summary

Enables a driver to access device registers in the memory space.

Prototype

```
typedef
EFI_STATUS
(EFIAPI * EFI_SMM_CPU_IO) (
  IN struct _EFI_SMM_CPU_IO_INTERFACE  *This,
  IN EFI_SMM_IO_WIDTH                  Width,
  IN UINT64                            Address,
  IN UINTN                             Count,
  IN OUT VOID                          *Buffer
  );
```

Parameters

This

The **EFI_SMM_CPU_IO_INTERFACE** instance.

Width

Signifies the width of the I/O operations. Type **EFI_SMM_IO_WIDTH** is defined in "Related Definitions" below.

Address

The base address of the I/O operations. The caller is responsible for aligning the *Address* if required.

Count

The number of I/O operations to perform. Bytes moved is *Width* size * *Count*, starting at *Address*.

Buffer

For read operations, the destination buffer to store the results. For write operations, the source buffer from which to write data.

Description

The **EFI_SMM_CPU_IO.Mem()** function enables a driver to access device registers in the memory.

The I/O operations are carried out exactly as requested. The caller is responsible for any alignment and I/O width issues that the bus, device, platform, or type of I/O might require. For example, on IA-32 platforms, width requests of **SMM_IO_UINT64** do not work.

The *Address* field is the bus relative address as seen by the device on the bus.

Related Definitions

```
//*******************************************************
// EFI_SMM_IO_WIDTH
//*******************************************************

typedef enum {
  SMM_IO_UINT8  = 0,
  SMM_IO_UINT16 = 1,
  SMM_IO_UINT32 = 2,
  SMM_IO_UINT64 = 3
} EFI_SMM_IO_WIDTH;
```

Status Codes Returned

EFI_SUCCESS	The data was read from or written to the device.
EFI_UNSUPPORTED	The *Address* is not valid for this system.
EFI_INVALID_PARAMETER	*Width* or *Count*, or both, were invalid.
EFI_OUT_OF_RESOURCES	The request could not be completed due to a lack of resources.

EFI_SMM_CPU_IO_INTERFACE.Io()

Summary

Enables a driver to access device registers in the I/O space.

Prototype

```
typedef
EFI_STATUS
(EFIAPI * EFI_SMM_CPU_IO) (
  IN struct _EFI_SMM_CPU_IO_INTERFACE   *This,
  IN EFI_SMM_IO_WIDTH                   Width,
  IN UINT64                             Address,
  IN UINTN                              Count,
  IN OUT VOID                           *Buffer
  );
```

Parameters

This

The **EFI_SMM_CPU_IO_INTERFACE** instance.

Width

Signifies the width of the I/O operations. Type **EFI_SMM_IO_WIDTH** is defined in **Mem()**.

Address

The base address of the I/O operations. The caller is responsible for aligning the *Address* if required.

Count

The number of I/O operations to perform. Bytes moved is *Width* size * *Count*, starting at *Address*.

Buffer

For read operations, the destination buffer to store the results. For write operations, the source buffer from which to write data.

Description

The **EFI_SMM_CPU_IO.Io()** function enables a driver to access device registers in the I/O space.

The I/O operations are carried out exactly as requested. The caller is responsible for any alignment and I/O width issues which the bus, device, platform, or type of I/O might require. For example, on IA-32 platforms, width requests of **SMM_IO_UINT64** do not work.

The caller must align the starting address to be on a proper width boundary.

Status Codes Returned

EFI_SUCCESS	The data was read from or written to the device.
EFI_UNSUPPORTED	The *Address* is not valid for this system.
EFI_INVALID_PARAMETER	*Width* or *Count*, or both, were invalid.
EFI_OUT_OF_RESOURCES	The request could not be completed due to a lack of resources.

SMM Runtime Memory Services

SmmAllocatePool()

Summary

Allocates pool memory from SMRAM for IA-32 or runtime memory for the Itanium® processor family.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMM_ALLOCATE_POOL) (
  IN EFI_MEMORY_TYPE                PoolType,
  IN UINTN                          Size,
  OUT VOID                          **Buffer
  );
```

Parameters

PoolType

The type of pool to allocate. The only supported type is **EfiRuntimeServicesData**; the interface will internally map this runtime request to SMRAM for IA-32 and leave it as this type for the Itanium processor family. Other types are ignorable. Type **EFI_MEMORY_TYPE** is defined in **AllocatePages()** in the *EFI 1.10 Specification.*

Size

The number of bytes to allocate from the pool.

Buffer

A pointer to a pointer to the allocated buffer if the call succeeds; undefined otherwise.

Description

The **SmmAllocatePool()** function allocates a memory region of *Size* bytes from memory of type *PoolType* and returns the address of the allocated memory in the location that is referenced by *Buffer*. This function allocates pages from EFI SMRAM memory for IA-32 as needed to grow the requested pool type. All allocations are 8-byte aligned.

PoolType can be ignored in that the type will always be SMRAM for IA-32 and runtime memory for the Itanium processor family.

The allocated pool memory is returned to the available pool with the **SmmFreePool()** function.

Status Codes Returned

EFI_SUCCESS	The requested number of bytes was allocated.
EFI_OUT_OF_RESOURCES	The pool requested could not be allocated.
EFI_UNSUPPORTED	In runtime.

SmmFreePool()

Summary

Returns pool memory to the system.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMM_FREE_POOL) (
  IN VOID                         *Buffer
  );
```

Parameters

Buffer

Pointer to the buffer to free.

Description

This function returns the memory specified by *Buffer* to the system. On return, the memory's type is EFI SMRAM memory. The *Buffer* that is freed must have been allocated by **SmmAllocatePool()**.

Status Codes Returned

EFI_SUCCESS	The memory was returned to the system.
EFI_INVALID_PARAMETER	*Buffer* was invalid.
EFI_UNSUPPORTED	In runtime.

SmmAllocatePages()

Summary

Allocates memory pages from the system.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMM_ALLOCATE_PAGES) (
  IN EFI_ALLOCATE_TYPE       Type,
  IN EFI_MEMORY_TYPE         MemoryType,
  IN UINTN                   NumberOfPages,
  OUT EFI_PHYSICAL_ADDRESS   *Memory
  );
```

Parameters

Type

The type of allocation to perform. Type **EFI_ALLOCATE_TYPE** is defined in **AllocatePages()** in the *EFI 1.10 Specification.*

MemoryType

This specification supports only **EfiRuntimeServicesData**. Type **EFI_MEMORY_TYPE** is defined in **AllocatePages()** in the *EFI 1.10 Specification.*

NumberofPages

The number of contiguous 4 KB pages to allocate.

Memory

Pointer to a physical address. On input, the way in which the address is used depends on the value of *Type*. See “Description” for more information. On output, the address is set to the base of the page range that was allocated. Type **EFI_PHYSICAL_ADDRESS** is defined in **AllocatePages()** in the *EFI 1.10 Specification.*

Description

The **SmmAllocatePages()** function allocates the requested number of pages and returns a pointer to the base address of the page range in the location referenced by *Memory*. The function scans the SMM infrastructure memory map to locate free pages. When it finds a physically contiguous block of pages that is large enough and also satisfies the value of *Type*, it changes the SMM infrastructure memory map to indicate that the pages are now of type *MemoryType*.

SMM drivers should allocate memory (and pool) of type **EfiRuntimeServicesData**.

Allocation requests of *Type* **AllocateAnyPages** allocate any available range of pages that satisfies the request. On input, the address pointed to by *Memory* is ignored. Allocation requests of *Type* **AllocateMaxAddress** allocate any available range of pages whose uppermost address is less than or equal to the address pointed to by *Memory* on input. Allocation requests of *Type* **AllocateAddress** allocate pages at the address pointed to by *Memory* on input.

Status Codes Returned

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.

SmmFreePages()

Summary

Frees memory pages for the system.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMM_FREE_PAGES) (
  IN EFI_PHYSICAL_ADDRESS    Memory,
  IN UINTN                   NumberOfPages
  );
```

Parameters

Memory

The base physical address of the pages to be freed. Type **EFI_PHYSICAL_ADDRESS** is defined in **AllocatePages()** in the *EFI 1.10 Specification.*

NumberOfPages

The number of contiguous 4 KB pages to free.

Description

The **SmmFreePages()** function returns memory that was allocated by **SmmAllocatePages()** to the firmware.

Status Codes Returned

EFI_SUCCESS	The requested memory pages were freed.
EFI_NOT_FOUND	The requested memory pages were not allocated with **SmmAllocatePages()**.
EFI_INVALID_PARAMETER	*Memory* is not a page-aligned address or *NumberOfPages* is invalid.

SMM CPU Information Records

SMM CPU Information Records Introduction

This section describes processor-specific information that is managed by the SMM infrastructure. These save-state structures are essentially descriptions of all of the operational processors in the system when the SMI or PMI activation was invoked.

SMM drivers use these structures to discern what type of processing needs to occur (such as the programmatic action that caused the SMI or PMI event). The SMM infrastructure also uses the information in these structures to restore the state of the processors after the system exits the SMM infrastructure and resumes its foreground operational activities.

Drivers can read from these structures but must take care in writing to them because the state of the machine will be affected by any updates that are performed in these structures.

EFI_SMM_CPU_SAVE_STATE and **EFI_SMM_FLOATING_POINT_SAVE_STATE** are the structures that define the processor save-state and floating-point save-state information, respectively. Each of these structures are unions for processor-specific data structures. The following table lists the CPU Information Records structures for each processor architecture. The next topics define these structures.

Table 4-1. Defined CPU Information Records

Platform	CPU Save-State Data Type	Floating-Point Save-State Data Type
IA-32	EFI_SMI_CPU_SAVE_STATE	EFI_PMI_SYSTEM_CONTEXT
Itanium® processor family	EFI_SMI_OPTIONAL_FPSAVE_STATE	EFI_PMI_OPTIONAL_FLOATING_POINT_CONTEXT

EFI_SMM_CPU_SAVE_STATE

EFI_SMU_CPU_SAVE_STATE

Summary

The processor save-state information for IA-32 and Itanium® processors.

Prototype

```
typedef union {
  EFI_SMI_CPU_SAVE_STATE      Ia32SaveState;
  EFI_PMI_SYSTEM_CONTEXT      ItaniumSaveState;
}  EFI_SMM_CPU_SAVE_STATE;
```

Parameters

Ia32SaveState

The processor save-state information for IA-32 processors. Type **EFI_SMI_CPU_SAVE_STATE** is defined in SMM CPU Information Records in Services - SMM.

ItaniumSaveState

The processor save-state information for Itanium processors. Type **EFI_PMI_SYSTEM_CONTEXT** is defined in SMM CPU Information Records in Services - SMM.

Description

The processor save-state information for IA-32 and Itanium processors. This information is important in that the SMM drivers may need to ascertain the state of the processor before invoking the SMI or PMI.

IA-32

EFI_SMI_CPU_SAVE_STATE

Summary

The processor save-state information for IA-32 processors. This information is important in that the SMM drivers may need to ascertain the state of the processor before invoking the SMI.

See the *IA-32 Intel® Architecture Software Developer's Manual*, volumes 1–3, for more information on the registers included in this data structure.

See **EFI_PMI_SYSTEM_CONTEXT** for the structure for the Itanium® processor family.

Prototype

```
typedef struct _EFI_SMI_CPU_SAVE_STATE {
  UINT8                      Reserved1[248];
  UINT32                     SMBASE;
  UINT32                     SMMRevId;
  UINT16                     IORestart;
  UINT16                     AutoHALTRestart;
  UINT8                      Reserved2[164];
  UINT32                     ES;
  UINT32                     CS;
  UINT32                     SS;
  UINT32                     DS;
  UINT32                     FS;
  UINT32                     GS;
  UINT32                     LDTBase;
  UINT32                     TR;
  UINT32                     DR7;
  UINT32                     DR6;
  UINT32                     EAX;
  UINT32                     ECX;
  UINT32                     EDX;
  UINT32                     EBX;
  UINT32                     ESP;
  UINT32                     EBP;
  UINT32                     ESI;
  UINT32                     EDI;
  UINT32                     EIP;
  UINT32                     EFLAGS;
  UINT32                     CR3;
  UINT32                     CR0;
  } EFI_SMI_CPU_SAVE_STATE;
```

Parameters

Reserved1

Reserved for future processors. As such, software should not attempt to interpret or write to this region.

SMBASE

The location of the processor SMBASE, which is the location where the processor will pass control upon receipt of an SMI.

SMMRevId

The revision of the SMM save state. This value is set by the processor.

IORestart

The value of the I/O restart field. Allows for restarting an in-process I/O instruction.

AutoHALTRestart

Describes behavior that should be commenced in response to a halt instruction.

Reserved2

Reserved for future processors. As such, software should not attempt to interpret or write to this region.

ES through *CR0*

Registers in IA-32 processors. See the *IA-32 Intel® Architecture Software Developer's Manual,* volumes 1–3, for more information.

Description

This data structure describes the processor save-state of an IA-32 processor. There will be a save-state structure for each processor, and the SMST shall reference the catenation of these structures. The processor will save this information upon receipt of the SMI, and the processor will restore this information to the processor upon receipt of the Resume (RSM) instruction.

Itanium® Processor Family

EFI_PMI_SYSTEM_CONTEXT

Summary

The processor save-state information for the Itanium® processor family. This information is important in that the SMM drivers may need to ascertain the state of the processor before invoking the PMI. This structure is mandatory and must be 512 byte aligned.

See the *Intel® Itanium® Architecture Software Developer's Manual*, volumes 1–4, for more information on the registers included in this data structure.

Prototype

```
typedef struct _EFI_PMI_SYSTEM_CONTEXT
{
  UINT64   reserved;
  UINT64   r1;
  UINT64   r2;
  UINT64   r3;
  UINT64   r4;
  UINT64   r5;
  UINT64   r6;
  UINT64   r7;
  UINT64   r8;
  UINT64   r9;
  UINT64   r10;
  UINT64   r11;
  UINT64   r12;
  UINT64   r13;
  UINT64   r14;
  UINT64   r15;
  UINT64   r16;
  UINT64   r17;
  UINT64   r18;
  UINT64   r19;
  UINT64   r20;
  UINT64   r21;
  UINT64   r22;
  UINT64   r23;
  UINT64   r24;
  UINT64   r25;
  UINT64   r26;
  UINT64   r27;
  UINT64   r28;
  UINT64   r29;
```

```
UINT64   r30;
UINT64   r31;

UINT64   pr;

UINT64   b0;
UINT64   b1;
UINT64   b2;
UINT64   b3;
UINT64   b4;
UINT64   b5;
UINT64   b6;
UINT64   b7;

// application registers
UINT64   ar_rsc;
UINT64   ar_bsp;
UINT64   ar_bspstore;
UINT64   ar_rnat;

UINT64   ar_fcr;

UINT64   ar_eflag;
UINT64   ar_csd;
UINT64   ar_ssd;
UINT64   ar_cflg;
UINT64   ar_fsr;
UINT64   ar_fir;
UINT64   ar_fdr;

UINT64   ar_ccv;

UINT64   ar_unat;

UINT64   ar_fpsr;

UINT64   ar_pfs;
UINT64   ar_lc;
UINT64   ar_ec;

// control registers
UINT64   cr_dcr;
UINT64   cr_itm;
UINT64   cr_iva;
UINT64   cr_pta;
UINT64   cr_ipsr;
UINT64   cr_isr;
UINT64   cr_iip;
UINT64   cr_ifa;
```

```
    UINT64    cr_itir;
    UINT64    cr_iipa;
    UINT64    cr_ifs;
    UINT64    cr_iim;
    UINT64    cr_iha;

    // debug registers
    UINT64    dbr0;
    UINT64    dbr1;
    UINT64    dbr2;
    UINT64    dbr3;
    UINT64    dbr4;
    UINT64    dbr5;
    UINT64    dbr6;
    UINT64    dbr7;

    UINT64    ibr0;
    UINT64    ibr1;
    UINT64    ibr2;
    UINT64    ibr3;
    UINT64    ibr4;
    UINT64    ibr5;
    UINT64    ibr6;
    UINT64    ibr7;

    // virtual registers
    UINT64    int_nat;          // nat bits for R1-R31

  } EFI_PMI_SYSTEM_CONTEXT;
```

EFI_SMM_OPTIONAL_FP_SAVE_STATE

EFI_SMM_FLOATING_POINT_SAVE_STATE

Summary

The processor save-state information for IA-32 and Itanium® processors.

Prototype

```
typedef union {
  EFI_SMI_OPTIONAL_FPSAVE_STATE                Ia32FpSave;
  EFI_PMI_OPTIONAL_FLOATING_POINT_CONTEXT      ItaniumFpSave;
} EFI_SMM_FLOATING_POINT_SAVE_STATE;
```

Parameters

Ia32FpSave

The optional floating point save-state information for IA-32 processors. Type **EFI_SMI_OPTIONAL_FPSAVE_STATE** is defined in SMM CPU Information Records in Services - SMM.

ItaniumFpSave

The optional floating point save-state information for Itanium processors. Type **EFI_PMI_OPTIONAL_FLOATING_POINT_CONTEXT** is defined in SMM CPU Information Records in Services - SMM.

Description

The processor save-state information for IA-32 and Itanium processors. If the optional floating point save is indicated for any handler, then this data structure must be preserved.

IA-32

EFI_SMI_OPTIONAL_FPSAVE_STATE

Summary

The optional floating point save-state information for IA-32 processors. If the optional floating point save is indicated for any handler, the following data structure must be preserved.

See the *IA-32 Intel® Architecture Software Developer's Manual*, volumes 1–3, for more information on the registers included in this data structure.

See **EFI_PMI_OPTIONAL_FLOATING_POINT_CONTEXT** for the structure for the Itanium® processor family.

Prototype

```
typedef struct _EFI_SMI_OPTIONAL_FPSAVE_STATE {
  UINT16                 Fcw;
  UINT16                 Fsw;
  UINT16                 Ftw;
  UINT16                 Opcode;
  UINT32                 Eip;
  UINT16                 Cs;
  UINT16                 Rsvd1;
  UINT32                 DataOffset;
  UINT16                 Ds;
  UINT8                  Rsvd2[10];
  UINT8                  St0Mm0[10], Rsvd3[6];
  UINT8                  St0Mm1[10], Rsvd4[6];
  UINT8                  St0Mm2[10], Rsvd5[6];
  UINT8                  St0Mm3[10], Rsvd6[6];
  UINT8                  St0Mm4[10], Rsvd7[6];
  UINT8                  St0Mm5[10], Rsvd8[6];
  UINT8                  St0Mm6[10], Rsvd9[6];
  UINT8                  St0Mm7[10], Rsvd10[6];
  UINT8                  Rsvd11[22*16];
} EFI_SMI_OPTIONAL_FPSAVE_STATE;
```

Itanium® Processor Family

EFI_PMI_OPTIONAL_FLOATING_POINT_CONTEXT

Summary

The optional floating point save-state information for the Itanium® processor family. If the optional floating point save is indicated for any handler, then this data structure must be preserved. See the *Intel® Itanium® Architecture Software Developer's Manual*, volumes 1–4, for more information on the registers included in this data structure.

Prototype

```
typedef struct {
  UINT64   f2[2];
  UINT64   f3[2];
  UINT64   f4[2];
  UINT64   f5[2];
  UINT64   f6[2];
  UINT64   f7[2];
  UINT64   f8[2];
  UINT64   f9[2];
  UINT64   f10[2];
  UINT64   f11[2];
  UINT64   f12[2];
  UINT64   f13[2];
  UINT64   f14[2];
  UINT64   f15[2];
  UINT64   f16[2];
  UINT64   f17[2];
  UINT64   f18[2];
  UINT64   f19[2];
  UINT64   f20[2];
  UINT64   f21[2];
  UINT64   f22[2];
  UINT64   f23[2];
  UINT64   f24[2];
  UINT64   f25[2];
  UINT64   f26[2];
  UINT64   f27[2];
  UINT64   f28[2];
  UINT64   f29[2];
  UINT64   f30[2];
  UINT64   f31[2];
} EFI_PMI_OPTIONAL_FLOATING_POINT_CONTEXT;
```

5
Services - SMM Library (SMLib)

Introduction

There is a share-nothing model that is employed between the management-mode application and the boot service/runtime EFI environment. As such, a minimum set of services needs to be available to the boot service agent.

The services described in this section are purposely coded to coexist with a foreground preboot or runtime environment. The latter can include both EFI and non-EFI aware operating systems. As such, the implementation of these services must save and restore any "shared" resources with the foreground environment or only use resources that are private to the SMM code.

This library should be used in place of the runtime or boot services library. It is specially coded to survive in an SMM environment.

Status Codes Services

EFI_SMM_STATUS_CODE_PROTOCOL

Summary

Provides status code services from SMM.

GUID

```
#define EFI_SMM_STATUS_CODE_PROTOCOL_GUID \
  { 0x6afd2b77, 0x98c1, 0x4acd, 0xa6, 0xf9, 0x8a, 0x94, 0x39,
   0xde, 0xf, 0xb1 }
```

Protocol Interface Structure

```
typedef struct _EFI_SMM_STATUS_CODE_PROTOCOL {
  EFI_SMM_REPORT_STATUS_CODE    ReportStatusCode;
} EFI_SMM_STATUS_CODE_PROTOCOL;
```

Parameters

ReportStatusCode

Allows for the SMM agent to produce a status code output. See the **ReportStatusCode()** function description.

Description

The **EFI_SMM_STATUS_CODE_PROTOCOL** provides the basic status code services while in SMRAM.

EFI_SMM_STATUS_CODE_PROTOCOL.ReportStatusCode()

Summary

Service to emit the status code in SMM.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMM_REPORT_STATUS_CODE) (
  IN struct _EFI_SMM_STATUS_CODE_PROTOCOL *This,
  IN EFI_STATUS_CODE_TYPE     CodeType,
  IN EFI_STATUS_CODE_VALUE    Value,
  IN UINT32                   Instance,
  IN EFI_GUID                 *CallerId,
  IN EFI_STATUS_CODE_DATA     *Data OPTIONAL
  );
```

Parameters

Type

Indicates the type of status code being reported. Type **EFI_STATUS_CODE_TYPE** is defined in "Related Definitions" below.

Value

Describes the current status of a hardware or software entity. This status includes information about the class and subclass that is used to classify the entity, as well as an operation. For progress codes, the operation is the current activity. For error codes, it is the exception. For debug codes, it is not defined at this time. Type **EFI_STATUS_CODE_VALUE** is defined in "Related Definitions" below. Specific values are discussed in the *Intel® Platform Innovation Framework for EFI Status Codes Specification.*

Instance

The enumeration of a hardware or software entity within the system. A system may contain multiple entities that match a class/subclass pairing. The instance differentiates between them. An instance of 0 indicates that instance information is unavailable, not meaningful, or not relevant. Valid instance numbers start with 1.

CallerId

This optional parameter may be used to identify the caller. This parameter allows the status code driver to apply different rules to different callers.

Data

This optional parameter may be used to pass additional data. Type **EFI_STATUS_CODE_DATA** is defined in "Related Definitions" below. The contents of this data type may have additional GUID-specific data. The standard GUIDs and their associated data structures are defined in the *Intel® Platform Innovation Framework for EFI Status Codes Specification*.

Description

The **EFI_SMM_STATUS_CODE_PROTOCOL.ReportStatusCode()** function enables a driver to emit a status code while in SMM. The reason that there is a separate protocol definition from the DXE variant of this service is that the publisher of this protocol will provide a service that is capability of coexisting with a foreground operational environment, such as an operating system after the termination of boot services.

In case of an error, the caller can specify the severity. In most cases, the entity that reports the error may not have a platform-wide view and may not be able to accurately assess the impact of the error condition. The DXE driver that produces the Status Code SMM Protocol is responsible for assessing the true severity level based on the reported severity and other information. This DXE driver may perform platform specific actions based on the type and severity of the status code being reported.

If *Data* is present, the driver treats it as read only data. The driver must copy *Data* to a local buffer in an atomic operation before performing any other actions. This is necessary to make this function re-entrant. The size of the local buffer may be limited. As a result, some of the *Data* can be lost. The size of the local buffer should at least be 256 bytes in size. Larger buffers will reduce the probability of losing part of the *Data*. If all of the local buffers are consumed, then this service may not be able to perform the platform specific action required by the status code being reported. As a result, if all the local buffers are consumed, the behavior of this service is undefined.

If the *CallerId* parameter is not **NULL**, then it is required to point to a constant GUID. In other words, the caller may not reuse or release the buffer pointed to by *CallerId*.

Related Definitions

```
//
// Status Code Type Definition
//
typedef UINT32 EFI_STATUS_CODE_TYPE;

//
// A Status Code Type is made up of the code type and severity
// All values masked by EFI_STATUS_CODE_RESERVED_MASK are
// reserved for use by this specification.
//
#define EFI_STATUS_CODE_TYPE_MASK       0x000000FF
#define EFI_STATUS_CODE_SEVERITY_MASK   0xFF000000
#define EFI_STATUS_CODE_RESERVED_MASK   0x00FFFF00

//
// Definition of code types, all other values masked by
// EFI_STATUS_CODE_TYPE_MASK are reserved for use by
// this specification.
//
#define EFI_PROGRESS_CODE               0x00000001
#define EFI_ERROR_CODE                  0x00000002
#define EFI_DEBUG_CODE                  0x00000003

//
// Definitions of severities, all other values masked by
// EFI_STATUS_CODE_SEVERITY_MASK are reserved for use by
// this specification.
// Uncontained errors are major errors that could not contained
// to the specific component that is reporting the error
// For example, if a memory error was not detected early enough,
// the bad data could be consumed by other drivers.
//
#define EFI_ERROR_MINOR                 0x40000000
#define EFI_ERROR_MAJOR                 0x80000000
#define EFI_ERROR_UNRECOVERED           0x90000000
#define EFI_ERROR_UNCONTAINED           0xa0000000

//
// Status Code Value Definition
//
typedef UINT32 EFI_STATUS_CODE_VALUE;
```

```
//
// A Status Code Value is made up of the class, subclass, and
// an operation.
//
#define EFI_STATUS_CODE_CLASS_MASK      0xFF000000
#define EFI_STATUS_CODE_SUBCLASS_MASK   0x00FF0000
#define EFI_STATUS_CODE_OPERATION_MASK  0x0000FFFF

//
// Definition of Status Code extended data header.
// The data will follow HeaderSize bytes from the beginning of
// the structure and is Size bytes long.
//
typedef struct {
  UINT16    HeaderSize;
  UINT16    Size;
  EFI_GUID  Type;
} EFI_STATUS_CODE_DATA;
```

HeaderSize

The size of the structure. This is specified to enable future expansion.

Size

The size of the data in bytes. This does not include the size of the header structure.

Type

The GUID defining the type of the data. The standard GUIDs and their associated data structures are defined in the *Intel® Platform Innovation Framework for EFI Status Codes Specification*.

Status Codes Returned

EFI_SUCCESS	The function completed successfully
EFI_DEVICE_ERROR	The function should not be completed due to a device error.

6
SMM Protocols

Introduction

The services described in this chapter describe a series of protocols that abstract the loading of DXE drivers into SMM, manipulation of the System Management RAM (SMRAM) apertures, and generation of System Management Interrupts (SMIs). These services have both boot services and runtime services.

The following protocols are defined in this chapter:

- **EFI_SMM_BASE_PROTOCOL**
- **EFI_SMM_ACCESS_PROTOCOL**
- **EFI_SMM_CONTROL_PROTOCOL**

EFI SMM Base Protocol

EFI_SMM_BASE_PROTOCOL

Summary

This protocol is used to install SMM handlers for support of subsequent SMI/PMI activations. This protocol is available on both IA-32 and Itanium®-based systems.

GUID

```
#define EFI_SMM_BASE_PROTOCOL_GUID \
  { 0x1390954D, 0xda95, 0x4227, 0x93, 0x28, 0x72, 0x82, 0xc2,
   0x17, 0xda, 0xa8 }
```

Protocol Interface Structure

```
typedef struct _EFI_SMM_BASE_PROTOCOL {
  EFI_SMM_REGISTER_HANDLER        Register;
  EFI_SMM_UNREGISTER_HANDLER      UnRegister;
  EFI_SMM_COMMUNICATE             Communicate;
  EFI_SMM_CALLBACK_SERVICE        RegisterCallback;
  EFI_SMM_INSIDE_OUT              InSmm;
  EFI_SMM_ALLOCATE_POOL           SmmAllocatePool;
  EFI_SMM_FREE_POOL               SmmFreePool;
  EFI_SMM_GET_SMST_LOCATION       GetSmstLocation;
} EFI_SMM_BASE_PROTOCOL;
```

Parameters

Register

Registers a handler to run in System Management RAM (SMRAM). See the **Register()** function description.

UnRegister

Removes a handler from execution in SMRAM. See the **UnRegister()** function description.

Communicate

Sends/receives a message for a registered handler. See the **Communicate()** function description.

RegisterCallback

Registers a callback from the constructor. See the **RegisterCallback()** function description.

InSmm

Detects whether the caller is inside or outside of SMM. See the **InSmm()** function description.

SmmAllocatePool

Allocates SMRAM. See the **SmmAllocatePool()** function description.

SmmFreePool

Deallocates SMRAM. See the **SmmFreePool()** function description.

GetSmstLocation

Retrieves the location of the System Management System Table (SMST). See the **GetSmstLocation()** function description.

Description

The **EFI_SMM_BASE_PROTOCOL** is a set of services that is exported by a processor device. It is a required protocol for the platform processor. This protocol can be used in both boot services and runtime mode. However, only the following member functions need to exist into runtime:

- **InSmm()**
- **Communicate()**

This protocol is responsible for registering the handler services. The order in which the handlers are executed is prescribed only with respect to the *MakeLast* flag in the **RegisterCallback()** service. The driver exports these registration and unregistration services in boot services mode, but the registered handlers will execute through the preboot and runtime. The only way to change the behavior of a registered driver after **ExitBootServices()** has been invoked is to use some private communication mechanism with the driver to order it to quiesce. This model permits typical use cases, such as invoking the handler to enter ACPI mode, where the OS loader would make this call before boot services are terminated. On the other hand, handlers for services such as chipset workarounds for the century rollover in CMOS should provide commensurate services throughout preboot and OS runtime.

For an IA-32 system, the dependency expression for the **EFI_SMM_BASE_PROTOCOL** driver might contain the **EFI_GUID** for the **EFI_SMM_CONTROL_PROTOCOL**, with a **DEPEX_AND** opcode combining this protocol with the **EFI_SMM_ACCESS_PROTOCOL**. For an Itanium-based system, the dependency expression might contain only the **EFI_GUID** for the **EFI_SMM_CONTROL_PROTOCOL**. This will allow the **EFI_SMM_BASE_PROTOCOL** driver to load only after the one (Itanium® processor family) or two (IA-32) supporting protocols have successfully loaded and installed their protocol interfaces.

An important additional aspect of the implementation of the driver that publishes the **EFI_SMM_BASE_PROTOCOL**, which we shall call the SMM infrastructure, concerns how it manages synchronous and asynchronous activations. Specifically, an SMI can be activated through the **Communicate()** interface, using inband software on a host processor that is manipulating the APM port through the **EFI_SMM_CONTROL_PROTOCOL**, for example. After the system has transitioned to SMM in response to a synchronous SMI, such as the **EFI_SMM_BASE_PROTOCOL.Communicate()**, there may be an asynchronous SMI activation, say from a periodic source in the I/O Controller Hub (ICH) device. The infrastructure must ensure that both activations are handled. As such, the SMM infrastructure will service the **Communicate()** request because there is a software handoff that it can parse. The SMM infrastructure, which is platform independent, will not be aware of the ICH-based pending SMI, because the latter is a platform source that should be managed by a child driver. As such, the SMM infrastructure should invoke all child handlers; it is up to the child handlers to decide if an exit can occur without activating a given source.

Given the information above, the SMM infrastructure cannot exit immediately after servicing a **Communicate()** or **RegisterCallback()** call.

EFI_SMM_BASE_PROTOCOL.Register()

Summary

Registers a given driver into System Management RAM (SMRAM). This function is the equivalent of performing the **LoadImage()**/**StartImage()** call (see the *EFI 1.10 Specification*, section 5.4) into SMM.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMM_REGISTER_HANDLER) (
  IN struct _EFI_SMM_BASE_PROTOCOL   *This,
  IN  EFI_DEVICE_PATH_PROTOCOL       *FilePath,
  IN  VOID                           *SourceBuffer OPTIONAL,
  IN  UINTN                          SourceSize,
  OUT EFI_HANDLE                     *ImageHandle,
  IN  BOOLEAN                        LegacyIA32Binary OPTIONAL
  )
```

Parameters

This

The **EFI_SMM_BASE_PROTOCOL** instance.

FilePath

Location of the image to be installed as the handler. Type **EFI_DEVICE_PATH** is defined in the *EFI 1.10 Specification.*

SourceBuffer

Memory location of image to be used as handler.

SourceSize

Size of the memory image to be used for handler.

ImageHandle

The handle that the base driver uses to decode the handler. Unique among SMM handlers only, not unique across DXE/EFI. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification.*

LegacyIA32Binary

An optional parameter that details that the associated file is a real-mode IA-32 binary. This flag should not be used on Itanium®-based systems.

Description

The **Register()** function is the equivalent of **LoadImage()** for the SMM execution phase. The SMM infrastructure code will invoke its equivalent of the start image immediately after the driver is loaded. The registered handlers are PE32+ images that conform to the EFI image specification. They export a single entry point that is used for both runtime dispatch and initialization.

As part of the initialization process, the driver will be passed in its image handle and the EFI System Table, as would any DXE driver. The driver constructor can use the EFI Boot Services to discover the instance of the **EFI_SMM_BASE_PROTOCOL** that loaded the image. Additionally, the boot service protocol services can be used to discover child dispatch protocols, and so on.

If the input handler is *LegacyIA32Binary*, the only interesting argument is *SourceBuffer*, which is simply a pointer to a 16-bit binary image handler. The SMM infrastructure code needs to maintain a list of 16-bit real-mode handlers that can exist only in the SMRAM locations below 1 MB such as the A- and B-segments. The handlers will be 16-bit code that expects to run in big-real mode or to have 32-bit pointer accessibility. The SMM infrastructure code should maintain an array of these 16-bit handlers that are dispatched before going into protected mode and dispatching the list of native-mode, PE32+ handlers. The 16-bit code can be relocated to any 16-byte boundary by way of fixing of the Code Segment (CS) register before invoking each handler.

The SMM infrastructure code will maintain a priority queue of the handlers for both 16-bit and 32-bit native handlers for IA-32 and native 64-bit handlers for Itanium® processors.

The scheduling model of this driver is rudimentary inasmuch as the SMM infrastructure code will exhaustively invoke every handler. This implementation will respect return codes and implement the appropriate exit policy; see "Related Definitions" below for defined SMM handler return codes. The behavior should be to continue exiting additional handlers on a return value of **EFI_HANDLER_SUCCESS**. Return values will be in the following registers:

- 16-bit handlers: Register AX
- 32-bit handlers: Register EAX
- Itanium processors: Register R8

For return values of **EFI_CRITICAL_EXIT**, the system should immediately return from SMM or the PMI state; the usage model here is that some latency-sensitive handler requires the context to immediately return to normal execution.

Finally, for returns of **EFI_HANDLER_SOURCE_QUIESCED**, the system believes that it has retired the SMI/PMI source. It is up to the main dispatcher to have acquired **at least** one handler return code with the value **EFI_HANDLER_SOURCE_QUIESCED**. If none are received, the SMM Dispatcher should reinvoke the handlers in case there are multiple pending sources. This re-scan strategy is used to revisit the handlers to avoid the latency involved in reinvoking the main SMI handler multiple times.

For native-mode handlers, the handler initialization might return a pointer to the actual handler. As such, the functional prototype (see SMM Infrastructure Code and Dispatcher) of the IA-32 16-bit handlers' initialization entry point will also be their call entry point. This dual nature means that there is no equivalent of a constructor for these service routines. The lowest address in the 16-bit handler is also the entry point that is always invoked.

NOTE

The SMM handlers should be stored in firmware files as DXE drivers. The entry point behavior of the driver will distinguish these drivers from other boot service DXE and runtime drivers. If the latter file type is used, then the standard DXE **EFI_DEPEX** *can be used to ensure that the driver is not dispatched until the appropriate time. The GUIDs in this dependency expression will be those of the other needed services. They are PE32+ images that have their subsystem type marked as Runtime Driver for purposes of construction. The reason that these drivers need to be put into special firmware files is to keep the DXE Dispatcher from attempting to load them autonomously.*

Related Definitions

```
//*******************************************************
//EFI SMM Handler Return Code
//*******************************************************
 #define EFI_HANDLER_SUCCESS                0x0000
 #define EFI_HANDLER_CRITICAL_EXIT          0x0001
 #define EFI_HANDLER_SOURCE_QUIESCED        0x0002
 #define EFI_HANDLER_SOURCE_PENDING         0x0003
```

Status Codes Returned

EFI_SUCCESS	The operation was successful.
EFI_OUT_OF_RESOURCES	There were no additional SMRAM resources to load the handler.
EFI_UNSUPPORTED	This platform does not support 16-bit handlers.
EFI_UNSUPPORTED	In runtime.
EFI_INVALID_PARAMETER	The handler was not the correct image type.

EFI_SMM_BASE_PROTOCOL.UnRegister()

Summary

Removes a handler from execution within SMRAM.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMM_UNREGISTER_HANDLER) (
  IN struct _EFI_SMM_BASE_PROTOCOL   *This,
  IN EFI_HANDLE                      ImageHandle
  )
```

Parameters

This

The **EFI_SMM_BASE_PROTOCOL** instance.

ImageHandle

The handler to be removed. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification.*

Description

This function unloads the image from SMRAM.

Status Codes Returned

EFI_SUCCESS	The operation was successful.
EFI_INVALID_PARAMETER	The handler did not exist.
EFI_UNSUPPORTED	In runtime.

EFI_SMM_BASE_PROTOCOL.Communicate()

Summary

Communicates with a registered handler

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMM_COMMUNICATE) (
  IN struct _EFI_SMM_BASE_PROTOCOL   *This,
  IN EFI_HANDLE                      ImageHandle,
  IN OUT VOID                        *CommunicationBuffer,
  IN OUT UINTN                       *SourceSize
  )
```

Parameters

This

The **EFI_SMM_BASE_PROTOCOL** instance.

ImageHandle

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

CommunicationBuffer

Pointer to the buffer to convey into SMRAM.

SourceSize

The size of the data buffer being passed in. On exit, the size of data being returned. Zero if the handler does not wish to reply with any data.

Description

This function provides a service to send and receive messages from a registered EFI service. The **EFI_SMM_BASE_PROTOCOL** driver is responsible for doing any of the copies such that the data lives in boot-service-accessible RAM.

A given implementation of the **EFI_SMM_BASE_PROTOCOL** may choose to use the **EFI_SMM_CONTROL_PROTOCOL** for effecting the mode transition, or it may use some processor protocol SMI/PMI Interprocessor Interrupt (IPI) protocol service.

The agent invoking the communication interface at runtime may be virtually mapped. The SMM infrastructure code and handlers, on the other hand, execute in physical mode. As a result, the non-SMM agent, which may be executing in the virtual-mode OS context (as a result of an OS invocation of the EFI 1.10 **SetVirtualAddressMap()** service), should use a contiguous memory buffer with a physical address before invoking this service. If the virtual address of the buffer is used, the SMM driver will not know how to do the appropriate virtual-to-physical conversion.

To avoid confusion in interpreting frames, the *CommunicateBuffer* parameter should always begin with **EFI_SMM_COMMUNICATE_HEADER**, which is defined in “Related Definitions” below. The header data is mandatory for messages sent **into** the SMM agent.

Related Definitions

```
//*******************************************************
// EFI_SMM_COMMUNICATE_HEADER
//*******************************************************
#define SMM_COMMUNICATE_HEADER_GUID \
{F328E36C-23B6-4a95-854B-32E19534CD75}

typedef struct {
  EFI_GUID                          HeaderGuid;
  UINTN                             MessageLength;
  UINT8                             Data[1];
} EFI_SMM_COMMUNICATE_HEADER;
```

HeaderGuid

Allows for disambiguation of the message format. See above for the definition of **SMM_COMMUNICATE_HEADER_GUID**. Type **EFI_GUID** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification.*

MessageLength

Describes the size of the message, not including the header.

Data

Designates an array of bytes that is *MessageLength* in size.

Status Codes Returned

EFI_SUCCESS	The message was successfully posted
EFI_INVALID_PARAMETER	The buffer was **NULL**.

EFI_SMM_BASE_PROTOCOL.RegisterCallback()

Summary

Registers a callback to execute within SMM. This allows receipt of messages created with **EFI_SMM_BASE_PROTOCOL.Communicate()**.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMM_CALLBACK_SERVICE) (
  IN struct _EFI_SMM_BASE_PROTOCOL    *This,
  IN EFI_HANDLE                       SmmImageHandle,
  IN EFI_SMM_CALLBACK_ENTRY_POINT     CallbackAddress,
  IN BOOLEAN                          MakeLast OPTIONAL,
  IN BOOLEAN                          FloatingPointSave OPTIONAL
  )
```

Parameters

This

The **EFI_SMM_BASE_PROTOCOL** instance.

SmmImageHandle

Handle of the callback service. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification.*

CallbackAddress

Address of the callback service. Type **EFI_SMM_CALLBACK_ENTRY_POINT** is defined in "Related Definitions" below.

MakeLast

If present, will stipulate that the handler is posted to be executed last in the dispatch table.

FloatingPointSave

An optional parameter that informs the **EFI_SMM_ACCESS_PROTOCOL** driver if it needs to save the floating point register state. If any of the handlers require this option, then the state will be saved for all of the handlers.

Description

This service allows the registration of a callback interface from within SMM. Calling this service from boot-services mode will result in an error. The purpose is to allow the handler to do the following:

- Operate in response to an SMI activation
- Receive a message from a non-SMM agent

The callback should have the **EFI_SMM_CALLBACK_ENTRY_POINT** interface defined; see “Related Definitions” below for its definition.

Each handler with the *MakeLast* flag should be sorted to the end of the list. In an IA-32 implementation, there is a separate queue for the 16-bit handlers that are dispatched prior to the queue for the native 32-bit handlers. The scope of the flags is for each queue.

There can be at most one first and one last. The expectation is that the first might be a dispatcher for child services, such as trap-register maintenance, and that the last would quiesce the source, such as setting the End of SMI (EOS) bit in the ICH.

Related Definitions

```
//*************************************************
// EFI_SMM_CALLBACK_ENTRY_POINT
//*************************************************

typedef
EFI_STATUS
(EFIAPI *EFI_SMM_CALLBACK_ENTRY_POINT) (
  IN EFI_HANDLE       SmmImageHandle,
  IN OUT VOID         *CommunicationBuffer OPTIONAL,
  IN OUT UINTN        *SourceSize OPTIONAL
  );
```

SmmImageHandle

A handle allocated by the SMM infrastructure code to uniquely designate a specific DXE SMM driver. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification.*

CommunicationBuffer

A pointer to a collection of data in memory that will be conveyed from a non-SMM environment into an SMM environment. The buffer must be contiguous, physically mapped, and be a physical address.

SourceSize

The size of the *CommunicationBuffer*.

Status Codes Returned

EFI_SUCCESS	The operation was successful
EFI_OUT_OF_RESOURCES	There was not enough space in the dispatch queue.
EFI_UNSUPPORTED	In runtime.
EFI_UNSUPPORTED	The caller is not in SMM.

EFI_SMM_BASE_PROTOCOL.InSmm()

Summary

Service to indicate whether the caller is already in SMM or not.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMM_INSIDE_OUT) (
  IN struct _EFI_SMM_BASE_PROTOCOL   *This,
  OUT BOOLEAN                        *InSmm
  )
```

Parameters

This

The **EFI_SMM_BASE_PROTOCOL** instance.

Boolean

Pointer to a Boolean. For IA-32, **TRUE** indicates that the caller is inside SMM. For the Itanium® processor family, **TRUE** indicates that the caller is servicing a PMI; **FALSE** if it is not.

Description

This service returns **TRUE** if the caller is inside SMM for IA-32 or servicing a PMI for the Itanium processor family. This function is useful because it allows the same constructor in the SMM driver to have the following two control paths:

- **InSmm == FALSE**: Can use boot services and allocate conventional memory.
- **InSmm == TRUE**: Can allocate SMRAM and perform other services.

Status Codes Returned

EFI_SUCCESS	The call returned successfully.
EFI_INVALID_PARAMETER	*InSmm* was **NULL**.

EFI_SMM_BASE_PROTOCOL.SmmAllocatePool()

Summary

Allocates pool memory from SMRAM for IA-32 or runtime memory for the Itanium® processor family.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMM_ALLOCATE_POOL) (
  IN struct _EFI_SMM_BASE_PROTOCOL   *This,
  IN EFI_MEMORY_TYPE                 PoolType,
  IN UINTN                           Size,
  OUT VOID                           **Buffer
  )
```

Parameters

This

The **EFI_SMM_BASE_PROTOCOL** instance.

PoolType

The type of pool to allocate. The only supported type is **EfiRuntimeServicesData**; the interface will internally map this runtime request to SMRAM for IA-32 and leave as this type for the Itanium processor family. Other types can be ignored. Type **EFI_MEMORY_TYPE** is defined in **AllocatePages()** in the *EFI 1.10 Specification.*

Size

The number of bytes to allocate from the pool.

Buffer

A pointer to a pointer to the allocated buffer if the call succeeds; undefined otherwise.

Description

This function allocates a memory region of *Size* bytes from memory of type *PoolType* and returns the address of the allocated memory in the location that is referenced by *Buffer*. This function allocates pages from EFI SMRAM memory for IA-32 as needed to grow the requested pool type. All allocations are 8-byte aligned.

PoolType can be ignored in that the type will always be SMRAM for IA-32 and runtime memory for the Itanium processor family.

The allocated pool memory is returned to the available pool with the **SmmFreePool()** function.

Status Codes Returned

EFI_SUCCESS	The requested number of bytes was allocated.
EFI_OUT_OF_RESOURCES	The pool requested could not be allocated.
EFI_UNSUPPORTED	In runtime.

EFI_SMM_BASE_PROTOCOL.SmmFreePool()

Summary

Returns pool memory to the system.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMM_FREE_POOL) (
  IN struct _EFI_SMM_BASE_PROTOCOL   *This,
  IN VOID                            *Buffer
  )
```

Parameters

This

The **EFI_SMM_BASE_PROTOCOL** instance.

Buffer

Pointer to the buffer to free.

Description

This function returns the memory specified by *Buffer* to the system. On return, the memory's type is EFI SMRAM memory. The *Buffer* that is freed must have been allocated by **SmmAllocatePool()**.

Status Codes Returned

EFI_SUCCESS	The memory was returned to the system.
EFI_INVALID_PARAMETER	*Buffer* was invalid.
EFI_UNSUPPORTED	In runtime.

EFI_SMM_BASE_PROTOCOL.GetSmstLocation()

Summary

Returns the location of the System Management Service Table (SMST).

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMM_GET_SMST_LOCATION) (
  IN struct _EFI_SMM_BASE_PROTOCOL   *This,
  IN OUT EFI_SMM_SYSTEM_TABLE        **Smst
  )
```

Parameters

This

The **EFI_SMM_BASE_PROTOCOL** instance.

Smst

Pointer to the SMST.

Description

This function returns the location of the System Management Service Table (SMST). The use of the API is such that a driver can discover the location of the SMST in its entry point and then cache it in some driver global variable so that the SMST can be invoked in subsequent callbacks.

Status Codes Returned

EFI_SUCCESS	The memory was returned to the system.
EFI_INVALID_PARAMETER	*Smst* was invalid.
EFI_UNSUPPORTED	Not in SMM.

SMM Access Protocol

EFI_SMM_ACCESS_PROTOCOL

Summary

This protocol is used to control the visibility of the SMRAM on the platform. The expectation is that the north bridge or memory controller would publish this protocol. For example, the Memory Controller Hub (MCH) has the hardware provision for this type of control. Because of the protected, distinguished class of memory for IA-32 systems, the expectation is that this protocol would be supported only on IA-32 systems.

GUID

```
#define EFI_SMM_ACCESS_PROTOCOL_GUID \
  { 0x3792095a, 0xe309, 0x4c1e, 0xaa, 0x01, 0x85, 0xf5, 0x65,
   0x5a, 0x17, 0xf1 }
```

Protocol Interface Structure

```
typedef struct _EFI_SMM_ACCESS_PROTOCOL {
  EFI_SMM_OPEN              Open;
  EFI_SMM_CLOSE             Close;
  EFI_SMM_LOCK              Lock;
  EFI_SMM_CAPABILITIES      GetCapabilities;
  BOOLEAN                   LockState;
  BOOLEAN                   OpenState;
} EFI_SMM_ACCESS_PROTOCOL;
```

Parameters

Open

Opens the SMRAM. See the **Open()** function description.

Close

Closes the SMRAM. See the **Close()** function description.

Lock

Locks the SMRAM. See the **Lock()** function description.

GetCapabilities

Gets information on possible SMRAM regions. See the **GetCapabilities()** function description.

LockState

Indicates the current state of the SMRAM. Set to **TRUE** if **any** region is locked.

OpenState

Indicates the current state of the SMRAM. Set to **TRUE** if **any** region is open.

Description

The **EFI_SMM_ACCESS_PROTOCOL** is used on the platform chipset device. It is a required protocol for a platform chipset. This protocol is useable only in boot-service mode. There is no analogous runtime protocol.

The principal role of this protocol interface is to provide an abstraction for the memory controller manipulation of SMRAM. This type of capability is available only on IA-32 platforms, where the SMRAM is an actual processor mode with bus cycles that allow the chipset to generate special SMRAM decodes. This being said, the principal functionality found in the memory controller includes the following:

- Exposing the SMRAM to all non-SMM agents, or the "open" state
- Shrouding the SMRAM to all but the SMM agents, or the "closed" state
- Preserving the system integrity, or "locking" the SMRAM, such that the settings cannot be perturbed by either boot service or runtime agents

This protocol will be published in the same fashion as other non-EFI Driver Model EFI drivers. It will not have a binding protocol. Instead, the driver should be stored in a firmware file as any other EFI driver.

EFI_SMM_ACCESS_PROTOCOL.Open()

Summary

Opens the SMRAM area to be accessible by a boot-service driver.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMM_OPEN) (
  IN struct _EFI_SMM_ACCESS_PROTOCOL  *This,
  UINTN                               DescriptorIndex
  );
```

Parameters

This

The **EFI_SMM_ACCESS_PROTOCOL** instance.

DescriptorIndex

Indicates that the driver wishes to open the memory tagged by this index. *DescriptorIndex* is an offset into the list of **EFI_SMRAM_DESCRIPTOR** data structures that describe the possible SMRAM mappings. Type **EFI_SMRAM_DESCRIPTOR** is defined in **EFI_SMM_ACCESS_PROTOCOL.GetCapabilities()**.

Description

This function enables access to the SMRAM region for purposes of copying handlers. This service is an abstraction of a programmatic access to some hardware that enables decode of the SMRAM from the boot service space.

Status Codes Returned

EFI_SUCCESS	The operation was successful.
EFI_INVALID_PARAMETER	The given *DescriptorIndex* is not supported.
EFI_NOT_STARTED	The SMM base service has not been initialized.

EFI_SMM_ACCESS_PROTOCOL.Close()

Summary

Inhibits access to the SMRAM.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMM_CLOSE) (
  IN struct _EFI_SMM_ACCESS_PROTOCOL   *This,
  UINTN                                DescriptorIndex
  );
```

Parameters

This

The **EFI_SMM_ACCESS_PROTOCOL** instance.

DescriptorIndex

Indicates that the driver wishes to open the memory tagged by this index. *DescriptorIndex* is an offset into the list of **EFI_SMRAM_DESCRIPTOR** data structures that describe the possible SMRAM mappings. Type **EFI_SMRAM_DESCRIPTOR** is defined in **EFI_SMM_ACCESS_PROTOCOL.GetCapabilities()**.

Related Definitions

This function disables access to the SMRAM region for purposes of copying handlers. This service is an abstraction of a programmatic access to some hardware that disables decode of the SMRAM from the boot service space.

Status Codes Returned

EFI_SUCCESS	The operation was successful.
EFI_DEVICE_ERROR	The given *DescriptorIndex* is not open.
EFI_INVALID_PARAMETER	The given *DescriptorIndex* is not supported.
EFI_NOT_STARTED	The SMM base service has not been initialized.

EFI_SMM_ACCESS_PROTOCOL.Lock()

Summary

Inhibits access to the SMRAM.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMM_LOCK) (
  IN struct _EFI_SMM_ACCESS_PROTOCOL  *This,
  UINTN                               DescriptorIndex
  );
```

Parameters

This

The **EFI_SMM_ACCESS_PROTOCOL** instance.

DescriptorIndex

Indicates that the driver wishes to open the memory tagged by this index. *DescriptorIndex* is an offset into the list of **EFI_SMRAM_DESCRIPTOR** data structures that describe the possible SMRAM mappings. Type **EFI_SMRAM_DESCRIPTOR** is defined in **EFI_SMM_ACCESS_PROTOCOL.GetCapabilities()**.

Related Definitions

This function prohibits access to the SMRAM region. This function is usually implemented such that it is a write-once operation. An implementation of the **EFI_SMM_ACCESS_PROTOCOL** should register a notification on **ExitBootServices()** to at least lock the system at this point, if it was not already locked by an earlier agent.

Status Codes Returned

EFI_SUCCESS	The device was successfully locked.
EFI_DEVICE_ERROR	The given *DescriptorIndex* is not open.
EFI_INVALID_PARAMETER	The given *DescriptorIndex* is not supported.
EFI_NOT_STARTED	The SMM base service has not been initialized.

EFI_SMM_ACCESS_PROTOCOL.GetCapabilities()

Summary

Queries the memory controller for the possible regions that will support SMRAM. This protocol is optional for Itanium®-based systems but mandatory for IA-32.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMM_CAPABILITIES) (
  IN struct _EFI_SMM_ACCESS_PROTOCOL      *This,
  IN OUT UINTN                            *SmramMapSize,
  IN OUT EFI_SMRAM_DESCRIPTOR             *SmramMap
  );
```

Parameters

This

The **EFI_SMM_ACCESS_PROTOCOL** instance.

SmramMapSize

A pointer to the size, in bytes, of the *SmramMemoryMap* buffer. On input, this value is the size of the buffer that is allocated by the caller. On output, it is the size of the buffer that was returned by the firmware if the buffer was large enough, or, if the buffer was too small, the size of the buffer that is needed to contain the map.

SmramMap

A pointer to the buffer in which firmware places the current memory map. The map is an array of **EFI_SMRAM_DESCRIPTOR**s. Type **EFI_SMRAM_DESCRIPTOR** is defined in "Related Definitions" below.

Description

This function enables access to the SMRAM region for purposes of copying handlers.

This data structure forms the contract between the **SMM_ACCESS** and **SMM_BASE** drivers. There is an ambiguity when any SMRAM region is remapped. For example, on some chipsets, H-SEG can be initialized at physical address 0xA0000–0xBFFFFh but is later accessed at the processor address 0xFEEA00000–0xFEEBFFFFF. There is currently no way for the **SMM_BASE** driver to know that it must use two different addresses depending on what it is trying to do. As a result, initial configuration and loading can use the physical address *PhysicalStart* while in non-SMM, boot services mode. However, once the region has been opened and needs to be accessed by agents in SMM, the *CpuStart* address must be used.

This protocol publishes the available memory that the chipset can shroud for the use of installing code. This API is not useful for Itanium-based systems in that there is no distinguished bus cycle from code running after a PMI is invoked, so in this case just runtime memory allocation should suffice. For IA-32, however, there are chipset provisions for providing SMRAM capability near the top of the physical memory or in locations such as behind the legacy frame buffer.

These regions serve the dual purpose of describing which regions have been open, closed, or locked. In addition, these regions may include overlapping memory ranges, depending on the chipset implementation. The latter might include a chipset that supports T-SEG, where memory near the top of the physical DRAM can be allocated for SMRAM too.

The key thing to note is that the regions that are described by the protocol are a subset of the capabilities of the hardware. The subset of the regions that are exposed include those that are conveyed in the platform-specific implementation of this driver or using the HOB handoff from a platform PEIM into a portable version of this driver. In the latter case, the HOB is defined in PEI Support.

Related Definitions

```
//*******************************************************
//EFI_SMRAM_STATE
//*******************************************************
//
// Hardware state
//
#define  EFI_SMRAM_OPEN                  0x00000001
#define  EFI_SMRAM_CLOSED                0x00000002
#define  EFI_SMRAM_LOCKED                0x00000004
//
// Capability
//
#define  EFI_CACHEABLE                   0x00000008
//
// Logical usage
//
#define  EFI_ALLOCATED                   0x00000010
//
// Directive prior to usage
//
#define  EFI_NEEDS_TESTING               0x00000020
#define  EFI_NEEDS_ECC_INITIALIZATION    0x00000040


//*******************************************************
// EFI_SMRAM_DESCRIPTOR
//*******************************************************
typedef struct _EFI_SMRAM_DESCRIPTOR {
  EFI_PHYSICAL_ADDRESS  PhysicalStart;
  EFI_PHYSICAL_ADDRESS  CpuStart;
  UINT64                PhysicalSize;
  UINT64                RegionState;
} EFI_SMRAM_DESCRIPTOR;
```

PhysicalStart

Designates the physical address of the SMRAM in memory. This view of memory is the same as seen by I/O-based agents, for example, but it may not be the address seen by the processors. Type **EFI_PHYSICAL_ADDRESS** is defined in **AllocatePages()** in the *EFI 1.10 Specification.*

CpuStart

Designates the address of the SMRAM, as seen by software executing on the processors. This address may or may not match *PhysicalStart*.

PhysicalSize

Describes the number of bytes in the SMRAM region.

RegionState

Describes the accessibility attributes of the SMRAM. These attributes include the hardware state (e.g., Open/Closed/Locked), capability (e.g., cacheable), logical allocation (e.g., allocated), and pre-use initialization (e.g., needs testing/ECC initialization).

Status Codes Returned

EFI_SUCCESS	The chipset supported the given resource.
EFI_BUFFER_TOO_SMALL	The *SmramMap* parameter was too small. The current buffer size needed to hold the memory map is returned in *SmramMapSize*.

SMM Control Protocol

EFI_SMM_CONTROL_PROTOCOL

Summary

This protocol is used initiate SMI/PMI activations. This protocol could be published by either of the following:

- A processor driver to abstract the SMI/PMI IPI
- The driver that abstracts the ASIC that is supporting the APM port, such as the ICH in an Intel® chipset

Because of the possibility of performing SMI or PMI IPI transactions, the ability to generate this event from a platform chipset agent is an optional capability for both IA-32 and Itanium®-based systems.

GUID

```
#define EFI_SMM_CONTROL_PROTOCOL_GUID \
  { 0x8d12e231, 0xc667, 0x4fd1, 0x98, 0xf2, 0x24, 0x49, 0xa7,
   0xe7, 0xb2, 0xe5 }
```

Protocol Interface Structure

```
typedef struct _EFI_SMM_CONTROL_PROTOCOL {
  EFI_SMM_ACTIVATE              Trigger;
  EFI_SMM_DEACTIVATE            Clear;
  EFI_SMM_GET_REGISTER_INFO     GetRegisterInfo;
  UINTN                         MinimumTriggerPeriod;
} EFI_SMM_CONTROL_PROTOCOL;
```

Parameters

Trigger

Initiates the SMI/PMI activation. See the **Trigger()** function description.

Clear

Quiesces the SMI/PMI activation. See the **Clear()** function description.

GetRegisterInfo

Provides data on the register used as the source of the SMI. See the **GetRegisterInfo()** function description.

MinimumTriggerPeriod

Minimum interval at which the platform can set the period. A maximum is not specified in that the SMM infrastructure code can emulate a maximum interval that is greater than the hardware capabilities by using software emulation in the SMM infrastructure code. Type **EFI_SMM_PERIOD** is defined in "Related Definitions" below.

Description

The **EFI_SMM_CONTROL_PROTOCOL** is used by the platform chipset or processor driver. This protocol is useable both in boot services and runtime. The runtime aspect is so that an implementation of **EFI_SMM_BASE_PROTOCOL.Communicate()** can layer upon this service and provide an SMI callback from a general EFI runtime driver.

The purpose of this protocol is to provide an abstraction to the platform hardware that generates an SMI or PMI. There are often I/O ports that, when accessed, will engender the SMI or PMI. Also, this hardware optionally supports the periodic generation of these signals.

Related Definitions

```
//*************************************************
// EFI_SMM_PERIOD
//*************************************************
typedef EFI_SMM_PERIOD UINTN
```

The period is in increments of 10 ns.

EFI_SMM_CONTROL_PROTOCOL.Trigger()

Summary

Invokes SMI activation from either the preboot or runtime environment.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMM_ACTIVATE) (
  IN struct _EFI_SMM_CONTROL_PROTOCOL   *This,
  IN OUT INT8                           *ArgumentBuffer OPTIONAL,
  IN OUT UINTN                          *ArgumentBufferSize
OPTIONAL,
  IN BOOLEAN                            Periodic OPTIONAL,
  IN UINTN                              ActivationInterval
OPTIONAL
  );
```

Parameters

This

The **EFI_SMM_CONTROL_PROTOCOL** instance.

ArgumentBuffer

Optional sized data to pass into the protocol activation. This data might be a value written to an APM port, for example.

ArgumentBufferSize

Optional size of the data.

Periodic

Optional mechanism to engender a periodic stream.

ActivationInterval

Optional parameter to repeat at this period one time or, if the *Periodic* Boolean is set, periodically.

Description

This function engenders the PMI/SMI activation.

Status Codes Returned

EFI_SUCCESS	The SMI/PMI has been engendered.
EFI_DEVICE_ERROR	The timing is unsupported.
EFI_INVALID_PARAMETER	The activation period is unsupported.
EFI_NOT_STARTED	The SMM base service has not been initialized.

EFI_SMM_CONTROL_PROTOCOL.Clear()

Summary

Clears any system state that was created in response to the **Trigger()** call.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMM_DEACTIVATE) (
  IN struct _EFI_SMM_CONTROL_PROTOCOL  *This,
  IN BOOLEAN                           Periodic OPTIONAL
  );
```

Parameters

This

The **EFI_SMM_CONTROL_PROTOCOL** instance.

Periodic

Optional parameter to repeat at this period one time or, if the *Periodic* Boolean is set, periodically.

Description

This function acknowledges and causes the deassertion of the PMI/SMI activation source.

Status Codes Returned

EFI_SUCCESS	The SMI/PMI has been engendered.
EFI_DEVICE_ERROR	The source could not be cleared.
EFI_INVALID_PARAMETER	The service did not support the *Periodic* input argument.

EFI_SMM_CONTROL_PROTOCOL.GetRegisterInfo()

Summary

Provides information on the source register used to generate the SMI.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMM_GET_REGISTER_INFO) (
  IN EFI_SMM_CONTROL_PROTOCOL          *This,
  IN OUT EFI_SMM_CONTROL_REGISTER     *SmiRegister
);
```

Parameters

This

Pointer to the **EFI_SMM_CONTROL_PROTOCOL** instance.

SmiRegister

Pointer to the SMI register description structure. Type **EFI_SMM_CONTROL_REGISTER** is defined in "Related Definitions" below.

Description

The **GetRegisterInfo()** function provides information on the state of the activation mechanism that is used for a synchronous SMI. Specifically, there are two types of SMI generation:

- Synchronous
- Asynchronous

The former would include **Trigger()** activations, and the latter would include periodic or I/O traps. See **EFI_SMM_PERIODIC_TIMER_DISPATCH_PROTOCOL** for more information on periodic traps.

This service can be used by a processor-specific driver that publishes the **EFI_SMM_BASE_PROTOCOL** to discriminate between synchronous and asynchronous sources.

Related Definitions

```
//*******************************************************
// EFI_SMM_CONTROL_REGISTER
//*******************************************************

typedef struct {
  UINT8     SmiTriggerRegister;
  UINT8     SmiDataRegister;
} EFI_SMM_CONTROL_REGISTER
```

SmiTriggerRegister

Describes the I/O location of the particular port that engendered the synchronous SMI. For example, this location can include but is not limited to the traditional PC-AT* APM port of 0B2h.

SmiDataRegister

Describes the value that was written to the respective activation port.

Status Codes Returned

EFI_SUCCESS	The register structure has been returned.

7
SMM Child Dispatch Protocols

Introduction

The services described in this chapter describe a series of protocols that abstract installation of handlers for a chipset-specific SMM design. As opposed to the **EFI_SMM_BASE_PROTOCOL.Register()** service, these services are called from the SMM driver constructors while in SMM. As such, these services are all scoped to be usable only from within SMRAM.

The following protocols are defined in this chapter:

- **EFI_SMM_SW_DISPATCH_PROTOCOL**
- **EFI_SMM_SX_DISPATCH_PROTOCOL**
- **EFI_SMM_PERIODIC_TIMER_DISPATCH_PROTOCOL**
- **EFI_SMM_USB_DISPATCH_PROTOCOL**
- **EFI_SMM_GPI_DISPATCH_PROTOCOL**
- **EFI_SMM_STANDBY_BUTTON_DISPATCH_PROTOCOL**
- **EFI_SMM_POWER_BUTTON_DISPATCH_PROTOCOL**

SMM Software Dispatch Protocol

EFI_SMM_SW_DISPATCH_PROTOCOL

Summary

Provides the parent dispatch service for a given SMI source generator.

GUID

```
#define EFI_SMM_SW_DISPATCH_PROTOCOL_GUID    \
  { 0xe541b773, 0xdd11, 0x420c, 0xb0, 0x26, 0xdf, 0x99, 0x36, 0x53,
   0xf8, 0xbf }
```

Protocol Interface Structure

```
typedef struct _EFI_SMM_ICHN_DISPATCH_PROTOCOL {
  EFI_SMM_SW_REGISTER        Register;
  EFI_SMM_SW_UNREGISTER      UnRegister;
  UINTN                      MaximumSwiValue;
} EFI_SMM_ICHN_DISPATCH_PROTOCOL;
```

Parameters

Register

Installs a child service to be dispatched by this protocol. See the **Register()** function description.

UnRegister

Removes a child service dispatched by this protocol. See the **UnRegister()** function description.

MaximumSwiValue

A read-only field that describes the maximum value that can be used in the **EFI_SMM_SW_DISPATCH_PROTOCOL.Register()** service.

Description

The **EFI_SMM_SW_DISPATCH_PROTOCOL** provides the ability to install child handlers for the given software. These handlers will respond to software interrupts, and the maximum software interrupt in the **EFI_SMM_SW_DISPATCH_CONTEXT** is denoted by *MaximumSwiValue*.

EFI_SMM_SW_DISPATCH_PROTOCOL.Register()

Summary

Provides the parent dispatch service for a given SMI source generator.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMM_SW_REGISTER) (
  IN  struct _EFI_SMM_SW_DISPATCH_PROTOCOL  *This,
  IN  EFI_SMM_SW_DISPATCH                   DispatchFunction,
  IN  EFI_SMM_SW_DISPATCH_CONTEXT           *DispatchContext,
  OUT EFI_HANDLE                            *DispatchHandle
  );
```

Parameters

This

Pointer to the **EFI_SMM_SW_DISPATCH_PROTOCOL** instance.

DispatchFunction

Function to install. Type **EFI_SMM_SW_DISPATCH** is defined in "Related Definitions" below.

DispatchContext

Pointer to the dispatch function's context. The caller fills in this context before calling the **Register()** function to indicate to the **Register()** function the software SMI input value for which the dispatch function should be invoked. Type **EFI_SMM_SW_DISPATCH_CONTEXT** is defined in "Related Definitions" below.

DispatchHandle

Handle generated by the dispatcher to track the function instance. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification.*

Description

This service registers a given instance of the given source.

Related Definitions

```
//*******************************************************
// EFI_SMM_SW_DISPATCH
//*******************************************************
typedef
VOID
(EFIAPI *EFI_SMM_SW_DISPATCH) (
  IN  EFI_HANDLE                   DispatchHandle,
  IN  EFI_SMM_SW_DISPATCH_CONTEXT  *DispatchContext
  );
```

DispatchHandle

Handle of this dispatch function. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification.*

DispatchContext

Pointer to the dispatch function's context. The *DispatchContext* fields are filled in by the software dispatching driver prior to invoking this dispatch function. The dispatch function will only be called for input values for which it is registered. Type **EFI_SMM_SW_DISPATCH_CONTEXT** is defined below.

```
//*******************************************************
// EFI_SMM_SW_DISPATCH_CONTEXT
//*******************************************************
//
// A particular chipset may not support all possible software SMI
// input values. For example, the ICH supports only values 00h to
// 0FFh.  The parent only allows a single child registration for
// each SwSmiInputValue.
//
typedef struct {
  UINTN     SwSmiInputValue;
} EFI_SMM_SW_DISPATCH_CONTEXT;
```

SwSmiInputValue

A number that is used during the registration process to tell the dispatcher which software input value to use to invoke the given handler.

Status Codes Returned

EFI_SUCCESS	The dispatch function has been successfully registered and the SMI source has been enabled.
EFI_DEVICE_ERROR	The driver was unable to enable the SMI source.
EFI_INVALID_PARAMETER	*DispatchContext* is invalid. The SW SMI input value is not within a valid range.
EFI_OUT_OF_RESOURCES	There is not enough memory (system or SMM) to manage this child.

EFI_SMM_SW_DISPATCH_PROTOCOL.UnRegister()

Summary

Unregisters a software service.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMM_SW_UNREGISTER) (
  IN  struct _EFI_SMM_SW_DISPATCH_PROTOCOL  *This,
  IN  EFI_HANDLE                            DispatchHandle
  );
```

Parameters

This

Pointer to the **EFI_SMM_SW_DISPATCH_PROTOCOL** instance.

DispatchHandle

Handle of the service to remove. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification.*

Description

This service will remove a handler.

Status Codes Returned

EFI_SUCCESS	The service has been successfully removed.
EFI_INVALID_PARAMETER	The *DispatchHandle* was not valid.

SMM Sx Dispatch Protocol

EFI_SMM_SX_DISPATCH_PROTOCOL

Summary

Provides the parent dispatch service for a given Sx-state source generator.

GUID

```
#define EFI_SMM_SX_DISPATCH_PROTOCOL_GUID    \
  { 0x14fc52be, 0x1dc, 0x426c, 0x91, 0xae, 0xa2, 0x3c, 0x3e,
0x22, 0xa, 0xe8 }
```

Protocol Interface Structure

```
typedef struct _EFI_SMM_SX_DISPATCH_PROTOCOL {
  EFI_SMM_SX_REGISTER     Register;
  EFI_SMM_SX_UNREGISTER   UnRegister;
} EFI_SMM_SX_DISPATCH_PROTOCOL;
```

Parameters

Register

Installs a child service to be dispatched by this protocol. See the **Register()** function description.

UnRegister

Removes a child service dispatched by this protocol. See the **UnRegister()** function description.

Description

The **EFI_SMM_SX_DISPATCH_PROTOCOL** provides the ability to install child handlers for the given event types.

EFI_SMM_SX_DISPATCH_PROTOCOL.Register()

Summary

Provides the parent dispatch service for a given Sx source generator.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMM_SX_REGISTER) (
  IN  struct _EFI_SMM_SX_DISPATCH_PROTOCOL     *This,
  IN  EFI_SMM_SX_DISPATCH                      DispatchFunction,
  IN  EFI_SMM_SX_DISPATCH_CONTEXT              *DispatchContext,
  OUT EFI_HANDLE                               *DispatchHandle
  );
```

Parameters

This

Pointer to the **EFI_SMM_SX_DISPATCH_PROTOCOL** instance.

DispatchFunction

Function to install. Type **EFI_SMM_SX_DISPATCH** is defined in "Related Definitions" below.

DispatchContext

Pointer to the dispatch function's context. The caller in fills this context before calling the **Register()** function to indicate to the **Register()** function on which Sx state type and phase the caller wishes to be called back. For this interface, the Sx driver will call the registered handlers for all Sx type and phases, so the Sx state handler(s) must check the *Type* and *Phase* field of **EFI_SMM_SX_DISPATCH_CONTEXT** and act accordingly.

DispatchHandle

Handle of the dispatch function, for when interfacing with the parent Sx state SMM driver. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification.*

Description

This service registers a given instance of the given source.

Related Definitions

```
//*******************************************************
// EFI_SMM_SX_DISPATCH
//*******************************************************
typedef
VOID
(EFIAPI *EFI_SMM_SX_DISPATCH) (
  IN  EFI_HANDLE                  DispatchHandle,
  IN  EFI_SMM_SX_DISPATCH_CONTEXT   *DispatchContext
  );
```

DispatchHandle

Handle of this dispatch function. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification.*

DispatchContext

Pointer to the dispatch function's context. The *Type* and *Phase* fields are filled in by the Sx dispatch driver prior to invoking this dispatch function. For this interface, the Sx driver will call the dispatch function for all Sx type and phases, so the Sx state handler(s) must check the *Type* and *Phase* field of **EFI_SMM_SX_DISPATCH_CONTEXT** and act accordingly. Type **EFI_SMM_SX_DISPATCH_CONTEXT** is defined below.

```
//*******************************************************
// EFI_SMM_SX_DISPATCH_CONTEXT
//*******************************************************
typedef struct {
  EFI_SLEEP_TYPE     Type;
  EFI_SLEEP_PHASE    Phase;
} EFI_SMM_SX_DISPATCH_CONTEXT;
```

```
//*******************************************************
// EFI_SLEEP_TYPE
//*******************************************************
typedef enum {
  SxS0,
  SxS1,
  SxS2,
  SxS3,
  SxS4,
  SxS5,
  EfiMaximumSleepType
} EFI_SLEEP_TYPE;
```

```
//*******************************************************
// EFI_SLEEP_PHASE
//*******************************************************
typedef enum {
  SxEntry,
  SxExit,
  EfiMaximumPhase
} EFI_SLEEP_PHASE;
```

Status Codes Returned

EFI_SUCCESS	The dispatch function has been successfully registered and the SMI source has been enabled.
EFI_UNSUPPORTED	The Sx driver or hardware does not support that Sx *Type*/*Phase*.
EFI_DEVICE_ERROR	The Sx driver was unable to enable the SMI source.
EFI_INVALID_PARAMETER	*DispatchContext* is invalid. The ICHN input value is not within a valid range.
EFI_OUT_OF_RESOURCES	There is not enough memory (system or SMM) to manage this child.

EFI_SMM_SX_DISPATCH_PROTOCOL.UnRegister()

Summary

Unregisters an Sx-state service.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMM_SX_UNREGISTER) (
  IN  struct _EFI_SMM_SX_DISPATCH_PROTOCOL_  *This,
  IN  EFI_HANDLE                             DispatchHandle
  );
```

Parameters

This

Pointer to the **EFI_SMM_SX_DISPATCH_PROTOCOL** instance.

DispatchHandle

Handle of the service to remove. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification.*

Description

This service removes a handler.

Status Codes Returned

EFI_SUCCESS	The service has been successfully removed.
EFI_INVALID_PARAMETER	The *DispatchHandle* was not valid.

SMM Periodic Timer Dispatch Protocol

EFI_SMM_PERIODIC_TIMER_DISPATCH_PROTOCOL

Summary

Provides the parent dispatch service for the periodical timer SMI source generator.

GUID

```
#define EFI_SMM_PERIODIC_TIMER_DISPATCH_PROTOCOL_GUID    \
  { 0x9cca03fc, 0x4c9e, 0x4a19, 0x9b, 0x6, 0xed, 0x7b, 0x47, 0x9b,
   0xde, 0x55 }
```

Protocol Interface Structure

```
typedef struct _EFI_SMM_PERIODIC_TIMER_DISPATCH_PROTOCOL {
  EFI_SMM_PERIODIC_TIMER_REGISTER      Register;
  EFI_SMM_PERIODIC_TIMER_UNREGISTER    UnRegister;
  EFI_SMM_PERIODIC_TIMER_INTERVAL      GetNextShorterInterval;
} EFI_SMM_PERIODIC_TIMER_DISPATCH_PROTOCOL;
```

Parameters

Register

Installs a child service to be dispatched by this protocol. See the **Register()** function description.

UnRegister

Removes a child service dispatched by this protocol. See the **UnRegister()** function description.

GetNextShorterInterval

Returns the next SMI tick period that is supported by the chipset. See the **GetNextShorterInterval()** function description.

Description

The **EFI_SMM_PERIODIC_TIMER_DISPATCH_PROTOCOL** provides the ability to install child handlers for the given event types.

EFI_SMM_PERIODIC_TIMER_DISPATCH_PROTOCOL.Register()

Summary

Provides the parent dispatch service for a given SMI source generator.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMM_PERIODIC_TIMER_REGISTER) (
  IN  struct _EFI_SMM_PERIODIC_TIMER_DISPATCH_PROTOCOL *This,
  IN  EFI_SMM_PERIODIC_TIMER_DISPATCH
       DispatchFunction,
  IN  EFI_SMM_PERIODIC_TIMER_DISPATCH_CONTEXT
         *DispatchContext,
  OUT EFI_HANDLE
       *DispatchHandle
  );
```

Parameters

This

Pointer to the **EFI_SMM_PERIODIC_TIMER_DISPATCH_PROTOCOL** instance.

DispatchFunction

Function to install. Type **EFI_SMM_PERIODIC_TIMER_DISPATCH** is defined in "Related Definitions" below.

DispatchContext

Pointer to the dispatch function's context. The caller fills this context in before calling the **Register()** function to indicate to the **Register()** function the period at which the dispatch function should be invoked. Type **EFI_SMM_PERIODIC_TIMER_DISPATCH_CONTEXT** is defined in "Related Definitions" below.

DispatchHandle

Handle generated by the dispatcher to track the function instance. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification.*

Description

This service registers a given instance of the given source.

Related Definitions

```
//*******************************************************
// EFI_SMM_PERIODIC_TIMER_DISPATCH
//*******************************************************

typedef
VOID
(EFIAPI *EFI_SMM_PERIODIC_TIMER_DISPATCH) (
  IN  EFI_HANDLE                             DispatchHandle,
  IN  EFI_SMM_PERIODIC_TIMER_DISPATCH_CONTEXT  *DispatchContext
  );
```

DispatchHandle

Handle of this dispatch function. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification.*

DispatchContext

Pointer to the dispatch function's context. The *DispatchContext* fields are filled in by the dispatching driver prior to invoking this dispatch function. Type **EFI_SMM_PERIODIC_TIMER_DISPATCH_CONTEXT** is defined in "Related Definitions" below.

```
//*******************************************************
// EFI_SMM_PERIODIC_TIMER_DISPATCH_CONTEXT
//*******************************************************

typedef struct {
  UINT64    Period;
  UINT64    SmiTickInterval;
  UINT64    ElapsedTime;
} EFI_SMM_PERIODIC_TIMER_DISPATCH_CONTEXT;
```

Period

The minimum period of time in 100 nanosecond units that the child gets called. The child will be called back after a time greater than the time *Period*.

SmiTickInterval

The period of time interval between SMIs. Children of this interface should use this field when registering for periodic timer intervals when a finer granularity periodic SMI is desired.

Example: A chipset supports periodic SMIs on every 64 ms or 2 seconds. A child wishes to schedule a periodic SMI to fire on a period of 3 seconds. There are several ways to approach the problem:

- The child may accept a 4 second periodic rate, in which case it registers with the following:

```
Period = 40000
SmiTickInterval = 20000
```

The resulting SMI will occur every 2 seconds with the child called back on every second SMI.

NOTE

The same result would occur if the child set `SmiTickInterval = 0`.

- The child may choose the finer granularity SMI (64 ms):

```
Period = 30000
SmiTickInterval = 640
```

The resulting SMI will occur every 64 ms with the child called back on every 47th SMI.

NOTE

The child driver should be aware that this will result in more SMIs occurring during system runtime, which can negatively impact system performance.

`ElapsedTime`

The actual time in 100 nanosecond units elapsed since last called. A value of 0 indicates an unknown amount of time.

Status Codes Returned

EFI_SUCCESS	The dispatch function has been successfully registered and the SMI source has been enabled.
EFI_DEVICE_ERROR	The driver was unable to enable the SMI source.
EFI_INVALID_PARAMETER	*`DispatchContext`* is invalid. The ICHN input value is not within a valid range.
EFI_OUT_OF_RESOURCES	There is not enough memory (system or SMM) to manage this child.

EFI_SMM_PERIODIC_TIMER_DISPATCH_PROTOCOL.UnRegister()

Summary

Unregisters a periodic timer service.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMM_PERIODIC_TIMER_UNREGISTER) (
  IN  struct _EFI_SMM_PERIODIC_TIMER_DISPATCH_PROTOCOL   *This,
  IN EFI_HANDLE
        DispatchHandle
  );
```

Parameters

This

Pointer to the **EFI_SMM_PERIODIC_TIMER_DISPATCH_PROTOCOL** instance.

DispatchHandle

Handle of the service to remove. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification.*

Description

This service removes a handler.

Status Codes Returned

EFI_SUCCESS	The service has been successfully removed.
EFI_INVALID_PARAMETER	The *DispatchHandle* was not valid.

EFI_SMM_PERIODIC_TIMER_DISPATCH_PROTOCOL.GetNextShorterInterval()

Summary

Returns the next SMI tick period that is supported by the chipset.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMM_PERIODIC_TIMER_INTERVAL) (
  IN  struct _EFI_SMM_PERIODIC_TIMER_DISPATCH_PROTOCOL   *This,
  IN OUT UINT64
                                              **SmiTickInterval
  );
```

Parameters

This

Pointer to the **EFI_SMM_PERIODIC_TIMER_DISPATCH_PROTOCOL** instance.

SmiTickInterval

Pointer to pointer of the next shorter SMI interval period that is supported by the child. This parameter works as a get-first, get-next field. The first time that this function is called, **SmiTickInterval* should be set to **NULL** to get the longest SMI interval. The returned **SmiTickInterval* should be passed in on subsequent calls to get the next shorter interval period until **SmiTickInterval* **= NULL**.

Description

This services returns the next SMI tick period that is supported by the chipset. The order returned is from longest to shortest interval period.

Status Codes Returned

EFI_SUCCESS	The service returned successfully.

SMM USB Dispatch Protocol

EFI_SMM_USB_DISPATCH_PROTOCOL

Summary

Provides the parent dispatch service for the USB SMI source generator.

GUID

```
#define EFI_SMM_USB_DISPATCH_PROTOCOL_GUID    \
  { 0xa05b6ffd, 0x87af, 0x4e42, 0x95, 0xc9, 0x62, 0x28, 0xb6, 0x3c,
   0xf3, 0xf3 }
```

Protocol Interface Structure

```
typedef struct _EFI_SMM_USB_DISPATCH_PROTOCOL {
  EFI_SMM_USB_REGISTER      Register;
  EFI_SMM_USB_UNREGISTER    UnRegister;
} EFI_SMM_USB_DISPATCH_PROTOCOL;
```

Parameters

Register

Installs a child service to be dispatched by this protocol. See the **Register()** function description.

UnRegister

Removes a child service dispatched by this protocol. See the **UnRegister()** function description.

Description

The **EFI_SMM_USB_DISPATCH_PROTOCOL** provides the ability to install child handlers for the given event types.

EFI_SMM_USB_DISPATCH_PROTOCOL.Register()

Summary

Provides the parent dispatch service for the USB SMI source generator.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMM_USB_REGISTER) (
  IN  struct _EFI_SMM_USB_DISPATCH_PROTOCOL    *This,
  IN  EFI_SMM_USB_DISPATCH                     DispatchFunction,
  IN  EFI_SMM_USB_DISPATCH_CONTEXT             *DispatchContext,
  OUT EFI_HANDLE                               *DispatchHandle
  );
```

Parameters

This

Pointer to the **EFI_SMM_USB_DISPATCH_PROTOCOL** instance.

DispatchFunction

Pointer to dispatch function to be invoked for this SMI source. Type **EFI_SMM_USB_DISPATCH** is defined in "Related Definitions" below.

DispatchContext

Pointer to the dispatch function's context. The caller fills this context in before calling the **Register()** function to indicate to the **Register()** function the USB SMI source for which the dispatch function should be invoked. Type **EFI_SMM_USB_DISPATCH_CONTEXT** is defined in "Related Definitions" below.

DispatchHandle

Handle generated by the dispatcher to track the function instance. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification.*

Description

This service registers a given instance of the given source.

Related Definitions

```
//*******************************************************
// EFI_SMM_USB_DISPATCH
//*******************************************************

typedef
VOID
(EFIAPI *EFI_SMM_USB_DISPATCH) (
  IN  EFI_HANDLE                    DispatchHandle,
  IN  EFI_SMM_USB_DISPATCH_CONTEXT  *DispatchContext
  );
```

DispatchHandle

Handle of this dispatch function. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification.*

DispatchContext

Pointer to the dispatch function's context. The *DispatchContext* fields are filled in by the dispatching driver prior to invoking this dispatch function. Type **EFI_SMM_USB_DISPATCH_CONTEXT** is defined below.

```
//*******************************************************
// EFI_SMM_USB_DISPATCH_CONTEXT
//*******************************************************

typedef struct {
  EFI_USB_SMI_TYPE            Type;
  EFI_DEVICE_PATH_PROTOCOL    *Device;
} EFI_SMM_USB_DISPATCH_CONTEXT;
```

Type

Describes whether this child handler will be invoked in response to a USB legacy emulation event, such as port-trap on the PS/2* keyboard control registers, or to a USB wake event, such as resumption from a sleep state. Type **EFI_USB_SMI_TYPE** is defined below.

Device

The device path is part of the context structure and describes the location of the particular USB host controller in the system for which this register event will occur. This location is important because of the possible integration of several USB host controllers in a system. Type **EFI_DEVICE_PATH** is defined in the *EFI 1.10 Specification.*

```
//*******************************************************
// EFI_USB_SMI_TYPE
//*******************************************************
typedef enum {
  UsbLegacy,
  UsbWake
} EFI_USB_SMI_TYPE;
```

Status Codes Returned

EFI_SUCCESS	The dispatch function has been successfully registered and the SMI source has been enabled.
EFI_DEVICE_ERROR	The driver was unable to enable the SMI source.
EFI_INVALID_PARAMETER	*DispatchContext* is invalid. The ICHN input value is not within valid range.
EFI_OUT_OF_RESOURCES	There is not enough memory (system or SMM) to manage this child.

EFI_SMM_USB_DISPATCH_PROTOCOL. UnRegister()

Summary

Unregisters a USB service.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMM_USB_UNREGISTER) (
  IN  struct _EFI_SMM_USB_DISPATCH_PROTOCOL     *This,
  IN EFI_HANDLE                                 DispatchHandle
  );
```

Parameters

This

Pointer to the **EFI_SMM_USB_DISPATCH_PROTOCOL** instance.

DispatchHandle

Handle of the service to remove. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification.*

Description

This service removes a handler.

Status Codes Returned

EFI_SUCCESS	The dispatch function has been successfully unregistered and the SMI source has been disabled, if there are no other registered child dispatch functions for this SMI source.
EFI_INVALID_PARAMETER	The *DispatchHandle* was not valid.

SMM General Purpose Input (GPI) Dispatch Protocol

EFI_SMM_GPI_DISPATCH_PROTOCOL

Summary

Provides the parent dispatch service for the General Purpose Input (GPI) SMI source generator.

GUID

```
#define EFI_SMM_GPI_DISPATCH_PROTOCOL_GUID    \
  { 0xe0744b81, 0x9513, 0x49cd, 0x8c, 0xea, 0xe9, 0x24, 0x5e, 0x70,
   0x39, 0xda }
```

Protocol Interface Structure

```
typedef struct _EFI_SMM_GPI_DISPATCH_PROTOCOL {
  EFI_SMM_GPI_REGISTER       Register;
  EFI_SMM_GPI_UNREGISTER     UnRegister;
  UINTN                      NumSupportedGpis;
} EFI_SMM_GPI_DISPATCH_PROTOCOL;
```

Parameters

Register

Installs a child service to be dispatched by this protocol. See the **Register()** function description.

UnRegister

Removes a child service dispatched by this protocol. See the **UnRegister()** function description.

NumSupportedGpis

Denotes the maximum value of inputs that can have handlers attached.

Description

The **EFI_SMM_GPI_DISPATCH_PROTOCOL** provides the ability to install child handlers for the given event types. Several inputs can be enabled. This purpose of this interface is to generate an SMI in response to any of these inputs having a true value provided.

EFI_SMM_GPI_DISPATCH_PROTOCOL.Register()

Summary

Registers a child SMI source dispatch function with a parent SMM driver.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMM_GPI_REGISTER) (
  IN  struct _EFI_SMM_GPI_DISPATCH_PROTOCOL     *This,
  IN  EFI_SMM_GPI_DISPATCH                      DispatchFunction,
  IN  EFI_SMM_GPI_DISPATCH_CONTEXT              *DispatchContext,
  OUT EFI_HANDLE                                *DispatchHandle
  );
```

Parameters

This

Pointer to the **EFI_SMM_GPI_DISPATCH_PROTOCOL** instance.

DispatchFunction

Function to install. Type **EFI_SMM_GPI_DISPATCH** is defined in "Related Definitions" below.

DispatchContext

Pointer to the dispatch function's context. The caller fills in this context before calling the **Register()** function to indicate to the **Register()** function the GPI SMI source for which the dispatch function should be invoked. Type **EFI_SMM_GPI_DISPATCH_CONTEXT** is defined in "Related Definitions" below.

DispatchHandle

Handle generated by the dispatcher to track the function instance. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification.*

Description

This service registers a given instance of the given source.

Related Definitions

```
//*******************************************************
// EFI_SMM_GPI_DISPATCH
//*******************************************************

typedef
VOID
(EFIAPI *EFI_SMM_GPI_DISPATCH) (
  IN  EFI_HANDLE                    DispatchHandle,
  IN  EFI_SMM_GPI_DISPATCH_CONTEXT  *DispatchContext
  );
```

DispatchHandle

Handle of this dispatch function. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification.*

DispatchContext

Pointer to the dispatch function's context. The *DispatchContext* fields are filled in by the dispatching driver prior to invoking this dispatch function. Type **EFI_SMM_GPI_DISPATCH_CONTEXT** is defined in "Related Definitions" below.

```
//*******************************************************
// EFI_SMM_GPI_DISPATCH_CONTEXT
//*******************************************************

typedef struct {
  UINTN      GpiNum;
} EFI_SMM_GPI_DISPATCH_CONTEXT;
```

GpiNum

A bit mask of 32 possible GPIs that can generate an SMI. Bit 0 corresponds to logical GPI[0], 1 corresponds to logical GPI[1], and so on.

The logical GPI index to a physical pin on the device is described by the GPI device name found on the same handle as the **EFI_SMM_GPI_DISPATCH_PROTOCOl**. The GPI device name is defined as protocol with a GUID name and **NULL** protocol pointer.

Status Codes Returned

EFI_SUCCESS	The dispatch function has been successfully registered and the SMI source has been enabled.
EFI_DEVICE_ERROR	The driver was unable to enable the SMI source.
EFI_INVALID_PARAMETER	*DispatchContext* is invalid. The GPI input value is not within valid range.
EFI_OUT_OF_RESOURCES	There is not enough memory (system or SMM) to manage this child.

EFI_SMM_GPI_DISPATCH_PROTOCOL.UnRegister()

Summary

Unregisters a General Purpose Input (GPI) service.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMM_GPI_UNREGISTER) (
  IN  struct _EFI_SMM_GPI_DISPATCH_PROTOCOL     *This,
  IN EFI_HANDLE                                 DispatchHandle
  );
```

Parameters

This

Pointer to the **EFI_SMM_GPI_DISPATCH_PROTOCOL** instance.

DispatchHandle

Handle of the service to remove. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification.*

Description

This service removes a handler.

Status Codes Returned

EFI_SUCCESS	The service has been successfully removed.
EFI_INVALID_PARAMETER	The *DispatchHandle* was not valid.

SMM Standby Button Dispatch Protocol

EFI_SMM_STANDBY_BUTTON_DISPATCH_PROTOCOL

Summary

Provides the parent dispatch service for the standby button SMI source generator.

GUID

```
#define EFI_SMM_STANDBY_BUTTON_DISPATCH_PROTOCOL_GUID    \
  { 0x78965b98, 0xb0bf, 0x449e, 0x8b, 0x22, 0xd2, 0x91, 0x4e, 0x49,
   0x8a, 0x98 }
```

Protocol Interface Structure

```
typedef struct _EFI_SMM_STANDBY_BUTTON_DISPATCH_PROTOCOL {
  EFI_SMM_STANDBY_BUTTON_REGISTER      Register;
  EFI_SMM_STANDBY_BUTTON_UNREGISTER    UnRegister;
} EFI_SMM_STANDBY_BUTTON_DISPATCH_PROTOCOL;
```

Parameters

Register

Installs a child service to be dispatched by this protocol. See the **Register()** function description.

UnRegister

Removes a child service dispatched by this protocol. See the **UnRegister()** function description.

Description

The **EFI_SMM_STANDBY_BUTTON_DISPATCH_PROTOCOL** provides the ability to install child handlers for the given event types.

EFI_SMM_STANDBY_BUTTON_DISPATCH_PROTOCOL.Register()

Summary

Provides the parent dispatch service for a given SMI source generator.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMM_STANDBY_BUTTON_REGISTER) (
  IN  struct _EFI_SMM_STANDBY_BUTTON_DISPATCH_PROTOCOL  *This,
  IN  EFI_SMM_STANDBY_BUTTON_DISPATCH
                  DispatchFunction,
  IN  EFI_SMM_STANDBY_BUTTON_DISPATCH_CONTEXT
          *DispatchContext,
  OUT EFI_HANDLE
                                      *DispatchHandle
  );
```

Parameters

This

Pointer to the **EFI_SMM_STANDBY_BUTTON_DISPATCH_PROTOCOL** instance.

DispatchFunction

Function to install. Type **EFI_SMM_STANDBY_BUTTON_DISPATCH** is defined in "Related Definitions" below.

DispatchContext

Pointer to the dispatch function's context. The caller fills in this context before calling the register function to indicate to the register function the standby button SMI source for which the dispatch function should be invoked. Type **EFI_SMM_STANDBY_BUTTON_DISPATCH_CONTEXT** is defined in "Related Definitions" below.

DispatchHandle

Handle generated by the dispatcher to track the function instance. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification.*

Description

This service registers a given instance of the given source.

Related Definitions

```
//*******************************************************
// EFI_SMM_STANDBY_BUTTON_DISPATCH
//*******************************************************
typedef
VOID
(EFIAPI *EFI_SMM_STANDBY_BUTTON_DISPATCH) (
  IN  EFI_HANDLE                              DispatchHandle,
  IN  EFI_SMM_STANDBY_BUTTON_DISPATCH_CONTEXT  *DispatchContext
  );
```

DispatchHandle

Handle of this dispatch function. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification.*

DispatchContext

Pointer to the dispatch function's context. The *DispatchContext* fields are filled in by the dispatching driver prior to invoking this dispatch function. Type **EFI_SMM_STANDBY_BUTTON_DISPATCH_CONTEXT** is defined below.

```
//*******************************************************
// EFI_SMM_STANDBY_BUTTON_DISPATCH_CONTEXT
//*******************************************************
typedef struct {
  EFI_STANDBY_BUTTON_PHASE  Phase;
} EFI_SMM_STANDBY_BUTTON_DISPATCH_CONTEXT;
```

Phase

Describes whether the child handler should be invoked upon the entry to the button activation or upon exit (i.e., upon receipt of the button press event or upon release of the event). This differentiation allows for workarounds or maintenance in each of these execution regimes. Type **EFI_STANDBY_BUTTON_PHASE** is defined below.

```
//*******************************************************
// EFI_STANDBY_BUTTON_PHASE;
//*******************************************************
typedef enum {
  Entry,
  Exit
} EFI_STANDBY_BUTTON_PHASE;
```

```
//
// Standby Button. Example, Use for changing LEDs before ACPI OS
// is on.
//    - DXE/BDS Phase
//    - OS Install Phase
//
```

Status Codes Returned

EFI_SUCCESS	The dispatch function has been successfully registered and the SMI source has been enabled.
EFI_DEVICE_ERROR	The driver was unable to enable the SMI source.
EFI_INVALID_PARAMETER	*DispatchContext* is invalid. The standby button input value is not within valid range.
EFI_OUT_OF_RESOURCES	There is not enough memory (system or SMM) to manage this child.

EFI_SMM_STANDBY_BUTTON_DISPATCH_PROTOCOL.UnRegister()

Summary

Unregisters a child SMI source dispatch function with a parent SMM driver.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMM_STANDBY_BUTTON_UNREGISTER) (
  IN  struct _EFI_SMM_STANDBY_BUTTON_DISPATCH_PROTOCOL   *This,
  IN EFI_HANDLE
        *DispatchHandle
  );
```

Parameters

This

Pointer to the **EFI_SMM_STANDBY_BUTTON_DISPATCH_PROTOCOL** instance.

DispatchHandle

Handle of the service to remove. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification.*

Description

This service removes a handler.

Status Codes Returned

EFI_SUCCESS	The service has been successfully removed.
EFI_INVALID_PARAMETER	The *DispatchHandle* was not valid.

SMM Power Button Dispatch Protocol

EFI_SMM_POWER_BUTTON_DISPATCH_PROTOCOL

Summary

Provides the parent dispatch service for the power button SMI source generator.

GUID

```
#define EFI_SMM_POWER_BUTTON_DISPATCH_PROTOCOL_GUID    \
  { 0xb709efa0, 0x47a6, 0x4b41, 0xb9, 0x31, 0x12, 0xec, 0xe7, 0xa8,
   0xee, 0x56 }
```

Protocol Interface Structure

```
typedef struct _EFI_SMM_POWER_BUTTON_DISPATCH_PROTOCOL {
  EFI_SMM_POWER_BUTTON_REGISTER      Register;
  EFI_SMM_POWER_BUTTON_UNREGISTER    UnRegister;
} EFI_SMM_POWER_BUTTON_DISPATCH_PROTOCOL;
```

Parameters

Register

Installs a child service to be dispatched by this protocol. See the **Register()** function description.

UnRegister

Removes a child service that was dispatched by this protocol. See the **UnRegister()** function description.

Description

The **EFI_SMM_POWER_BUTTON_DISPATCH_PROTOCOL** provides the ability to install child handlers for the given event types.

EFI_SMM_POWER_BUTTON_DISPATCH_PROTOCOL. Register()

Summary

Provides the parent dispatch service for a given SMI source generator.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMM_POWER_BUTTON_REGISTER) (
  IN  struct _EFI_SMM_POWER_BUTTON_DISPATCH_PROTOCOL  *This,
  IN  EFI_SMM_POWER_BUTTON_DISPATCH
          DispatchFunction,
  IN  EFI_SMM_POWER_BUTTON_DISPATCH_CONTEXT
          *DispatchContext,
  OUT EFI_HANDLE
          *DispatchHandle
  );
```

Parameters

This

Pointer to the **EFI_SMM_POWER_BUTTON_DISPATCH_PROTOCOL** instance.

DispatchFunction

Function to install. Type **EFI_SMM_POWER_BUTTON_DISPATCH** is defined in "Related Definitions" below.

DispatchContext

Pointer to the dispatch function's context. The caller fills in this context before calling the **Register()** function to indicate to the **Register()** function the power button SMI phase for which the dispatch function should be invoked. Type **EFI_SMM_POWER_BUTTON_DISPATCH_CONTEXT** is defined in "Related Definitions" below.

DispatchHandle

Handle generated by the dispatcher to track the function instance. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification.*

Description

This service registers a given instance of the given source.

Related Definitions

```
//*******************************************************
// EFI_SMM_POWER_BUTTON_DISPATCH
//*******************************************************
typedef
VOID
(EFIAPI *EFI_SMM_POWER_BUTTON_DISPATCH) (
  IN  EFI_HANDLE                            DispatchHandle,
  IN  EFI_SMM_POWER_BUTTON_DISPATCH_CONTEXT  *DispatchContext
  );
```

DispatchHandle

Handle of this dispatch function. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification.*

DispatchContext

Pointer to the dispatch function's context. The *DispatchContext* fields are filled in by the dispatching driver prior to invoking this dispatch function. Type **EFI_SMM_POWER_BUTTON_DISPATCH_CONTEXT** is defined below.

```
//*******************************************************
// EFI_SMM_POWER_BUTTON_DISPATCH_CONTEXT
//*******************************************************
typedef struct {
  EFI_POWER_BUTTON_PHASE  Phase;
} EFI_SMM_POWER_BUTTON_DISPATCH_CONTEXT;
```

Phase

Designates whether this handler should be invoked upon entry or exit. Type **EFI_POWER_BUTTON_PHASE** is defined in "Related Definitions" below.

```
//*******************************************************
// EFI_POWER_BUTTON_PHASE
//*******************************************************
typedef enum {
  PowerButtonEntry,
  PowerButtonExit
} EFI_POWER_BUTTON_PHASE;

// Power Button. Example, Use for changing LEDs before ACPI OS is
// on.
//    - DXE/BDS Phase
//    - OS Install Phase
```

Status Codes Returned

EFI_SUCCESS	The dispatch function has been successfully registered and the SMI source has been enabled.
EFI_DEVICE_ERROR	The driver was unable to enable the SMI source.
EFI_INVALID_PARAMETER	*DispatchContext* is invalid. The power button input value is not within valid range.
EFI_OUT_OF_RESOURCES	There is not enough memory (system or SMM) to manage this child.

EFI_SMM_POWER_BUTTON_DISPATCH_PROTOCOL. UnRegister()

Summary

Unregisters a power-button service.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMM_POWER_BUTTON_UNREGISTER) (
  IN  struct _EFI_SMM_POWER_BUTTON_DISPATCH_PROTOCOL    *This,
  IN  EFI_HANDLE
                                      DispatchHandle
  );
```

Parameters

This

Pointer to the **EFI_SMM_POWER_BUTTON_DISPATCH_PROTOCOL** instance.

DispatchHandle

Handle of the service to remove. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification.*

Description

This service removes a handler.

Status Codes Returned

EFI_SUCCESS	The service has been successfully removed.
EFI_INVALID_PARAMETER	The *DispatchHandle* was not valid.

8
Interactions with PEI, DXE, and BDS

Introduction

This chapter describes issues related to image verification and interactions between SMM and other Framework phases, including Hand-Off Blocks (HOBs) that describe the SMRAM regions to use.

Verification (Security)

Introduction

The SMM phase must preserve the chain of trust initiated in the previous phase. To do so, it must validate the modules that it loads for the subsequent dispatcher.

Execution

Once the final SMM handler has been loaded and before the system enters the Boot Device Selection (BDS) phase, the SMRAM must be locked down if possible on the platform.

SMM Chain of Trust

The current mechanism that is proposed for validating SMM images is using a Hashed Message Authentication Code (HMAC). This mechanism is reasonably secure and has the advantage that it does not require much memory.

See FAQ Question 2.1.7 on the RSA Security web site for more information (see References for the URL).

PEI Support

Introduction

To support T-SEG, H-SEG, and other memory decode mechanisms on IA-32 systems, there needs to be a PEIM that does the following:

- Updates the **`EFI_HOB_SMRAM_DESCRIPTOR_BLOCK`**, which describes the memory map while in SMM
- Exports policy information

This policy includes reservation of a given memory range at the top of physical memory for T-SEG, whether to use AB-SEG or H-SEG, and so on.

EFI_HOB_SMRAM_DESCRIPTOR_BLOCK

Summary

To convey the existence of the T-SEG reservation and H-SEG usage, there shall be a GUIDed Hand-Off Block (HOB) with GUID listed below. See the *Intel® Platform Innovation Framework for EFI Hand-Off Block (HOB) Specification* for more information on HOBs.

GUID

```
#define EFI_SMM_PEI_SMRAM_MEMORY_RESERVE \
  { 0x6dadf1d1, 0xd4cc, 0x4910, 0xbb, 0x6e, 0x82, 0xb1, 0xfd, 0x80,
   0xff, 0x3d }
```

Prototype

```
typedef struct _EFI_HOB_SMRAM_HOB_DESCRIPTOR_BLOCK {
  UINTN                  NumberOfSmmReservedRegions;
  EFI_SMRAM_DESCRIPTOR   Descriptor[1];
} EFI_HOB_SMRAM_DESCRIPTOR_BLOCK;
```

Parameters

NumberOfSmmReservedRegions

Designates the number of possible regions in the system that can be usable for SMRAM. This value can be greater than one in that the processor-chipset complex may expose several options for SMRAM support. The multiplicity of options is embodied in the possibly greater than one **EFI_SMRAM_DESCRIPTOR** data structures. Type **EFI_SMRAM_DESCRIPTOR** is defined in **EFI_SMM_ACCESS_PROTOCOL.GetCapabilities()**.

Descriptor

Used throughout this protocol to describe the candidate regions for SMRAM that are supported by this platform. Type **EFI_SMRAM_DESCRIPTOR** is defined in **EFI_SMM_ACCESS_PROTOCOL.GetCapabilities()**.
If the *RegionState* field has either **EFI_NEEDS_TESTING** or **EFI_NEEDS_ECC_INITIALIZATION**, then the implementation of the **EFI_SMM_BASE_PROTOCOL** should perform the appropriate action on the region prior to usage. In the case of Error Correction Code (ECC) initialization, the SMRAM region should have its contents written to zero prior to usage. The early platform initialization code may not have SMRAM decode enabled, such as AB Segment, so it will create region state option fields detailing the needed actions of the later component.

Description

This data structure will be created by a platform PEIM during the PEI phase of execution. The PEIM is also responsible for ensuring that the physical memory description is consistent with the capability of the chipset. If T-SEG is desired, for example, the memory range shall do one of the following:

- Be removed from the HOBs that were created by the memory controller
- Be marked as Firmware Reserved using a memory allocation

The **EFI_HOB_SMRAM_DESCRIPTOR_BLOCK** will be consumed by the implementation of the **EFI_SMM_ACCESS_PROTOCOL** during the DXE phase of execution. The DXE driver that abstracts the memory controller SMRAM capabilities will search through the HOB list that is referenced from the GUID/pointer pair in the EFI System Table. The memory that is described in this descriptor-set **EFI_HOB_SMRAM_DESCRIPTOR_BLOCK** is invisible to DXE for purposes of memory management and will not appear in the EFI memory map at all. This aspect of being outside of the DXE/EFI memory map is the uniqueness of this capability for IA-32, whereas for the Itanium® processor family, memory for PMI handlers can be firmware reserved.

The **EFI_HOB_SMRAM_DESCRIPTOR_BLOCK** HOB must exist and it is expected that the DXE driver that publishes the **SMM_ACCESS** driver will publish all of the SMRAM modality of the controller that is described therein.

Also, any region among the possible regions that are decoded by the hardware will be described in this data structure. For example, a chipset that had a cacheable high region and uncacheable high region might only describe the latter as being available because of the desire to trade performance for security concerns. These various capabilities cannot be understood by the **SMM_BASE** driver implementation via policy defaults; instead, the **SMM_ACCESS** driver constrains the possible ranges that the former can request.

See the *Intel® Platform Innovation Framework for EFI Hand-Off Block (HOB) Specification* for more information on HOBs.

SMM and DXE

SMM-to-DXE/EFI Communication

During the boot service phase of DXE/EFI, there will be a messaging mechanism between SMM and DXE drivers. This mechanism will allow a gradual state evolution of the SMM handlers during the boot phase.

The purpose of the DXE/EFI communication is to allow interfaces from either runtime or boot services to be proxied into SMM. For example, a vendor may choose to implement their EFI Variable Services in SMM. The motivation to do so would include a design in which the SMM code performed error logging by writing data to an EFI variable in flash. The error generation would be asynchronous with respect to the foreground operating system (OS). A problem is that the OS could be writing an EFI variable when the error condition, such as a Single-Bit Error (SBE) that was generated from main memory, occurred. To avoid two agents—SMM and EFI Runtime—both trying to write to flash at the same time, the runtime implementation of the **`SetVariable()`** EFI call would simply be an invocation of the **`EFI_SMM_BASE_PROTOCOL.Communicate()`** interface. Then, the SMM code would internally serialize the error logging flash write request and the OS **`SetVariable()`** request.

See the **`EFI_SMM_BASE_PROTOCOL.Communicate()`** service for more information on this interface.

9 Appendix

Introduction

This section provides the following supplemental information:

- An additional child dispatch protocol, the SMM ICHn Dispatch Protocol.
- Processor-specific information

The SMM ICHn Dispatch Protocol is not included with the architectural protocols listed in SMM Child Dispatch Protocols because the ICHn and its respective child sources are based on a given set of SMI activation sources in a particular platform implementation. The other protocols listed in SMM Child Dispatch Protocols represent a more generic set of capabilities, such as S-state transition and software-source generation. For this reason, the expectation is that the SMM ICHn Dispatch Protocol will serve as an interface to be used in today's platforms and as a model for future proliferations of this interface.

The processor-specific information in this appendix includes a discussion of multiprocessor issues and register summaries for IA-32 and Itanium® processors.

SMM ICHn Dispatch Protocol

SMM ICHn Dispatch Protocol

The architectural dispatch protocols that are defined in the SMM Child Dispatch Protocols chapter describe a class of system transitions, including power state transitions, periodic activations, and so on.

Beyond these more generic transitions, however, there are a collection of chipset-specific SMI activations that do not lend themselves to a simple abstraction. As such, there should be an additional dispatch protocol that supports a collection of these chipset-specific activations, such as watchdog timeout and ECC memory error signaling. This final class of errors will be contained in the context field for this dispatch protocol.

The **`EFI_SMM_ICHN_DISPATCH_PROTOCOL`** describes an example of this class of interface. The heterogeneous class of activation types are described in the enumeration **`EFI_SMM_ICHN_SMI_TYPE`**.

EFI_SMM_ICHN_DISPATCH_PROTOCOL

Summary

Provides the parent dispatch service for a given SMI source generator.

GUID

```
#define EFI_SMM_ICHN_DISPATCH_PROTOCOL_GUID    \
  { 0xc50b323e, 0x9075, 0x4f2a, 0xac, 0x8e, 0xd2, 0x59, 0x6a, 0x10,
   0x85, 0xcc }
```

Protocol Interface Structure

```
typedef struct _EFI_SMM_ICHN_DISPATCH_PROTOCOL {
  EFI_SMM_ICHN_REGISTER      Register;
  EFI_SMM_ICHN_UNREGISTER    UnRegister;
} EFI_SMM_ICHN_DISPATCH_PROTOCOL;
```

Parameters

Register

Installs a child service to be dispatched by this protocol. See the **Register()** function description.

UnRegister

Removes a child service dispatched by this protocol. See the **UnRegister()** function description.

Description

The **EFI_SMM_ICHN_DISPATCH_PROTOCOL** provides the ability to install child handlers for the given event types.

EFI_SMM_ICHN_DISPATCH_PROTOCOL.Register()

Summary

Provides the parent dispatch service for a given SMI source generator.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMM_ICHN_REGISTER) (
  IN  struct _EFI_SMM_ICHN_DISPATCH_PROTOCOL     *This,
  IN  EFI_SMM_ICHN_DISPATCH                      DispatchFunction,
  IN  EFI_SMM_ICHN_DISPATCH_CONTEXT              *DispatchContext,
  OUT EFI_HANDLE                                 *DispatchHandle
  );
```

Parameters

This

Pointer to the **EFI_SMM_ICHN_DISPATCH_PROTOCOL** instance.

DispatchFunction

Function to install. Type **EFI_SMM_ICHN_DISPATCH** is defined in "Related Definitions" below.

DispatchContext

Pointer to the dispatch function's context. The caller fills in this context before calling the **Register()** function to indicate to the **Register()** function the ICHN SMI source for which the dispatch function should be invoked. Type **EFI_SMM_ICHN_DISPATCH_CONTEXT** is defined in "Related Definitions" below.

DispatchHandle

Handle generated by the dispatcher to track the function instance. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification.*

Description

This service registers a given instance of the given source.

Related Definitions

```
//*****************************************************
// EFI_SMM_ICHN_DISPATCH
//*****************************************************

typedef
VOID
(EFIAPI *EFI_SMM_ICHN_DISPATCH) (
  IN  EFI_HANDLE                       DispatchHandle,
  IN  EFI_SMM_ICHN_DISPATCH_CONTEXT    *DispatchContext
  );
```

DispatchHandle

Handle of this dispatch function. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification.*

DispatchContext

Pointer to the dispatch function's context. The *DispatchContext* fields are filled in by the dispatching driver prior to invoking this dispatch function. Type **EFI_SMM_ICHN_DISPATCH_CONTEXT** is defined below.

```
//*****************************************************
// EFI_SMM_ICHN_DISPATCH_CONTEXT
//*****************************************************

typedef struct {
  EFI_SMM_ICHN_SMI_TYPE  Type;
} EFI_SMM_ICHN_DISPATCH_CONTEXT;
```

Type

ICHN-specific SMIs. These are miscellaneous SMI sources that are supported by the ICHN-specific SMI implementation. These may change over time. The trap number is valid only if the *Type* is trap. Type **EFI_SMM_ICHN_SMI_TYPE** is defined below.

```
//****************************************************
// EFI_SMM_ICHN_SMI_TYPE
//****************************************************
typedef enum {
  // NOTE: NEVER delete items from this list/enumeration!
  // Doing so will prevent other versions of the code
  // from compiling.  If the ICH version for which your driver
  // is written does not support some of these SMIs, then
  // simply return EFI_UNSUPPORTED when a child/client tries
  // to register for them.
  IchnMch,
  IchnPme,
  IchnRtcAlarm,
  IchnRingIndicate,
  IchnAc97Wake,
  IchnSerialIrq,
  IchnY2KRollover,
  IchnTcoTimeout,
  IchnOsTco,
  IchnNmi,
  IchnIntruderDetect,
  IchnBiosWp,
  IchnMcSmi,
  IchnPmeB0,
  IchnThrmSts,
  IchnSmBus,
  IchnIntelUsb2,
  IchnMonSmi7,
  IchnMonSmi6,
  IchnMonSmi5,
  IchnMonSmi4,
  IchnDevTrap13,
  IchnDevTrap12,
  IchnDevTrap11,
  IchnDevTrap10,
  IchnDevTrap9,
  IchnDevTrap8,
  IchnDevTrap7,
  IchnDevTrap6,
  IchnDevTrap5,
  IchnDevTrap3,
  IchnDevTrap2,
  IchnDevTrap1,
  IchnDevTrap0,
  // INSERT NEW ITEMS JUST BEFORE THIS LINE
  NUM_ICHN_TYPES // the number of items in this enumeration
} EFI_SMM_ICHN_SMI_TYPE;
```

Status Codes Returned

EFI_SUCCESS	The dispatch function has been successfully registered and the SMI source has been enabled.
EFI_DEVICE_ERROR	The driver was unable to enable the SMI source.
EFI_INVALID_PARAMETER	*DispatchContext* is invalid. The ICHN input value is not within a valid range.
EFI_OUT_OF_RESOURCES	There is not enough memory (system or SMM) to manage this child.

EFI_SMM_ICHN_DISPATCH_PROTOCOL. UnRegister()

Summary

Unregisters a child SMI source dispatch function with a parent SMM driver.

Prototype

```
typedef
  EFIAPI
  (EFIAPI *EFI_SMM_ICHN_UNREGISTER) (
    IN   struct _EFI_SMM_ICHN_DISPATCH_PROTOCOL     *This,
    IN   EFI_HANDLE                                 DispatchHandle
    );
```

Parameters

This

Pointer to the **EFI_SMM_ICHN_DISPATCH_PROTOCOL** instance.

DispatchHandle

Handle of the service to remove. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification.*

Description

This function unregisters a child SMI source dispatch function with a parent SMM driver.

Status Codes Returned

EFI_SUCCESS	The dispatch function has been successfully unregistered and the SMI source has been disabled, if there are no other registered child dispatch functions for this SMI source.
EFI_INVALID_PARAMETER	The *DispatchHandle* is invalid.

Processor-Specific Information

Introduction

The sections in this chapter discusses multiprocessor issues and provides register summaries for IA-32 and Itanium® processors.

For information on processor save-state information, see the SMM CPU Information Records section in Services - SMM. This information is important in that the SMM drivers may need to ascertain the state of the processor before invoking the SMI or PMI, respectively.

Multiprocessor Issues

The design of the SMM infrastructure is such that the bulk of the SMM infrastructure code and the dispatched SMM drivers will all execute in a single-processor, single-threaded environment. This execution is in contrast to the initiation of the SMI or PMI hardware event, which is visible to all processors. Because of the multiprocessor nature of the hardware activation, this prescription for single-threaded execution is enforced by preamble software in the SMM infrastructure. Specifically, during any SMI/PMI activation, all of the application processors (APs) will rendezvous while the boot-strap processor (BSP) services the SMI-initiated event.

The SMM design assumes that there is a preamble set of code that receives the machine state of the PMI or SMI activation in native mode. The code herein will rendezvous all of the processors using some atomic instructions on a semaphore. This election processor will only allow one processor to execute all of the handlers. When this single processor finishes executing all of the handlers, it will release the APs from this synchronization variable.

A future instance of this specification may speak to the concurrent, parallel dispatch of handlers. However, for this protocol suite, the dispatch will be serial.

Register Summaries

IA-32

IA-32 Register Summary

IA-32 architecture provides a limited number of registers that are visible to the programmer, as follows:

- 8 general purpose registers
- 6 segment registers
- 2 status and control registers
- 8 MMX registers (only processors that support Intel® MMX™ technology)
- 8 SIMD floating-point registers (only processors with streaming Single Instruction, Multiple Data (SIMD) extension support)

The table below lists the IA-32 architecture registers and provides more detailed information on each register type. See the *IA-32 Intel® Architecture Software Developer's Manual* for a more detailed description of the registers available with the IA-32 architecture. See the figures in General IA-32 Register Usage and SMM IA-32 Register Usage for how the IA-32 register sets are used in the SMM environment.

Table 9-1. IA-32 Register Summary

Register Description	Size	Quantity	Description
General Purpose registers EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP	32-bit	8 registers total	Each register is referred to by a mnemonic (for example, EAX and EBX) that corresponds to the register set found in 16-bit Intel® processors such as the Intel® 8086 and 80286 processors. During normal operation, each register performs the following functions: EAX: Accumulator for operands and results data EBX: Pointer to data in the DS segment register ECX: Counter for string and loop operations EDX: I/O pointer ESI: Pointer to data in the segment pointed to by the DS register; source pointer for string operations EDI: Pointer to data (or destination) in the segment pointed to by the ES register; destination pointer for string operations EBP: Pointer to data on the stack (in the SS segment register) ESP: Stack pointer (in the SS segment register)
Segment registers CS, DS, SS, ES, FS, GS	16-bit	6 registers total	Normally hold 16-bit segment selectors that point to a segment in memory.
Status and Control registers EFLAGS, EIP	32-bit	2 registers total	**EFLAGS register:** Normally contains a group of status, control, and system flags. **EIP (instruction pointer) register:** Normally contains the offset in the current code segment for the next instruction to be executed.
MMX registers (MM0 – MM7)	64-bit	8 registers total	Newer Intel® Pentium® processors with MMX technology have an additional eight 64-bit registers that can be used during the SMM phase.
SIMD floating-point registers (XMM0 – XMM7)	128-bit	8 registers total	Processors that support streaming SIMD extensions have an additional eight 128-bit registers over those with earlier MMX technology.

General IA-32 Register Usage

The figure below shows the general usage of the IA-32 register sets.

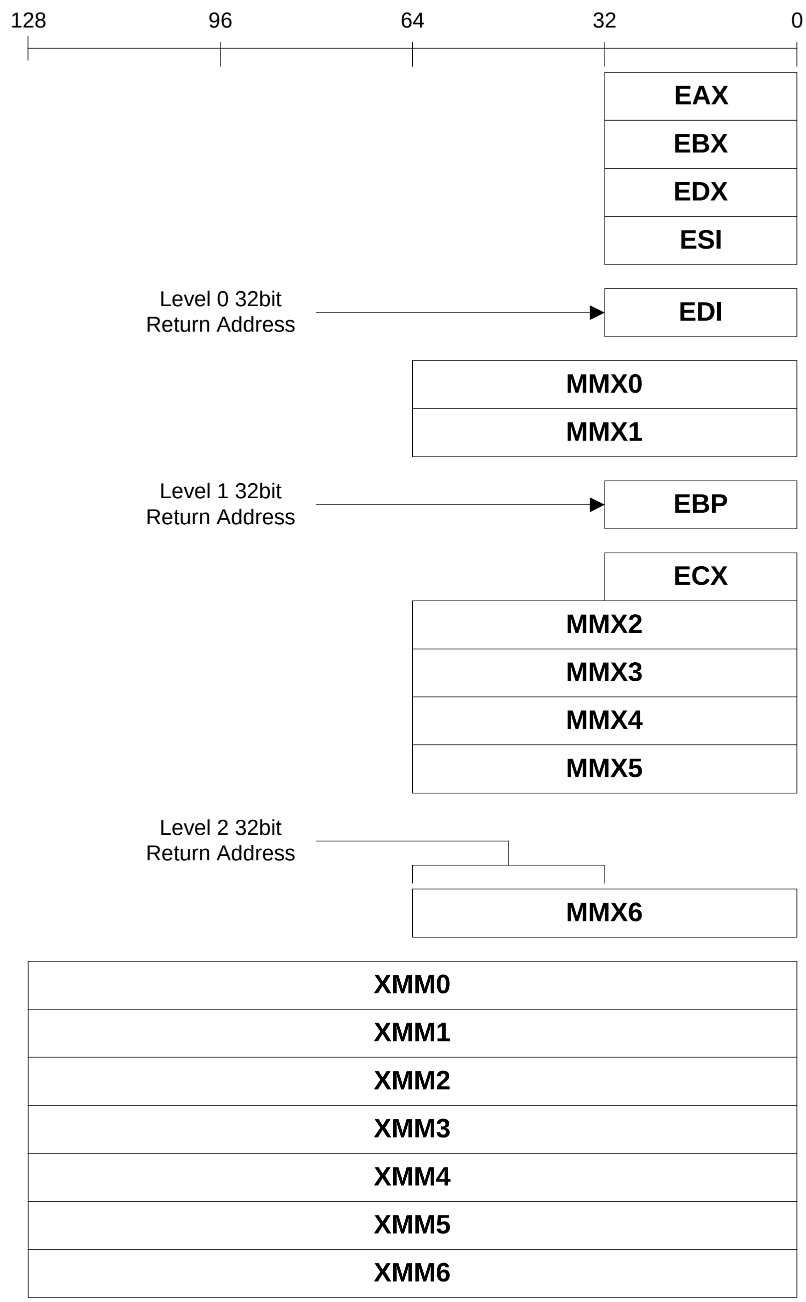

Figure 9-1. General IA-32 Register Usage

SMM IA-32 Register Usage

The figure below shows how the IA-32 register sets are used in the SMM environment.

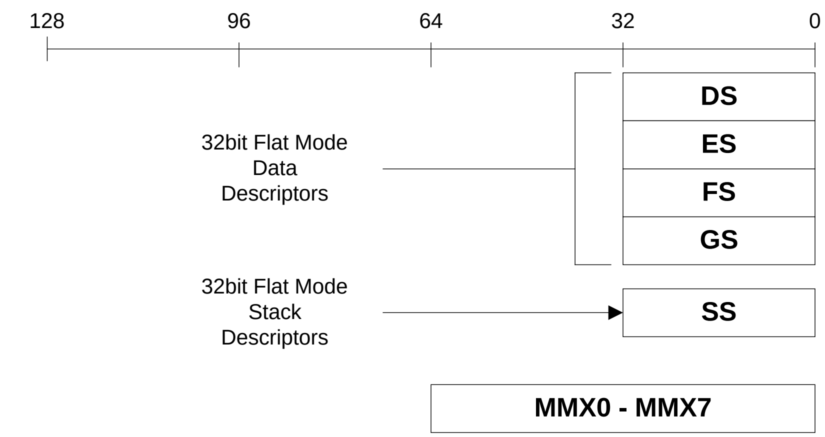

Figure 9-2. SMM IA-32 Register Usage

Itanium® Processor Family

Itanium® Processor Family Register Summary

Itanium® architecture provides several register files that are visible to the programmer, as follows:

- 128 general registers
- 128 floating-point registers
- 64 predicate registers
- 8 branch registers
- 128 application registers
- 1 instruction pointer (IP) register

Registers are referred to by a mnemonic denoting the register type and a number. For example, general register 32 is named gr32. The table below lists the Itanium architecture registers; see the following topics for more detailed information on each register type.

Table 9-2. Itanium® Processor Family Register Summary

Register Name	Size	Quantity
General registers (gr0 – gr127)	64-bit	32 static and global 96 stacked 128 registers total
Floating-point registers (fr0 – fr127)	82-bit	32 static and global 96 rotating (SW pipelining) 128 registers total

continued

Table 9-2. Itanium® Processor Family Register Summary (continued)

Register Name	Size	Quantity
Predicate registers (pr0 – pr63)	1-bit	16 static 48 rotating (SW pipeline control) 64 registers total
Branch registers (br0 – br7)	64-bit	8 registers total
Application registers (ar0 – ar127)	64-bit	128 registers total
Instruction pointer (IP) register	64-bit	One register, not directly accessible, that is always 16-byte aligned.

Itanium® Processor Family: General Registers (gr0 – gr127)

Itanium® architecture provides 128 64-bit general purpose registers for all integer and multimedia computation.

Register gr0 is a read-only register and is always zero (0). The first 32 registers are static and global to the process. The remaining 96 registers are stacked. These registers are for argument passing and local register stack frame. A portion of these registers can also be used for software pipelining.

Each register has an associated Not a Thing (NaT) bit, indicating whether the value stored in the register is valid.

Itanium® Processor Family: General Register Stack (gr32 – gr127)

There are 96 general registers, starting at gr32, that are used to pass parameters to the called procedure and store local variables for the currently executing procedure.

Itanium® Processor Family: Floating-Point Registers (fr0 – fr127)

Itanium® architecture provides 128 82-bit floating-point registers, for floating-point computations. All floating-point registers are globally accessible within the process. The floating-point registers are broken up as follows:

- 32 static floating-point registers
- 96 rotating floating-point registers, for software pipelining

The first two registers (fr0 and fr1) are read-only:

- fr0 is read as +0.0
- fr1 is read as +1.0

Each register contains the following three fields:

- 64-bit significand field
- 17-bit exponent field
- 1-bit sign field

Itanium® Processor Family: Predicate Registers (pr0 – pr63)

There are 64 1-bit predicate registers to enable controlling the execution of instructions. When the value of a predicate register is true (1), the instruction is executed. The predicate registers enable the following:

- Validating/invalidating instructions
- Eliminating branches in if/then/else logic blocks

The predicate registers are broken up as follows:

- 16 static predicate registers
- 48 rotating predicate registers for controlling software pipelining

Instructions that are not explicitly preceded by a predicate default to the first predicate register, pr0, which is read-only and is always true (1).

Itanium® Processor Family: Branch Registers (br0 – br7)

Eight 64-bit branch registers are used to specify the branch target addresses for indirect branches.

The branch registers streamline call/return branching.

Itanium® Processor Family: Application Registers (ar0 – ar127)

There are 128 64-bit special purpose registers that are used for various functions. Some of the more commonly used application registers have assembler aliases. For example, ar66 is used as the Epilogue Counter (EC) and is called ar.ec.

Itanium® Processor Family: Instruction Pointer (IP) Register

The 64-bit instruction pointer (IP) holds the address of the bundle of the currently executing instruction. The IP cannot be directly read or written; it increments as instructions are executed. Branch instructions set the IP to a new value. The IP is always 16-byte aligned.