VOLUME 4: Platform Initialization Specification

System Management Mode Core Interface

Version 1.4 Errata A

3/15/2016

Revision History

Revision	Revision History	Date
1.0	Initial public release.	8/21/06
1.0 errata	Mantis tickets: • M47 dxe_dispatcher_load_image_behavior • M48 Make spec more consistent GUID & filename. • M155 FV_FILE and FV_ONLY: Change subtype number back to th:e original one. • M171Remove 10 us lower bound restriction for the TickPeriod in the Metronome • M178 Remove references to tail in file header and made file checksum for the data • M183 Vol 1-Vol 5: Make spec more consistent. • M192 Change PAD files to have an undefined GUID file name and update all FV	10/29/07
1.1	Mantis tickets: • M39 (Updates PCI Hostbridge & PCI Platform) • M41 (Duplicate 167) • M42 Add the definition of theDXE CIS Capsule AP & Variable AP • M43 (SMbios) • M46 (SMM error codes) • M163 (Add Volume 4--SMM • M167 (Vol2: adds the DXE Boot Services Protocols--new Chapter 12) • M179 (S3 boot script) • M180 (PMI ECR) • M195 (Remove PMI references from SMM CIS) • M196 (disposable-section type to the FFS)	11/05/07
1.1 correction	Restore (missing) MP protocol	03/12/08
1.1 Errata	Revises typographical errors and minor omissions--see Errata for details	04/25/08

1.1 Errata	Mantis tickets • 204 Stack HOB update 1.1errata • 225 Correct references from EFI_FIRMWARE_VOLUME_PROTOCOL to EFI_FIRMWARE_VOLUME2_PROTOCOL • 226 Remove references to Framework • 227 Correct protocol name GUIDED_SECTION_EXTRACTION_PROTOCOL • 228 insert"typedef" missing from some typedefs in Volume 3 • 243 Define interface "EFI_PEI_FV_PPI" declaration in PI1.0 FfsFindNextVolume() • 285 Time quality of service in S3 boot script poll operation • 287 Correct MP spec, PIVOLUME 2:Chapter 13.3 and 13.4 - return error language • 290 PI Errata • 305 Remove Datahub reference • 336 SMM Control Protocol update • 345 PI Errata • 353 PI Errata • 360 S3RestoreConfig description is missing • 363 PI Volume 1 Errata • 367 PCI Hot Plug Init errata • 369 Volume 4 Errata • 380 SMM Development errata • 381 Errata on EFI_SMM_SAVE_STATE_IO_INFO	01/13/09
1.1 Errata	• 247 Clarification regarding use of dependency expression section types with firmware volume image files • 399 SMBIOS Protocol Errata • 405 PIWG Volume 5 incorrectly refers to EFI_PCI_OVERRIDE_PROTOCOL • 422 TEMPORARY_RAM_SUPPORT_PPI is misnamed • 428 Volume 5 PCI issue • 430 Clarify behavior w/ the FV extended header	02/23/09
1.2	• 271 Support For Large Firmware Files And Firmware File Sections • 284 CPU I/O protocol update • 286 Legacy Region protocol • 289 Recovery API • 292 PCD Specification Update • 354 ACPI Manipulation Protocol • 355 EFI_SIO_PROTOCOL Errata • 365 UEFI Capsule HOB • 382 IDE Controller Specification • 385 Report Status Code Router Specification • 386 Status Code Specification	01/19/09

1.2	• 401 SMM Volume 4 issue • 402 SMM PI spec issue w.r.t. CRC • 407 Add LMA Pseudo-Register to SMM Save State Protocol • 409 PCD_PROTOCOL Errata • 411 Draft Errata, Volume 5, Section 8 • 412 Comment: PEI_S3_RESUME_PPI should be EFI_PEI_S3_RESUME_PPI • 414 Draft Chapter 7 Comments • 415 Comment: Report Status Code Routers • 416 EFI_CPU_IO_PROTOCOL2 Name should be EFI_CPU_IO2_PROTOCOL • 417 Volume 5, Chapter 4 & 5 order is reversed • 423 Comment: Section 15.2.1 Formatting Issues vol5 • 424 Comments: Volume 5, Appendix A.1 formatting issues • 425 Comment: Formatting in Section 6.1 of Volume 3 • 426 Comments: Volume 2 • 427 Comment: Volume 3, Section 6 • 433 Editorial issues in PI 1.2 draft	02/23/09
1.2	• 407 Comment: additional change to LMA Pseudo-Register • 441 Comment: PI Volume 3, Incorrect Struct Declaration (esp PCD_PPI) • 455 Comment: Errata - Clarification of InstallPeiMemory() • 465 Comment: Errata on PMI interface • 466 Comment: Vol 4 EXTENDED_SAL_PROC definition • 467 Comments: PI1.1 errata • 480 Comment: FIX to PCD_PROTOCOL and PCD_PPI	05/13/09

1.2 errata	• 345 PI1.0 errata • 468 Issues on proposed PI1.2 ACPI System Description Table Protocol • 492 Add Resource HOB Protectability Attributes • 494 Vol. 2 Appendix A Clean up • 495 Vol 1: update HOB reference • 380 PI1.1 errata from SMM development • 501 Clean Up SetMemoryAttributes() language Per Mantis 489 (from USWG) • 502 Disk info • 503 typo • 504 remove support for fixed address resources • 509 PCI errata – execution phase • 510 PCI errata - platform policy • 511 PIC TE Image clarification/errata • 520 PI Errata • 521Add help text for EFI_PCD_PROTOCOL for GetNextTokenSpace • 525 Itanium ESAL, MCA/INIT/PMI errata • 526 PI SMM errata • 529 PCD issues in Volume 3 of the PI1.2 Specification • 541 Volume 5 Typo • 543 Clarification around usage of FV Extended header • 550 Naming conflicts w/ PI SMM	12/16/09

1.2 errata A	• 363 PI volume 1 errata • 365 UEFI Capsule HOB • 381 PI1.1 Errata on EFI_SMM_SAVE_STATE_IO_INFO • 482 One other naming inconsistency in the PCD PPI declaration • 483 PCD Protocol / PPI function name synchronization..... • 496 Boot mode description • 497 Status Code additions • 548 Boot firmware volume clarification • 551 Name conflicts w/ Legacy region • 552 MP services • 553 Update text to PEI • 554 update return code from PEI AllocatePages • 555 Inconsistency in the S3 protocol • 561 Minor update to PCD->SetPointer • 565 CANCEL_CALL_BACK should be CANCEL_CALLBACK • 569 Recovery: EFI_PEI_GET_NUMBER_BLOCK_DEVICES decl has EFI_STATUS w/o return code & errror on stage 3 recovery description • 571 duplicate definition of EFI_AP_PROCEDURE in DXE MP (volume2) and SMM (volume 4) • 581 EFI_HOB_TYPE_LOAD_PEIM ambiguity • 591ACPI Protocol Name collision • 592 More SMM name conflicts • 593 A couple of ISA I/O clarifications • 594 ATA/ATAPI clarification • 595 SMM driver entry point clarification • 596 Clarify ESAL return codes • 602 SEC->PEI hand-off update • 604 EFI_NOT_SUPPORTED versus EFI_UNSUPPORTED	2/24/10
1.2 errata B	• 628 ACPI SDT protocol errata • 629 Typos in PCD GetSize() • 630EFI_SMM_PCI_ROOT_BRIDGE_IO_PROTOCOL service clarification • 631 System Management System Table (SMST) MP-related field clarification	5/27/10

1.2 Errata C	• 550 Naming conflicts w/ PI SMM • 571 duplicate definition of EFI_AP_PROCEDURE in DXE MP (volume2) and SMM (volume 4) • 654 UEFI PI specific handle for SMBIOS is now available • 688 Status Code errata • 690 Clarify agent in IDE Controller chapter • 691 SMM a priori file and SOR support • 692 Clarify the SMM SW Register API • 694 PEI Temp RAM PPI ambiguity • 703 End of PEI phase PPI publication for the S3 boot mode case • 706 GetPeiServicesTablePointer () changes for the ARM architecture • 714 PI Service Table Versions • 717 PI Extended File Size Errata • 718 PI Extended Header cleanup / Errata • 730 typo in EFI_SMM_CPU_PROTOCOL.ReadSaveState() return code • ERROR: listed by mistake:737 • 738 Errata to Volume 2 of the PI1.2 specification • 739 Errata for PI SMM Volume 4 Control protocol • 742 Errata for SMBUS chapter in Volume 5 • 743 Errata - PCD_PPI declaration • 745 Errata – PI Firmware Section declarations • 746 Errata - PI status code • 747 Errata - Text for deprecated HOB • 752 Binary Prefix change • ERROR: listed by mistake: 753 • 764 PI Volume 4 SMM naming errata • 775 errata/typo in EFI_STATUS_CODE_EXCEP_SYSTEM_CONTEXT, Volume 3 • 781 S3 Save State Protocol Errata • 782 Format Insert(), Compare() and Label() as for Write() • 783 TemporaryRamMigration Errata • 784 Typos in status code definitions • 787 S3 Save State Protocol Errata 2 • 810 Set Memory Attributes return code clarification • 811 SMBIOS API Clarification • 814 PI SMBIOS Errata • 821Location conflict for EFI_RESOURCE_ATTRIBUTE_xxx_PROTECTABLE #defines • 823 Clarify max length of SMBIOS Strings in SMBIOS Protocol • 824 EFI_SMM_SW_DISPATCH2_PROTOCOL.Register() Errata • 837 ARM Vector table can not support arbitrary 32-bit address • 838 Vol 3 EFI_FVB2_ALIGNMNET_512K should be EFI_FVB2_ALIGNMENT_512K • 840 Vol 3 Table 5 Supported FFS Alignments contains values not supported by FFS • 844 correct references to Platform Initialization Hand-Off Block Specification	10/27/11

1.2.1	• 527 PI Volume 2 DXE Security Architecture Protocol (SAP) clarification • 562 Add SetMemoryCapabilities to GCD interface • 719 End of DXE event • 731 Volume 4 SMM - clarify the meaning of NumberOfCpus • 737 Remove SMM Communication ACPI Table definition . • 753 SIO PEI and UEFI-Driver Model Architecture • 769 Signed PI sections • 813 Add a new EFI_GET_PCD_INFO_PROTOCOL and EFI_GET_PCD_INFO_PPI instance. • 818 New SAP2 return code • 822 Method to disable Temporary RAM when Temp RAM Migration is not required • 833 Method to Reserve Interrupt and Exception Vectors • 839 Add support for weakly aligned FVs • 892 EFI_PCI_ENUMERATION_COMPLETE_GUID Protocol • 894 SAP2 Update • 895 Status Code Data Structures Errata • 902 Errata on signed firmware volume/file • 903 Update • 906 Volume 3 errata - Freeform type • 916 Service table revisions	05/02/12
1.2.1 Errata A	• 922 Add a "Boot with Manufacturing" boot mode setting • 925 Errata on signed FV/Files • 931 DXE Volume 2 - Clarify memory map construction from the GCD • 936 Clarify memory usage in PEI on S3 • 937 SMM report protocol notify issue errata • 951 Root Handler Processing by SmiManage • 958 Omissions in PI1.2.1 integration for M816 and M894 • 969Vol 1 errata: TE Header parameters	10/26/12
1.3	• 945 Integrated Circuit (I2C) Bus Protocol • 998 PI Status Code additions • 999 PCI enumeration complete GUID • 1005 NVMe Disk Info guid • 1006 Security Ppi Fixes • 1025 PI table revisions	3/29/13

1.3 Errata	• 1041 typo in HOB Overview • 1067 PI1.3 Errata for SetBootMode • 1068 Updates to PEI Service table/M1006 • 1069 SIO Errata - pnp end node definition • 1070 Typo in SIO chapter • 1072 Errata – SMM register protocol notify clarification/errata • 1093 Extended File Size Errata • 1095 typos/errata • 1097 PI SMM GPI Errata • 1098 Errata on I2C IO status code • 1099 I2C Protocol stop behavior errata • 1104 ACPI System Description Table Protocol Errata • 1105 ACPI errata - supported table revision • 1177 PI errata - make CPU IO optional • 1178 errata - allow PEI to report an additional memory type • 1283 Errata - clarify sequencing of events	2/19/15
1.4	• 1210 Adding persistence attribute to GCD • 1235 PI.Next Feature - no execute support • 1236 PI.Next feature - Graphics PPI • 1237 PI.Next feature - add reset2 PPI • 1239 PI.Next feature - Disk Info Guid UFS • 1240 PI.Next feature - Recovery Block IO PPI - UFS • 1259 PI.Next feature - MP PPI • 1273 PI.Next feature - capsule PPI • 1274 Recovery Block I/O PPI Update • 1275 GetMemoryMap Update • 1277 PI1.next feature - multiple CPU health info • 1278 PI1.next - Memory relative reliability definition • 1305 PI1.next - specification number encoding • 1331 Remove left-over Boot Firmware Volume references in the SEC Platform Information PPI • 1366 PI 1.4 draft - M1277 issue BIST / CPU. So health record needs to be indexed / CPU.	2/20/15

1.4 Errata A	• 1596 Mantis1489 GCD issue • 1574 Fix artificial limitation in the PCD.SetSku support • 1565 Update status code to include AArch64 exception error codes • 1564 SMM Software Dispatch Protocol Errata • 1562 Errata to remove statement from DXE vol about PEI dispatch behavior • 1561 Errata to provide Equivalent of DXE-CIS Mantis 247 for the PEI-CIS • 1532 Allow S3 Resume without having installed permanent memory (via InstallPeiMemory) • 1530 errata on dxe report status code • 1529 address space granularity errata • 1525 PEI Services Table Retrieval for AArch64 • 1515 EFI_PEIM_NOTIFY_ENTRY_POINT return values are undefined • 1497 Fixing language in SMMStartupThisAP • 1489 GCD Conflict errata • 1485 Minor Errata in SMM Vo2 description of SMMStartupThisAP • 1397 PEI 1.4 specification revision errata • 1394 Errata to Relax requirements on CPU rendez in SEC • 1351 EndOfDxe and SmmReadyToLock • 1322 Minor Updates to handle Asynchronous CPU Entry Into SMM	3/15/16

Specification Volumes

The **Platform Initialization Specification** is divided into volumes to enable logical organization, future growth, and printing convenience. The **Platform Initialization Specification** consists of the following volumes:

VOLUME 1: Pre-EFI Initialization Core Interface

VOLUME 2: Driver Execution Environment Core Interface

VOLUME 3: Shared Architectural Elements

VOLUME 4: System Management Mode

VOLUME 5: Standards

Each volume should be viewed in the context of all other volumes, and readers are strongly encouraged to consult the entire specification when researching areas of interest. Additionally, a single-file version of the **Platform Initialization Specification** is available to aid search functions through the entire specification.

Contents

Figures

Tables

1 Overview

1.1 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.

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.

MTRR

Memory Type Range Register.

RSM

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

SMI

System Management Interrupt. Generic term for a non-maskable, high priority interrupt which transitions the system into System Management Mode.

SMM

System Management Mode. Generic term for the execution mode entered when a CPU detects an SMI. The firmware, in response to the interrupt type, will gain control in physical mode. For the purpose of this document, “SMM” will be used to describe the operational regime for IA32 and x64 processors that share the OS-transparent characteristics.

SMM Driver

A driver launched directly into SMRAM, with access to the SMM interfaces.

SMM handler

A DXE driver that is loaded into and executed from SMRAM. SMM handlers are dispatched during boot services time and invoked synchronously or asynchronously thereafter. SMM handlers remain present during runtime.

SMM Initialization

The phase of SMM Driver initialization which starts with the call to the driver’s entry point and ends with the return from the driver’s entry point.

SMM Runtime

The phase of SMM Driver initialization which starts after the return from the driver's entry point.

SMST

System Management System Table. Hand-off to handler.

1.2 System Management Mode (SMM)

System Management Mode (SMM) is a generic term used to describe a unique operating mode of the processor which is entered when the CPU detects a special high priority System Management Interrupt (SMI). Upon detection of an SMI, a CPU will switch into SMM, jump to a pre-defined entry vector and save some portion of its state (the "save state") such that execution can be resumed.

The SMI can be generated synchronously by software or asynchronously by a hardware event. Each SMI source can be detected, cleared and disabled.

Some systems provide for special memory (SMRAM) which is set aside for software running in SMM. Usually the SMRAM is hidden during normal CPU execution, but this is not required. Usually, the SMRAM is locked after initialization so that it cannot be exposed until the next system reset.

1.3 SMM Driver Execution Environment

The SMM Core Interface Specification describes the optional SMM *phase,* which starts during the DXE phase and runs in parallel with the other PI Architecture phases into runtime.

The SMM Core Interface Specification describes two pieces of the PI SMM architecture:

SMRAM Initialization

During DXE, an SMM related driver opens SMRAM, creates the SMRAM memory map and provides the necessary services to launch SMM-related drivers and then, before boot, close and lock SMRAM.

SMI Management

When an SMI generated, the driver execution environment is created and then the SMI sources are detected and SMI handlers called.

The figure below shows the SMM architecture.

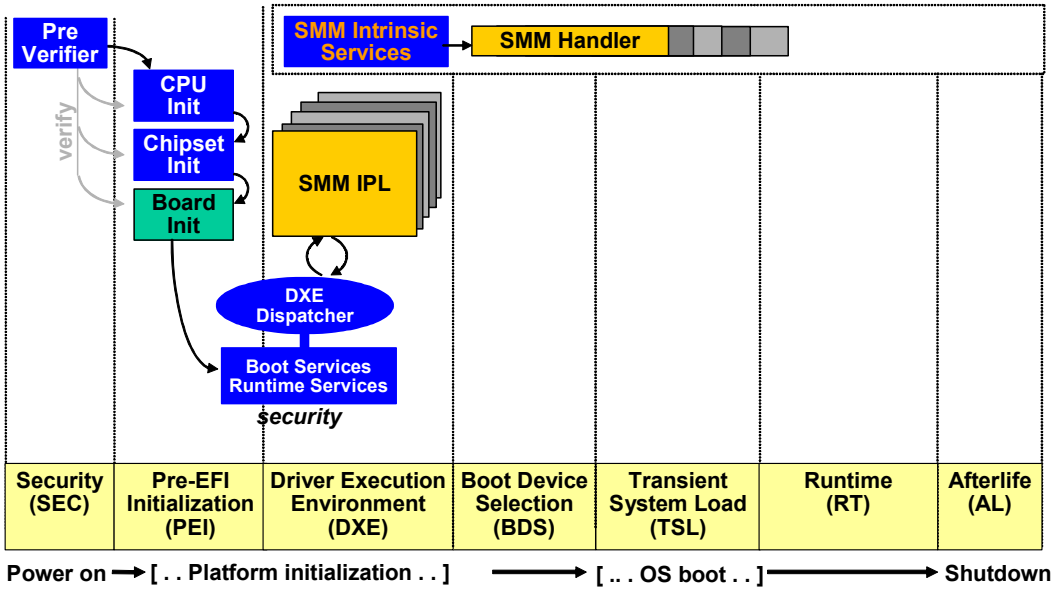

Figure 1. SMM Architecture

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

1.4 Initializing System Management Mode

System Management Mode initialization prepares the hardware for SMI generation and creates the necessary data structures for managing the SMM resources such as SMRAM. It is initialized with the cooperation of several DXE drivers.

1. A DXE driver produces the **`EFI_SMM_ACCESS2_PROTOCOL`**, which describes the different SMRAM regions available in the system.
2. A DXE driver produces the **`EFI_SMM_CONTROL2_PROTOCOL`**, which allows synchronous SMIs to be generated.
3. A DXE driver (dependent on the **`EFI_SMM_ACCESS2_PROTOCOL`** and, perhaps, the **`EFI_SMM_CONTROL2_PROTOCOL`**), does the following:
   - Initializes the SMM entry vector with the code necessary to meet the entry point requirements described in “Entering & Exiting SMM”.

- Produces the **EFI_SMM_CONFIGURATION _PROTOCOL**, which describes those areas of SMRAM which should be excluded from the memory map.

4. The SMM IPL DXE driver (dependent on the **EFI_SMM_ACCESS2_PROTOCOL**, **EFI_SMM_CONTROL2_PROTOCOL** and **EFI_SMM_CONFIGURATION_PROTOCOL**) does the following:
   - Opens SMRAM
   - Creates the SMRAM heap, excluding any areas listed in **EFI_SMM_CONFIGURATION_PROTOCOL** *SmramReservedRegions* field.
   - Loads the SMM Foundation into SMRAM. The SMM Foundation produces the SMST.
   - Invokes the **EFI_SMM_CONFIGURATION_PROTOCOL.***RegisterSmmEntry()* function with the SMM Foundation entry point.
   - Publishes the **EFI_SMM_BASE2_PROTOCOL** in the UEFI Protocol Database
   - At this point SMM is initially configured and SMIs can be generated.
   - Register for notification upon installation of the **EFI_DXE_SMM_READY_TO_LOCK_PROTOCOL** in the UEFI protocol database.
5. During the remainder of the DXE phase, additional drivers may load and be initialized in SMRAM.
6. At some point prior to the processing of boot options, a DXE driver will install the **EFI_DXE_SMM_READY_TO_LOCK_PROTOCOL** protocol in the UEFI protocol database. (outside of SMM).
7. As a result, some DXE driver will cause the **EFI_SMM_READY_TO_LOCK_PROTOCOL** protocol to be installed in the SMM protocol database.
   - Optionally, close the SMRAM so that it is no longer visible using the **EFI_SMM_ACCESS2_PROTOCOL**. Closing SMRAM may not be supported on all platforms.
   - Optionally, lock the SMRAM so that its configuration can no longer be altered using the **EFI_SMM_ACCESS2_PROTCOL**. Locking SMRAM may not be supported on all platforms.

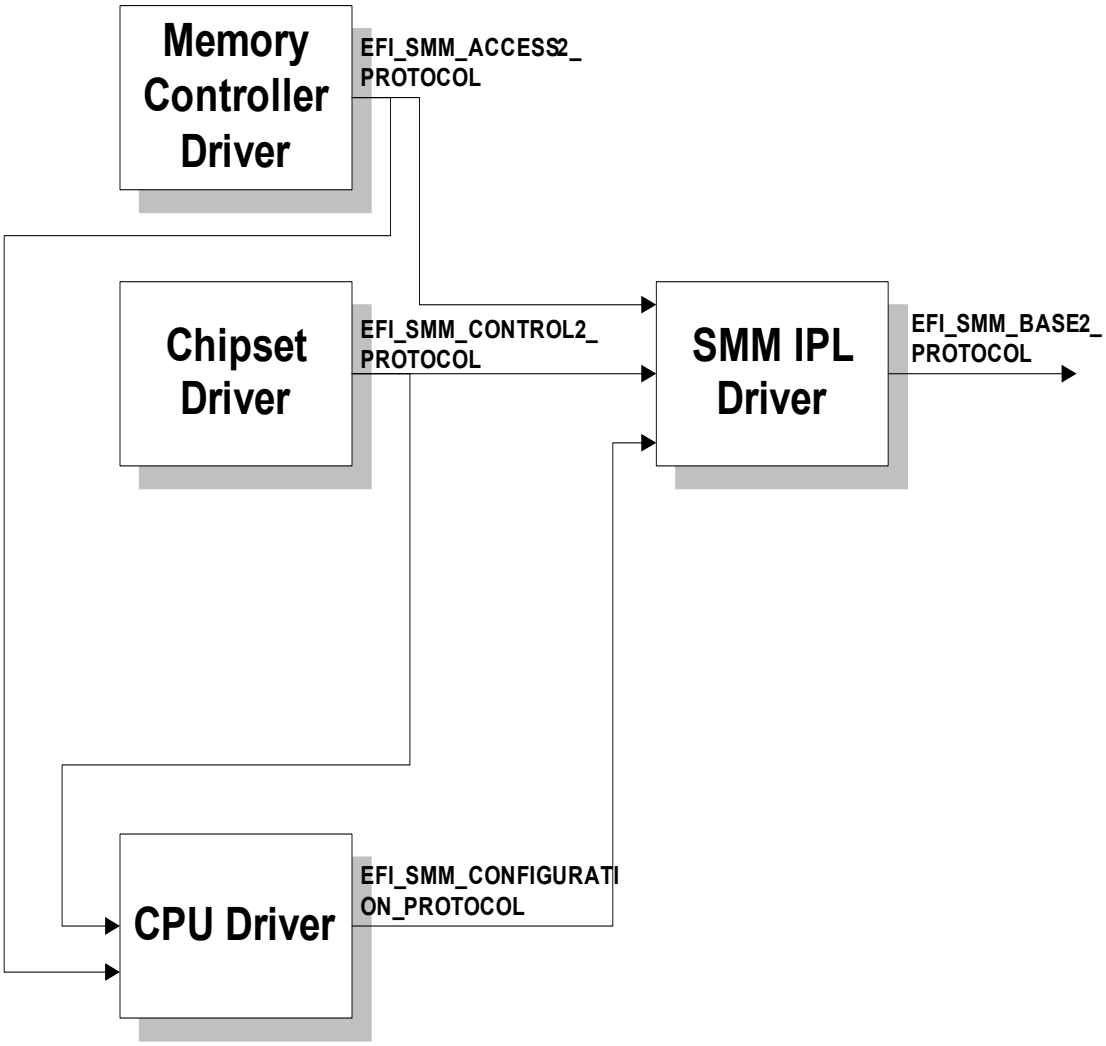

Figure 2. Example SMM Initialization Components

1.5 Entering & Exiting SMM

The code at the entry vector must:

- Save any CPU state necessary for supporting the EFI_SMM_CPU_ PROTOCOL
- Save any CPU state so that the normal operation can be resumed.
- Select a single CPU to enter the SMM Foundation.
- If an entry point has been registered via *RegisterSmmEntry()*, switch to the same CPU mode as the SMM Foundation and call the SMM Foundation entry point.

The SMM Foundation entry point must:

- Update the SMST with the CPU information passed to the entry point.
- Call all root SMI controller handlers using SmiManage(NULL)

- Return to the entry vector code.

After returning from the SMM Foundation entry point, the code at the entry vector must:

- Restore any CPU state information necessary for normal operation.
- Resume normal operation

1.6 SMM Drivers

There are two types of SMM-related drivers: SMM Drivers and Combination SMM/DXE Drivers. Both types of drivers are initialized by calling their main entry point.

The entry point of the driver is the same as a UEFI specification **EFI_IMAGE_ENTRY_POINT**.

1.6.1 SMM Drivers

SMM Drivers must have the file type **EFI_FV_FILETYPE_SMM**. SMM Drivers are launched once, directly into SMRAM. SMM Drivers cannot be launched until the dependency expression in the file section EFI_SECTION_SMM_DEPEX evaluates to true. This dependency expression can refer to both UEFI and SMM protocols.

The entry point of the driver is the same as a UEFI specification **EFI_IMAGE_ENTRY_POINT**.

1.6.2 Combination SMM/DXE Drivers

Combination SMM/DXE Drivers must have the file type **EFI_FV_FILETYPE_COMBINED_SMM_DXE**. Combination Drivers are launched twice.

They are launched by the DXE Dispatcher as a normal DXE driver outside of SMRAM after the dependency expression in the file section **EFI_SECTION_DXE_DEPEX** evaluates to true. As DXE Drivers, they have access to the normal UEFI interfaces.

Combination Drivers are also launched as SMM Drivers inside of SMRAM after the dependency expression in the file section EFI_SECTION_SMM_DEPEX evaluates to true. Combination Drivers have access to DXE, UEFI and SMM services during SMM Initialization. Combination Drivers have access to SMM services during SMM Runtime.

Combination Drivers can determine whether or not they are executing during SMM Initialization or SMM Runtime by locating the **EFI_SMM_READY_TO_LOCK_SMM_PROTOCOL**.

On the first load, the entry point of the driver is the same as a UEFI specification **EFI_IMAGE_ENTRY_POINT** since the driver is loaded by the DXE core.

On the second load, the entry point of the driver is the same as a UEFI specification **EFI_IMAGE_ENTRY_POINT**.

1.6.3 SOR and Dependency Expressions for SMM

The Apiori file can also contain DXE and SMM FFS files. The implementation doesn't support SOR for the SMM drivers, though.

1.7 SMM Driver Initialization

An SMM Driver's initialization phase begins when the driver has been loaded into SMRAM and its entry point is called. An SMM Driver's initialization phase ends when the entry point returns.

During SMM Driver initialization, SMM Drivers have access to two sets of protocols: UEFI and SMM. UEFI protocols are those which are installed and discovered using the UEFI Boot Services. UEFI protocols can be located and used by SMM drivers only during SMM Initialization. SMM protocols are those which are installed and discovered using the System Management Services Table (SMST). SMM protocols can be discovered by SMM drivers during initialization time and accessed while inside of SMM.

SMM Drivers should not use the following UEFI Boot Services during SMM Driver Initialization:

- Exit()
- ExitBootServices()

1.8 SMM Driver Runtime

During SMM Driver runtime, SMM drivers only have access to SMM protocols. In addition, depending on the platform architecture, memory areas outside of SMRAM may not be accessible to SMM Drivers. Likewise, memory areas inside of SMRAM may not be accessible to UEFI drivers.

These SMM Driver Runtime characteristics lead to several restrictions regarding the usage of UEFI services:

- UEFI interfaces and services which are located during SMM Driver Initialization should not be called or referenced during SMM Driver Runtime. This includes the EFI System Table, the UEFI Boot Services and the UEFI Runtime Services.
- Installed UEFI protocols should be uninstalled before exiting the driver entry point OR the UEFI protocol should refer to addresses which are not within SMRAM..
- Events created during SMM Driver Initialization should be closed before exiting the drier entry point..

1.9 Dispatching SMI Handlers

SMI handlers are registered using the SMST's **`SmiHandlerRegister()`** function. SMI handlers fall into three categories:

Root SMI Controller Handlers

These are handlers for devices which directly control SMI generation for the CPU(s). The handlers have the ability to detect, clear and disable one or more SMI sources. They are registered by calling **`SmiHandlerRegister()`** with *`HandlerType`* set to NULL. After an SMI source has been detected, the Root SMI handler calls the Child SMI Controllers or SMI Handlers whose handler functions were registered using either a SMM Child Dispatch protocols or using **`SmiHandlerRegister()`**. To call the latter, it calls **`Manage()`** with a GUID identifying the SMI source so that any registered Child SMI Handlers or Leaf SMI Handlers will be called. If the handler returns **`EFI_INTERRUPT_PENDING`**, it indicates that the interrupt source could not be quiesced. If possible, the Root SMI handler should disable

and clear the SMI source. If the handler does not return an error, the Root SMI Handler should clear the SMI source.

Child SMI Controller Handlers

These are SMI handlers which handle a single interrupt source from a Root or Child SMI handler and, in turn, control one or more child SMI sources which can be detected, cleared and disabled. They are registered by calling the **`SmiHandlerRegister()`** function with *`HandlerType`* set to the GUID of the Parent SMI Controller SMI source. Handlers for this SMI handler's SMI sources are called in the same manner as Root SMI Handlers.

SMI Handlers

These SMI handlers perform basic software or hardware services based on the SMI source received. If the SMI handler manages a device outside the control of the Parent SMI Controller, it must make sure that the device is quiesced, especially if the device drives a level-active input.

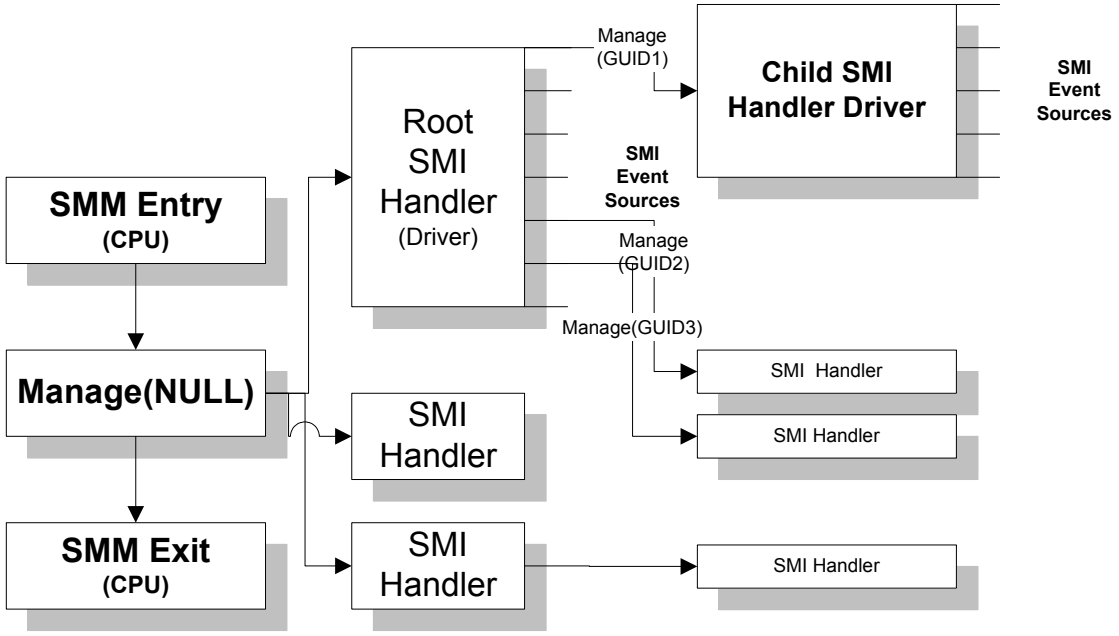

Figure 3. SMI Handler Relationships

1.10 SMM Services

1.10.1 SMM Driver Model

The SMM driver model has similar constraints to those of UEFI runtime drivers. Specifically, while inside of SMM, the drivers may not be able to use core protocol services. There will be SMST-based services, which the drivers can access, but the UEFI System Table and other protocols installed during boot services may not necessarily be available.

Instead, the full collection of UEFI Boot Services and UEFI Runtime Services are available only during the driver initialization phase. This visibility is useful so that the SMM driver can leverage the rich set of UEFI services to do the following:

- Marshall interfaces to other UEFI services.

  This design makes the UEFI 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
- A minimal protocol database for protocols for use inside of SMM.

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

1.10.2 SMM Protocols

Additional standard protocols are exposed as SMM protocols that are located during the initialization phase of the SMM driver in SMM. For example, the status code equivalent in SMM is simply a UEFI 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.

1.11 SMM UEFI Protocols

1.11.1 UEFI Protocols

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

- SMM Base Protocol
- SMM Access Protocol
- SMM Control Protocol

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

- Opening SMRAM
- Creating the SMRAM heap
- Registering the handlers

The *SMM Access Protocol* understands the particular enable and locking mechanisms that memory controller might support while executing in SMM.

The *SMM Control Protocol* understands how to trigger synchronous SMIs either once or periodically.

1.11.2 SMM Protocols

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

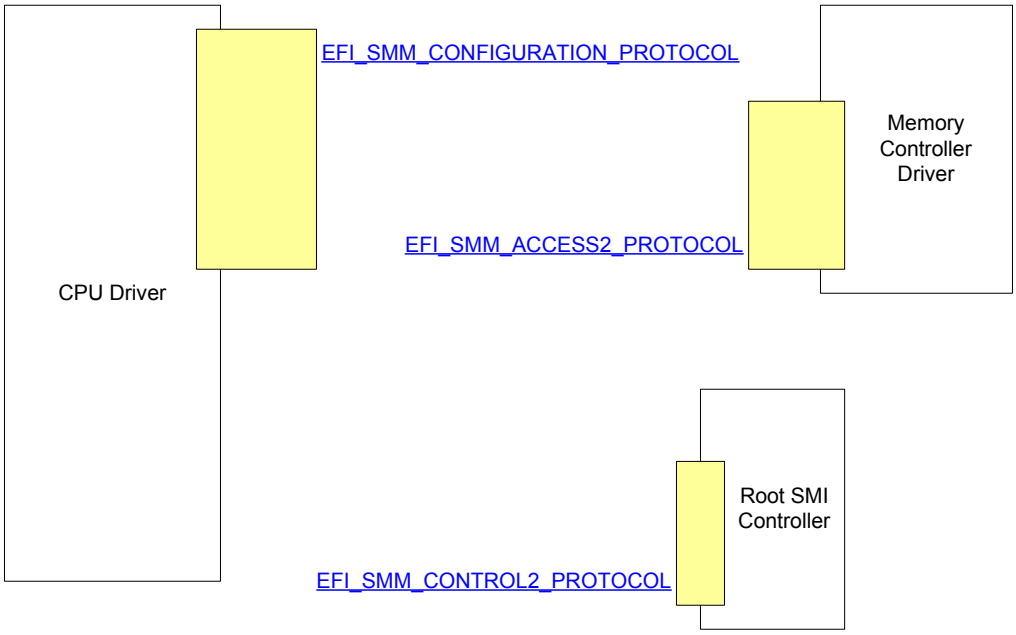

Figure 4. Published Protocols for IA-32 Systems

2
SMM Foundation Entry Point

2.1 EFI_SMM_ENTRY_POINT

Summary

This function is the main entry point to the SMM Foundation.

Prototype

```
typedef
VOID
(EFIAPI *EFI_SMM_ENTRY_POINT) (
  IN CONST EFI_SMM_ENTRY_CONTEXT *SmmEntryContext
  );
```

Parameters

SmmEntryContext

Processor information and functionality needed by SMM Foundation.

Description

This function is the entry point to the SMM Foundation. The processor SMM entry code will call this function with the processor information and functionality necessary for SMM

Related Definitions

```
typedef struct _EFI_SMM_ENTRY_CONTEXT {
  EFI_SMM_STARTUP_THIS_AP SmmStartupThisAp;
  UINTN                   CurrentlyExecutingCpu;
  UINTN                   NumberOfCpus;
  UINTN                   *CpuSaveStateSize;
  VOID                    **CpuSaveState;
} EFI_SMM_ENTRY_CONTEXT;
```

SmmStartupThisAp

Initiate a procedure on an application processor while in SMM. See the **SmmStartupThisAp()** function description.

CurrentlyExecutingCpu

A number between zero and the *NumberOfCpus* field. This field designates which processor is executing the SMM Foundation.

NumberOfCpus

The number of current operational processors in the platform. This is a 1 based counter. This does not indicate the number of processors that entered SMM.

CpuSaveStateSize

Points to an array, where each element describes the number of bytes in the corresponding save state specified by *CpuSaveState*. There are always *NumberOfCpus* entries in the array.

CpuSaveState

Points to an array, where each element is a pointer to a CPU save state. The corresponding element in *CpuSaveStateSize* specifies the number of bytes in the save state area. There are always *NumberOfCpus* entries in the array.

3 System Management System Table (SMST)

3.1 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 similar to the UEFI System Table. It is a fixed set of services and data that are designed to provide basic services for SMM drivers. The SMST is provided by the SMM IPL driver, which also manages the following:

- Dispatch of drivers in SMM
- Allocations of SMRAM
- Installation/discovery of SMM protocols

3.2 EFI_SMM_SYSTEM_TABLE2

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 SMM_SPECIFICATION_MAJOR_REVISION  1
#define SMM_SPECIFICATION_MINOR_REVISION  14
#define EFI_SMM_SYSTEM_TABLE2_REVISION
((SMM_SPECIFICATION_MAJOR_REVISION<<16) |
(SMM_SPECIFICATION_MINOR_REVISION)

typedef struct _EFI_SMM_SYSTEM_TABLE2 {
  EFI_TABLE_HEADER                      Hdr;

  CHAR16                                *SmmFirmwareVendor;
  UINT32                                SmmFirmwareRevision;

  EFI_SMM_INSTALL_CONFIGURATION_TABLE2 SmmInstallConfigurationTable;

EFI_SMM_CPU_IO2_PROTOCOL                SmmIo;

  //
  // Runtime memory service
```

```
  //
  EFI_ALLOCATE_POOL                 SmmAllocatePool;
  EFI_FREE_POOL                     SmmFreePool;
  EFI_ALLOCATE_PAGES                SmmAllocatePages;
  EFI_FREE_PAGES                    SmmFreePages;

  //
  // MP service
  //
  EFI_SMM_STARTUP_THIS_AP           SmmStartupThisAp;

  //
  // CPU information records
  //
  UINTN                             CurrentlyExecutingCpu;
  UINTN                             NumberOfCpus;
UINTN                               *CpuSaveStateSize;
VOID                                **CpuSaveState;

  //
  // Extensibility table
  //
  UINTN                             NumberOfTableEntries;
  EFI_CONFIGURATION_TABLE           *SmmConfigurationTable;

//
// Protocol services
//
  EFI_INSTALL_PROTOCOL_INTERFACE    SmmInstallProtocolInterface;
  EFI_UNINSTALL_PROTOCOL_INTERFACE  SmmUninstallProtocolInterface;
  EFI_HANDLE_PROTOCOL               SmmHandleProtocol;
  EFI_SMM_REGISTER_PROTOCOL_NOTIFY  SmmRegisterProtocolNotify;
EFI_LOCATE_HANDLE                   SmmLocateHandle;
EFI_LOCATE_PROTOCOL                 SmmLocateProtocol;

//
// SMI management functions
//
EFI_SMM_INTERRUPT_MANAGE            SmiManage;
EFI_SMM_INTERRUPT_REGISTER          SmiHandlerRegister;
EFI_SMM_INTERRUPT_UNREGISTER        SmiHandlerUnRegister;
} EFI_SMM_SYSTEM_TABLE2;
```

Parameters

Hdr

The table header for the System Management System Table (SMST). This header contains the **SMM_SMST_SIGNATURE** and **EFI_SMM_SYSTEM_TABLE2_REVISION** values along with the size of the

EFI_SMM_SYSTEM_TABLE2 structure and a 32-bit CRC to verify that the contents of the SMST are valid.

Note: *In the SMM Foundation use of the* **EFI_TABLE_HEADER** *for the System Management Services Table (SMST), there is special treatment of the CRC32 field. This value is ignorable for SMM and should be set to zero*

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.

SmmIo

Provides the basic memory and I/O interfaces that are used to abstract accesses to devices. The I/O services are provided by the driver which produces the SMM CPU I/O Protocol. If that driver has not been loaded yet, this function pointer will return **EFI_UNSUPPORTED**.

SmmAllocatePool

Allocates SMRAM.

SmmFreePool

Returns pool memory to the system.

SmmAllocatePages

Allocates pages from SMRAM.

SmmFreePages

Returns pages of memory to the system.

SmmStartupThisAp

Initiate a procedure on an application processor while in SMM. See the **SmmStartupThisAp()** function description. *SmmStartupThisAp* may not be used in the entry point of an SMM driver and must be considered "undefined". This service only defined while an SMI is being processed.

CurrentlyExecutingCpu

A number between zero and and the *NumberOfCpus* field. This field designates which processor is executing the SMM infrastructure. *CurrentlyExecutingCpu* may not be used in the entry point of an SMM driver and must be considered "undefined". This field is only defined while an SMI is being processed.

NumberOfCpus

The number of possible processors in the platform. This is a 1 based counter. *NumberOfCpus* may not be used in the entry point of an SMM driver and must be considered "undefined". This field is only defined while an SMI is being processed.

CpuSaveStateSize

Points to an array, where each element describes the number of bytes in the corresponding save state specified by *CpuSaveState*. There are always *NumberOfCpus* entries in the array. *CpuSaveStateSize* may not be used in the entry point of an SMM driver and must be considered "undefined". This field is only defined while an SMI is being processed.

CpuSaveState

Points to an array, where each element is a pointer to a CPU save state. The corresponding element in *CpuSaveStateSize* specifies the number of bytes in the save state area. There are always *NumberOfCpus* entries in the array. *CpuSaveState* may not be used in the entry point of an SMM driver and must be considered "undefined". This field is only defined while an SMI is being processed.

NumberOfTableEntries

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

SmmConfigurationTable

A pointer to the UEFI Configuration Tables. The number of entries in the table is *NumberOfTableEntries*. Type **EFI_CONFIGURATION_TABLE** is defined in the UEFI 2.1 specification, section 4.6.

SmmInstallProtocolInterface

Installs an SMM protocol interface on a device handle. Type **EFI_INSTALL_PROTOCOL_INTERFACE** is defined in the UEFI specification, section 4.4.

SmmUninstallProtocolInterface

Removes a SMM protocol interface from a device handle. Type **EFI_UNINSTALL_PROTOCOL_INTERFACE** is defined in the UEFI 2.1 specification, section 4.4.

SmmHandleProtocol

Queries a handle to determine if it supports a specified SMM protocol. Type **EFI_HANDLE_PROTOCOL** is defined in the UEFI 2.1 specification, section 4.4.

SmmRegisterProtocolNotify

Registers a callback routine that will be called whenever an interface is installed for a specified SMM protocol.

SmmLocateHandle

Returns an array of handles that support a specified SMM protocol. Type **EFI_LOCATE_HANDLE** is defined in the UEFI 2.1 specification, section 4.4.

SmmLocateProtocol

Returns the first installed interface for a specific SMM protocol. Type **EFI_LOCATE_PROTOCOL** is defined in the UEFI 2.1 specification, section 4.4.

SmiManage

Manage SMI sources of a particular type.

SmiHandlerRegister

Registers an SMI handler for an SMI source.

SmiHandlerUnRegister

Unregisters an SMI handler for an SMI source.

Description

The *CurrentlyExecutingCpu* parameter is a value that is less than the *NumberOfCpus* field. The *CpuSaveState* is a pointer to an array of CPU save states 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 **EFI_SMM_SYSTEM_TABLE2** provides support for SMRAM allocation. The functions have the same function prototypes as UEFI Boot Services, but are only effective in allocating and freeing SMRAM. Drivers cannot allocate or free UEFI memory using these services. Drivers cannot allocate or free SMRAM using the UEFI Boot Services. The functions are:

- **SmmAllocatePages()**
- **SmmFreePages()**
- **SmmAllocatePool()**
- **SmmFreePool()**

The **EFI_SMM_SYSTEM_TABLE2** provides support for SMM protocols, which are runtime protocols designed to execute exclusively inside of SMM. Drivers cannot access protocols installed using the UEFI Boot Services through this interface. Drivers cannot access protocols installed using these interfaces through the UEFI Boot Services interfaces.

Five of the standard protocol-related functions from the UEFI boot services table are provided in the SMST and perform in a similar fashion. These functions are required to be available until the **EFI_SMM_READY_TO_LOCK_PROTOCOL** notification has been installed. The functions are:

- **SmmInstallProtocolInterface()**
- **SmmUninstallProtocolInterface()**
- **SmmLocateHandle()**
- **SmmHandleProtocol()**
- **SmmLocateProtocol()**.

Noticeably absent are services which support the UEFI driver model. The function **SmmRegisterProtocolNotify()**, works in a similar fashion to the UEFI 2.1 function except that it does not use an event.

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_TABLE2) (
  IN CONST EFI_SMM_SYSTEM_TABLE2  *SystemTable,
  IN CONST 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.*

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.

SmmAllocatePool()

Summary

Allocates pool memory from SMRAM.

Prototype

Type **EFI_ALLOCATE_POOL** is defined in the UEFI 2.1 specification, section 4.4. The function description is found in the UEFI 2.1 specification, section 6.2.

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 referenced by *Buffer*. This function allocates pages from **EfiConventionalMemory** as needed to grow the requested pool type. All allocations are eight-byte aligned.

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

Note: *All allocations of SMRAM should use* **EfiRuntimeServicesCode** *or* **EfiRuntimeServicesData**.

Status Codes Returned

EFI_SUCCESS	The requested number of bytes was allocated.
EFI_OUT_OF_RESOURCES	The pool requested could not be allocated.
EFI_INVALID_PARAMETER	*PoolType* was invalid.

SmmFreePool()

Summary

Returns pool memory to the system.

Prototype

Type **EFI_FREE_POOL** is defined in the UEFI 2.1 specification, section 4.4. The function description is found in the UEFI 2.1 specification, section 6.2.

Description

The **SmmFreePool()** function returns the memory specified by *Buffer* to the SMRAM heap. 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.

SmmAllocatePages()

Summary

Allocates page memory from SMRAM.

Prototype

Type **EFI_ALLOCATE_PAGES** is defined in the UEFI 2.1 specifications, section 4.4. The function description is found in the UEFI 2.1 specification, section 6.2.

Description

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

All allocations of SMRAM should use **EfiRuntimeServicesCode** or **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.
- **AllocateMaxAddress** allocate any available range of pages whose uppermost address is less than or equal to the address pointed to by *Memory* on input.
- **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

Returns pages of memory to the system.

Protocol

Type **EFI_FREE_PAGES** is defined in the UEFI 2.1 specifications, section 4.4. The function description is found in the UEFI 2.1 specification, section 6.2.

Description

The **SmmFreePages()** function returns memory allocated by **SmmAllocatePages()** to the SMRAM heap.

Status Codes Returned

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

SmmStartupThisAp()

Summary

This service lets the caller to get one distinct application processor (AP) to execute a caller-provided code stream while in SMM.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMM_STARTUP_THIS_AP) (
  IN  EFI_AP_PROCEDURE           Procedure
  IN  UINTN                      CpuNumber,
  IN  OUT VOID                   *ProcArguments OPTIONAL
  );
```

Parameters

Procedure

A pointer to the code stream to be run on the designated AP of the system. Type **EFI_AP_PROCEDURE** is defined below.

CpuNumber

The zero-based index of the processor number of the AP on which the code stream is supposed to run. If the processor number points to the current processor, then it will not run the supplied code.

ProcArguments

Allows the caller to pass a list of parameters to the code that is run by the AP. It is an optional common mailbox between APs and the caller to share information.

Related Definitions

See Volume 2, EFI_MP_SERVICES_PROTOCOL.StartupAllAPs, Related definitions.

Description

This function is used to dispatch one specific, healthy, enabled, and non-busy AP out of the processor pool to the code stream that is provided by the caller while in SMM. The recovery of a failed AP is optional and the recovery mechanism is implementation dependent.

Status Codes Returned

EFI_SUCCESS	The call was successful and the return parameters are valid.
EFI_INVALID_PARAMETER	The input arguments are out of range.
EFI_INVALID_PARAMETER	The CPU requested is not available on this SMI invocation.
EFI_INVALID_PARAMETER	The CPU cannot support an additional service invocation.

SmmInstallProtocolInterface()

Summary

Installs a SMM protocol interface on a device handle. If the handle does not exist, it is created and added to the list of handles in the system.

Prototype

Type **EFI_INSTALL_PROTOCOL_INTERFACE** is defined in the UEFI 2.1 specification, section 4.4. The function description is found in the UEFI 2.1 specification, section 6.3.1.

Description

The **SmmInstallProtocolInterface()** function installs a protocol interface (a GUID/Protocol Interface structure pair) on an SMM device handle. The same GUID cannot be installed more than once onto the same handle. If installation of a duplicate GUID on a handle is attempted, an **EFI_INVALID_PARAMETER** will result. Installing a protocol interface allows other SMM drivers to locate the *Handle*, and the interfaces installed on it.

When a protocol interface is installed, the firmware calls all notification functions that have registered to wait for the installation of *Protocol*. For more information, see the **SmmRegisterProtocolNotify()** function description.

Status Codes Returned

EFI_SUCCESS	The protocol interface was installed.
EFI_OUT_OF_RESOURCES	Space for a new handle could not be allocated.
EFI_INVALID_PARAMETER	*Handle* is **NULL**.
EFI_INVALID_PARAMETER	*Protocol* is **NULL**.
EFI_INVALID_PARAMETER	*InterfaceType* is not **EFI_NATIVE_INTERFACE**.
EFI_INVALID_PARAMETER	*Protocol* is already installed on the handle specified by *Handle*.

SmmUninstallProtocolInterface()

Summary

Removes a SMM protocol interface from a device handle.

Prototype

Type **EFI_UNINSTALL_PROTOCOL_INTERFACE** is defined in the UEFI 2.1 specification, section 4.4. The function description is found in the UEFI 2.1 specification, section 6.3.1.

Description

The **SmmUninstallProtocolInterface()** function removes a protocol interface from the handle on which it was previously installed. The *Protocol* and *Interface* values define the protocol interface to remove from the handle.

The caller is responsible for ensuring that there are no references to a protocol interface that has been removed. If the last protocol interface is removed from a handle, the handle is freed and is no longer valid.

Status Codes Returned

EFI_SUCCESS	The interface was removed.
EFI_NOT_FOUND	The interface was not found.
EFI_ACCESS_DENIED	The interface was not removed because the interface is still being used by a driver.
EFI_INVALID_PARAMETER	*Handle* is not a valid **EFI_HANDLE**.
EFI_INVALID_PARAMETER	*Protocol* is **NULL**.

SmmHandleProtocol()

Summary

Queries a handle to determine if it supports a specified SMM protocol.

Prototype

Type **EFI_HANDLE_PROTOCOL** is defined in the UEFI 2.1 specification, section 4.4. The function description is found in the UEFI 2.1 specification, section 6.3.1.

Description

The **SmmHandleProtocol()** function queries *Handle* to determine if it supports *Protocol*. If it does, then, on return, *Interface* points to a pointer to the corresponding Protocol Interface.

Interface can then be passed to any protocol service to identify the context of the request.

Status Codes Returned

EFI_SUCCESS	The interface information for the specified protocol was returned.
EFI_UNSUPPORTED	The device does not support the specified protocol.
EFI_INVALID_PARAMETER	*Handle* is not a valid **EFI_HANDLE**.
EFI_INVALID_PARAMETER	*Protocol* is **NULL**.
EFI_INVALID_PARAMETER	*Interface* is **NULL**.

SmmRegisterProtocolNotify()

Summary

Register a callback function be called when a particular protocol interface is installed.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMM_REGISTER_PROTOCOL_NOTIFY)(
  IN  CONST EFI_GUID        *Protocol,
  IN  EFI_SMM_NOTIFY_FN     Function,
  IN OUT VOID               **Registration
);
```

Parameters

Protocol

The unique ID of the protocol for which the event is to be registered. Type **EFI_GUID** is defined in the **InstallProtocolInterface()** function description.

Function

Points to the notification function, which is described below.

Registration

A pointer to a memory location to receive the registration value. This value must be saved and used by the notification function to retrieve the list of handles that have added a protocol interface of type *Protocol*.

Description

The **SmmRegisterProtocolNotify()** function creates a registration *Function* that is to be called whenever a protocol interface is installed for *Protocol* by **SmmInstallProtocolInterface()**.

When *Function* has been called, the **SmmLocateHandle()** function can be called to identify the newly installed handles that support *Protocol*. The *Registration* parameter in **SmmRegisterProtocolNotify()** corresponds to the *SearchKey* parameter in **SmmLocateHandle()**. Note that the same handle may be returned multiple times if the handle reinstalls the target protocol ID multiple times.

If *Function* == NULL and *Registration* is an existing registration, then the callback is unhooked. **Protocol* must be validated it with **Registration*. If *Registration* is not found then **EFI_NOT_FOUND** is returned.

Related Definitions

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMM_NOTIFY_FN)(
  IN CONST EFI_GUID    *Protocol,
```

```
IN VOID              *Interface,
IN EFI_HANDLE        Handle
);
```

Protocol

Points to the protocol's unique identifier.

Interface

Points to the interface instance.

Handle

The handle on which the interface was installed.

Status Codes Returned

EFI_SUCCESS	Successfully returned the registration record that has been added or unhooked.
EFI_INVALID_PARAMETER	*Protocol* is NULL or *Registration* is NULL.
EFI_OUT_OF_RESOURCES	Not enough memory resource to finish the request.
EFI_NOT_FOUND	If the registration is not found when Function == NULL

SmmLocateHandle()

Summary

Returns an array of handles that support a specified protocol.

Prototype

Type **EFI_LOCATE_HANDLE** is defined in the UEFI 2.1 specification, section 4.4. The function description is found in the UEFI 2.1 specification, section 6.3.1.

Description

The **SmmLocateHandle()** function returns an array of handles that match the *SearchType* request. If the input value of *BufferSize* is too small, the function returns **EFI_BUFFER_TOO_SMALL** and updates *BufferSize* to the size of the buffer needed to obtain the array.

Status Codes Returned

EFI_SUCCESS	The array of handles was returned.
EFI_NOT_FOUND	No handles match the search.
EFI_BUFFER_TOO_SMALL	The *BufferSize* is too small for the result. *BufferSize* has been updated with the size needed to complete the request.
EFI_INVALID_PARAMETER	*SearchType* is not a member of **EFI_LOCATE_SEARCH_TYPE**.
EFI_INVALID_PARAMETER	*SearchType* is **ByRegisterNotify** and *SearchKey* is **NULL**.
EFI_INVALID_PARAMETER	*SearchType* is **ByProtocol** and *Protocol* is **NULL**.
EFI_INVALID_PARAMETER	One or more matches are found and *BufferSize* is **NULL**.
EFI_INVALID_PARAMETER	*BufferSize* is large enough for the result and *Buffer* is **NULL**.

SmmLocateProtocol()

Summary

Returns the first SMM protocol instance that matches the given protocol.

Prototype

Type **EFI_LOCATE_PROTOCOL** is defined in the UEFI 2.1 specification, section 4.4. The function description is found in the UEFI 2.1 specification, section 6.3.1.

Description

The **SmmLocateProtocol()** function finds the first device handle that support *Protocol*, and returns a pointer to the protocol interface from that handle in *Interface*. If no protocol instances are found, then *Interface* is set to **NULL**.

If *Interface* is **NULL**, then **EFI_INVALID_PARAMETER** is returned.

If *Registration* is **NULL**, and there are no handles in the handle database that support *Protocol*, then **EFI_NOT_FOUND** is returned.

If *Registration* is not **NULL**, and there are no new handles for *Registration*, then **EFI_NOT_FOUND** is returned.

Status Codes Returned

EFI_SUCCESS	A protocol instance matching *Protocol* was found and returned in *Interface*.
EFI_INVALID_PARAMETER	*Interface* is **NULL**.
EFI_NOT_FOUND	No protocol instances were found that match *Protocol* and *Registration*.

SmiManage()

Summary

Manage SMI of a particular type.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMM_INTERRUPT_MANAGE)(
 IN CONST EFI_GUID      *HandlerType,
 IN CONST VOID          *Context          OPTIONAL,
 IN OUT VOID            *CommBuffer       OPTIONAL,
 IN OUT UINTN           *CommBufferSize   OPTIONAL
 );
```

Parameters

HandlerType

Points to the handler type or NULL for root SMI handlers.

Context

Points to an optional context buffer. The format of the contents of the context buffer depends on *HandlerType*.

CommBuffer

Points to the optional communication buffer. The format of the contents of the communication buffer depends on *HandlerType*. The contents of the buffer (and its size) may be altered if **EFI_SUCCESS** is returned.

CommBufferSize

Points to the size of the optional communication buffer. The size of the buffer may be altered if **EFI_SUCCESS** is returned.

Description

This function will call the registered handler functions which match the specified interrupt type.

If NULL is passed in *HandlerType*, then only those registered handler functions which passed NULL as their *HandlerType* will be called. If NULL is passed in *HandlerType*, then Context should be NULL, *CommBuffer* should point to an instance of **EFI_SMM_ENTRY_CONTEXT** and *CommBufferSize* should point to the size of that structure. Type **EFI_SMM_ENTRY_CONTEXT** is defined in "Related Definitions" below.

If at least one of the handlers returns **EFI_WARN_INTERRUPT_SOURCE_QUIESCED** or **EFI_SUCCESS** then the function will return **EFI_SUCCESS**. If a handler returns **EFI_SUCCESS** and *HandlerType* is not NULL then no additional handlers will be processed.

If a handler returns **EFI_INTERRUPT_PENDING** and *HandlerType* is not NULL then no additional handlers will be processed and **EFI_INTERRUPT_PENDING** will be returned.

If all the handlers returned **EFI_WARN_INTERRUPT_SOURCE_PENDING** then **EFI_WARN_INTERRUPT_SOURCE_PENDING** will be returned.

If no handlers of *HandlerType* are found then **EFI_NOT_FOUND** will be returned.

Status Codes Returned

EFI_WARN_INTERRUPT_SOURCE_PENDING	Interrupt source was processed successfully but not quiesced.
EFI_INTERRUPT_PENDING	One or more SMI sources could not be quiesced.
EFI_NOT_FOUND	Interrupt source was not handled or quiesced.
EFI_SUCCESS	Interrupt source was handled and quiesced.

SmiHandlerRegister()

Summary

Registers a handler to execute within SMM.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMM_INTERRUPT_REGISTER) (
  IN  EFI_SMM_HANDLER_ENTRY_POINT2  Handler,
  IN  CONST EFI_GUID                *HandlerType OPTIONAL,
  OUT EFI_HANDLE                    *DispatchHandle
  );
```

Parameters

Handler

Handler service function pointer. Type **EFI_SMM_HANDLER_ENTRY_POINT2** is defined in "Related Definitions" below.

HandlerType

Points to an **EFI_GUID** which describes the type of interrupt that this handler is for or **NULL** to indicate a root SMI handler.

DispatchHandle

On return, contains a unique handle which can be used to later unregister the handler function. It is also passed to the handler function itself.

Description

This service allows the registration of a SMI handling function from within SMM.

The handler should have the **EFI_SMM_HANDLER_ENTRY_POINT2** interface defined in "Related Definitions" below.

Related Definitions

```
//**********************************************
// EFI_SMM_HANDLER_ENTRY_POINT2
//**********************************************

typedef
EFI_STATUS
(EFIAPI *EFI_SMM_HANDLER_ENTRY_POINT2) (
IN EFI_HANDLE       DispatchHandle,
IN CONST VOID       *Context           OPTIONAL,
  IN OUT VOID       *CommBuffer        OPTIONAL,
  IN OUT UINTN      *CommBufferSize    OPTIONAL
  );
```

DispatchHandle

The unique handle assigned to this handler by **SmiHandlerRegister()**.Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *UEFI 2.1 Specification.*

Context

Points to the optional handler context which was specified when the handler was registered.

CommBuffer

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.

CommBufferSize

The size of the *CommBuffer*.

SmiHandlerRegister() returns one of two status codes:

Status Codes Returned (SmiHandlerRegister())

EFI_SUCCESS	SMI handler added successfully.
EFI_INVALID_PARAMETER	Handler is **NULL** or *DispatchHandle* is **NULL**

EFI_SMM_HANDLER_ENTRY_POINT2 returns one of four status codes:

Status Codes Returned (EFI_SMM_HANDLER_ENTRY_POINT2)

EFI_SUCCESS	The interrupt was handled and quiesced. No other handlers should still be called.
EFI_WARN_INTERRUPT_SOURCE_QUIESCED	The interrupt has been quiesced but other handlers should still be called.
EFI_WARN_INTERRUPT_SOURCE_PENDING	The interrupt is still pending and other handlers should still be called.
EFI_INTERRUPT_PENDING	The interrupt could not be quiesced.

SmiHandlerUnRegister()

Summary

Unregister a handler in SMM.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMM_INTERRUPT_UNREGISTER)(
  IN EFI_HANDLE                       DispatchHandle,
  );
```

Parameters

DispatchHandle

The handle that was specified when the handler was registered.

Description

This function unregisters the specified handler function.

Status Codes Returned

EFI_SUCCESS	Handler function was successfully unregistered.
EFI_INVALID_PARAMETER	*DispatchHandle* does not refer to a valid handle.

4
SMM Protocols

4.1 Introduction

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

The services described in this section coexist with a foreground pre-boot or runtime environment. The latter can include both UEFI and non-UEFI 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.

4.2 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 CONST EFI_SMM_STATUS_CODE_PROTOCOL *This,
  IN EFI_STATUS_CODE_TYPE               CodeType,
  IN EFI_STATUS_CODE_VALUE              Value,
  IN UINT32                             Instance,
  IN CONST EFI_GUID                     *CallerId,
  IN EFI_STATUS_CODE_DATA               *Data OPTIONAL
  );
```

Parameters

This

Points to this instance of the **EFI_SMM_STATUS_CODE_PROTOCOL**.

CodeType

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.

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.

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 SMM 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 SMM 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.*

Status Codes Returned

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

4.3 CPU Save State Access Services

EFI_SMM_CPU_PROTOCOL

Summary

Provides access to CPU-related information while in SMM.

GUID

```
#define EFI_SMM_CPU_PROTOCOL_GUID \
  { 0xeb346b97, 0x975f, 0x4a9f, \
  0x8b, 0x22, 0xf8, 0xe9, 0x2b, 0xb3, 0xd5, 0x69 }
```

Prototype

```
typedef struct _EFI_SMM_CPU_PROTOCOL {
  EFI_SMM_READ_SAVE_STATE  ReadSaveState;
  EFI_SMM_WRITE_SAVE_STATE WriteSaveState;
} EFI_SMM_CPU_PROTOCOL;
```

Members

ReadSaveState

Read information from the CPU save state. See **ReadSaveState()** for more information.

WriteSaveState

Write information to the CPU save state. See **WriteSaveState()** for more information.

Description

This protocol allows SMM drivers to access architecture-standard registers from any of the CPU save state areas. In some cases, difference processors provide the same information in the save state, but not in the same format. These so-called pseudo-registers provide this information in a standard format.

EFI_SMM_CPU_PROTOCOL.ReadSaveState()

Summary

Read data from the CPU save state.

Prototype

```
typedef
  EFI_STATUS
(EFIAPI *EFI_SMM_READ_SAVE_STATE (
  IN  CONST EFI_SMM_CPU_PROTOCOL  *This,
  IN  UINTN                       Width,
  IN  EFI_SMM_SAVE_STATE_REGISTER Register,
  IN  UINTN                       CpuIndex,
  OUT VOID                        *Buffer
  );
```

Parameters

Width

The number of bytes to read from the CPU save state. If the register specified by *Register* does not support the size specified by *Width*, then **EFI_INVALID_PARAMETER** is returned.

Register

Specifies the CPU register to read form the save state. The type **EFI_SMM_SAVE_STATE_REGISTER** is defined in "Related Definitions" below. If the specified register is not implemented in the CPU save state map then **EFI_NOT_FOUND** error will be returned.

CpuIndex

Specifies the zero-based index of the CPU save state

**Buffer*

Upon return, this holds the CPU register value read from the save state.

Description

This function is used to read the specified number of bytes of the specified register from the CPU save state of the specified CPU and place the value into the buffer. If the CPU does not support the specified register *Register*, then **EFI_NOT_FOUND** should be returned. If the CPU does not support the specified register width *Width*, then **EFI_INVALID_PARAMETER** is returned.

Related Definitions

```
typedef enum {

  //
  // x86/X64 standard registers
  //
```

```
EFI_SMM_SAVE_STATE_REGISTER_GDTBASE       = 4,
EFI_SMM_SAVE_STATE_REGISTER_IDTBASE       = 5,
EFI_SMM_SAVE_STATE_REGISTER_LDTBASE       = 6,
EFI_SMM_SAVE_STATE_REGISTER_GDTLIMIT      = 7,
EFI_SMM_SAVE_STATE_REGISTER_IDTLIMIT      = 8,
EFI_SMM_SAVE_STATE_REGISTER_LDTLIMIT      = 9,
EFI_SMM_SAVE_STATE_REGISTER_LDTINFO       = 10,

EFI_SMM_SAVE_STATE_REGISTER_ES            = 20,
EFI_SMM_SAVE_STATE_REGISTER_CS            = 21,
EFI_SMM_SAVE_STATE_REGISTER_SS            = 22,
EFI_SMM_SAVE_STATE_REGISTER_DS            = 23,
EFI_SMM_SAVE_STATE_REGISTER_FS            = 24,
EFI_SMM_SAVE_STATE_REGISTER_GS            = 25,
EFI_SMM_SAVE_STATE_REGISTER_LDTR_SEL      = 26,
EFI_SMM_SAVE_STATE_REGISTER_TR_SEL        = 27,
EFI_SMM_SAVE_STATE_REGISTER_DR7           = 28,
EFI_SMM_SAVE_STATE_REGISTER_DR6           = 29,

EFI_SMM_SAVE_STATE_REGISTER_R8            = 30,
EFI_SMM_SAVE_STATE_REGISTER_R9            = 31,
EFI_SMM_SAVE_STATE_REGISTER_R10           = 32,
EFI_SMM_SAVE_STATE_REGISTER_R11           = 33,
EFI_SMM_SAVE_STATE_REGISTER_R12           = 34,
EFI_SMM_SAVE_STATE_REGISTER_R13           = 35,
EFI_SMM_SAVE_STATE_REGISTER_R14           = 36,
EFI_SMM_SAVE_STATE_REGISTER_R15           = 37,

EFI_SMM_SAVE_STATE_REGISTER_RAX           = 38,
EFI_SMM_SAVE_STATE_REGISTER_RBX           = 39,
EFI_SMM_SAVE_STATE_REGISTER_RCX           = 40,
EFI_SMM_SAVE_STATE_REGISTER_RDX           = 41,
EFI_SMM_SAVE_STATE_REGISTER_RSP           = 42,
EFI_SMM_SAVE_STATE_REGISTER_RBP           = 43,
EFI_SMM_SAVE_STATE_REGISTER_RSI           = 44,
EFI_SMM_SAVE_STATE_REGISTER_RDI           = 45,
EFI_SMM_SAVE_STATE_REGISTER_RIP           = 46,

EFI_SMM_SAVE_STATE_REGISTER_RFLAGS        = 51,
EFI_SMM_SAVE_STATE_REGISTER_CR0           = 52,
EFI_SMM_SAVE_STATE_REGISTER_CR3           = 53,
EFI_SMM_SAVE_STATE_REGISTER_CR4           = 54,

EFI_SMM_SAVE_STATE_REGISTER_FCW           = 256,
EFI_SMM_SAVE_STATE_REGISTER_FSW           = 257,
EFI_SMM_SAVE_STATE_REGISTER_FTW           = 258,
EFI_SMM_SAVE_STATE_REGISTER_OPCODE        = 259,
```

```
  EFI_SMM_SAVE_STATE_REGISTER_FP_EIP        = 260,
  EFI_SMM_SAVE_STATE_REGISTER_FP_CS         = 261,
  EFI_SMM_SAVE_STATE_REGISTER_DATAOFFSET    = 262,
  EFI_SMM_SAVE_STATE_REGISTER_FP_DS         = 263,
  EFI_SMM_SAVE_STATE_REGISTER_MM0           = 264,
  EFI_SMM_SAVE_STATE_REGISTER_MM1           = 265,
  EFI_SMM_SAVE_STATE_REGISTER_MM2           = 266,
  EFI_SMM_SAVE_STATE_REGISTER_MM3           = 267,
  EFI_SMM_SAVE_STATE_REGISTER_MM4           = 268,
  EFI_SMM_SAVE_STATE_REGISTER_MM5           = 269,
  EFI_SMM_SAVE_STATE_REGISTER_MM6           = 270,
  EFI_SMM_SAVE_STATE_REGISTER_MM7           = 271,
  EFI_SMM_SAVE_STATE_REGISTER_XMM0          = 272,
  EFI_SMM_SAVE_STATE_REGISTER_XMM1          = 273,
  EFI_SMM_SAVE_STATE_REGISTER_XMM2          = 274,
  EFI_SMM_SAVE_STATE_REGISTER_XMM3          = 275,
  EFI_SMM_SAVE_STATE_REGISTER_XMM4          = 276,
  EFI_SMM_SAVE_STATE_REGISTER_XMM5          = 277,
  EFI_SMM_SAVE_STATE_REGISTER_XMM6          = 278,
  EFI_SMM_SAVE_STATE_REGISTER_XMM7          = 279,
  EFI_SMM_SAVE_STATE_REGISTER_XMM8          = 280,
  EFI_SMM_SAVE_STATE_REGISTER_XMM9          = 281,
  EFI_SMM_SAVE_STATE_REGISTER_XMM10         = 282,
  EFI_SMM_SAVE_STATE_REGISTER_XMM11         = 283,
  EFI_SMM_SAVE_STATE_REGISTER_XMM12         = 284,
  EFI_SMM_SAVE_STATE_REGISTER_XMM13         = 285,
  EFI_SMM_SAVE_STATE_REGISTER_XMM14         = 286,
  EFI_SMM_SAVE_STATE_REGISTER_XMM15         = 287,

  //
  // Pseudo-Registers
  //
  EFI_SMM_SAVE_STATE_REGISTER_IO            = 512
  EFI_SMM_SAVE_STATE_REGISTER_LMA           = 513
  EFI_SMM_SAVE_STATE_REGISTER_PROCESSOR_ID  = 514
} EFI_SMM_SAVE_STATE_REGISTER;
```

The Read/Write interface for the pseudo-register **EFI_SMM_SAVE_STATE_REGISTER_PROCESSOR_ID** follows these rules:

For **ReadSaveState()**:

The pseudo-register only supports the 64-bit size specified by *Width*.

If the processor is in SMM at the time the SMI occurred, the pseudo register value **EFI_SMM_SAVE_STATE_REGISTER_PROCESSOR_ID** is returned in Buffer. The value should match the *ProcessorId* value, as described in the **EFI_PROCESSOR_INFORMATION** record defined in Volume 2 of the *Platform Initialization Specification*.

For **WriteSaveState**():

Write operations to this pseudo-register are ignored.

The Read/Write interface for the pseudo-register **EFI_SMM_SAVE_STATE_REGISTER_LMA** follows these rules:

For **ReadSaveState()**:

The pseudo-register only supports the single Byte size specified by *Width*. If the processor acts in 32-bit mode at the time the SMI occurred, the pseudo register value **EFI_SMM_SAVE_STATE_REGISTER_LMA_32BIT** is returned in *Buffer*. Otherwise, **EFI_SMM_SAVE_STATE_REGISTER_LMA_64BIT** is returned in *Buffer*.

```
#define EFI_SMM_SAVE_STATE_REGISTER_LMA_32BIT = 32
#define EFI_SMM_SAVE_STATE_REGISTER_LMA_64BIT = 64
```

For **WriteSaveState()**:

Write operations to this pseudo-register are ignored.

Status Codes Returned

EFI_SUCCESS	The register was read or written from Save State
EFI_NOT_FOUND	The register is not defined for the Save State of Processor
EFI_NOT_FOUND	The processor is not in SMM.
EFI_INVALID_PARAMETER	Input parameters are not valid. For ex: Processor No or register width is not correct. *This* or *Buffer* is **NULL**.

EFI_SMM_CPU_PROTOCOL.WriteSaveState()

Summary

Write data to the CPU save state.

Prototype

```
typedef
  EFI_STATUS
(EFIAPI *EFI_SMM_WRITE_SAVE_STATE (
  IN  CONST EFI_SMM_CPU_PROTOCOL  *This,
  IN  UINTN                       Width,
  IN  EFI_SMM_SAVE_STATE_REGISTER Register,
  IN  UINTN                       CpuIndex,
  IN  CONST VOID                  *Buffer
  );
```

Parameters

Width

The number of bytes to write to the CPU save state. If the register specified by *Register* does not support the size specified by *Width*, then **EFI_INVALID_PARAMETER** s returned.

Register

Specifies the CPU register to write to the save state. The type **EFI_SMM_SAVE_STATE_REGISTER** is defined in **ReadSaveState()** above. If the specified register is not implemented in the CPU save state map then **EFI_NOT_FOUND** error will be returned.

CpuIndex

Specifies the zero-based index of the CPU save state.

Buffer

Upon entry, this holds the new CPU register value.

Description

This function is used to write the specified number of bytes of the specified register to the CPU save state of the specified CPU and place the value into the buffer. If the CPU does not support the specified register *Register*, then **EFI_NOT_FOUND** should be returned. If the CPU does not support the specified register width *Width*, then **EFI_INVALID_PARAMETER** is returned.

Status Codes Returned

EFI_SUCCESS	The register was read or written from Save State
EFI_NOT_FOUND	The register *Register* is not defined for the Save State of Processor
EFI_INVALID_PARAMETER	Input parameters are not valid. For example: *ProcessorIndex* or *Width* is not correct. *This* or *Buffer* is **NULL**.

4.3.1 SMM Save State IO Info

EFI_SMM_SAVE_STATE_IO_INFO

Summary

Describes the I/O operation which was in process when the SMI was generated.

Prototype

```
typedef struct _EFI_SMM_SAVE_STATE_IO_INFO {
  UINT64                          IoData;
  UINT16                          IoPort;
  EFI_SMM_SAVE_STATE_IO_WIDTH     IoWidth;
  EFI_SMM_SAVE_STATE_IO_TYPE      IoType;
} EFI_SMM_SAVE_STATE_IO_INFO
```

Parameters

IoData

For input instruction (IN, INS), this is data read before the SMI occurred. For output instructions (OUT, OUTS) this is data that was written before the SMI occurred. The width of the data is specified by *IoWidth*. The data buffer is allocated by the Called function, and it is the Caller's responsibility to free this buffer.

IoPort

The I/O port that was being accessed when the SMI was triggered.

IoWidth

Defines the size width (UINT8, UINT16, UINT32, UINT64) for *IoData*. See Related Definitions.

IoType

Defines type of I/O instruction. See Related Definitions.

Description

This is the structure of the data which is returned when **ReadSaveState()** is called with **EFI_SMM_SAVE_STATE_REGISTER_IO**. If there was no I/O then **ReadSaveState()** will return **EFI_NOT_FOUND**.

Related Definitions

```
typedef enum {
  EFI_SMM_SAVE_STATE_IO_WIDTH_UINT8        = 0,
  EFI_SMM_SAVE_STATE_IO_WIDTH_UINT16       = 1,
  EFI_SMM_SAVE_STATE_IO_WIDTH_UINT32       = 2,
  EFI_SMM_SAVE_STATE_IO_WIDTH_UINT64       = 3
} EFI_SMM_SAVE_STATE_IO_WIDTH


typedef enum {
  EFI_SMM_SAVE_STATE_IO_TYPE_INPUT         = 1,
  EFI_SMM_SAVE_STATE_IO_TYPE_OUTPUT        = 2,
  EFI_SMM_SAVE_STATE_IO_TYPE_STRING        = 4,
  EFI_SMM_SAVE_STATE_IO_TYPE_REP_PREFIX    = 8
} EFI_SMM_SAVE_STATE_IO_TYPE
```

4.4 SMM CPU I/O Protocol

EFI_SMM_CPU_IO2_PROTOCOL

Summary

Provides CPU I/O and memory access within SMM

GUID

```
#define EFI_SMM_CPU_IO2_PROTOCOL_GUID \
  { 0x3242a9d8, 0xce70, 0x4aa0, \
  0x95, 0x5d, 0x5e, 0x7b, 0x14, 0xd, 0xe4, 0xd2 }
```

Protocol Interface Structure

```
typedef struct _EFI_SMM_CPU_IO2_PROTOCOL {
  EFI_SMM_IO_ACCESS2    Mem;
  EFI_SMM_IO_ACCESS2    Io;
} EFI_SMM_CPU_IO2_PROTOCOL;
```

Parameters

Mem

Allows reads and writes to memory-mapped I/O space. See the **Mem()** function description. Type **EFI_SMM_IO_ACCESS2** is defined in "Related Definitions" below.

Io

Allows reads and writes to I/O space. See the **Io()** function description. Type **EFI_SMM_IO_ACCESS2** is defined in "Related Definitions" below.

Description

The **EFI_SMM_CPU_IO2_PROTOCOL** service provides the basic memory, I/O, and PCI interfaces that are used to abstract accesses to devices.

The interfaces provided in **EFI_SMM_CPU_IO2_PROTOCOL** are for performing basic operations to memory and I/O. The **EFI_SMM_CPU_IO2_PROTOCOL** 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.

Related Definitions

```
//*******************************************************
// EFI_SMM_IO_ACCESS2
//*******************************************************
typedef struct {
  EFI_SMM_CPU_IO2  Read;
  EFI_SMM_CPU_IO2  Write;
} EFI_SMM_IO_ACCESS2;
```

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_IO2_PROTOCOL.Mem()

Summary

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

Prototype

```
typedef
EFI_STATUS
(EFIAPI * EFI_SMM_CPU_IO2) (
  IN CONST EFI_SMM_CPU_IO2_PROTOCOL     *This,
  IN EFI_SMM_IO_WIDTH                   Width,
  IN UINT64                             Address,
  IN UINTN                              Count,
  IN OUT VOID                           *Buffer
  );
```

Parameters

This

The **EFI_SMM_CPU_IO2_PROTOCOL** 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_IO2.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_IO2_PROTOCOL.Io()

Summary

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

Prototype

```
typedef
EFI_STATUS
(EFIAPI * EFI_SMM_CPU_IO2) (
  IN CONST EFI_SMM_CPU_IO2_PROTOCOL     *This,
  IN EFI_SMM_IO_WIDTH                   Width,
  IN UINT64                             Address,
  IN UINTN                              Count,
  IN OUT VOID                           *Buffer
  );
```

Parameters

This

The **EFI_SMM_CPU_IO2_PROTOCOL** 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_IO2.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.

4.5 SMM PCI I/O Protocol

EFI_SMM_PCI_ROOT_BRIDGE_IO_PROTOCOL

Summary

Provides access to PCI I/O, memory and configuration space inside of SMM.

GUID

```
#define EFI_SMM_PCI_ROOT_BRIDGE_IO_PROTOCOL_GUID \
  {0x8bc1714d, 0xffcb, 0x41c3, \
  0x89, 0xdc, 0x6c, 0x74, 0xd0, 0x6d, 0x98, 0xea}
```

Prototype

```
typedef EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL
EFI_SMM_PCI_ROOT_BRIDGE_IO_PROTOCOL;
```

Description

This protocol provides the same functionality as the PCI Root Bridge I/O Protocol defined in the UEFI 2.1 Specifcation, section 13.2, except that the functions for **Map()**, **Unmap()**, **Flush()**, **AllocateBuffer()**, **FreeBuffer()**, **SetAttributes()**, and **Configuration()** may return **EFI_UNSUPPORTED**.

4.6 SMM Ready to Lock Protocol

EFI_SMM_READY_TO_LOCK_SMM_PROTOCOL

Summary

Indicates that SMM resources and services that should not be used by the third party code are about to be locked.

GUID

```
#define EFI_SMM_READY_TO_LOCK_PROTOCOL_GUID \
  { 0x47b7fa8c, 0xf4bd, 0x4af6, \
  {0x82, 0x0, 0x33, 0x30, 0x86, 0xf0, 0xd2, 0xc8 } }
```

Prototype

```
NULL
```

Description

This protocol is a mandatory protocol published by the SMM Foundation code when the system is preparing to lock certain resources and interfaces in anticipation of the invocation of 3rd party extensible modules. This protocol is an SMM counterpart of the *DXE SMM Ready to Lock Protocol*. This protocol prorogates resource locking notification into SMM environment. This protocol is installed after installation of the *SMM End of DXE Protocol*.

4.7 SMM End of DXE Protocol

EFI_SMM_END_OF_DXE_PROTOCOL

Summary

Indicates end of the execution phase when all of the components are under the authority of the platform manufacturer.

GUID

```
#define EFI_SMM_END_OF_DXE_PROTOCOL_GUID \
{ 0x24e70042, 0xd5c5, 0x4260, \
{ 0x8c, 0x39, 0xa, 0xd3, 0xaa, 0x32, 0xe9, 0x3d } }
```

Prototype

NULL

Description

This protocol is a mandatory protocol published by SMM Foundation code. This protocol is an SMM counterpart of the End of DXE Event. This protocol prorogates End of DXE notification into SMM environment. This protocol is installed prior to installation of the SMM Ready to Lock Protocol.

5
UEFI Protocols

5.1 Introduction

The services described in this chapter describe a series of protocols that locate the SMST, manipulate the System Management RAM (SMRAM) apertures, and generate System Management Interrupts (SMIs). Some of these protocols provide only boot services while others have both boot services and runtime services.

The following protocols are defined in this chapter:

- **EFI_SMM_BASE2_PROTOCOL**
- **EFI_SMM_ACCESS2_PROTOCOL**
- **EFI_SMM_CONTROL2_PROTOCOL**
- **EFI_SMM_CONFIGURATION_PROTOCOL**
- **EFI_SMM_COMMUNICATION_PROTOCOL**

5.2 EFI SMM Base Protocol

EFI_SMM_BASE2_PROTOCOL

Summary

This protocol is used to locate the SMST during SMM driver initialization.

GUID

```
#define EFI_SMM_BASE2_PROTOCOL_GUID \
  { 0xf4ccbfb7, 0xf6e0, 0x47fd, \
  0x9d, 0xd4, 0x10, 0xa8, 0xf1, 0x50, 0xc1, 0x91 }
```

Protocol Interface Structure

```
typedef struct _EFI_SMM_BASE2_PROTOCOL {
  EFI_SMM_INSIDE_OUT2                 InSmm;
  EFI_SMM_GET_SMST_LOCATION2          GetSmstLocation;
} EFI_SMM_BASE2_PROTOCOL;
```

Parameters

InSmm

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

GetSmstLocation

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

Description

The **EFI_SMM_BASE2_PROTOCOL** is provided by the SMM IPL driver. It is a required protocol. It will be utilized by all SMM drivers to locate the SMM infrastructure services and determine whether the driver is being invoked inside SMRAM or outside of SMRAM.

EFI_SMM_BASE2_PROTOCOL.InSmm()

Summary

Service to indicate whether the driver is currently executing in the SMM Initialization phase.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMM_INSIDE_OUT2) (
  IN CONST EFI_SMM_BASE2_PROTOCOL   *This,
  OUT BOOLEAN                       *InSmram
  )
```

Parameters

This

The **EFI_SMM_BASE2_PROTOCOL** instance.

InSmram

Pointer to a Boolean which, on return, indicates that the driver is currently executing inside of SMRAM (TRUE) or outside of SMRAM (FALSE).

Description

This service returns whether the caller is being executed in the SMM Initialization phase. For SMM drivers, this will return **TRUE** in *InSmram* while inside the driver's entry point and otherwise **FALSE**. For combination SMM/DXE drivers, this will return **FALSE** in the DXE launch. For the SMM launch, it behaves as an SMM driver.

Status Codes Returned

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

EFI_SMM_BASE2_PROTOCOL.GetSmstLocation()

Summary

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

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMM_GET_SMST_LOCATION2) (
  IN     CONST EFI_SMM_BASE2_PROTOCOL   *This,
  IN OUT EFI_SMM_SYSTEM_TABLE2          **Smst
  )
```

Parameters

This

The **EFI_SMM_BASE2_PROTOCOL** instance.

Smst

On return, points to a pointer to the System Management Service Table (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 handlers.

Status Codes Returned

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

5.3 SMM Access Protocol

EFI_SMM_ACCESS2_PROTOCOL

Summary

This protocol is used to control the visibility of the SMRAM on the platform.

GUID

```
#define EFI_SMM_ACCESS2_PROTOCOL_GUID \
  { 0xc2702b74, 0x800c, 0x4131, \
  0x87, 0x46, 0x8f, 0xb5, 0xb8, 0x9c, 0xe4, 0xac }
```

Protocol Interface Structure

```
typedef struct _EFI_SMM_ACCESS2_PROTOCOL {
  EFI_SMM_OPEN2           Open;
  EFI_SMM_CLOSE2          Close;
  EFI_SMM_LOCK2           Lock;
  EFI_SMM_CAPABILITIES2   GetCapabilities;
  BOOLEAN                 LockState;
  BOOLEAN                 OpenState;
} EFI_SMM_ACCESS2_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 about all SMRAM regions. See the **GetCapabilities()** function description.

LockState

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

OpenState

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

Description

The **EFI_SMM_ACCESS2_PROTOCOL** abstracts the location and characteristics of SMRAM. 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

EFI_SMM_ACCESS2_PROTOCOL.Open()

Summary

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

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMM_OPEN2) (
  IN EFI_SMM_ACCESS2_PROTOCOL *This
  );
```

Parameters

This

The **EFI_SMM_ACCESS2_PROTOCOL** instance.

Description

This function "opens" SMRAM so that it is visible while not inside of SMM. The function should return **EFI_UNSUPPORTED** if the hardware does not support hiding of SMRAM. The function should return **EFI_DEVICE_ERROR** if the SMRAM configuration is locked.

Status Codes Returned

EFI_SUCCESS	The operation was successful.
EFI_UNSUPPORTED	The system does not support opening and closing of SMRAM.
EFI_DEVICE_ERROR	SMRAM cannot be opened, perhaps because it is locked.

EFI_SMM_ACCESS2_PROTOCOL.Close()

Summary

Inhibits access to the SMRAM.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMM_CLOSE2) (
  IN EFI_SMM_ACCESS2_PROTOCOL   *This
  );
```

Parameters

This

The **EFI_SMM_ACCESS2_PROTOCOL** instance.

Description

This function “closes” SMRAM so that it is not visible while outside of SMM. The function should return **EFI_UNSUPPORTED** if the hardware does not support hiding of SMRAM.

Status Codes Returned

EFI_SUCCESS	The operation was successful.
EFI_UNSUPPORTED	The system does not support opening and closing of SMRAM.
EFI_DEVICE_ERROR	SMRAM cannot be closed.

EFI_SMM_ACCESS2_PROTOCOL.Lock()

Summary

Inhibits access to the SMRAM.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMM_LOCK2) (
  IN EFI_SMM_ACCESS2_PROTOCOL *This
  );
```

Parameters

This

The **EFI_SMM_ACCESS2_PROTOCOL** instance.

Description

This function prohibits access to the SMRAM region. This function is usually implemented such that it is a write-once operation.

Status Codes Returned

EFI_SUCCESS	The device was successfully locked.
EFI_UNSUPPORTED	The system does not support locking of SMRAM.

EFI_SMM_ACCESS2_PROTOCOL.GetCapabilities()

Summary

Queries the memory controller for the regions that will support SMRAM.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMM_CAPABILITIES2) (
  IN CONST EFI_SMM_ACCESS2_PROTOCOL     *This,
  IN OUT UINTN                          *SmramMapSize,
  IN OUT EFI_SMRAM_DESCRIPTOR           *SmramMap
  );
```

Parameters

This

The **EFI_SMM_ACCESS2_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 describes the SMRAM regions.

This data structure forms the contract between the **SMM_ACCESS2** and **SMM_IPL** drivers. There is an ambiguity when any SMRAM region is remapped. For example, on some chipsets, some SMRAM regions can be initialized at one physical address but is later accessed at another processor address. There is currently no way for the SMM IPL 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 SMRAM is open. However, once the region has been closed 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.

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.

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 *UEFI 2.1 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*.

5.4 SMM Control Protocol

EFI_SMM_CONTROL2_PROTOCOL

Summary

This protocol is used initiate synchronous SMI activations. This protocol could be published by a processor driver to abstract the SMI IPI or a driver which abstracts the ASIC that is supporting the APM port.

Because of the possibility of performing SMI IPI transactions, the ability to generate this event from a platform chipset agent is an optional capability for both IA-32 and x64-based systems.

GUID

```
#define EFI_SMM_CONTROL2_PROTOCOL_GUID \
  { 0x843dc720, 0xab1e, 0x42cb, \
  0x93, 0x57, 0x8a, 0x0, 0x78, 0xf3, 0x56, 0x1b }
```

Protocol Interface Structure

```
typedef struct _EFI_SMM_CONTROL2_PROTOCOL {
  EFI_SMM_ACTIVATE2           Trigger;
  EFI_SMM_DEACTIVATE2         Clear;
  EFI_SMM_PERIOD              MinimumTriggerPeriod;
  } EFI_SMM_CONTROL2_PROTOCOL;
```

Parameters

Trigger

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

Clear

Quiesces the SMI activation. See the **Clear()** 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_CONTROL2_PROTOCOL** is produced by a runtime driver. It provides an abstraction of the platform hardware that generates an SMI. There are often I/O ports that, when accessed, will generate the SMI. Also, the hardware optionally supports the periodic generation of these signals.

Related Definitions

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

Note: *The period is in increments of 10 ns.*

EFI_SMM_CONTROL2_PROTOCOL.Trigger()

Summary

Invokes SMI activation from either the preboot or runtime environment.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMM_ACTIVATE2) (
  IN CONST EFI_SMM_CONTROL2_PROTOCOL *This,
  IN OUT UINT8                       *CommandPort      OPTIONAL,
  IN OUT UINT8                       *DataPort         OPTIONAL,
  IN BOOLEAN                         Periodic          OPTIONAL,
  IN UINTN                           ActivationInterval  OPTIONAL
  );
```

Parameters

This

The **EFI_SMM_CONTROL2_PROTOCOL** instance.

CommandPort

The value written to the command port; this value corresponds to the *SwSmiInputValue* in the *RegisterContext* parameter for the **Register()** function in the **EFI_SMM_SW_DISPATCH2_PROTOCOL** and in the *Context* parameter in the call to the **DispatchFunction**, see section 6.2.

DataPort

The value written to the data port; this value corresponds to the *DataPort* member in the *CommBuffer* parameter in the call to the **DispatchFunction**, see section 6.2.

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 generates an SMI.

Status Codes Returned

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

EFI_SMM_CONTROL2_PROTOCOL.Clear()

Summary

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

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMM_DEACTIVATE2) (
  IN CONST EFI_SMM_CONTROL2_PROTOCOL  *This,
  IN BOOLEAN                           Periodic OPTIONAL
  );
```

Parameters

This

The **EFI_SMM_CONTROL2_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 SMI activation source.that was initiated by a preceding *Trigger* invocation.

The results of this function update the software state of the communication infrastructure in the runtime code, but it is ignorable from the perspective of the hardware state, though. This distinction stems from the fact that many implementations clear the hardware acknowledge in the SMM-resident infrastructure itself and may also have other actions using that same activation hardware generated by SMM drivers. This clear-in-SMM distinction also avoids having the possible pathology of an asynchronous SMI being received in the time window between the RSM that followed the flows engendered by the *Trigger* and the subsequent non-SMM resident runtime driver code invocation of the *Clear*.

Status Codes Returned

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

5.5 SMM Configuration Protocol

EFI_SMM_CONFIGURATION_PROTOCOL

Summary

Reports the portions of SMRAM regions which cannot be used for the SMRAM heap.

GUID

```
#define EFI_SMM_CONFIGURATION_PROTOCOL_GUID \
  { 0x26eeb3de, 0xb689, 0x492e, \
  0x80, 0xf0, 0xbe, 0x8b, 0xd7, 0xda, 0x4b, 0xa7 }
```

Prototype

```
typedef struct _EFI_SMM_CONFIGURATION_PROTOCOL {
  EFI_SMM_RESERVED_SMRAM_REGION    *SmramReservedRegions;
  EFI_SMM_REGISTER_SMM_ENTRY       RegisterSmmEntry;
} EFI_SMM_CONFIGURATION_PROTOCOL;
```

Members

SmramReservedRegions

A pointer to an array SMRAM ranges used by the initial SMM entry code.

RegisterSmmEntry

A function to register the SMM Foundation entry point.

Description

This protocol is a mandatory protocol published by a DXE CPU driver to indicate which areas within SMRAM are reserved for use by the CPU for any purpose, such as stack, save state or SMM entry point.

The *SmramReservedRegions* points to an array of one or more **EFI_SMM_RESERVED_SMRAM_REGION** structures, with the last structure having the *SmramReservedSize* set to 0. An empty array would contain only the last structure.

The *RegisterSmmEntry()* function allows the SMM IPL DXE driver to register the SMM Foundation entry point with the SMM entry vector code.

Related Definitions

```
typedef struct _EFI_SMM_RESERVED_SMRAM_REGION {
  EFI_PHYSICAL_ADDRESS SmramReservedStart;
  UINT64               SmramReservedSize;
} EFI_SMM_RESERVED_SMRAM_REGION;
```

SmramReservedStart

Starting address of the reserved SMRAM area, as it appears while SMRAM is open. Ignored if *SmramReservedSize* is 0.

SmramReservedSize

Number of bytes occupied by the reserved SMRAM area. A size of zero indicates the last SMRAM area.

EFI_SMM_CONFIGURATION_PROTOCOL.RegisterSmmEntry()

Summary

Register the SMM Foundation entry point.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMM_REGISTER_SMM_ENTRY) (
  IN CONST EFI_SMM_CONFIGURATION_PROTOCOL   *This,
  IN EFI_SMM_ENTRY_POINT                    SmmEntryPoint
  )
```

Parameters

This

The **EFI_SMM_CONFIGURATION_PROTOCOL** instance.

SmmEntryPoint

SMM Foundation entry point.

Description

This function registers the SMM Foundation entry point with the processor code. This entry point will be invoked by the SMM Processor entry code as defined in section 2.5.

Status Codes Returned

EFI_SUCCESS	The entry-point was successfully registered.

5.6 DXE SMM Ready to Lock Protocol

EFI_DXE_SMM_READY_TO_LOCK_PROTOCOL

Summary

Indicates that resources and services that should not be used by the third party code are about to be locked.

GUID

```
#define EFI_DXE_SMM_READY_TO_LOCK_PROTOCOL_GUID \
 { 0x60ff8964, 0xe906, 0x41d0, \
 0xaf, 0xed, 0xf2, 0x41, 0xe9, 0x74, 0xe0, 0x8e}
```

Prototype

```
NULL
```

Description

This protocol is a mandatory protocol published by PI platform code.

This protocol in tandem with the *End of DXE Even*t facilitates transition of the platform from the environment where all of the components are under the authority of the platform manufacturer to the environment where third party extensible modules such as UEFI drivers and UEFI applications are executed.

The protocol is published immediately after signaling of the *End of DXE Event*.

PI modules that need to lock or protect their resources in anticipation of the invocation of 3rd party extensible modules should register for notification on installation of this protocol and effect the appropriate protections in their notification handlers. For example, PI platform code may choose to use notification handler to lock SMM by invoking **EFI_SMM_ACCESS2_PROTOCOL.Lock()** function.

5.7 SMM Communication Protocol

EFI_SMM_COMMUNICATION_PROTOCOL

Summary

This protocol provides a means of communicating between drivers outside of SMM and SMI handlers inside of SMM.

GUID

```
#define EFI_SMM_COMMUNICATION_PROTOCOL_GUID \
  { 0xc68ed8e2, 0x9dc6, 0x4cbd, 0x9d, 0x94, 0xdb, 0x65, \
  0xac, 0xc5, 0xc3, 0x32 }
```

Prototype

```
typedef struct _EFI_SMM_COMMUNICATION_PROTOCOL {
  EFI_SMM_COMMUNICATE2           Communicate;
} EFI_SMM_COMMUNICATION_PROTOCOL;
```

Members

Communicate

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

Description

This protocol provides runtime services for communicating between DXE drivers and a registered SMI handler.

EFI_SMM_COMMUNICATION_PROTOCOL.Communicate()

Summary

Communicates with a registered handler.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMM_COMMUNICATE2) (
  IN CONST EFI_SMM_COMMUNICATION_PROTOCOL   *This,
  IN OUT VOID                               *CommBuffer,
  IN OUT UINTN                              *CommSize
  );
```

Parameters

This

The **EFI_SMM_COMMUNICATION_PROTOCOL** instance.

CommBuffer

Pointer to the buffer to convey into SMRAM.

CommSize

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 UEFI service. The **EFI_SMM_COMMUNICATION_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_COMMUNICATION_PROTOCOL** may choose to use the **EFI_SMM_CONTROL2_PROTOCOL** for effecting the mode transition, or it may use some other method.

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 UEFI **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 may 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.

Once inside of SMM, the SMM infrastructure will call all registered handlers with the same *HandlerType* as the GUID specified by *HeaderGuid* and the *CommBuffer* pointing to *Data*.

This function is not reentrant.

Related Definitions

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

HeaderGuid

Allows for disambiguation of the message format. Type **EFI_GUID** is defined in **InstallProtocolInterface()** in the *UEFI 2.1 Specification.*

MessageLength

Describes the size of *Data* (in bytes) and does not include the size of 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**.

6
SMM Child Dispatch Protocols

6.1 Introduction

The services described in this chapter describe a series of protocols that abstract installation of handlers for a chipset-specific SMM design. These services are all scoped to be usable only from within SMRAM.

The following protocols are defined in this chapter:

- **EFI_SMM_SW_DISPATCH2_PROTOCOL**
- **EFI_SMM_SX_DISPATCH2_PROTOCOL**
- **EFI_SMM_PERIODIC_TIMER_DISPATCH2_PROTOCOL**
- **EFI_SMM_USB_DISPATCH2_PROTOCOL**
- **EFI_SMM_GPI_DISPATCH2_PROTOCOL**
- **EFI_SMM_STANDBY_BUTTON_DISPATCH2_PROTOCOL**
- **EFI_SMM_POWER_BUTTON_DISPATCH2_PROTOCOL**
- **EFI_SMM_IO_TRAP_DISPATCH2_PROTOCOL**

SMM drivers which create instances of these protocols should install an instance of the **EFI_DEVICE_PATH_PROTOCOL** on the same handle. This allows other SMM drivers to distinguish between multiple instances of the same child dispatch protocol

6.2 SMM Software Dispatch Protocol

EFI_SMM_SW_DISPATCH2_PROTOCOL

Summary

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

GUID

```
#define EFI_SMM_SW_DISPATCH2_PROTOCOL_GUID    \
{ 0x18a3c6dc, 0x5eea, 0x48c8, \
0xa1, 0xc1, 0xb5, 0x33, 0x89, 0xf9, 0x89, 0x99}
```

Protocol Interface Structure

```
typedef struct _EFI_SMM_SW_DISPATCH2_PROTOCOL {
  EFI_SMM_SW_REGISTER2     Register;
  EFI_SMM_SW_UNREGISTER2   UnRegister;
  UINTN                    MaximumSwiValue;
} EFI_SMM_SW_DISPATCH2_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_DISPATCH2_PROTOCOL.Register()** service.

Description

The **EFI_SMM_SW_DISPATCH2_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_REGISTER_CONTEXT** is denoted by *MaximumSwiValue*.

EFI_SMM_SW_DISPATCH2_PROTOCOL.Register()

Summary

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

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMM_SW_REGISTER2) (
  IN  CONST EFI_SMM_SW_DISPATCH2_PROTOCOL  *This,
  IN  EFI_SMM_HANDLER_ENTRY_POINT2         DispatchFunction,
  IN  EFI_SMM_SW_REGISTER_CONTEXT          *RegisterContext,
  OUT EFI_HANDLE                           *DispatchHandle
  );
```

Parameters

This

Pointer to the **EFI_SMM_SW_DISPATCH2_PROTOCOL** instance.

DispatchFunction

Function to register for handler when the specified software SMI is generated. Type **EFI_SMM_HANDLER_ENTRY_POINT2** is defined in "Related Definitions" in **SmiHandlerRegister()**.

RegisterContext

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_REGISTER_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 *UEFI 2.1 Specification.*

Description

This service registers a function (*DispatchFunction*) which will be called when the software SMI source specified by *RegisterContext->SwSmiCpuIndex* is detected. On return, *DispatchHandle* contains a unique handle which may be used later to unregister the function using UnRegister().

If *SwSmiInputValue* is set to **(UINTN) -1** then a unique value will be assigned and returned in the structure. If no unique value can be assigned then **EFI_OUT_OF_RESOURCES** will be returned.

The *DispatchFunction* will be called with *Context* set to the same value as was passed into this function in *RegisterContext* and with *CommBuffer* (and *CommBufferSize*) pointing

to an instance of **EFI_SMM_SW_CONTEXT** indicating the index of the CPU which generated the software SMI.

Related Definitions

```
//*******************************************************
// EFI_SMM_SW_CONTEXT
//*******************************************************
typedef struct {
 UINTN       SwSmiCpuIndex;
 UINT8       CommandPort;
 UINT8       DataPort;
} EFI_SMM_SW_CONTEXT;
```

SwSmiCpuIndex

The 0-based index of the CPU which generated the software SMI.

CommandPort

This value corresponds directly to the *CommandPort* parameter used in the call to **Trigger()**, see section 5.4.

DataPort

This value corresponds directly to the *DataPort* parameter used in the call to **Trigger()**, see section 5.4.

```
//*******************************************************
// EFI_SMM_SW_REGISTER_CONTEXT
//*******************************************************
typedef struct {
  UINTN      SwSmiInputValue;
} EFI_SMM_SW_REGISTER_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	*RegisterContext* is invalid. The SW SMI input value is not within a valid range or is already in use.
EFI_OUT_OF_RESOURCES	There is not enough memory (system or SMM) to manage this child.
EFI_OUT_OF_RESOURCES	A unique software SMI value could not be assigned for this dispatch.

EFI_SMM_SW_DISPATCH2_PROTOCOL.UnRegister()

Summary

Unregisters a software service.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMM_SW_UNREGISTER2) (
  IN  CONST EFI_SMM_SW_DISPATCH2_PROTOCOL  *This,
  IN  EFI_HANDLE                           DispatchHandle
  );
```

Parameters

This

Pointer to the **EFI_SMM_SW_DISPATCH2_PROTOCOL** instance.

DispatchHandle

Handle of the service to remove. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *UEFI 2.1 Specification.*

Description

This service removes the handler associated with *DispatchHandle* so that it will no longer be called in response to a software SMI.

Status Codes Returned

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

6.3 SMM Sx Dispatch Protocol

EFI_SMM_SX_DISPATCH2_PROTOCOL

Summary

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

GUID

```
#define EFI_SMM_SX_DISPATCH2_PROTOCOL_GUID    \
{ 0x456d2859, 0xa84b, 0x4e47, \
0xa2, 0xee, 0x32, 0x76, 0xd8, 0x86, 0x99, 0x7d }
```

Protocol Interface Structure

```
typedef struct _EFI_SMM_SX_DISPATCH2_PROTOCOL {
```

```
  EFI_SMM_SX_REGISTER2    Register;
  EFI_SMM_SX_UNREGISTER2  UnRegister;
} EFI_SMM_SX_DISPATCH2_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_DISPATCH2_PROTOCOL** provides the ability to install child handlers to respond to sleep state related events.

EFI_SMM_SX_DISPATCH2_PROTOCOL.Register()

Summary

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

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMM_SX_REGISTER2) (
  IN  CONST EFI_SMM_SX_DISPATCH2_PROTOCOL     *This,
  IN  EFI_SMM_HANDLER_ENTRY_POINT2            DispatchFunction,
  IN  CONST EFI_SMM_SX_REGISTER_CONTEXT       *RegisterContext,
  OUT EFI_HANDLE                              *DispatchHandle
  );
```

Parameters

This

Pointer to the **EFI_SMM_SX_DISPATCH2_PROTOCOL** instance.

DispatchFunction

Function to register for handler when the specified sleep state event occurs. Type **EFI_SMM_HANDLER_ENTRY_POINT2** is defined in "Related Definitions" in **SmiHandlerRegister()** in the SMST.

RegisterContext

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_REGISTER_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 *UEFI 2.1 Specification.*

Description

This service registers a function (*DispatchFunction*) which will be called when the sleep state event specified by *RegisterContext* is detected. On return, *DispatchHandle* contains a unique handle which may be used later to unregister the function using **UnRegister()**.

The *DispatchFunction* will be called with *Context* set to the same value as was passed into this function in *RegisterContext* and with *CommBuffer* and *CommBufferSize* set to NULL and 0 respectively.

Related Definitions

```
//*******************************************************
// EFI_SMM_SX_REGISTER_CONTEXT
//*******************************************************
typedef struct {
  EFI_SLEEP_TYPE    Type;
  EFI_SLEEP_PHASE   Phase;
} EFI_SMM_SX_REGISTER_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	*RegisterContext* 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_DISPATCH2_PROTOCOL.UnRegister()

Summary

Unregisters an Sx-state service.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMM_SX_UNREGISTER2) (
  IN  CONST EFI_SMM_SX_DISPATCH2_PROTOCOL  *This,
  IN  EFI_HANDLE                           DispatchHandle
  );
```

Parameters

This

Pointer to the **EFI_SMM_SX_DISPATCH2_PROTOCOL** instance.

DispatchHandle

Handle of the service to remove. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *UEFI 2.1 Specification.*

Description

This service removes the handler associated with *DispatchHandle* so that it will no longer be called in response to sleep event.

Status Codes Returned

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

6.4 SMM Periodic Timer Dispatch Protocol

EFI_SMM_PERIODIC_TIMER_DISPATCH2_PROTOCOL

Summary

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

GUID

```
#define EFI_SMM_PERIODIC_TIMER_DISPATCH2_PROTOCOL_GUID    \
  { 0x4cec368e, 0x8e8e, 0x4d71, \
  0x8b, 0xe1, 0x95, 0x8c, 0x45, 0xfc, 0x8a, 0x53}
```

Protocol Interface Structure

```
typedef struct _EFI_SMM_PERIODIC_TIMER_DISPATCH2_PROTOCOL {
```

```
  EFI_SMM_PERIODIC_TIMER_REGISTER2     Register;
  EFI_SMM_PERIODIC_TIMER_UNREGISTER2   UnRegister;
  EFI_SMM_PERIODIC_TIMER_INTERVAL2     GetNextShorterInterval;
} EFI_SMM_PERIODIC_TIMER_DISPATCH2_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_DISPATCH2_PROTOCOL** provides the ability to install child handlers for the given event types.

EFI_SMM_PERIODIC_TIMER_DISPATCH2_PROTOCOL.Register()

Summary

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

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMM_PERIODIC_TIMER_REGISTER2) (
  IN  CONST EFI_SMM_PERIODIC_TIMER_DISPATCH2_PROTOCOL *This,
  IN  EFI_SMM_HANDLER_ENTRY_POINT2              DispatchFunction,
  IN  CONST EFI_SMM_PERIODIC_TIMER_REGISTER_CONTEXT
*RegisterContext,
  OUT EFI_HANDLE                                *DispatchHandle
  );
```

Parameters

This

Pointer to the **EFI_SMM_PERIODIC_TIMER_DISPATCH2_PROTOCOL** instance.

DispatchFunction

Function to register for handler when at least the specified amount of time has elapsed. Type **EFI_SMM_HANDLER_ENTRY_POINT2** is defined in "Related Definitions" in **SmiHandlerRegister()** in the SMST.

RegisterContext

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_REGISTER_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 *UEFI 2.1 Specification.*

Description

This service registers a function (*DispatchFunction*) which will be called when at least the amount of time specified by *RegisterContext* has elapsed. On return, *DispatchHandle* contains a unique handle which may be used later to unregister the function using **UnRegister()**.

The *DispatchFunction* will be called with *Context* set to the same value as was passed into this function in *RegisterContext* and with *CommBuffer* pointing to an instance of **EFI_SMM_PERIODIC_TIMER_CONTEXT** and *CommBufferSize* pointing to its size.

Related Definitions

```
//*******************************************************
// EFI_SMM_PERIODIC_TIMER_REGISTER_CONTEXT
//*******************************************************

typedef struct {
  UINT64    Period;
  UINT64    SmiTickInterval;
} EFI_SMM_PERIODIC_TIMER_REGISTER_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.*

```
typedef struct _EFI_SMM_PERIODIC_TIMER_CONTEXT {
 UINT64              ElapsedTime;
} EFI_SMM_PERIODIC_TIMER_CONTEXT;
```

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	*RegisterContext* 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_DISPATCH2_PROTOCOL.UnRegister()

Summary

Unregisters a periodic timer service.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMM_PERIODIC_TIMER_UNREGISTER2) (
  IN CONST EFI_SMM_PERIODIC_TIMER_DISPATCH2_PROTOCOL  *This,
  IN EFI_HANDLE                                       DispatchHandle
  );
```

Parameters

This

Pointer to the **EFI_SMM_PERIODIC_TIMER_DISPATCH2_PROTOCOL** instance.

DispatchHandle

Handle of the service to remove. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *UEFI 2.1 Specification.*

Description

This service removes the handler associated with *DispatchHandle* so that it will no longer be called when the time has elapsed.

Status Codes Returned

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

EFI_SMM_PERIODIC_TIMER_DISPATCH2_PROTOCOL. GetNextShorterInterval()

Summary

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

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMM_PERIODIC_TIMER_INTERVAL2) (
  IN     CONST EFI_SMM_PERIODIC_TIMER_DISPATCH2_PROTOCOL  *This,
  IN OUT UINT64                                          **SmiTickInterval
  );
```

Parameters

This

Pointer to the **EFI_SMM_PERIODIC_TIMER_DISPATCH2_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 service returns the next SMI tick period that is supported by the device. The order returned is from longest to shortest interval period.

Status Codes Returned

EFI_SUCCESS	The service returned successfully.

6.5 SMM USB Dispatch Protocol

EFI_SMM_USB_DISPATCH2_PROTOCOL

Summary

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

GUID

```
#define EFI_SMM_USB_DISPATCH2_PROTOCOL_GUID    \
  { 0xee9b8d90, 0xc5a6, 0x40a2, \
  0xbd, 0xe2, 0x52, 0x55, 0x8d, 0x33, 0xcc, 0xa1 }
```

Protocol Interface Structure

```
typedef struct _EFI_SMM_USB_DISPATCH2_PROTOCOL {
  EFI_SMM_USB_REGISTER2     Register;
  EFI_SMM_USB_UNREGISTER2   UnRegister;
} EFI_SMM_USB_DISPATCH2_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_DISPATCH2_PROTOCOL** provides the ability to install child handlers for the given event types.

EFI_SMM_USB_DISPATCH2_PROTOCOL.Register()

Summary

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

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMM_USB_REGISTER2) (
  IN  CONST EFI_SMM_USB_DISPATCH2_PROTOCOL    *This,
  IN  EFI_SMM_HANDLER_ENTRY_POINT2            DispatchFunction,
  IN  CONST EFI_SMM_USB_REGISTER_CONTEXT      *RegisterContext,
  OUT EFI_HANDLE                              *DispatchHandle
  );
```

Parameters

This

Pointer to the **EFI_SMM_USB_DISPATCH2_PROTOCOL** instance.

DispatchFunction

Function to register for handler when a USB-related SMI occurs. Type **EFI_SMM_HANDLER_ENTRY_POINT2** is defined in "Related Definitions" in **SmiHandlerRegister()** in the SMST.

RegisterContext

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_REGISTER_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 *UEFI 2.1 Specification.*

Description

This service registers a function (*DispatchFunction*) which will be called when the USB-related SMI specified by *RegisterContext* has occurred. On return, *DispatchHandle* contains a unique handle which may be used later to unregister the function using **UnRegister()**.

The *DispatchFunction* will be called with *Context* set to the same value as was passed into this function in *RegisterContext* and with *CommBuffer* containing NULL and *CommBufferSize* containing zero.

Related Definitions

```
//*******************************************************
// EFI_SMM_USB_REGISTER_CONTEXT
```

```
//*******************************************************

typedef struct {
  EFI_USB_SMI_TYPE            Type;
  EFI_DEVICE_PATH_PROTOCOL    *Device;
} EFI_SMM_USB_REGISTER_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 *UEFI 2.1 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	*RegisterContext* 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_DISPATCH2_PROTOCOL.UnRegister()

Summary

Unregisters a USB service.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMM_USB_UNREGISTER2) (
  IN  CONST EFI_SMM_USB_DISPATCH2_PROTOCOL     *This,
  IN EFI_HANDLE                                 DispatchHandle
  );
```

Parameters

This

Pointer to the **EFI_SMM_USB_DISPATCH2_PROTOCOL** instance.

DispatchHandle

Handle of the service to remove. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *UEFI 2.1 Specification.*

Description

This service removes the handler associated with *DispatchHandle* so that it will no longer be called when the USB event occurs. .

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.

6.6 SMM General Purpose Input (GPI) Dispatch Protocol

EFI_SMM_GPI_DISPATCH2_PROTOCOL

Summary

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

GUID

```
#define EFI_SMM_GPI_DISPATCH2_PROTOCOL_GUID    \
{ 0x25566b03, 0xb577, 0x4cbf, \
0x95, 0x8c, 0xed, 0x66, 0x3e, 0xa2, 0x43, 0x80 }
```

Protocol Interface Structure

```
typedef struct _EFI_SMM_GPI_DISPATCH2_PROTOCOL {
  EFI_SMM_GPI_REGISTER2     Register;
  EFI_SMM_GPI_UNREGISTER2   UnRegister;
  UINTN                     NumSupportedGpis;
} EFI_SMM_GPI_DISPATCH2_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_DISPATCH2_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_DISPATCH2_PROTOCOL.Register()

Summary

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

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMM_GPI_REGISTER2) (
  IN  CONST EFI_SMM_GPI_DISPATCH2_PROTOCOL    *This,
  IN  EFI_SMM_HANDLER_ENTRY_POINT2            DispatchFunction,
  IN  CONST EFI_SMM_GPI_REGISTER_CONTEXT      *RegisterContext,
  OUT EFI_HANDLE                              *DispatchHandle
  );
```

Parameters

This

Pointer to the **EFI_SMM_GPI_DISPATCH2_PROTOCOL** instance.

DispatchFunction

Function to register for handler when the specified GPI causes an SMI. Type **EFI_SMM_HANDLER_ENTRY_POINT2** is defined in "Related Definitions" in **SmiHandlerRegister()** in the SMST.

RegisterContext

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_REGISTER_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 *UEFI 2.1 Specification.*

Description

This service registers a function (*DispatchFunction*) which will be called when an SMI is generated because of one or more of the GPIs specified by *RegisterContext*. On return, *DispatchHandle* contains a unique handle which may be used later to unregister the function using **UnRegister()**.

The *DispatchFunction* will be called with *Context* set to the same value as was passed into this function in *RegisterContext* and with *CommBuffer* pointing to another instance of **EFI_SMM_GPI_REGISTER_CONTEXT** describing the GPIs which actually caused the SMI and *CommBufferSize* pointing to the size of the structure.

Related Definitions

```
//*****************************************
// EFI_SMM_GPI_REGISTER_CONTEXT
//*****************************************

typedef struct {
  UINT64      GpiNum;
} EFI_SMM_GPI_REGISTER_CONTEXT;
```

GpiNum

A number from one of 2^64 possible GPIs that can generate an SMI. A 0 corresponds to logical GPI[0]; 1 corresponds to logical GPI[1]; and *GpiNum* of N corresponds to GPI[N], where N can span from 0 to 2^64-1.

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	*RegisterContext* 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_DISPATCH2_PROTOCOL.UnRegister()

Summary

Unregisters a General Purpose Input (GPI) service.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMM_GPI_UNREGISTER2) (
  IN  CONST EFI_SMM_GPI_DISPATCH2_PROTOCOL     *This,
  IN EFI_HANDLE                                 DispatchHandle
  );
```

Parameters

This

Pointer to the **EFI_SMM_GPI_DISPATCH2_PROTOCOL** instance.

DispatchHandle

Handle of the service to remove. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *UEFI 2.1 Specification.*

Description

This service removes the handler associated with *DispatchHandle* so that it will no longer be called when the GPI triggers an SMI.

Status Codes Returned

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

6.7 SMM Standby Button Dispatch Protocol

EFI_SMM_STANDBY_BUTTON_DISPATCH2_PROTOCOL

Summary

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

GUID

```
#define EFI_SMM_STANDBY_BUTTON_DISPATCH2_PROTOCOL_GUID    \
  { 0x7300c4a1, 0x43f2, 0x4017, \
  0xa5, 0x1b, 0xc8, 0x1a, 0x7f, 0x40, 0x58, 0x5b }
```

Protocol Interface Structure

```
typedef struct _EFI_SMM_STANDBY_BUTTON_DISPATCH2_PROTOCOL {
```

```
  EFI_SMM_STANDBY_BUTTON_REGISTER2    Register;
  EFI_SMM_STANDBY_BUTTON_UNREGISTER2  UnRegister;
} EFI_SMM_STANDBY_BUTTON_DISPATCH2_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_DISPATCH2_PROTOCOL** provides the ability to install child handlers for the given event types.

EFI_SMM_STANDBY_BUTTON_DISPATCH2_PROTOCOL.Register()

Summary

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

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMM_STANDBY_BUTTON_REGISTER2) (
  IN  CONST EFI_SMM_STANDBY_BUTTON_DISPATCH2_PROTOCOL  *This,
  IN  EFI_SMM_HANDLER_ENTRY_POINT2                     DispatchFunction,
  IN  EFI_SMM_STANDBY_BUTTON_REGISTER_CONTEXT          *RegisterContext,
  OUT EFI_HANDLE                                       *DispatchHandle
  );
```

Parameters

This

Pointer to the **EFI_SMM_STANDBY_BUTTON_DISPATCH2_PROTOCOL** instance.

DispatchFunction

Function to register for handler when the standby button is pressed or released. Type **EFI_SMM_HANDLER_ENTRY_POINT2** is defined in "Related Definitions" in **SmiHandlerRegister()** in the SMST.

RegisterContext

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_REGISTER_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 *UEFI 2.1 Specification.*

Description

This service registers a function (*DispatchFunction*) which will be called when an SMI is generated because the standby button was pressed or released, as specified by *RegisterContext*. On return, *DispatchHandle* contains a unique handle which may be used later to unregister the function using **UnRegister()**.

The *DispatchFunction* will be called with *Context* set to the same value as was passed into this function in *RegisterContext* and with *CommBuffer* and *CommBufferSize* set to **NULL**.

Related Definitions

```
//*****************************************************
// EFI_SMM_STANDBY_BUTTON_REGISTER_CONTEXT
//*****************************************************
typedef struct {
  EFI_STANDBY_BUTTON_PHASE  Phase;
} EFI_SMM_STANDBY_BUTTON_REGISTER_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 {
  EfiStandbyButtonEntry,
  EfiStandbyButtonExit,
  EfiStandbyButtonMax
} EFI_STANDBY_BUTTON_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	*RegisterContext* 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_DISPATCH2_PROTOCOL.UnRegister()

Summary

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

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMM_STANDBY_BUTTON_UNREGISTER2) (
  IN CONST EFI_SMM_STANDBY_BUTTON_DISPATCH2_PROTOCOL *This,
  IN EFI_HANDLE                                      *DispatchHandle
  );
```

Parameters

This

Pointer to the **EFI_SMM_STANDBY_BUTTON_DISPATCH2_PROTOCOL** instance.

DispatchHandle

Handle of the service to remove. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *UEFI 2.1 Specification.*

Description

This service removes the handler associated with *DispatchHandle* so that it will no longer be called when the standby button is pressed or released.

Status Codes Returned

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

6.8 SMM Power Button Dispatch Protocol

EFI_SMM_POWER_BUTTON_DISPATCH2_PROTOCOL

Summary

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

GUID

```
#define EFI_SMM_POWER_BUTTON_DISPATCH2_PROTOCOL_GUID    \
  { 0x1b1183fa, 0x1823, 0x46a7, \
  0x88, 0x72, 0x9c, 0x57, 0x87, 0x55, 0x40, 0x9d }
```

Protocol Interface Structure

```
typedef struct _EFI_SMM_POWER_BUTTON_DISPATCH2_PROTOCOL {
```

```
  EFI_SMM_POWER_BUTTON_REGISTER2     Register;
  EFI_SMM_POWER_BUTTON_UNREGISTER2   UnRegister;
} EFI_SMM_POWER_BUTTON_DISPATCH2_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_DISPATCH2_PROTOCOL** provides the ability to install child handlers for the given event types.

EFI_SMM_POWER_BUTTON_DISPATCH2_PROTOCOL. Register()

Summary

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

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMM_POWER_BUTTON_REGISTER2) (
  IN  CONST EFI_SMM_POWER_BUTTON_DISPATCH2_PROTOCOL  *This,
  IN  EFI_SMM_HANDLER_ENTRY_POINT2                   DispatchFunction,
  IN  EFI_SMM_POWER_BUTTON_REGISTER_CONTEXT          *RegisterContext,
  OUT EFI_HANDLE                                     *DispatchHandle
  );
```

Parameters

This

Pointer to the **EFI_SMM_POWER_BUTTON_DISPATCH2_PROTOCOL** instance.

DispatchFunction

Function to register for handler when power button is pressed or released. Type **EFI_SMM_HANDLER_ENTRY_POINT2** is defined in "Related Definitions" in **SmiHandlerRegister()** in the SMST.

RegisterContext

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_REGISTER_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 *UEFI 2.1 Specification.*

Description

This service registers a function (*DispatchFunction*) which will be called when an SMI is generated because the power button was pressed or released, as specified by *RegisterContext*. On return, *DispatchHandle* contains a unique handle which may be used later to unregister the function using **UnRegister()**.

The *DispatchFunction* will be called with *Context* set to the same value as was passed into this function in *RegisterContext* and with *CommBuffer* and *CommBufferSize* set to **NULL**.

Related Definitions

```
//*******************************************************
// EFI_SMM_POWER_BUTTON_REGISTER_CONTEXT
//*******************************************************
typedef struct {
  EFI_POWER_BUTTON_PHASE  Phase;
} EFI_SMM_POWER_BUTTON_REGISTER_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 {
  EfiPowerButtonEntry,
EfiPowerButtonExit,
EfiPowerButtonMax
} EFI_POWER_BUTTON_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	*RegisterContext* 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_DISPATCH2_PROTOCOL.UnRegister()

Summary

Unregisters a power-button service.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMM_POWER_BUTTON_UNREGISTER2) (
  IN  CONST EFI_SMM_POWER_BUTTON_DISPATCH2_PROTOCOL  *This,
  IN  EFI_HANDLE                                     DispatchHandle
  );
```

Parameters

This

Pointer to the **EFI_SMM_POWER_BUTTON_DISPATCH2_PROTOCOL** instance.

DispatchHandle

Handle of the service to remove. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *UEFI 2.1 Specification.*

Description

This service removes the handler associated with *DispatchHandle* so that it will no longer be called when the standby button is pressed or released.

Status Codes Returned

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

6.9 SMM IO Trap Dispatch Protocol

EFI_SMM_IO_TRAP_DISPATCH2_PROTOCOL

Summary

This protocol provides a parent dispatch service for IO trap SMI sources.

GUID

```
#define EFI_SMM_IO_TRAP_DISPATCH2_PROTOCOL_GUID \
  { 0x58dc368d, 0x7bfa, 0x4e77, \
  0xab, 0xbc, 0xe, 0x29, 0x41, 0x8d, 0xf9, 0x30 }
```

Protocol Interface Structure

```
struct _EFI_SMM_IO_TRAP_DISPATCH2_PROTOCOL {
  EFI_SMM_IO_TRAP_DISPATCH2_REGISTER              Register;
  EFI_SMM_IO_TRAP_DISPATCH2_UNREGISTER            UnRegister;
} EFI_SMM_IO_TRAP_DISPATCH2_PROTOCOL;
```

Parameters

Register

Installs a child service to be dispatched when the requested IO trap SMI occurs. See the **Register()** function description.

UnRegister

Removes a previously registered child service. See the *Register()* and **UnRegister()** function descriptions.

Description

This protocol provides the ability to install child handlers for IO trap SMI. These handlers will be invoked to respond to specific IO trap SMI. IO trap SMI would typically be generated on reads or writes to specific processor IO space addresses or ranges. This protocol will typically abstract a limited hardware resource, so callers should handle errors gracefully.

EFI_SMM_IO_TRAP_DISPATCH2_PROTOCOL.Register ()

Summary

Register an IO trap SMI child handler for a specified SMI.

Prototype

```
EFI_STATUS
(EFIAPI *EFI_SMM_IO_TRAP_DISPATCH2_REGISTER) (
  IN     CONST EFI_SMM_IO_TRAP_DISPATCH2_PROTOCOL      *This,
  IN     EFI_SMM_HANDLER_ENTRY_POINT2           DispatchFunction,
  IN OUT EFI_SMM_IO_TRAP_REGISTER_CONTEXT       *RegisterContext,
  OUT EFI_HANDLE                               *DispatchHandle
  );
```

Parameters

This

Pointer to the **EFI_SMM_IO_TRAP_DISPATCH2_PROTOCOL** instance.

DispatchFunction

Function to register for handler when I/O trap location is accessed. Type **EFI_SMM_HANDLER_ENTRY_POINT2** is defined in "Related Definitions" in **SmiHandlerRegister()** in the SMST.

RegisterContext

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 IO trap SMI source for which the dispatch function should be invoked.

DispatchHandle

Handle of the dispatch function, for when interfacing with the parent SMM driver. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *UEFI 2.1 Specification*.

Description

This service registers a function (*DispatchFunction*) which will be called when an SMI is generated because of an access to an I/O port specified by *RegisterContext*. On return, *DispatchHandle* contains a unique handle which may be used later to unregister the function using **UnRegister()**. If the base of the I/O range specified is zero, then an I/O range with the specified length and characteristics will be allocated and the Address field in *RegisterContext* updated. If no range could be allocated, then **EFI_OUT_OF_RESOURCES** will be returned.

The service will not perform GCD allocation if the base address is non-zero or EFI_SMM_READY_TO_LOCK has been installed. In this case, the caller is responsible for the existence and allocation of the specific IO range.

An error may be returned if some or all of the requested resources conflict with an existing IO trap child handler.

It is not required that implementations will allow multiple children for a single IO trap SMI source. Some implementations may support multiple children.

The *DispatchFunction* will be called with *Context* updated to contain information concerning the I/O action that actually happened and is passed in *RegisterContext,* with *CommBuffer* pointing to the data actually written and *CommBufferSize* pointing to the size of the data in *CommBuffer*.

Related Definitions

```
//
// IO Trap valid types
//
typedef enum {
  WriteTrap,
  ReadTrap,
  ReadWriteTrap,
  IoTrapTypeMaximum
} EFI_SMM_IO_TRAP_DISPATCH_TYPE;

//
// IO Trap context structure containing information about the
// IO trap event that should invoke the handler
//
typedef struct {
  UINT16                                Address;
  UINT16                                Length;
  EFI_SMM_IO_TRAP_DISPATCH_TYPE         Type;
} EFI_SMM_IO_TRAP_REGISTER_CONTEXT;

//
// IO Trap context structure containing information about the IO
trap that occurred
//
typedef struct {
  UINT32                                WriteData;
} EFI_SMM_IO_TRAP_CONTEXT;
```

Status Codes Returned

EFI_SUCCESS	The dispatch function has been successfully registered.
EFI_DEVICE_ERROR	The driver was unable to complete due to hardware error.
EFI_OUT_OF_RESOURCES	Insufficient resources are available to fulfill the IO trap range request.
EFI_INVALID_PARAMETER	*RegisterContext* is invalid. The input value is not within a valid range.

EFI_SMM_IO_TRAP_DISPATCH2_PROTOCOL.UnRegister ()

Summary

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

Prototype

```
EFI_STATUS
(EFIAPI *EFI_SMM_IO_TRAP_DISPATCH2_UNREGISTER) (
  IN CONST EFI_SMM_IO_TRAP_DISPATCH2_PROTOCOL    *This,
  IN EFI_HANDLE                                  DispatchHandle
  );
```

Parameters

This

Pointer to the **EFI_SMM_IO_TRAP_DISPATCH2_PROTOCOL** instance.

DispatchHandle

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

Description

This service removes a previously installed child dispatch handler. This does not guarantee that the system resources will be freed from the GCD.

Related Definitions

None

Status Codes Returned

EFI_SUCCESS	The dispatch function has been successfully unregistered.
EFI_INVALID_PARAMETER	The *DispatchHandle* was not valid.

7
Interactions with PEI, DXE, and BDS

7.1 Introduction

This chapter describes issues related to image verification and interactions between SMM and other PI Architecture phases.

7.2 SMM and DXE

7.2.1 Software SMI Communication Interface (Method #1)

During the boot service phase of DXE/UEFI, 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/UEFI 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 UEFI 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 UEFI 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 UEFI 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 UEFI Runtime—both trying to write to flash at the same time, the runtime implementation of the **`SetVariable()`** UEFI call would simply be an invocation of the **`EFI_SMM_COMMUNICATION_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_COMMUNICATION_PROTOCOL.Communicate()`** service for more information on this interface.

7.2.2 Software SMI Communication Interface (Method #2)

This section describes an alternative mechanism that can be used to initiate inter-mode communication. This mechanism can be used in the OS present environment by non-firmware agents. Inter-mode communication can be initiated using special software SMI.

Details regarding the SMI are described in the SMM Communication ACPI Table. This table is described in Appendix O of the *UEFI Specification*.

Firmware processes this software SMI in the same manner it processes direct invocation of the **`Communicate()`** function.

8
Other Related Notes For Support Of SMM Drivers

8.1 File Types

The following new file type is added:

```
#define EFI_FV_FILETYPE_SMM  0x0A
#define EFI_FV_FILETYPE_COMBINED_SMM_DXE  0x0C
```

8.1.1 File Type EFI_FV_FILETYPE_SMM

The file type **EFI_FV_FILETYPE_SMM** denotes a file that contains a PE32+ image that will be loaded into SMRAM.

This file type is a sectioned file that must be constructed in accordance with the following rules:

• The file must contain at least one **EFI_SECTION_PE32** section. There are no restrictions on encapsulation of this section.

• The file must contain no more than one **EFI_SECTION_VERSION** section.

• The file must contain no more than one **EFI_SECTION_SMM_DEPEX** section.

There are no restrictions on the encapsulation of the leaf sections. In the event that more than one **EFI_SECTION_PE32** section is present in the file, the selection algorithm for choosing which one represents the DXE driver that will be dispatched is defined by the **LoadImage()** boot service, which is used by the DXE Dispatcher. See the *Platform Initialization Specification, Volume 2* for details. The file may contain other leaf and encapsulation sections as required or enabled by the platform design.

8.1.2 File Type EFI_FV_FILETYPE_COMBINED_SMM_DXE

The file type **EFI_FV_FILETYPE_COMBINED_SMM_DXE** denotes a file that contains a PE32+ image that will be dispatched by the DXE Dispatcher and will also be loaded into SMRAM.

This file type is a sectioned file that must be constructed in accordance with the following rules:

- The file must contain at least one **EFI_SECTION_PE32** section. There are no restrictions on encapsulation of this section.
- The file must contain no more than one **EFI_SECTION_VERSION** section.
- The file must contain no more than one **EFI_SECTION_DXE_DEPEX** section. This section is ignored when the file is loaded into SMRAM.
- The file must contain no more than one **EFI_SECTION_SMM_DEPEX** section. This section is ignored when the file is dispatched by the DXE Dispatcher.

There are no restrictions on the encapsulation of the leaf sections. In the event that more than one **EFI_SECTION_PE32** section is present in the file, the selection algorithm for choosing which one represents the DXE driver that will be dispatched is defined by the **LoadImage()** boot service, which is used by the DXE Dispatcher. See the *Platform Initialization Specification, Volume 2* for

details. The file may contain other leaf and encapsulation sections as required or enabled by the platform design.

8.2 File Section Types

The following new section type must be added:

```
#define EFI_SECTION_SMM_DEPEX 0x1c
```

8.2.1 File Section Type EFI_SECTION_SMM_DEPEX

Summary

A leaf section type that is used to determine the dispatch order for an SMM driver.

Prototype

```
typedef EFI_COMMON_SECTION_HEADER EFI_SMM_DEPEX_SECTION;
```

Description

The *SMM dependency expression section* is a leaf section that contains a dependency expression that is used to determine the dispatch order for SMM drivers. Before the SMRAM invocation of the SMM driver's entry point, this dependency expression must evaluate to TRUE. See the *Platform Initialization Specification, Volume 2*for details regarding the format of the dependency expression.

The dependency expression may refer to protocols installed in either the UEFI or the SMM protocol database.

9
MCA/INIT/PMI Protocol

This document defines the basic plumbing required to run the MCA, PMI & INIT in a generic framework. They have been group together since MCA and INIT follows a very similar flow and all three have access to the min-state as defined by PAL.

It makes an attempt to bind the platform knowledge by the way of generic abstraction to the SAL MCA, PMI & INIT code. We have tried to create a private & public data structures for each CPU. For example, any CPU knowledge that should remain within the context of that CPU should be private. Any CPU knowledge that may be accessed by another CPU should be a Global Structure that can be accessed by any CPU for that domain. There are some flags that may be required globally (Sal Proc, Runtime Services, PMI, INIT, MCA) are made accessible through a protocol pointer that is described in section 5.

9.1 Machine Check and INIT

This section describes how Machine Check Abort Interrupt and INIT are handled in a UEFI 2.0 compliant system.

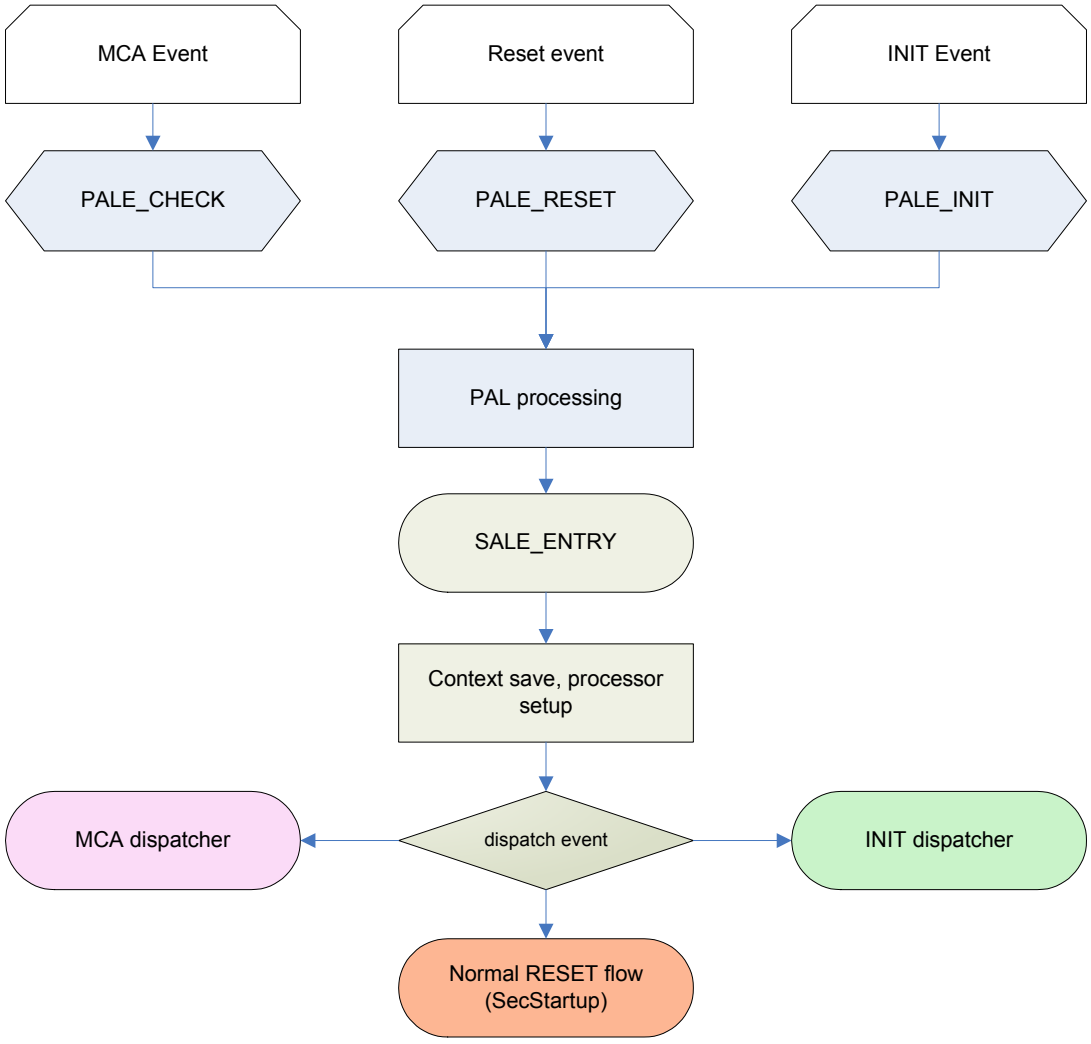

Figure 5. Early Reset, MCA and INIT flow

As shown in Figure 5 resets, MCA and INIT follow a near identical early flow. For all three events, PAL first processes the event, save some states if needed in the min-state before jumping to SAL through the common SALE_ENTRY entry point. SAL performs some early processor initialization, save some extra states to set up an environment in which the event can be handled and then branch to the appropriate event dispatcher (normal reset flow, MCA, INIT).

MCA/INIT handling per say consists of a generic dispatcher and one or more platform specific handlers. The dispatcher is responsible for handling tasks specified in SAL specification, such as performing rendezvous, before calling the event handlers in a fixed order. The handlers are responsible for error logging, error correction and any other platform specific task required to properly handle a MCA or INIT event.

9.2 MCA Handling

The machine check (MCA) code path in a typical machine based on IPF architecture is shown in the diagram below (see Figure 6).

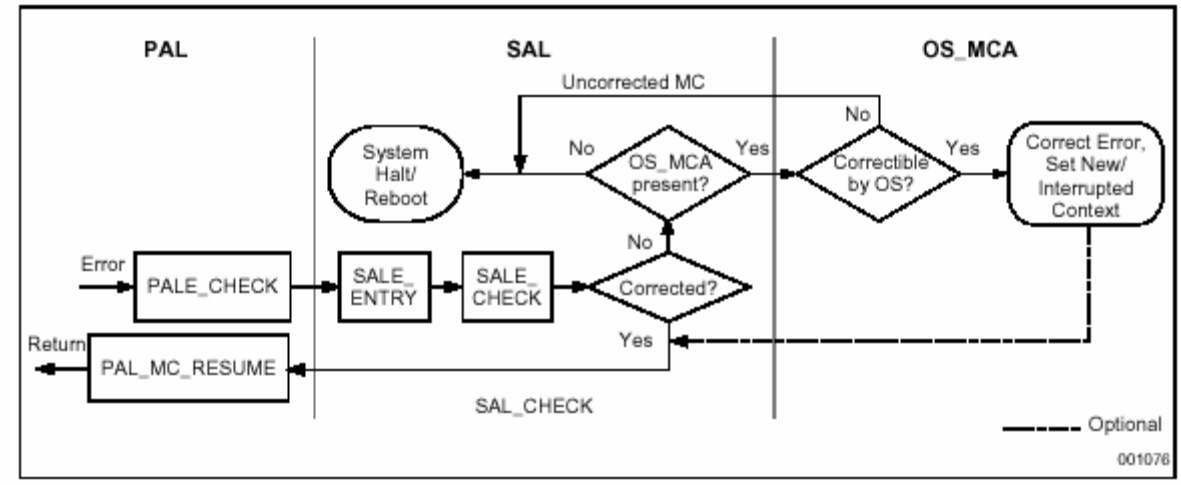

Figure 6. Basic MCA processing flow

MCA processing starts in PAL, running in physical mode. Control is then pass to SAL through the SALE_ENTRY entry point which in turn, after processing the MCA, pass control to the OS MCA handler.

In the PI architecture, OEMs have the choice to process MCA events in either entirely in ROM code, entirely in the RAM code or partly in ROM and partly in RAM. The early part of the MCA flow follow the SEC->PEI boot flow, with SALE_ENTRY residing in SEC while the MCA dispatcher is a PEIM dispatcher (see Figure 7). From that point on the rest of the code can reside in ROM or RAM.

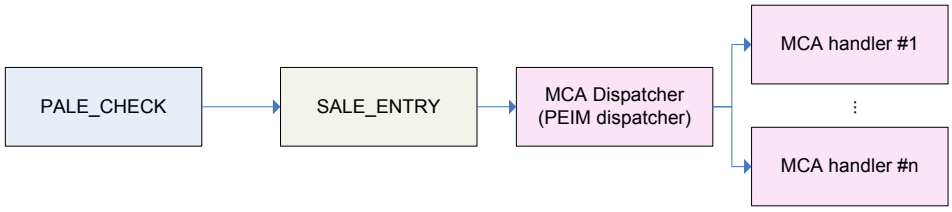

Figure 7. PI MCA processing flow

When PAL hands off control to SALE_ENTRY, it will supply unique hand off state in the processor registers as well as the minimum state saved buffer area pointer called “min-state pointer”. The min-state pointer is the only context available to SALE_ENTRY. This buffer is a unique per processor save area registered to each processor during normal OS boot path.

A sample implementation is described below to clarify some of the finer points of MCA/INIT/PMI. Actual implementations may vary.

Usually, we can anchor some extra data (the **MCA_INIT_PMI_PER_PROCESSOR_DATA** data structure) required by the PEIM dispatcher and the MCA and INIT dispatchers to the min-state (see Figure 8).

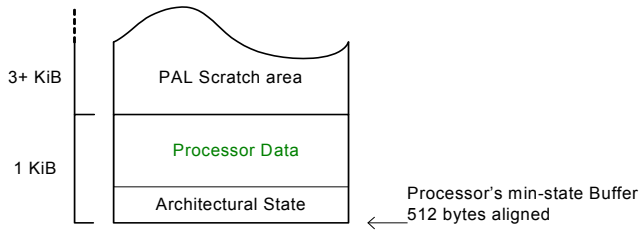

Figure 8. PI architectural data in the min-state

The software component (a PEIM or a DXE module) that includes the MCA and INIT dispatchers is responsible for registering the min-state on all processors and initializing **MCA_INIT_PMI_PER_PROCESSOR_DATA** data structures. Only then can MCA be properly handled by the platform. To guarantee proper MCA and INIT handling, at least one handler is required to be registered with the MCA dispatcher. OEM might decide to use a monolithic handler or use multiple handlers.

The register state at the MCA dispatcher entry point is the same as the PALE_CHECK exit state with the following exceptions -

- GR1 contains GP for the *McaDispatcherProc*.
- PAL saves b0 in the min-state and can be used as scratch. b0 contains the address of the *McaDispatcherProc*.
- PAL saves static registers to the min-state. All static registers in both banks except GR16-GR20 in bank 0 can be used as scratch registers. SALE_ENTRY may freely modify these registers.

The MCA dispatcher is responsible for setting up a stack and backing store based on the values in the **MCA_INIT_PMI_PER_PROCESSOR_DATA** data structure. The OS stack and backing store cannot be used since they might point to virtual addresses. The MCA dispatcher is also responsible for saving any registers not saved in the min-state that may be used by the MCA handling code in the PI per processor data. Since we want to use high-level language such as C, floating point registers f2 to f31 as well as branch registers b6 and b7 must be saved. Code used during MCA handling must be compiled with /f32 option to prevent the use of registers f33-f127. Otherwise, such code is responsible for saving and restoring floating point registers f33-f127 as well as any other registers not saved in the min-state or the PI per processor data.

Note that nested MCA recovery is not supported by the Itanium architecture as PAL uses the same min-state for every MCA and INIT event. As a result, the same context within the min-state is used by PI every time the MCA dispatcher is entered.

All the MCA handles are presented in a form of an Ordered List. The head of the Ordered List is a member of the Private Data Structure. In order to reach the MCA handle Ordered List the following steps are used:

1. PerCpuInfoPointer = MinStatePointer (From SALE_CHECK) + 4K
2. ThisCpuMcaPrivateData = PerCpuInfoPointer->Private
3. McaHandleListHead = ThisCpuMcaPrivateData->McaList

Or **`((EFI_MCA_SAVE_DATA*)(((UINT8*) MinStatePointer) + 4*1024))->Private-> McaList`**

On reaching the Ordered List from the private data we can obtain Plabel & MCA Handle Context. Using that we can execute each handle as they appear in the ordered list.

Once the last handler has completed execution, the MCA dispatcher is responsible for deciding whether to resume execution, halt the platform or reset the platform. This is based on the OS request and platform policies. Resuming to the interrupted context is accomplished by calling **`PAL_MC_RESUME`**.

As shown in Figure 6, the MCA handling flow requires access to certain shared hardware and software resources to support things such as error logging, error handling/correction and processor rendezvous. In addition, since MCAs are asynchronous, they might happen while other parts of the system are using those shared resources or while accessing those resources (for example during the execution of a SAL_PROC like PCI config write). We thus need a mechanism to allow shared access to two isolated model which are not aware of each others.

This is handled through the use of common code (libraries) and semaphores. The SAL PROCs and the MCAA/INIT code use the same libraries to implement any functionality shared between them such as platform reset, stall, PCI read/write. Semaphores are used to gate access to critical portion of the code and prevent multiple accesses to the same HW resource at the same time. To prevent deadlocks and guarantee proper OS handling of an MCA it might be necessary for the MCA/INIT handler to break semaphore or gets priority access to protected resources.

In addition to the previously mentioned semaphores used for gating access to HW resource, the multithreaded/MP MCA model may require an MCA specific semaphore to support things like monarch processor selection and log access. This semaphore should be visible from all processors. In addition some global are required for MCA processing to indicate a processor status (entering MCA, in MCA rendezvous, ready to enter OS MCA) with regards to the current MCA. This flags need to have a global scope since the MCA monarch may need to access them to make sure all processor are where they are supposed to be.

9.3 INIT Handling

Most of what have been defined for the MCA handling and dispatcher applies to the INIT code path. The early part of the INIT code path, up to the INIT dispatcher is identical to the MCA code path while some of the INIT handler code, like logging, can be shared with the MCA handler.

The INIT code path in a typical machine based on IPF architecture is shown in the diagram below.

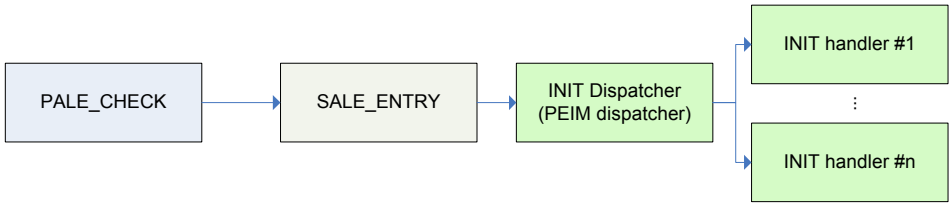

Figure 9. PI INIT processing flow

Like MCA, INIT processing starts in the PAL code in physical mode and then flows into PI code (OEM firmware code) through SALE_ENTRY. The INIT dispatcher is responsible for setting up a stack and backing store, saving the floating point registers before calling any code that may be written in higher level languages. At that point the dispatcher is ready to call the INIT handlers. As with MCA only one handler is required to exist but OEMs are free to implement a monolithic handler or use multiple handlers. Once the last handler has been executed, the dispatcher will resume to the interrupted context or reset the platform based on the OS request.

The MCA handler limitations regarding access to shared HW and SW resources applies to the INIT handler, as such library code and common semaphores should be used.

INIT events are always local to each processor. As a result we do not need INIT specific flags or semaphore in the **`MCA_INIT_PMI_PER_PROCESSOR_DATA`** data structures.

9.4 PMI

This section describes how PMI, platform management interrupts, are handled in EFI 2.0 compliant system. PMIs provide an operating system-independent interrupt mechanism to support OEM and vendor specific hardware event.

Figure 10. PMI handling flow

As shown in Figure 10, PMI handling is pretty similar to MCA and INIT handling in such that it consists of a generic dispatcher and one of more platform specific handlers. The dispatchers is the SAL PMI entry point (SALE_PMI) and is responsible for saving state and setting up the environment for the handler to execute. Contrary to MCA and INIT, PAL does not save any context in the min-state and it is the responsibility of the PMI dispatcher to save state. Since the min-state is available during PMI handling (PAL provides its address to the SAL PMI handler) the

MCA_INIT_PMI_PER_PROCESSOR_DATA data structure present in the min-state can be used. However an MCA/INIT event occurring while PMI is being would preclude the system from resuming from the PMI event. To alleviate this, a platform may decide to implement a separate copy of the **MCA_INIT_PMI_PER_PROCESSOR_DATA** data structure out side of the min-state, to be used for PMI state saving.

Once the state is saved, the platform specific PMI handlers are found using the order handler list provided in the private data structure. The mechanism used is the same one used in MCA and INIT handling.

9.5 Event Handlers

The events handlers are called by the various dispatchers.

9.5.1 MCA Handlers

MCA Handler

```
typedef
EFI_STATUS
SAL_RUNTIMESERVICE
(EFIAPI *EFI_SAL_MCA_HANDLER) (
  IN  VOID                      *ModuleGlobal,
  IN  UINT64                    ProcessorStateParameters,
  IN  EFI_PHYSICAL_ADDRESS      MinstateBase,
  IN  UINT64                    RendezvouseStateInformation,
  IN  UINT64                    CpuIndex,
  IN  SAL_MCA_COUNT_STRUCTURE   *McaCountStructure,
  IN OUT BOOLEAN                *CorrectedMachineCheck
  );
```

Parameters

ModuleGlobal

The context of MCA Handler.

ProcessorStateParameters

The processor state parameters (PSP),

MinstateBase

Base address of the min-state.

RendezvousStateInformation

Rendezvous state information to be passed to the OS on OS MCA entry. Refer to the *Sal Specification 3.0 , section 4.8* for more information.

CpuIndex

Index of the logical processor

McaCountStructure

Pointer to the MCA records structure

CorrectedMachineCheck

This flag is set to **TRUE** is the MCA has been corrected by the handler or by a previous handler.

```
#pragma pack(1)
//
// MCA Records Structure
//
typedef struct {
  UINT64  First : 1;
  UINT64  Last : 1;
  UINT64  EntryCount : 16;
  UINT64  DispatchedCount : 16;
  UINT64  Reserved : 30;
} SAL_MCA_COUNT_STRUCTURE;

#pragma pack()
```

9.5.2 INIT Handlers

INIT Handler

```
typedef
EFI_STATUS
SAL_RUNTIMESERVICE
(EFIAPI *EFI_SAL_INIT_HANDLER) (
  IN  VOID                      *ModuleGlobal,
  IN  UINT64                    ProcessorStateParameters,
  IN  EFI_PHYSICAL_ADDRESS      MinstateBase,
  IN  BOOLEAN                   McaInProgress,
  IN  UINT64                    CpuIndex,
  IN  SAL_MCA_COUNT_STRUCTURE   *McaCountStructure,
  OUT BOOLEAN                  *DumpSwitchPressed
  );
```

Parameters

ModuleGlobal

The context of MCA Handler.

ProcessorStateParameters

The processor state parameters (PSP),

MinstateBase

Base address of the min-state.

McaInProgress

This flag indicates if an MCA is in progress.

CpuIndex

Index of the logical processor

McaCountStructure

Pointer to the MCA records structure

DumpSwitchPressed

This flag indicates the crash dump switch has been pressed.

9.5.3 PMI Handlers

PMI Handler

```
typedef
EFI_STATUS
(EFIAPI *SAL_PMI_HANDLER) (
  IN  VOID                      *ModuleGlobal,
  IN  UINT64                    CpuIndex,
  IN  UINT64                    PmiVector
);
```

Description

ModuleGlobal

The context of MCA Handler.

CpuIndex

Index of the logical processor

PmiVector

The PMI vector number as received from the PALE_PMI exit state (GR24).

9.6 MCA PMI INIT Protocol

Summary

This protocol is used to register MCA, INIT and PMI handlers with their respective dispatcher.

GUID

```
#define EFI_SAL_MCA_INIT_PMI_PROTOCOL_GUID \
     {
0xb60dc6e8,0x3b6f,0x11d5,0xaf,0x9,0x0,0xa0,0xc9,0x44,0xa0,0x5b }
```

Protocol Interface Structure

```
typedef struct {
  EFI_SAL_REGISTER_MCA_HANDLER  RegisterMcaHandler;
  EFI_SAL_REGISTER_INIT_HANDLER RegisterInitHandler;
  EFI_SAL_REGISTER_PMI_HANDLER  RegisterPmiHandler;
  BOOLEAN                       McaInProgress;
  BOOLEAN                       InitInProgress;
  BOOLEAN                       PmiInProgress;
} EFI_SAL_MCA_INIT_PMI_PROTOCOL;
```

Parameters

RegisterMcaHandler

Function to register a MCA handler.

RegisterInitHandler

Function to register an INIT handler.

RegisterPmiHandler

Function to register a PMI hander.

McaInProgress

Whether MCA handler is in progress

InitInProgress

Whether Init handler is in progress

PmiInProgress

Whether Pmi handler is in progress

EFI_SAL_MCA_INIT_PMI_PROTOCOL. RegisterMcaHandler ()

Summary

Register a MCA handler with the MCA dispatcher.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SAL_REGISTER_MCA_HANDLER) (
  IN  struct _EFI_SAL_MCA_INIT_PMI_PROTOCOL   *This,
  IN  EFI_SAL_MCA_HANDLER                     McaHandler,
  IN  VOID                                    ModuleGlobal
  IN  BOOLEAN                                 MakeFirst,
  IN  BOOLEAN                                 MakeLast
  );
```

Parameters

This

The **EFI_SAL_MCA_INIT_PMI_PROTOCOL** instance.

McaHandler

The MCA handler to register as defined in section 9.5.1.

ModuleGlobal

The context of the MCA Handler.

MakeFirst

This flag specifies the handler should be made first in the list.

MakeLast

This flag specifies the handler should be made last in the list.

Status Codes Returned

EFI_SUCCESS	MCA Handle was registered
EFI_OUT_OF_RESOURCES	No more resources to register an MCA handler
EFI_INVALID_PARAMETER	Invalid parameters were passed.

EFI_SAL_MCA_INIT_PMI_PROTOCOL. RegisterInitHandler ()

Summary

Register an INIT handler with the INIT dispatcher.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SAL_REGISTER_INIT_HANDLER) (
  IN  struct _EFI_SAL_MCA_INIT_PMI_PROTOCOL   *This,
  IN  EFI_SAL_INIT_HANDLER                    InitHandler,
  IN  VOID                                    ModuleGlobal
  IN  BOOLEAN                                 MakeFirst,
  IN  BOOLEAN                                 MakeLast
  );
```

Parameters

This

The **EFI_SAL_MCA_INIT_PMI_PROTOCOL** instance.

InitHandlerT

The INIT handler to register as defined in section 9.5.2

ModuleGlobal

The context of the INIT Handler.

MakeFirst

This flag specifies the handler should be made first in the list.

MakeLast

This flag specifies the handler should be made last in the list.

Status Codes Returned

EFI_SUCCESS	INIT Handle was registered
EFI_OUT_OF_RESOURCES	No more resources to register an INIT handler
EFI_INVALID_PARAMETER	Invalid parameters were passed.

EFI_SAL_MCA_INIT_PMI_PROTOCOL. RegisterPmiHandler ()

Summary

Register a PMI handler with the PMI dispatcher.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SAL_REGISTER_PMI_HANDLER) (
  IN  struct _EFI_SAL_MCA_INIT_PMI_PROTOCOL    *This,
  IN  EFI_SAL_PMI_HANDLER                      PmiHandler,
  IN  VOID                                     ModuleGlobal
  IN  BOOLEAN                                  MakeFirst,
  IN  BOOLEAN                                  MakeLast
  );
```

Parameters

This

The **EFI_SAL_MCA_INIT_PMI_PROTOCOL** instance.

PmiHandler

The PMI handler to register as defined in section 9.5.3.

ModuleGlobal

The context of the PMI Handler.

MakeFirst

This flag specifies the handler should be made first in the list.

MakeLast

This flag specifies the handler should be made last in the list.

Status Codes Returned

EFI_SUCCESS	INIT Handle was registered
EFI_OUT_OF_RESOURCES	No more resources to register a PMI handler
EFI_INVALID_PARAMETER	Invalid parameters were passed.

10 Extended SAL Services

This document describes the Extended SAL support for the EDK II. The Extended SAL uses a calling convention that is very similar to the SAL calling convention. This includes the ability to call Extended SAL Procedures in physical mode prior to **`SetVirtualAddressMap()`**, and the ability to call Extended SAL Procedures in physical mode or virtual mode after **`SetVirtualAddressMap()`**.

10.1 SAL Overview

The Extended SAL can be used to implement the following services:

- SAL Procedures required by the *Intel Itanium Processor Family System Abstraction Layer Specification.*
- EFI Runtime Services required by the *UEFI 2.0 Specification*, that may also be required by SAL Procedures, other Extended SAL Procedures, or MCA, INIT, and PMI flows.
- Services required to abstract hardware accesses from SAL Procedures and Extended SAL Procedures. This includes I/O port accesses, MMIO accesses, PCI Configuration Cycles, and access to non-volatile storage for logging purposes.
- Services required during the MCA, INIT, and PMI flows.

Note: *Arguments to SAL procedures are formatted the same as arguments and paramenters in this document. Example "`address` parameter to . . ."*

The Extended SAL support includes a DXE Protocol that supports the publishing of the SAL System Table along with services to register and call Extended SAL Procedures. It also includes a number of standard Extended SAL Service Classes that are required to implement EFI Runtime Services, the minimum set of required SAL Procedures, services to abstract hardware accesses, and services to support the MSA, INIT, and PMI flows. Platform developer may define addition Extended SAL Service Classes to provide platform specific functionality that requires the Extended SAL calling conventions.The SAL calling convention requires operation in both physical and virtual mode. Standard EFI runtime services work in either physical mode or virtual mode at a time. Therefore, the EFI code can call the SAL code, but not vice versa. To reduce code duplication resulting out of multiple operating modes, additional procedures called Extended SAL Procedures are implemented. Architected SAL procedures are a subset of the Extended SAL procedures. The individual Extended SAL procedures can be called through the entry point **`ExtendedSalProc()`** in the **`EXTENDED_SAL_BOOT_SERVICE_PROTOCOL`**. The cost of writing dual mode code is that one must strictly follow the SAL runtime coding rules. Experience on prior IPF platform shows us that the benefits outweigh the cost.

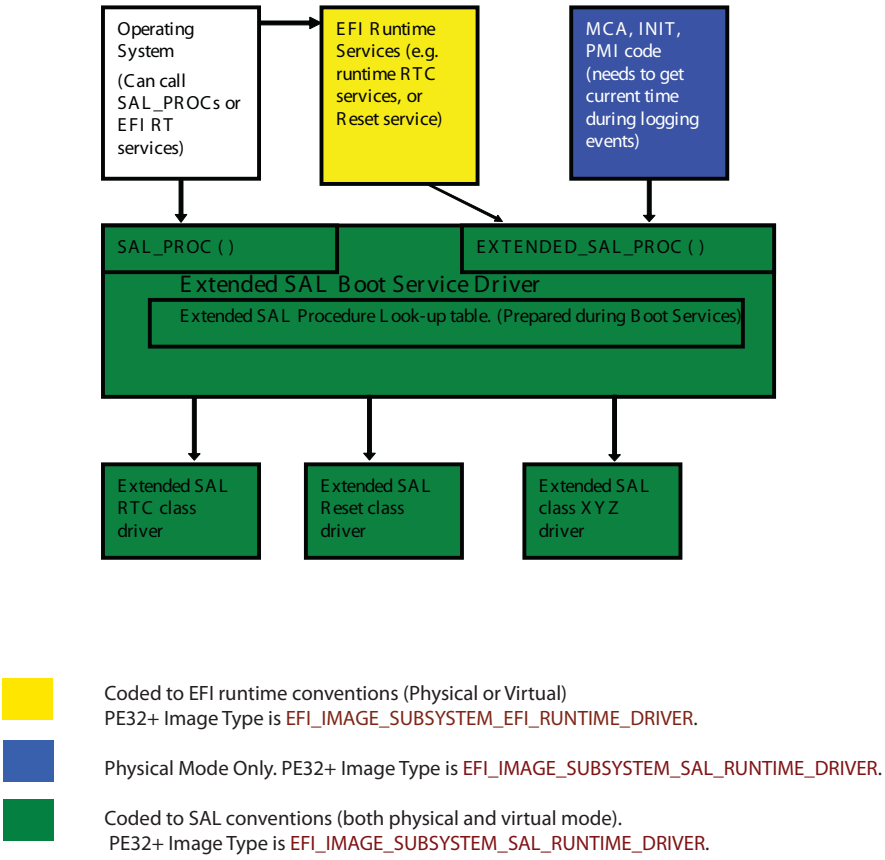

Figure 11. SAL Calling Diagram

Note: *In the figure above, arrows indicate the direction of calling. For example, OS code may call EFI runtime services or* **`SAL_PROCs`**. *Extended SAL functions are divided in several classes based on their functionality, with no defined hierarchy. It is legal for an EFI Boot Service Code to call* **`ExtendedSalProc()`**. *It is also legal for an Extended SAL procedure to call another Extended SAL Procedure via* **`ExtendedSalProc()`**. *These details are not shown in the figure in order to maintain clarity.*

A driver with a module type of **`DXE_SAL_DRIVER`** is required to produce the **`EXTENDED_SAL_BOOT_SERVICE_PROTOCOL`**. This driver contains the entry point of the Extended SAL Procedures and dispatches previously registered procedures. It also provides services to register Extended SAL Procedures and functions to help construct the SAL System Table.

Drivers with a module type of **`DXE_SAL_DRIVER`** are required to produce the various Extended SAL Service Classes. It is expected that a single driver will supply all the Extended SAL Procedures that belong to a single Extended SAL Service Class. As each Extended SAL Service Class is registered, the GUID associated with that class is also installed into the EFI Handle Database. This allows other DXE drivers to use the Extended SAL Service Class GUIDs in their dependency expressions, so they only execute once their dependent Extended SAL Service Classes are available.

Drivers register the set of Extended SAL Procedures they produce with the **EXTENDED_SAL_BOOT_SERVICE_PROTOCOL**. Once this registration step is complete, the Extended SAL Procedure are available for use by other drivers.

10.2 Extended SAL Boot Service Protocol

This protocol supports the creation of the SAL System Table, and provides services to register and call Extended SAL Procedures. The driver that produces this protocol is required to allocate and initialize the SAL System Table. The SAL System Table must also be registered in the list of EFI System Configuration tables. The driver that produces this protocol must be of type **DXE_SAL_DRIVER**. This is required because the entry point to the **ExtendedSalProc()** function is always available, even after the OS assumes control of the platform at **ExitBootServices()**.

EXTENDED_SAL_BOOT_SERVICE_PROTOCOL

Summary

This section provides a detailed description of the **EXTENDED_SAL_BOOT_SERVICE_PROTOCOL**.

GUID

```
#define EXTENDED_SAL_BOOT_SERVICE_PROTOCOL_GUID \
  {0xde0ee9a4,0x3c7a,0x44f2, \
  {0xb7,0x8b,0xe3,0xcc,0xd6,0x9c,0x3a,0xf7}}
```

Protocol Interface Structure

```
typedef struct _EXTENDED_SAL_BOOT_SERVICE_PROTOCOL {
  EXTENDED_SAL_ADD_SST_INFO            AddSalSystemTableInfo;
  EXTENDED_SAL_ADD_SST_ENTRY           AddSalSystemTableEntry;
  EXTENDED_SAL_REGISTER_INTERNAL_PROC  RegisterExtendedSalProc;
  EXTENDED_SAL_PROC                    ExtendedSalProc;
} EXTENDED_SAL_BOOT_SERVICE_PROTOCOL;
```

Parameters

AddSalSystemTableInfo

Adds platform specific information to the to the header of the SAL System Table. Only available prior to **ExitBootServices()**.

AddSalSystemTableEntry

Add an entry into the SAL System Table. Only available prior to **ExitBootServices()**.

RegisterExtendedSalProc

Registers an Extended SAL Procedure. Extended SAL Procedures are named by a (GUID, FunctionID) pair. Extended SAL Procedures are divided into classes based on the functionality they provide. Extended SAL Procedures are callable only in

physical mode prior to **SetVirtualAddressMap()**, and are callable in both virtual and physical mode after **SetVirtualAddressMap()**. Only available prior to **ExitBootServices()**.

ExtendedSalProc

Entry point for all extended SAL procedures. This entry point is always available.

Description

The **EXTENDED_SAL_BOOT_SERVICE_PROTOCOL** provides a mechanisms for platform specific drivers to update the SAL System Table and register Extended SAL Procedures that are callable in physical or virtual mode using the SAL calling convention. The services exported by the SAL System Table are typically implemented as Extended SAL Procedures. Services required by MCA, INIT, and PMI flows that are also required in the implementation of EFI Runtime Services are also typically implemented as Extended SAL Procedures. Extended SAL Procedures are named by a (GUID, FunctionID) pair. A standard set of these (GUID, FunctionID) pairs are defined in this specification. Platforms that require additional functionality from their Extended SAL Procedures may define additional (GUID, FunctionID) pairs.

EXTENDED_SAL_BOOT_SERVICE_PROTOCOL.AddSalSystemTableInfo()

Summary

Adds platform specific information to the to the header of the SAL System Table.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EXTENDED_SAL_ADD_SST_INFO) (
  IN EXTENDED_SAL_BOOT_SERVICE_PROTOCOL  *This,
  IN UINT16                              SalAVersion,
  IN UINT16                              SalBVersion,
  IN CHAR8                               *OemId,
  IN CHAR8                               *ProductId
  );
```

Parameters

This

A pointer to the **EXTENDED_SAL_BOOT_SERVICE_PROTOCOL** instance.

SalAVersion

Version of recovery SAL PEIM(s) in BCD format. Higher byte contains the major revision and the lower byte contains the minor revision.

SalBVersion

Version of DXE SAL Driver in BCD format. Higher byte contains the major revision and the lower byte contains the minor revision.

OemId

A pointer to a Null-terminated ASCII string that contains OEM unique string. The string cannot be longer than 32 bytes in total length.

ProductId

A pointer to a Null-terminated ASCII string that uniquely identifies a family of compatible products. The string cannot be longer than 32 bytes in total length.

Description

This function updates the platform specific information in the SAL System Table header. The **SAL_A_VERSION** field of the SAL System Table is set to the value specified by *SalAVersion*. The **SAL_B_VERSION** field of the SAL System Table is set to the value specified by *SalBVersion*. The **OEM_ID** field of the SAL System Table is filled in with the contents of the Null-terminated ASCII string specified by *OemId*. If *OemId* is **NULL** or the length of *OemId* is greater than 32 characters, then **EFI_INVALID_PARAMETER** is returned. The **PRODUCT_ID** field of the SAL System Table is filled in with the contents of the Null-terminated ASCII string specified by *ProductId*. If *ProductId* is **NULL** or the length of *ProductId* is greater than 32 characters, then **EFI_INVALID_PARAMETER** is returned. This function is also responsible for re-

computing the **CHECKSUM** field of the SAL System Table after the **SAL_A_REVISION**, **SAL_B_REVISION**, **OEM_ID**, and **PRODUCT_ID** fields have been filled in. Once the **CHEKSUM** field has been updated, **EFI_SUCCESS** is returned.

Status Codes Returned

EFI_SUCCESS	The SAL System Table header was updated successfully.
EFI_INVALID_PARAMETER	OemId is **NULL**.
EFI_INVALID_PARAMETER	ProductId is **NULL**.
EFI_INVALID_PARAMETER	The length of *OemId* is greater than 32 characters.
EFI_INVALID_PARAMETER	The length of *ProductId* is greater than 32 characters.

EXTENDED_SAL_BOOT_SERVICE_PROTOCOL.AddSalSystemTableEntry()

Summary

Adds an entry to the SAL System Table.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EXTENDED_SAL_ADD_SST_ENTRY) (
  IN EXTENDED_SAL_BOOT_SERVICE_PROTOCOL  *This,
  IN UINT8                               *TableEntry,
  IN UINTN                               EntrySize
  );
```

Parameters

This

A pointer to the **EXTENDED_SAL_BOOT_SERVICE_PROTOCOL** instance.

TableEntry

Pointer to a buffer containing a SAL System Table entry that is *EntrySize* bytes in length. The first byte of the *TableEntry* describes the type of entry. See the *Intel Itanium Processor Family System Abstraction Layer Specification* for more details.

EntrySize

The size, in bytes, of *TableEntry*.

Description

This function adds the SAL System Table Entry specified by *TableEntry* and *EntrySize* to the SAL System Table. If *TableEntry* is **NULL**, then **EFI_INVALID_PARAMETER** is returned. If the entry type specified in *TableEntry* is invalid, then **EFI_INVALID_PARAMETER** is returned. If the length of the *TableEntry* is not valid for the entry type specified in *TableEntry*, then **EFI_INVALID_PARAMETER** is returned. Otherwise, *TableEntry* is added to the SAL System Table. This function is also responsible for re-computing the **CHECKSUM** field of the SAL System Table. Once the **CHEKSUM** field has been updated, **EFI_SUCCESS** is returned.

Status Codes Returned

EFI_SUCCESS	The SAL System Table was updated successfully
EFI_INVALID_PARAMETER	*TableEntry* is NULL.
EFI_INVALID_PARAMETER	*TableEntry* specifies an invalid entry type.
EFI_INVALID_PARAMETER	*EntrySize* is not valid for this type of entry.

EXTENDED_SAL_BOOT_SERVICE_PROTOCOL.AddExtendedSalProc( )

Summary

Registers an Extended SAL Procedure.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EXTENDED_SAL_REGISTER_INTERNAL_PROC) (
  IN EXTENDED_SAL_BOOT_SERVICE_PROTOCOL       *This,
  IN UINT64                                   ClassGuidLo,
  IN UINT64                                   ClassGuidHi,
  IN UINT64                                   FunctionId,
  IN SAL_INTERNAL_EXTENDED_SAL_PROC           InternalSalProc,
  IN VOID \
    *PhysicalModuleGlobal OPTIONAL
  );
```

Parameters

This

A pointer to the **EXTENDED_SAL_BOOT_SERVICE_PROTOCOL** instance.

ClassGuidLo

The lower 64-bits of the class GUID for the Extended SAL Procedure being added. Each class GUID contains one or more functions specified by a Function ID.

ClassGuidHi

The upper 64-bits of the class GUID for the Extended SAL Procedure being added. Each class GUID contains one or more functions specified by a Function ID.

FunctionId

The Function ID for the Extended SAL Procedure that is being added. This Function ID is a member of the Extended SAL Procedure class specified by *ClassGuidLo* and *ClassGuidHi*.

InternalSalProc

A pointer to the Extended SAL Procedure being added. The Extended SAL Procedure is named by the GUID and Function ID specified by *ClassGuidLo*, *ClassGuidHi*, and *FunctionId*.

PhysicalModuleGlobal

Pointer to a module global structure. This is a physical mode pointer. This pointer is passed to the Extended SAL Procedure specified by *ClassGuidLo*, *ClassGuidHi*, *FunctionId*, and *InternalSalProc*. If the system is in physical mode, then this pointer is passed unmodified to *InternalSalProc*. If the system is in virtual mode, then the virtual address associated with this pointer is

passed to *InternalSalProc*. This parameter is optional and may be **NULL**. If it is **NULL**, then **NULL** is always passed to *InternalSalProc*.

Related Definitions

```
typedef
SAL_RETURN_REGS
(EFIAPI *SAL_INTERNAL_EXTENDED_SAL_PROC) (
  IN UINT64    FunctionId,
  IN UINT64    Arg2,
  IN UINT64    Arg3,
  IN UINT64    Arg4,
  IN UINT64    Arg5,
  IN UINT64    Arg6,
  IN UINT64    Arg7,
  IN UINT64    Arg8,
  IN BOOLEAN   VirtualMode,
  IN VOID      *ModuleGlobal  OPTIONAL
  );
```

FunctionId

The Function ID associated with this Extended SAL Procedure.

Arg2

Second argument to the Extended SAL procedure.

Arg3

Third argument to the Extended SAL procedure.

Arg4

Fourth argument to the Extended SAL procedure.

Arg5

Fifth argument to the Extended SAL procedure.

Arg6

Sixth argument to the Extended SAL procedure.

Arg7

Seventh argument to the Extended SAL procedure.

Arg8

Eighth argument to the Extended SAL procedure.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure.

Description

The Extended SAL Procedure *specified by InternalSalProc and named by ClassGuidLo*, *ClassGuidHi*, and *FunctionId* is added to the set of available Extended SAL Procedures. Each Extended SAL Procedure is allowed one module global to record any state information required during the execution of the Extended SAL Procedure. This module global is specified by *PhysicalModuleGlobal*.

If there are not enough resource available to add the Extended SAL Procedure, then **EFI_OUT_OF_RESOURCES** is returned.

If the Extended SAL Procedure specified by *InternalSalProc* and named by *ClassGuidLo*, *ClassGuidHi*, and *FunctionId* was not previously registered, then the Extended SAL Procedure along with its module global specified by *PhysicalModuleGlobal* is added to the set of Extended SAL Procedures, and **EFI_SUCCESS** is returned.

If the Extended SAL Procedure specified by *InternalSalProc* and named by *ClassGuidLo*, *ClassGuidHi*, and *FunctionId* was previously registered, then the module global is replaced with *PhysicalModuleGlobal*, and *EFI_SUCCESS* is returned.

Status Codes Returned

EFI_SUCCESS	The Extended SAL Procedure was added.
EFI_OUT_OF_RESOURCES	There are not enough resources available to add the Extended SAL Procedure.

EXTENDED_SAL_BOOT_SERVICE_PROTOCOL.ExtendedSalProc()

Summary

Calls a previously registered Extended SAL Procedure.

Prototype

```
typedef
SAL_RETURN_REGS
(EFIAPI *EXTENDED_SAL_PROC) (
  IN UINT64     ClassGuidLo,
  IN UINT64     ClassGuidHi,
  IN UINT64     FunctionId,
  IN UINT64     Arg2,
  IN UINT64     Arg3,
  IN UINT64     Arg4,
  IN UINT64     Arg5,
  IN UINT64     Arg6,
  IN UINT64     Arg7,
  IN UINT64     Arg8
  );
```

Parameters

ClassGuidLo

The lower 64-bits of the class GUID for the Extended SAL Procedure that is being called.

ClassGuidHi

The upper 64-bits of the class GUID for the Extended SAL Procedure that is being called.

FunctionId

Function ID for the Extended SAL Procedure being called.

Arg2

Second argument to the Extended SAL procedure.

Arg3

Third argument to the Extended SAL procedure.

Arg4

Fourth argument to the Extended SAL procedure.

Arg5

Fifth argument to the Extended SAL procedure.

Arg6

Sixth argument to the Extended SAL procedure.

Arg7

Seventh argument to the Extended SAL procedure.

Arg8

Eighth argument to the Extended SAL procedure.

Description

This function calls the Extended SAL Procedure specified by *ClassGuidLo*, *ClassGuidHi*, and *FunctionId*. The set of previously registered Extended SAL Procedures is searched for a matching *ClassGuidLo*, *ClassGuidHi*, and *FunctionId*. If a match is not found, then **EFI_SAL_NOT_IMPLEMENTED** is returned. The module global associated with *ClassGuidLo*, *ClassGuidHi*, and *FunctionId* is retrieved. If that module global is not **NULL** and the system is in virtual mode, and the virtual address of the module global is not available, then **EFI_SAL_VIRTUAL_ADDRESS_ERROR** is returned. Otherwise, the Extended SAL Procedure associated with *ClassGuidLo*, *ClassGuidHi*, and *FunctionId* is called. The arguments specified by *FunctionId*, *Arg2*, *Arg3*, *Arg4*, *Arg5*, *Arg6*, *Arg7*, and *Arg8* are passed into the Extended SAL Procedure along with the *VirtrualMode* flag and *ModuleGlobal* pointer.

If the system is in physical mode, then the *ModuleGlobal* that was originally registered with **AddExtendedSalProc()** is passed into the Extended SAL Procedure. If the system is in virtual mode, then the virtual address associated with *ModuleGlobal* is passed to the Extended SAL Procedure. The EFI Runtime Service **ConvertPointer()** is used to convert the physical address of *ModuleGlobal* to a virtual address. If *ModuleGlobal* was registered as **NULL**, then **NULL** is always passed into the Extended SAL Procedure.

The return status from this Extended SAL Procedure is returned.

Status Codes Returned

EFI_SAL_NOT_IMPLEMENTED	The Extended SAL Procedure specified by *ClassGuidLo*, *ClassGuidHi*, and *FunctionId* has not been registered.
EFI_SAL_VIRTUAL_ADDRESS_ERROR	This function was called in virtual mode before virtual mappings for the specified Extended SAL Procedure are available.
Other	The result returned from the specified Extended SAL Procedure

10.3 Extended SAL Service Classes

This chapter contains the standard set of Extended SAL service classes. These include EFI Runtime Services in the *UEFI 2.0 Specification*, SAL Procedures required by the *Intel Itanium Processor Family System Abstraction Layer Specification* , services required to abstract access to hardware devices, and services required in the handling of MCA, INIT, and PMI flows. Extended SAL Service Classes behave like PPIs and Protocols. They are named by GUID and contain a set of services for each GUID. This also allows platform developers to add new Extended SAL service classes over time to implement platform specific features that require the Extended SAL capabilities.

The following tables list the Extended SAL Service Classes defined by this specification. The following sections contain detailed descriptions of the functions in each of the classes.

Table 1. Extended SAL Service Classes – EFI Runtime Services

Name	Description
Real Time Clock Services Class	The Extended SAL Real Time Clock Services Class provides functions to access the real time clock.
Reset Services Class	The Extended SAL Reset Services Class provides platform reset services.
Status Code Services Class	The Extended SAL Status Code Services Class provides services to report status code information.
Monotonic Counter Services Class	The Extended SAL Monotonic Counter Services Class provides functions to access the monotonic counter.
Variable Services Class	The Extended SAL Variable Services Class provides functions to access EFI variables.

Table 2. Extended SAL Service Classes – SAL Procedures

Name	Description
Base Services Class	The Extended SAL Base Services Class provides base services that do not have any hardware dependencies including a number of SAL Procedures required by the *Intel Itanium Processor Family System Abstraction Layer Specification*.
Cache Services Class	The Extended SAL Cache Services Class provides services to initialize and flush the caches.
PAL Services Class	The Extended SAL PAL Services Class provides services to make PAL calls.
PCI Services Class	The Extended SAL PCI Services Class provides services to perform PCI configuration cycles.
MCA Log Services Class	The Extended SAL MCA Log Services Class provides logging services for MCA events.

Table 3. Extended SAL Service Classes – Hardware Abstractions

Name	Description
Base I/O Services Class	The Extended SAL Base I/O Services Class provides the basic abstractions for accessing I/O ports and MMIO.
Stall Services Class	The Extended SAL Stall Services Class provides functions to perform calibrated delays.
Firmware Volume Block Services Class	The Extended SAL Firmware Volume Block Services Class provides services that are equivalent to the Firmware Volume Block Protocol in the *Platform Initialization Specification*.

Table 4. Extended SAL Service Classes – Other

Name	Description
MP Services Class	The Extended SAL MP Services Class provides services for managing multiple CPUs.

MCA Services Class	TBD

10.3.1 Extended SAL Base I/O Services Class

Summary

The Extended SAL Base I/O Services Class provides the basic abstractions for accessing I/O ports and MMIO.

GUID

```
#define EFI_EXTENDED_SAL_BASE_IO_SERVICES_PROTOCOL_GUID_LO \
  0x451531e15aea42b5
#define EFI_EXTENDED_SAL_BASE_IO_SERVICES_PROTOCOL_GUID_HI \
  0xa6657525d5b831bc
#define EFI_EXTENDED_SAL_BASE_IO_SERVICES_PROTOCOL_GUID \
  {0x5aea42b5,0x31e1,0x4515,
  {0xbc,0x31,0xb8,0xd5,0x25,0x75,0x65,0xa6}}
```

Related Definitions

```
typedef enum {
  IoReadFunctionId,
  IoWriteFunctionId,
  MemReadFunctionId,
  MemWriteFunctionId,
} EFI_EXTENDED_SAL_BASE_IO_SERVICES_FUNC_ID;
```

Description

Table 5. Extended SAL Base I/O Services Class

Name	Description
ExtendedSalIoRead	This function is equivalent in functionality to the `Io.Read()` function of the CPU I/O PPI. See *Volume1:Platform Initialization Specification* Section 7.2. The function prototype for the `Io.Read()` service is shown in Related Definitions.
ExtendedSalIoWrite	This function is equivalent in functionality to the `Io.Write()` function of the CPU I/O PPI. See *Volume1:Platform Initialization Specification* Section 7.2. The function prototype for the `Io.Write()` service is shown in Related Definitions.
ExtendedSalMemRead	This function is equivalent in functionality to the `Mem.Read()` function of the CPU I/O PPI. See *Volume1:Platform Initialization Specification* Section 7.2. The function prototype for the `Mem.Read()` service is shown in Related Definitions.
ExtendedSalMemWrite	This function is equivalent in functionality to the `Mem.Write()` function of the CPU I/O PPI. See *Volume1:Platform Initialization Specification* Section 7.2. The function prototype for the `Mem.Write()` service is shown in Related Definitions.

ExtendedSalIoRead

Summary

This function is equivalent in functionality to the **Io.Read()** function of the CPU I/O PPI. See *Volume1:Platform Initialization Specification* Section 7.2. The function prototype for the **Io.Read()** service is shown in Related Definitions.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalIoRead (
  IN UINT64    FunctionId,
  IN UINT64    Arg2,
  IN UINT64    Arg3,
  IN UINT64    Arg4,
  IN UINT64    Arg5,
  IN UINT64    Arg6,
  IN UINT64    Arg7,
  IN UINT64    Arg8,
  IN BOOLEAN   VirtualMode,
  IN VOID      *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalIoReadFunctionId**.

Arg2

Signifies the width of the I/O read operation. This argument is interpreted as type **EFI_PEI_CPU_IO_PPI_WIDTH**. See the *Width* parameter in Related Definitions.

Arg3

The base address of the I/O read operation. This argument is interpreted as a **UINT64**. See the *Address* parameter in Related Definitions.

Arg4

The number of I/O read operations to perform. This argument is interpreted as a **UINTN**. See the *Count* parameter in Related Definitions.

Arg5

The destination buffer to store the results. This argument is interpreted as a **VOID *.** See the *Buffer* parameter in Related Definitions.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Related Definitions

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_CPU_IO_PPI_IO_MEM) (
  IN  EFI_PEI_SERVICES          **PeiServices,
  IN  EFI_PEI_CPU_IO_PPI        *This,
  IN  EFI_PEI_CPU_IO_PPI_WIDTH  Width,
  IN  UINT64                    Address,
  IN  UINTN                     Count,
  IN  OUT VOID                  *Buffer
  );
```

Description

This function performs the equivalent operation as the **Io.Read()** function in the CPU I/O PPI. If this function is called in virtual mode before any required mapping have been converted to virtual addresses, then **EFI_SAL_VIRTUAL_ADDRESS_ERROR** is returned. Otherwise, the status from performing the **Io.Read()** function of the CPU I/O PPI is returned.

Status Codes Returned

EFI_SAL_VIRTUAL_ADDRESS_ERROR	This function was called in virtual mode before virtual mappings for the specified Extended SAL Procedure are available.
Other	See the return status codes for the Io.Read() function in the CPU I/O PPI.

ExtendedSalIoWrite

Summary

This function is equivalent in functionality to the **Io.Write()** function of the CPU I/O PPI. See *Volume1:Platform Initialization Specification* Section 7.2. The function prototype for the **Io.Write()** service is shown in Related Definitions.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalIoWrite (
  IN UINT64    FunctionId,
  IN UINT64    Arg2,
  IN UINT64    Arg3,
  IN UINT64    Arg4,
  IN UINT64    Arg5,
  IN UINT64    Arg6,
  IN UINT64    Arg7,
  IN UINT64    Arg8,
  IN BOOLEAN   VirtualMode,
  IN VOID      *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalIoWriteFunctionId**.

Arg2

Signifies the width of the I/O write operation. This argument is interpreted as type **EFI_PEI_CPU_IO_PPI_WIDTH**. See the *Width* parameter in Related Definitions.

Arg3

The base address of the I/O write operation. This argument is interpreted as a **UINT64**. See the *Address* parameter in Related Definitions.

Arg4

The number of I/O write operations to perform. This argument is interpreted as a **UINTN**. See the *Count* parameter in Related Definitions.

Arg5

The source buffer of the value to write. This argument is interpreted as a **VOID *** . See the *Buffer* parameter in Related Definitions.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Related Definitions

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_CPU_IO_PPI_IO_MEM) (
  IN  EFI_PEI_SERVICES          **PeiServices,
  IN  EFI_PEI_CPU_IO_PPI        *This,
  IN  EFI_PEI_CPU_IO_PPI_WIDTH  Width,
  IN  UINT64                    Address,
  IN  UINTN                     Count,
  IN  OUT VOID                  *Buffer
  );
```

Description

This function performs the equivalent operation as the **Io.Write()** function in the CPU I/O PPI. If this function is called in virtual mode before any required mapping have been converted to virtual addresses, then **EFI_SAL_VIRTUAL_ADDRESS_ERROR** is returned. Otherwise, the status from performing the **Io.Write()** function of the CPU I/O PPI is returned.

Status Codes Returned

EFI_SAL_VIRTUAL_ADDRESS_ERROR	This function was called in virtual mode before virtual mappings for the specified Extended SAL Procedure are available.
Other	See the return status codes for the Io.Write() function in the CPU I/O PPI.

ExtendedSalMemRead

Summary

This function is equivalent in functionality to the **Mem.Read()** function of the CPU I/O PPI. See *Volume 1:Platform Initialization Specification* Section 7.2. The function prototype for the **Mem.Read()** service is shown in Related Definitions.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalMemRead (
  IN UINT64    FunctionId,
  IN UINT64    Arg2,
  IN UINT64    Arg3,
  IN UINT64    Arg4,
  IN UINT64    Arg5,
  IN UINT64    Arg6,
  IN UINT64    Arg7,
  IN UINT64    Arg8,
  IN BOOLEAN   VirtualMode,
  IN VOID      *ModuleGlobal  OPTIONAL
);
```

Parameters

FunctionId

Must be **EsalMemReadFunctionId**.

Arg2

Signifies the width of the MMIO read operation. This argument is interpreted as type **EFI_PEI_CPU_IO_PPI_WIDTH**. See the *Width* parameter in Related Definitions.

Arg3

The base address of the MMIO read operation. This argument is interpreted as a **UINT64**. See the *Address* parameter in Related Definitions.

Arg4

The number of MMIO read operations to perform. This argument is interpreted as a **UINTN**. See the *Count* parameter in Related Definitions.

Arg5

The destination buffer to store the results. This argument is interpreted as a **VOID *.** See the *Buffer* parameter in Related Definitions.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Related Definitions

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_CPU_IO_PPI_IO_MEM) (
  IN  EFI_PEI_SERVICES          **PeiServices,
  IN  EFI_PEI_CPU_IO_PPI        *This,
  IN  EFI_PEI_CPU_IO_PPI_WIDTH  Width,
  IN  UINT64                    Address,
  IN  UINTN                     Count,
  IN  OUT VOID                  *Buffer
  );
```

Description

This function performs the equivalent operation as the **Mem.Read()** function in the CPU I/O PPI. If this function is called in virtual mode before any required mapping have been converted to virtual addresses, then **EFI_SAL_VIRTUAL_ADDRESS_ERROR** is returned. Otherwise, the status from performing the **Mem.Read()** function of the CPU I/O PPI is returned.

Status Codes Returned

EFI_SAL_VIRTUAL_ADDRESS_ERROR	This function was called in virtual mode before virtual mappings for the specified Extended SAL Procedure are available.
Other	See the return status codes for the Mem.Read() function in the CPU I/O PPI.

ExtendedSalMemWrite

Summary

This function is equivalent in functionality to the **Mem.Write()** function of the CPU I/O PPI. See *Volume 1:Platform Initialization Specification* Section 7.2. The function prototype for the **Mem.Write()** service is shown in Related Definitions.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalMemWrite (
  IN UINT64    FunctionId,
  IN UINT64    Arg2,
  IN UINT64    Arg3,
  IN UINT64    Arg4,
  IN UINT64    Arg5,
  IN UINT64    Arg6,
  IN UINT64    Arg7,
  IN UINT64    Arg8,
  IN BOOLEAN   VirtualMode,
  IN VOID      *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalMemWriteFunctionId**.

Arg2

Signifies the width of the MMIO write operation. This argument is interpreted as type **EFI_PEI_CPU_IO_PPI_WIDTH**. See the *Width* parameter in Related Definitions.

Arg3

The base address of the MMIO write operation. This argument is interpreted as a **UINT64**. See the *Address* parameter in Related Definitions.

Arg4

The number of MMIO write operations to perform. This argument is interpreted as a **UINTN**. See the *Count* parameter in Related Definitions.

Arg5

The source buffer of the value to write. This argument is interpreted as a **VOID *** . See the *Buffer* parameter in Related Definitions.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Related Definitions

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_CPU_IO_PPI_IO_MEM) (
  IN  EFI_PEI_SERVICES          **PeiServices,
  IN  EFI_PEI_CPU_IO_PPI        *This,
  IN  EFI_PEI_CPU_IO_PPI_WIDTH  Width,
  IN  UINT64                    Address,
  IN  UINTN                     Count,
  IN  OUT VOID                  *Buffer
  );
```

Description

This function performs the equivalent operation as the **Mem.Write()** function in the CPU I/O PPI. If this function is called in virtual mode before any required mapping have been converted to virtual addresses, then **EFI_SAL_VIRTUAL_ADDRESS_ERROR** is returned. Otherwise, the status from performing the **Mem.Write()** function of the CPU I/O PPI is returned.

Status Codes Returned

EFI_SAL_VIRTUAL_ADDRESS_ERROR	This function was called in virtual mode before virtual mappings for the specified Extended SAL Procedure are available.
Other	See the return status codes for the Mem.Write() function in the CPU I/O PPI.

10.4 Extended SAL Stall Services Class

Summary

The Extended SAL Stall Services Class provides functions to perform calibrated delays.

GUID

```
#define EFI_EXTENDED_SAL_STALL_SERVICES_PROTOCOL_GUID_LO \
```

```
  0x4d8cac2753a58d06
#define EFI_EXTENDED_SAL_STALL_SERVICES_PROTOCOL_GUID_HI \
  0x704165808af0e9b5
#define EFI_EXTENDED_SAL_STALL_SERVICES_PROTOCOL_GUID \
  {0x53a58d06,0xac27,0x4d8c,\
  {0xb5,0xe9,0xf0,0x8a,0x80,0x65,0x41,0x70}}
```

Related Definitions

```
typedef enum {
  StallFunctionId,
} EFI_EXTENDED_SAL_STALL_FUNC_ID;
```

Description

Table 6. Extended SAL Stall Services Class

Name	Description
ExtendedSalStall	This function is equivalent in functionality to the EFI Boot Service **`Stall()`**. See *UEFI 2.0 Specification* Section 6.5. The function prototype for the **`Stall()`** service is shown in Related Definitions.

ExtendedSalStall

Summary

This function is equivalent in functionality to the EFI Boot Service **Stall()**. See *UEFI 2.0 Specification* Section 6.5. The function prototype for the **Stall()** service is shown in Related Definitions.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalStall (
  IN UINT64    FunctionId,
  IN UINT64    Arg2,
  IN UINT64    Arg3,
  IN UINT64    Arg4,
  IN UINT64    Arg5,
  IN UINT64    Arg6,
  IN UINT64    Arg7,
  IN UINT64    Arg8,
  IN BOOLEAN   VirtualMode,
  IN VOID      *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalStallFunctionId**.

Arg2

Specifies the delay in microseconds. This argument is interpreted as type **UINTN**. See *Microseconds* in Related Definitions.

Arg3

Reserved. Must be zero.

Arg4

Reserved. Must be zero.

Arg5

Reserved. Must be zero.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Related Definitions

```
typedef
EFI_STATUS
(EFIAPI *EFI_STALL) (
  IN  UINTN  Microseconds
  );
```

Description

This function performs the equivalent operation as the **Stall()** function in the EFI Boot Services Table. If this function is called in virtual mode before any required mapping have been converted to virtual addresses, then **EFI_SAL_VIRTUAL_ADDRESS_ERROR** is returned. Otherwise, the one of the status codes defined in the **Stall()** function of the EFI Boot Services Table is returned.

Status Codes Returned

EFI_SAL_VIRTUAL_ADDRESS_ERROR	This function was called in virtual mode before virtual mappings for the specified Extended SAL Procedure are available.
Other	See the return status codes for the Stall() function in the EFI Boot Services Table.

10.4.1 Extended SAL Real Time Clock Services Class

Summary

The Extended SAL Real Time Clock Services Class provides functions to access the real time clock.

GUID

```
#define EFI_EXTENDED_SAL_RTC_SERVICES_PROTOCOL_GUID_LO \
  0x4d02efdb7e97a470
#define EFI_EXTENDED_SAL_RTC_SERVICES_PROTOCOL_GUID_HI \
  0x96a27bd29061ce8f
#define EFI_EXTENDED_SAL_RTC_SERVICES_PROTOCOL_GUID  \
  {0x7e97a470,0xefdb,0x4d02, \
  {0x8f,0xce,0x61,0x90,0xd2,0x7b,0xa2,0x96}
```

Related Definitions

```
typedef enum {
  GetTimeFunctionId,
  SetTimeFunctionId,
```

```
    GetWakeupTimeFunctionId,
    SetWakeupTimeFunctionId,
    GetRtcClassMaxFunctionId
    InitializeThresholdFunctionId,
    BumpThresholdCountFunctionId,
    GetThresholdCountFunctionId
  } EFI_EXTENDED_SAL_RTC_SERVICES_FUNC_ID;
```

Description

Table 7. Extended SAL Real Time Clock Services Class

Name	Description
ExtendedSalGetTime	This function is equivalent in functionality to the EFI Boot Service **`GetTime()`**. See *UEFI 2.0 Specification* Section 7.2. The function prototype for the **`GetTime()`** service is shown in Related Definitions.
ExtendedSalSetTime	This function is equivalent in functionality to the EFI Runtime Service **`SetTime()`**. See *UEFI 2.0 Specification* Section 7.2. The function prototype for the **`SetTime()`** service is shown in Related Definitions.
ExtendedSalGetWakeupTime	This function is equivalent in functionality to the EFI Runtime Service **`GetWakeupTime()`**. See *UEFI 2.0 Specification* Section 7.2. The function prototype for the **`GetWakeupTime()`** service is shown in Related Definitions.
ExtendedSalSetWakeupTime	This function is equivalent in functionality to the EFI Runtime Service **`SetWakeupTime()`**. See *UEFI 2.0 Specification* Section 7.2. The function prototype for the **`SetWakeupTime()`** service is shown in Related Definitions.

ExtendedSalGetTime

Summary

This function is equivalent in functionality to the EFI Runtime Service **GetTime()**. See *UEFI 2.0 Specification* Section 7.2. The function prototype for the **GetTime()** service is shown in Related Definitions.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalGetTime (
  IN UINT64    FunctionId,
  IN UINT64    Arg2,
  IN UINT64    Arg3,
  IN UINT64    Arg4,
  IN UINT64    Arg5,
  IN UINT64    Arg6,
  IN UINT64    Arg7,
  IN UINT64    Arg8,
  IN BOOLEAN   VirtualMode,
  IN VOID      *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalGetTimeFunctionId**.

Arg2

This argument is interpreted as a pointer to an **EFI_TIME** structure. See *Time* in Related Definitions.

Arg3

This argument is interpreted as a pointer to an **EFI_TIME_CAPABILITIES** structure. See *Capabilities* in Related Definitions.

Arg4

Reserved. Must be zero.

Arg5

Reserved. Must be zero.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Related Definitions

```
typedef
EFI_STATUS
(EFIAPI *EFI_GET_TIME) (
  OUT EFI_TIME                *Time,
  OUT EFI_TIME_CAPABILITIES   *Capabilities  OPTIONAL
  );
```

Description

This function performs the equivalent operation as the **GetTime()** function in the EFI Runtime Services Table. If this function is called in virtual mode before any required mapping have been converted to virtual addresses, then **EFI_SAL_VIRTUAL_ADDRESS_ERROR** is returned. Otherwise, the one of the status codes defined in the **GetTime()** function of the EFI Runtime Services Table is returned.

Status Codes Returned

EFI_SAL_VIRTUAL_ADDRESS_ERROR	This function was called in virtual mode before virtual mappings for the specified Extended SAL Procedure are available.
Other	See the return status codes for the GetTime() function in the EFI Runtime Services Table.

ExtendedSalSetTime

Summary

This function is equivalent in functionality to the EFI Runtime Service **SetTime()**. See *UEFI 2.0 Specification* Section 7.2. The function prototype for the **SetTime()** service is shown in Related Definitions.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalSetTime (
  IN UINT64    FunctionId,
  IN UINT64    Arg2,
  IN UINT64    Arg3,
  IN UINT64    Arg4,
  IN UINT64    Arg5,
  IN UINT64    Arg6,
  IN UINT64    Arg7,
  IN UINT64    Arg8,
  IN BOOLEAN   VirtualMode,
  IN VOID      *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalGetTimeFunctionId**.

Arg2

This argument is interpreted as a pointer to an **EFI_TIME** structure. See *Time* in Related Definitions.

Arg3

Reserved. Must be zero.

Arg4

Reserved. Must be zero.

Arg5

Reserved. Must be zero.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Related Definitions

```
typedef
EFI_STATUS
(EFIAPI *EFI_SET_TIME) (
  IN EFI_TIME   *Time
  );
```

Description

This function performs the equivalent operation as the **SetTime()** function in the EFI Runtime Services Table. If this function is called in virtual mode before any required mapping have been converted to virtual addresses, then **EFI_SAL_VIRTUAL_ADDRESS_ERROR** is returned. Otherwise, the one of the status codes defined in the **SetTime()** function of the EFI Runtime Services Table is returned.

Status Codes Returned

EFI_SAL_VIRTUAL_ADDRESS_ERROR	This function was called in virtual mode before virtual mappings for the specified Extended SAL Procedure are available.
Other	See the return status codes for the SetTime() function in the EFI Runtime Services Table.

ExtendedSalGetWakeupTime

Summary

This function is equivalent in functionality to the EFI Runtime Service **GetWakeupTime()**. See *UEFI 2.0 Specification* Section 7.2. The function prototype for the **GetWakeupTime()** service is shown in Related Definitions.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalGetWakeupTime (
  IN UINT64    FunctionId,
  IN UINT64    Arg2,
  IN UINT64    Arg3,
  IN UINT64    Arg4,
  IN UINT64    Arg5,
  IN UINT64    Arg6,
  IN UINT64    Arg7,
  IN UINT64    Arg8,
  IN BOOLEAN   VirtualMode,
  IN VOID      *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalGetWakeupTimeFunctionId**.

Arg2

This argument is interpreted as a pointer to a **BOOLEAN** value. See *Enabled* in Related Definitions.

Arg3

This argument is interpreted as a pointer to a **BOOLEAN** value. See *Pending* in Related Definitions.

Arg4

This argument is interpreted as a pointer to an **EFI_TIME** structure. See *Time* in Related Definitions.

Arg5

Reserved. Must be zero.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Related Definitions

```
typedef
EFI_STATUS
(EFIAPI *EFI_GET_WAKEUP_TIME) (
  OUT BOOLEAN    *Enabled,
  OUT BOOLEAN    *Pending,
  OUT EFI_TIME   *Time
  );
```

Description

This function performs the equivalent operation as the **GetWakeupTime()** function in the EFI Runtime Services Table. If this function is called in virtual mode before any required mapping have been converted to virtual addresses, then **EFI_SAL_VIRTUAL_ADDRESS_ERROR** is returned. Otherwise, the one of the status codes defined in the **GetWakeupTime()** function of the EFI Runtime Services Table is returned.

Status Codes Returned

EFI_SAL_VIRTUAL_ADDRESS_ERROR	This function was called in virtual mode before virtual mappings for the specified Extended SAL Procedure are available.
Other	See the return status codes for the GetWakeupTime() function in the EFI Runtime Services Table.

ExtendedSalSetWakeupTime

Summary

This function is equivalent in functionality to the EFI Runtime Service **SetWakeupTime()**. See *UEFI 2.0 Specification* Section 7.2. The function prototype for the **SetWakeupTime()** service is shown in Related Definitions.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalSetWakeupTime (
  IN UINT64    FunctionId,
  IN UINT64    Arg2,
  IN UINT64    Arg3,
  IN UINT64    Arg4,
  IN UINT64    Arg5,
  IN UINT64    Arg6,
  IN UINT64    Arg7,
  IN UINT64    Arg8,
  IN BOOLEAN   VirtualMode,
  IN VOID      *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalSetWakeupTimeFunctionId**.

Arg2

This argument is interpreted as a **BOOLEAN** value. See *Enable* in Related Definitions.

Arg3

This argument is interpreted as a pointer to an **EFI_TIME** structure. See *Time* in Related Definitions.

Arg4

Reserved. Must be zero.

Arg5

Reserved. Must be zero.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Related Definitions

```
typedef
EFI_STATUS
(EFIAPI *EFI_SET_WAKEUP_TIME) (
  IN BOOLEAN   Enable,
  IN EFI_TIME  *Time   OPTIONAL
  );
```

Description

This function performs the equivalent operation as the **SetWakeupTime()** function in the EFI Runtime Services Table. If this function is called in virtual mode before any required mapping have been converted to virtual addresses, then **EFI_SAL_VIRTUAL_ADDRESS_ERROR** is returned. Otherwise, the one of the status codes defined in the **SetWakeupTime()** function of the EFI Runtime Services Table is returned.

Status Codes Returned

EFI_SAL_VIRTUAL_ADDRESS_ERROR	This function was called in virtual mode before virtual mappings for the specified Extended SAL Procedure are available.
Other	See the return status codes for the SetWakeupTime() function in the EFI Runtime Services Table.

10.4.2 Extended SAL Reset Services Class

Summary

The Extended SAL Reset Services Class provides platform reset services.

GUID

```
#define EFI_EXTENDED_SAL_RESET_SERVICES_PROTOCOL_GUID_LO \
  0x46f58ce17d019990
#define EFI_EXTENDED_SAL_RESET_SERVICES_PROTOCOL_GUID_HI \
  0xa06a6798513c76a7
#define EFI_EXTENDED_SAL_RESET_SERVICES_PROTOCOL_GUID \
  {0x7d019990,0x8ce1,0x46f5,
  {0xa7,0x76,0x3c,0x51,0x98,0x67,0x6a,0xa0}}
```

Related Definitions

```
typedef enum {
  ResetSystemFunctionId,
} EFI_EXTENDED_SAL_RESET_FUNC_ID;
```

Description

Table 8. Extended SAL Reset Services Class

Name	Description
ExtendedSalResetSystem	This function is equivalent in functionality to the EFI Runtime Service **`ResetSystem()`**. See *UEFI 2.0 Specification* Section 7.4.1. The function prototype for the **`ResetSystem()`** service is shown in Related Definitions.

ExtendedSalResetSystem

Summary

This function is equivalent in functionality to the EFI Runtime Service **ResetSystem()**. See *UEFI 2.0 Specification* Section 7.4.1. The function prototype for the **ResetSystem()** service is shown in Related Definitions.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalResetSystem (
  IN UINT64    FunctionId,
  IN UINT64    Arg2,
  IN UINT64    Arg3,
  IN UINT64    Arg4,
  IN UINT64    Arg5,
  IN UINT64    Arg6,
  IN UINT64    Arg7,
  IN UINT64    Arg8,
  IN BOOLEAN   VirtualMode,
  IN VOID      *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalResetSystemFunctionId**.

Arg2

This argument is interpreted as a **EFI_RESET_TYPE** value. See *ResetType* in Related Definitions.

Arg3

This argument is interpreted as **EFI_STATUS** value. See *ResetStatus* in Related Definitions.

Arg4

This argument is interpreted as **UINTN** value. See *DataSize* in Related Definitions.

Arg5

This argument is interpreted a pointer to a Unicode string. See *ResetData* in Related Definitions.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Related Definitions

```
typedef
VOID
(EFIAPI *EFI_RESET_SYSTEM) (
  IN EFI_RESET_TYPE   ResetType,
  IN EFI_STATUS       ResetStatus,
  IN UINTN            DataSize,
  IN CHAR16           *ResetData  OPTIONAL
  );
```

Description

This function performs the equivalent operation as the **ResetSystem()** function in the EFI Runtime Services Table. If this function is called in virtual mode before any required mapping have been converted to virtual addresses, then **EFI_SAL_VIRTUAL_ADDRESS_ERROR** is returned. Otherwise, the one of the status codes defined in the **ResetSystem()** function of the EFI Runtime Services Table is returned.

Status Codes Returned

EFI_SAL_VIRTUAL_ADDRESS_ERROR	This function was called in virtual mode before virtual mappings for the specified Extended SAL Procedure are available.
Other	See the return status codes for the ResetSystem() function in the EFI Runtime Services Table.

10.4.3 Extended SAL PCI Services Class

Summary

The Extended SAL PCI Services Class provides services to perform PCI configuration cycles.

GUID

```
#define EFI_EXTENDED_SAL_PCI_SERVICES_PROTOCOL_GUID_LO \
  0x4905ad66a46b1a31
#define EFI_EXTENDED_SAL_PCI_SERVICES_PROTOCOL_GUID_HI \
  0x6330dc59462bf692
#define EFI_EXTENDED_SAL_PCI_SERVICES_PROTOCOL_GUID \
```

```
  {0xa46b1a31,0xad66,0x4905,
  {0x92,0xf6,0x2b,0x46,0x59,0xdc,0x30,0x63}}
```

Related Definitions

```
typedef enum {
  SalPciConfigReadFunctionId,
  SalPciConfigWriteFunctionId,
} EFI_EXTENDED_SAL_PCI_SERVICES_FUNC_ID;
```

Description

Table 9. Extended SAL PCI Services Class

Name	Description
ExtendedSalPciRead	This function is equivalent in functionality to the SAL Procedure **SAL_PCI_CONFIG_READ**. See the *Intel Itanium Processor Family System Abstraction Layer Specification* Chapter 9.
ExtendedSalPciWrite	This function is equivalent in functionality to the SAL Procedure **SAL_PCI_CONFIG_WRITE**. See the *Intel Itanium Processor Family System Abstraction Layer Specification* Chapter 9.

ExtendedSalPciRead

Summary

This function is equivalent in functionality to the SAL Procedure **SAL_PCI_CONFIG_READ**. See the *Intel Itanium Processor Family System Abstraction Layer Specification* Chapter 9.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalPciRead (
  IN UINT64    FunctionId,
  IN UINT64    Arg2,
  IN UINT64    Arg3,
  IN UINT64    Arg4,
  IN UINT64    Arg5,
  IN UINT64    Arg6,
  IN UINT64    Arg7,
  IN UINT64    Arg8,
  IN BOOLEAN   VirtualMode,
  IN VOID      *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalPciReadFunctionId**.

Arg2

address parameter to **SAL_PCI_CONFIG_WRITE**.

Arg3

size parameter to **SAL_PCI_CONFIG_WRITE**.

Arg4

address_type parameter to **SAL_PCI_CONFIG_WRITE**.

Arg5

Reserved. Must be zero.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

ExtendedSalPciWrite

Summary

This function is equivalent in functionality to the SAL Procedure **SAL_PCI_CONFIG_WRITE**. See the *Intel Itanium Processor Family System Abstraction Layer Specification* Chapter 9.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalPciWrite (
  IN UINT64    FunctionId,
  IN UINT64    Arg2,
  IN UINT64    Arg3,
  IN UINT64    Arg4,
  IN UINT64    Arg5,
  IN UINT64    Arg6,
  IN UINT64    Arg7,
  IN UINT64    Arg8,
  IN BOOLEAN   VirtualMode,
  IN VOID      *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalPciWriteFunctionId**.

Arg2

address parameter to **SAL_PCI_CONFIG_WRITE**.

Arg3

size parameter to **SAL_PCI_CONFIG_WRITE**.

Arg4

value parameter to **SAL_PCI_CONFIG_WRITE**.

Arg5

address_type parameter to **SAL_PCI_CONFIG_WRITE**.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

10.4.4 Extended SAL Cache Services Class

Summary

The Extended SAL Cache Services Class provides services to initialize and flush the caches.

GUID

```
#define EFI_EXTENDED_SAL_CACHE_SERVICES_PROTOCOL_GUID_LO \
  0x4ba52743edc9494
#define EFI_EXTENDED_SAL_CACHE_SERVICES_PROTOCOL_GUID_HI \
  0x88f11352ef0a1888
#define EFI_EXTENDED_SAL_CACHE_SERVICES_PROTOCOL_GUID \
  {0xedc9494,0x2743,0x4ba5,\
  {0x88,0x18,0x0a,0xef,0x52,0x13,0xf1,0x88}}
```

Related Definitions

```
typedef enum {
  SalCacheInitFunctionId,
  SalCacheFlushFunctionId,
  SalCacheClassMaxFunctionId
} EFI_EXTENDED_SAL_CACHE_SERVICES_FUNC_ID;
```

Description

Table 10. Extended SAL Cache Services Class

Name	Description
ExtendedSalCacheInit	This function is equivalent in functionality to the SAL Procedure **SAL_CACHE_INIT**. See the *Intel Itanium Processor Family System Abstraction Layer Specification* Chapter 9.
ExtendedSalCacheFlush	This function is equivalent in functionality to the SAL Procedure **SAL_CACHE_FLUSH**. See the *Intel Itanium Processor Family System Abstraction Layer Specification* Chapter 9.

ExtendedSalCacheInit

Summary

This function is equivalent in functionality to the SAL Procedure **SAL_CACHE_INIT**. See the *Intel Itanium Processor Family System Abstraction Layer Specification* Chapter 9.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalCacheInit (
  IN UINT64    FunctionId,
  IN UINT64    Arg2,
  IN UINT64    Arg3,
  IN UINT64    Arg4,
  IN UINT64    Arg5,
  IN UINT64    Arg6,
  IN UINT64    Arg7,
  IN UINT64    Arg8,
  IN BOOLEAN   VirtualMode,
  IN VOID      *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalCacheInitFunctionId**.

Arg2

Reserved. Must be zero.

Arg3

Reserved. Must be zero.

Arg4

Reserved. Must be zero.

Arg5

Reserved. Must be zero.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

ExtendedSalCacheFlush

Summary

This function is equivalent in functionality to the SAL Procedure **SAL_CACHE_FLUSH**. See the *Intel Itanium Processor Family System Abstraction Layer Specification* Chapter 9.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalCacheFlush (
  IN UINT64    FunctionId,
  IN UINT64    Arg2,
  IN UINT64    Arg3,
  IN UINT64    Arg4,
  IN UINT64    Arg5,
  IN UINT64    Arg6,
  IN UINT64    Arg7,
  IN UINT64    Arg8,
  IN BOOLEAN   VirtualMode,
  IN VOID      *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalCacheFlushFunctionId**.

Arg2

i_or_d parameter in **SAL_CACHE_FLUSH**.

Arg3

Reserved. Must be zero.

Arg4

Reserved. Must be zero.

Arg5

Reserved. Must be zero.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

10.4.5 Extended SAL PAL Services Class

Summary

The Extended SAL PAL Services Class provides services to make PAL calls.

GUID

```
#define EFI_EXTENDED_SAL_PAL_SERVICES_PROTOCOL_GUID_LO \
  0x438d0fc2e1cd9d21
#define EFI_EXTENDED_SAL_PAL_SERVICES_PROTOCOL_GUID_HI \
  0x571e966de6040397
#define EFI_EXTENDED_SAL_PAL_SERVICES_PROTOCOL_GUID \
  {0xe1cd9d21,0x0fc2,0x438d, \
  {0x97,0x03,0x04,0xe6,0x6d,0x96,0x1e,0x57}}
```

Related Definitions

```
typedef enum {
  PalProcFunctionId,
  SetNewPalEntryFunctionId,
  GetNewPalEntryFunctionId,
  EsalUpdatePalFunctionId,
} EFI_EXTENDED_SAL_PAL_SERVICES_FUNC_ID;
```

Description

Table 11. Extended SAL PAL Services Class

Name	Description
ExtendedSalPalProc	This function provides a C wrapper for making PAL Procedure calls. See the *Intel Itanium Architecture Software Developers Manual Volume2: System Architecture* Section 11.10 for details on the PAL calling conventions and the set of PAL Procedures.
ExtendedSalSetNewPalEntry	This function records the physical or virtual PAL entry point.
ExtendedSalSetNewPalEntry	This function retrieves the physical or virtual PAL entry point.

ExtendedSalPalProc

Summary

This function provides a C wrapper for making PAL Procedure calls. See the *Intel Itanium Architecture Software Developers Manual Volume2: System Architecture* Section 11.10 for details on the PAL calling conventions and the set of PAL Procedures.

Prototype

```
PAL_PROC_RETURN
EFIAPI
ExtendedSalPalProc (
  IN UINT64    FunctionId,
  IN UINT64    Arg2,
  IN UINT64    Arg3,
  IN UINT64    Arg4,
  IN UINT64    Arg5,
  IN UINT64    Arg6,
  IN UINT64    Arg7,
  IN UINT64    Arg8,
  IN BOOLEAN   VirtualMode,
  IN VOID      *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalPalProcFunctionId**.

Arg2

PAL_PROC Function ID.

Arg3

Arg2of the **PAL_PROC**.

Arg4

Arg3 of the **PAL_PROC**.

Arg5

Arg4 of the **PAL_PROC**.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Description

This function provide a C wrapper for making PAL Procedure calls. The **PAL_PROC** Function ID in Arg2 is used to determine if the **PAL_PROC** is stacked or static. If the PAL has been shadowed, then the memory copy of the PAL is called. Otherwise, the ROM version of the PAL is called. The caller does not need to worry whether or not the PAL has been shadowed or not (except for the fact that some of the PAL calls don't work until PAL has been shadowed). If this function is called in virtual mode before any required mapping have been converted to virtual addresses, then **EFI_SAL_VIRTUAL_ADDRESS_ERROR** is returned. Otherwise, the return status from the **PAL_PROC** is returned.

ExtendedSalSetNewPalEntry

Summary

This function records the physical or virtual PAL entry point.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalSetNewPalEntry (
  IN UINT64    FunctionId,
  IN UINT64    Arg2,
  IN UINT64    Arg3,
  IN UINT64    Arg4,
  IN UINT64    Arg5,
  IN UINT64    Arg6,
  IN UINT64    Arg7,
  IN UINT64    Arg8,
  IN BOOLEAN   VirtualMode,
  IN VOID      *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalSetNewPalEntryFunctionId**.

Arg2

This parameter is interpreted as a **BOOLEAN**. If it is **TRUE**, then PAL Entry Point specified by *Arg3* is a physical address. If it is **FALSE**, then the Pal Entry Point specified by *Arg3* is a virtual address.

Arg3

The PAL Entry Point that is being set.

Arg4

Reserved. Must be zero.

Arg5

Reserved. Must be zero.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Description

This function records the PAL Entry Point specified by *Arg3*, so **PAL_PROC** calls can be made with the **EsalPalProcFunctionId** Function ID. If *Arg2* is **TRUE**, then *Arg3* is the physical address of the PAL Entry Point. If *Arg2* is **FALSE**, then *Arg3* is the virtual address of the PAL Entry Point. If this function is called in virtual mode before any required mapping have been converted to virtual addresses, then **EFI_SAL_VIRTUAL_ADDRESS_ERROR** is returned. Otherwise, the **EFI_SAL_SUCCESS** is returned.

Status Codes Returned

EFI_SAL_SUCCESS	The PAL Entry Point was set
EFI_SAL_VIRTUAL_ADDRESS_ERROR	This function was called in virtual mode before virtual mappings for the specified Extended SAL Procedure are available.

ExtendedSalGetNewPalEntry

Summary

This function retrieves the physical or virtual PAL entry point.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalGetNewPalEntry (
  IN UINT64    FunctionId,
  IN UINT64    Arg2,
  IN UINT64    Arg3,
  IN UINT64    Arg4,
  IN UINT64    Arg5,
  IN UINT64    Arg6,
  IN UINT64    Arg7,
  IN UINT64    Arg8,
  IN BOOLEAN   VirtualMode,
  IN VOID      *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalGetNewPalEntryFunctionId**.

Arg2

This parameter is interpreted as a **BOOLEAN.** If it is **TRUE**, then physical address of the PAL Entry Point is retrieved. If it is **FALSE**, then the virtual address of the Pal Entry Point is retrieved.

Arg3

Reserved. Must be zero.

Arg4

Reserved. Must be zero.

Arg5

Reserved. Must be zero.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Description

This function retrieves the PAL Entry Point that as previously set with **EsalSetNewPalEntryFunctionId**. If *Arg2* is **TRUE**, then the physical address of the PAL Entry Point is returned in **SAL_RETURN_REGS**.*r9* and **EFI_SAL_SUCCESS** is returned. If *Arg2* is **FALSE** and a virtual mapping for the PAL Entry Point is not available, then **EFI_SAL_VIRTUAL_ADDRESS_ERROR** is returned. If *Arg2* is **FALSE** and a virtual mapping for the PAL Entry Point is available, then the virtual address of the PAL Entry Point is returned in **SAL_RETURN_REGS.***r9* and **EFI_SAL_SUCCESS** is returned.

Status Codes Returned

EFI_SAL_SUCCESS	The PAL Entry Point was retrieved and returned in SAL_RETURN_REGS.r9.
EFI_SAL_VIRTUAL_ADDRESS_ERROR	A request for the virtual mapping of the PAL Entry Point was requested, and a virtual mapping is not currently available.

ExtendedSalUpdatePal

Summary

This function is equivalent in functionality to the SAL Procedure **SAL_UPDATE_PAL**. See the *Intel Itanium Processor Family System Abstraction Layer Specification* Chapter 9.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalUpdatePal (
  IN UINT64    FunctionId,
  IN UINT64    Arg2,
  IN UINT64    Arg3,
  IN UINT64    Arg4,
  IN UINT64    Arg5,
  IN UINT64    Arg6,
  IN UINT64    Arg7,
  IN UINT64    Arg8,
  IN BOOLEAN   VirtualMode,
  IN VOID      *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalUpdatePal**.

Arg2

param_buf parameter to **SAL_UPDATE_PAL**.

Arg3

scratch_buf parameter to **SAL_UPDATE_PAL**.

Arg4

scratch_buf_size parameter to **SAL_UPDATE_PAL**.

Arg5

Reserved. Must be zero.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

10.4.6 Extended SAL Status Code Services Class

Summary

The Extended SAL Status Code Services Class provides services to report status code information.

GUID

```
#define EFI_EXTENDED_SAL_STATUS_CODE_SERVICES_PROTOCOL_GUID_LO \
  0x420f55e9dbd91d
#define EFI_EXTENDED_SAL_STATUS_CODE_SERVICES_PROTOCOL_GUID_HI \
  0x4fb437849f5e3996
#define EFI_EXTENDED_SAL_STATUS_CODE_SERVICES_PROTOCOL_GUID  \
  {0xdbd91d,0x55e9,0x420f,
  {0x96,0x39,0x5e,0x9f,0x84,0x37,0xb4,0x4f}}
```

Related Definitions

```
typedef enum {
  ReportStatusCodeServiceFunctionId,
} EFI_EXTENDED_SAL_STATUS_CODE_SERVICES_FUNC_ID;
```

Description

Table 12. Extended SAL Status Code Services Class

Name	Description
ExtendedSalReportStatusCode	This function is equivalent in functionality to the **ReportStatusCode()** service of the Status Code Runtime Protocol. See Section 12.2 of the *Volume 2:Platform Initialization Specification, Driver Execution Environment, Core Interface*. The function prototype for the **ReportStatusCode()** service is shown in Related Definitions.

ExtendedSalReportStatusCode

Summary

This function is equivalent in functionality to the **ReportStatusCode()** service of the Status Code Runtime Protocol. See Section 12.2 of the *Volume 2:Platform Initialization Specification, Driver Execution Environment, Core Interface*. The function prototype for the **ReportStatusCode()** service is shown in Related Definitions.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalReportStatusCode (
  IN UINT64    FunctionId,
  IN UINT64    Arg2,
  IN UINT64    Arg3,
  IN UINT64    Arg4,
  IN UINT64    Arg5,
  IN UINT64    Arg6,
  IN UINT64    Arg7,
  IN UINT64    Arg8,
  IN BOOLEAN   VirtualMode,
  IN VOID      *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalReportStatusCodeFunctionId**.

Arg2

This argument is interpreted as type **EFI_STATUS_CODE_TYPE**. See the *Type* parameter in Related Definitions.

Arg3 *T*

his argument is interpreted as type **EFI_STATUS_CODE_VALUE**. See the *Value* parameter in Related Definitions.

Arg4

This argument is interpreted as type **UINT32**. See the *Instance* parameter in Related Definitions.

Arg5

This argument is interpreted as a pointer to type **CONST EFI_GUID**. See the *CallerId* parameter in Related Definitions.

Arg6

This argument is interpreted as pointer to type **CONST EFI_STATUS_CODE_DATA**. See the *Data* parameter in Related Definitions.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Related Definitions

```
typedef
EFI_STATUS
(EFIAPI *EFI_REPORT_STATUS_CODE) (
  IN EFI_STATUS_CODE_TYPE         Type,
  IN EFI_STATUS_CODE_VALUE        Value,
  IN UINT32                       Instance,
  IN CONST EFI_GUID               *CallerId  OPTIONAL,
  IN CONST EFI_STATUS_CODE_DATA   *Data      OPTIONAL
  );
```

Description

This function performs the equivalent operation as the **ReportStatusCode** function of the Status Code Runtime Protocol. If this function is called in virtual mode before any required mapping have been converted to virtual addresses, then **EFI_SAL_VIRTUAL_ADDRESS_ERROR** is returned. Otherwise, the one of the status codes defined in the **ReportStatusCode()** function of the Status Code Runtime Protocol is returned.

Status Codes Returned

EFI_SAL_VIRTUAL_ADDRESS_ERROR	This function was called in virtual mode before virtual mappings for the specified Extended SAL Procedure are available.
Other	See the return status codes for the ReportStatusCode() function in the Status Code Runtime Protocol.

10.4.7 Extended SAL Monotonic Counter Services Class

Summary

The Extended SAL Monotonic Counter Services Class provides functions to access the monotonic counter.

GUID

```
#define EFI_EXTENDED_SAL_MTC_SERVICES_PROTOCOL_GUID_LO \
```

```
  0x408b75e8899afd18
#define EFI_EXTENDED_SAL_MTC_SERVICES_PROTOCOL_GUID_HI \
  0x54f4cd7e2e6e1aa4
#define EFI_EXTENDED_SAL_MTC_SERVICES_PROTOCOL_GUID \
  {0x899afd18,0x75e8,0x408b,\
  {0xa4,0x1a,0x6e,0x2e,0x7e,0xcd,0xf4,0x54}}
```

Related Definitions

```
typedef enum {
  GetNextHighMotonicCountFunctionId,
} EFI_EXTENDED_SAL_MTC_SERVICES_FUNC_ID;
```

Description

Table 13. Extended SAL Monotonic Counter Services Class

Name	Description
ExtendedSalGetNextHighMtc	This function is equivalent in functionality to the EFI Runtime Service **`GetNextHighMonotonicCount()`**. See *UEFI 2.0 Specification* Section 7.4.2. The function prototype for the **`GetNextHighMonotonicCount()`** service is shown in Related Definitions.

ExtendedSalGetNextHighMtc

Summary

This function is equivalent in functionality to the EFI Runtime Service **GetNextHighMonotonicCount()**. See *UEFI 2.0 Specification* Section 7.4.2. The function prototype for the **GetNextHighMonotonicCount()** service is shown in Related Definitions.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalGetNextHighMtc (
  IN UINT64    FunctionId,
  IN UINT64    Arg2,
  IN UINT64    Arg3,
  IN UINT64    Arg4,
  IN UINT64    Arg5,
  IN UINT64    Arg6,
  IN UINT64    Arg7,
  IN UINT64    Arg8,
  IN BOOLEAN   VirtualMode,
  IN VOID      *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalGetNextHighMtcFunctionId**.

Arg2

This argument is interpreted as a pointer to a **UINT32**. See the *HighCount* parameter in Related Definitions.

Arg3

Reserved. Must be zero.

Arg4

Reserved. Must be zero.

Arg5

Reserved. Must be zero.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Related Definitions

```
typedef
EFI_STATUS
(EFIAPI *EFI_GET_NEXT_HIGH_MONO_COUNT) (
  OUT UINT32  *HighCount
  );
```

Description

This function performs the equivalent operation as the **GetNextHighMonotoicCount()** function in the EFI Runtime Services Table. If this function is called in virtual mode before any required mapping have been converted to virtual addresses, then **EFI_SAL_VIRTUAL_ADDRESS_ERROR** is returned. Otherwise, the one of the status codes defined in the **GetNextHighMonotonicCount()** function of the EFI Runtime Services Table is returned.

Status Codes Returned

EFI_SAL_VIRTUAL_ADDRESS_ERROR	This function was called in virtual mode before virtual mappings for the specified Extended SAL Procedure are available.
Other	See the return status codes for the GetNextHighMonotonicCount() function in the EFI Runtime Services Table.

10.4.8 Extended SAL Variable Services Class

Summary

The Extended SAL Variable Services Class provides functions to access EFI variables.

GUID

```
#define EFI_EXTENDED_SAL_VARIABLE_SERVICES_PROTOCOL_GUID_LO \
  0x4370c6414ecb6c53
#define EFI_EXTENDED_SAL_VARIABLE_SERVICES_PROTOCOL_GUID_HI \
  0x78836e490e3bb28c
#define EFI_EXTENDED_SAL_VARIABLE_SERVICES_PROTOCOL_GUID \
  {0x4ecb6c53,0xc641,0x4370, \
  {0x8c,0xb2,0x3b,0x0e,0x49,0x6e,0x83,0x78}}
```

Related Definitions

```
typedef enum {
  EsalGetVariableFunctionId,
  EsalGetNextVariableNameFunctionId,
  EsalSetVariableFunctionId,
  EsalQueryVariableInfoFunctionId,
} EFI_EXTENDED_SAL_VARIABLE_SERVICES_FUNC_ID;
```

Description

Table 14. Extended SAL Variable Services Class

Name	Description
ExtendedSalGetVariable	This function is equivalent in functionality to the EFI Runtime Service **`GetVariable()`**. See *UEFI 2.0 Specification* Section 7.1. The function prototype for the **`GetVariable()`** service is shown in Related Definitions.
ExtendedSalGetNextVariableName	This function is equivalent in functionality to the EFI Runtime Service **`GetNextVariableName()`**. See *UEFI 2.0 Specification* Section 7.1. The function prototype for the **`GetNextVariableName()`** service is shown in Related Definitions.
ExtendedSalSetVariable	This function is equivalent in functionality to the EFI Runtime Service **`SetVariable()`**. See *UEFI 2.0 Specification* Section 7.1. The function prototype for the **`SetVariable()`** service is shown in Related Definitions.
ExtendedSalQueryVariableInfo	This function is equivalent in functionality to the EFI Runtime Service **`QueryVariableInfo()`**. See *UEFI 2.0 Specification* Section 7.1. The function prototype for the **`QueryVariableInfo()`** service is shown in Related Definitions.

ExtendedSalGetVariable

Summary

This function is equivalent in functionality to the EFI Runtime Service **GetVariable()**. See *UEFI 2.0 Specification* Section 7.1. The function prototype for the **GetVariable()** service is shown in Related Definitions.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalGetVariable (
  IN UINT64    FunctionId,
  IN UINT64    Arg2,
  IN UINT64    Arg3,
  IN UINT64    Arg4,
  IN UINT64    Arg5,
  IN UINT64    Arg6,
  IN UINT64    Arg7,
  IN UINT64    Arg8,
  IN BOOLEAN   VirtualMode,
  IN VOID      *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalGetVariableFunctionId**.

Arg2

This argument is interpreted as a pointer to a Unicode string. See the *VariableName* parameter in Related Definitions.

Arg3

This argument is interpreted as a pointer to an **EFI_GUID**. See the *VendorGuid* parameter in Related Definitions.

Arg4

This argument is interpreted as a pointer to a value of type **UINT32**. See the *Attributes* parameter in Related Definitions.

Arg5

This argument is interpreted as a pointer to a value of type **UINTN**. See the *DataSize* parameter in Related Definitions.

Arg6

This argument is interpreted as a pointer to a buffer with type **VOID ***. See the *Data* parameter in Related Definitions.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Related Definitions

```
typedef
EFI_STATUS
(EFIAPI *EFI_GET_VARIABLE) (
  IN      CHAR16     *VariableName,
  IN      EFI_GUID   *VendorGuid,
  OUT     UINT32     *Attributes,     OPTIONAL
  IN OUT  UINTN      *DataSize,
  OUT     VOID       *Data
  );
```

Description

This function performs the equivalent operation as the **GetVariable()** function in the EFI Runtime Services Table. If this function is called in virtual mode before any required mapping have been converted to virtual addresses, then **EFI_SAL_VIRTUAL_ADDRESS_ERROR** is returned. Otherwise, the one of the status codes defined in the **GetVariable()** function of the EFI Runtime Services Table is returned.

Status Codes Returned

EFI_SAL_VIRTUAL_ADDRESS_ERROR	This function was called in virtual mode before virtual mappings for the specified Extended SAL Procedure are available.
Other	See the return status codes for the GetVariable() function in the EFI Runtime Services Table.

ExtendedSalGetNextVariableName

Summary

This function is equivalent in functionality to the EFI Runtime Service **GetNextVariableName()**. See *UEFI 2.0 Specification* Section 7.1. The function prototype for the **GetNextVariableName()** service is shown in Related Definitions.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalGetNextVariableName (
  IN UINT64    FunctionId,
  IN UINT64    Arg2,
  IN UINT64    Arg3,
  IN UINT64    Arg4,
  IN UINT64    Arg5,
  IN UINT64    Arg6,
  IN UINT64    Arg7,
  IN UINT64    Arg8,
  IN BOOLEAN   VirtualMode,
  IN VOID      *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalGetNextVariableNameFunctionId**.

Arg2

This argument is interpreted as a pointer to value of type **UINTN**. See the *VariableNameSize* parameter in Related Definitions.

Arg3

This argument is interpreted as a pointer to a Unicode string. See the *VendorName* parameter in Related Definitions.

Arg4

This argument is interpreted as a pointer to a value of type **EFI_GUID**. See the *VendorGuid* parameter in Related Definitions.

Arg5

Reserved. Must be zero.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Related Definitions

```
typedef
EFI_STATUS
(EFIAPI *EFI_GET_NEXT_VARIABLE_NAME) (
  IN OUT UINTN      *VariableNameSize,
  IN OUT CHAR16     *VariableName,
  IN OUT EFI_GUID   *VendorGuid
  );
```

Description

This function performs the equivalent operation as the **GetNextVariableName()** function in the EFI Runtime Services Table. If this function is called in virtual mode before any required mapping have been converted to virtual addresses, then **EFI_SAL_VIRTUAL_ADDRESS_ERROR** is returned. Otherwise, the one of the status codes defined in the **GetNextVariableName()** function of the EFI Runtime Services Table is returned.

Status Codes Returned

EFI_SAL_VIRTUAL_ADDRESS_ERROR	This function was called in virtual mode before virtual mappings for the specified Extended SAL Procedure are available.
Other	See the return status codes for the GetNextVariableName() function in the EFI Runtime Services Table.

ExtendedSalSetVariable

Summary

This function is equivalent in functionality to the EFI Runtime Service **SetVariable()**. See *UEFI 2.0 Specification* Section 7.1. The function prototype for the **SetVariable()** service is shown in Related Definitions.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalSetVariable (
  IN UINT64    FunctionId,
  IN UINT64    Arg2,
  IN UINT64    Arg3,
  IN UINT64    Arg4,
  IN UINT64    Arg5,
  IN UINT64    Arg6,
  IN UINT64    Arg7,
  IN UINT64    Arg8,
  IN BOOLEAN   VirtualMode,
  IN VOID      *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalSetVariableFunctionId**.

Arg2

This argument is interpreted as a pointer to a Unicode string. See the *VariableName* parameter in Related Definitions.

Arg3

This argument is interpreted as a pointer to an **EFI_GUID**. See the *VendorGuid* parameter in Related Definitions.

Arg4

This argument is interpreted as a value of type **UINT32**. See the *Attributes* parameter in Related Definitions.

Arg5

This argument is interpreted as a value of type **UINTN**. See the *DataSize* parameter in Related Definitions.

Arg6

This argument is interpreted as a pointer to a buffer with type **VOID ***. See the *Data* parameter in Related Definitions.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Related Definitions

```
typedef
EFI_STATUS
(EFIAPI *EFI_SET_VARIABLE) (
  IN  CHAR16     *VariableName,
  IN  EFI_GUID   *VendorGuid,
  IN  UINT32     Attributes,
  IN  UINTN      DataSize,
  IN  VOID       *Data
  );
```

Description

This function performs the equivalent operation as the **SetVariable()** function in the EFI Runtime Services Table. If this function is called in virtual mode before any required mapping have been converted to virtual addresses, then **EFI_SAL_VIRTUAL_ADDRESS_ERROR** is returned. Otherwise, the one of the status codes defined in the **SetVariable()** function of the EFI Runtime Services Table is returned.

Status Codes Returned

EFI_SAL_VIRTUAL_ADDRESS_ERROR	This function was called in virtual mode before virtual mappings for the specified Extended SAL Procedure are available.
Other	See the return status codes for the SetVariable() function in the EFI Runtime Services Table.

ExtendedSalQueryVariableInfo

Summary

This function is equivalent in functionality to the EFI Runtime Service **QueryVariableInfo()**. See *UEFI 2.0 Specification* Section 7.1. The function prototype for the **QueryVariableInfo()** service is shown in Related Definitions.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalQueryVariableInfo (
  IN UINT64    FunctionId,
  IN UINT64    Arg2,
  IN UINT64    Arg3,
  IN UINT64    Arg4,
  IN UINT64    Arg5,
  IN UINT64    Arg6,
  IN UINT64    Arg7,
  IN UINT64    Arg8,
  IN BOOLEAN   VirtualMode,
  IN VOID      *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalQueryVariableInfoFunctionId**.

Arg2

This argument is interpreted as a value of type **UINT32**. See the *Attributes* parameter in Related Definitions.

Arg3

This argument is interpreted as a pointer to a value of type **UINT64**. See the *MaximumVariableStorageSize* parameter in Related Definitions.

Arg4

This argument is interpreted as a pointer to a value of type **UINT64**. See the *RemainingVariableStorageSize* parameter in Related Definitions.

Arg5

This argument is interpreted as a pointer to a value of type **UINT64**. See the *MaximumVariableSize* parameter in Related Definitions.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Related Definitions

```
typedef
EFI_STATUS
(EFIAPI *EFI_QUERY_VARIABLE_INFO) (
  IN  UINT32     Attributes,
  OUT UINT64     *MaximumVariableStorageSize,
  OUT UINT64     *RemainingVariableStorageSize,
  OUT UINT64     *MaximumVariableSize
  );
```

Description

This function performs the equivalent operation as the **QueryVariableInfo()** function in the EFI Runtime Services Table. If this function is called in virtual mode before any required mapping have been converted to virtual addresses, then **EFI_SAL_VIRTUAL_ADDRESS_ERROR** is returned. Otherwise, the one of the status codes defined in the **QueryVariableInfo()** function of the EFI Runtime Services Table is returned.

Status Codes Returned

EFI_SAL_VIRTUAL_ADDRESS_ERROR	This function was called in virtual mode before virtual mappings for the specified Extended SAL Procedure are available.
Other	See the return status codes for the QueryVariableInfo() function in the EFI Runtime Services Table.

10.4.9 Extended SAL Firmware Volume Block Services Class

Summary

The Extended SAL Firmware Volume Block Services Class provides services that are equivalent to the Firmware Volume Block Protocol in the *Platform Initialization Specification*.

GUID

```
#define EFI_EXTENDED_SAL_FVB_SERVICES_PROTOCOL_GUID_LO \
  0x4f1dbcbba2271df1
#define EFI_EXTENDED_SAL_FVB_SERVICES_PROTOCOL_GUID_HI \
  0x1a072f17bc06a998
```

```
#define EFI_EXTENDED_SAL_FVB_SERVICES_PROTOCOL_GUID \
  {0xa2271df1,0xbcbb,0x4f1d,\
  {0x98,0xa9,0x06,0xbc,0x17,0x2f,0x07,0x1a}}
```

Related Definitions

```
typedef enum {
  ReadFunctionId,
  WriteFunctionId,
  EraseBlockFunctionId,
  GetVolumeAttributesFunctionId,
  SetVolumeAttributesFunctionId,
  GetPhysicalAddressFunctionId,
  GetBlockSizeFunctionId,
} EFI_EXTENDED_SAL_FV_BLOCK_SERVICES_FUNC_ID;
```

Description

Table 15. Extended SAL Variable Services Class

Name	Description
ExtendedSalRead	This function is equivalent in functionality to the **`Read()`** service of the EFI Firmware Volume Block Protocol. See Section 2.4 of the *Volume 3:Platform Initialization Specification, Shared Architectural Elements.* The function prototype for the **`Read()`** service is shown in Related Definitions.
ExtendedSalWrite	This function is equivalent in functionality to the **`Write()`** service of the EFI Firmware Volume Block Protocol. See Section 2.4 of the *Volume 3:Platform Initialization Specification, Shared Architectural Elements.* The function prototype for the **`Write()`** service is shown in Related Definitions.
ExtendedSalEraseBlock	This function is equivalent in functionality to the **`EraseBlocks()`** service of the EFI Firmware Volume Block Protocol except this function can only erase one block per request. See Section 2.4 of the *Volume 3:Platform Initialization Specification, Shared Architectural Elements.* The function prototype for the **`EraseBlock()`** service is shown in Related Definitions.
ExtendedSalGetAttributes	This function is equivalent in functionality to the **`GetAttributes()`** service of the EFI Firmware Volume Block Protocol. See Section 2.4 of the *Volume 3:Platform Initialization Specification, Shared Architectural Elements.* The function prototype for the **`GetAttributes()`** service is shown in Related Definitions.
ExtendedSalSetAttributes	This function is equivalent in functionality to the **`SetAttributes()`** service of the EFI Firmware Volume Block Protocol. See Section 2.4 of the *Volume 3:Platform Initialization Specification, Shared Architectural Elements*. The function prototype for the **`SetAttributes()`** service is shown in Related Definitions.

ExtendedSalGetPhysicalAddress	This function is equivalent in functionality to the **`GetPhysicalAddress()`** service of the EFI Firmware Volume Block Protocol. See Section 2.4 of the *Volume 3:Platform Initialization Specification, Shared Architectural Elements.* The function prototype for the **`GetPhysicalAddress()`** service is shown in Related Definitions.
ExtendedSalGetBlockSize	This function is equivalent in functionality to the **`GetBlockSize()`** service of the EFI Firmware Volume Block Protocol. See Section 2.4 of the *Volume 3:Platform Initialization Specification, Shared Architectural Elements*. The function prototype for the **`GetBlockSize()`** service is shown in Related Definitions.
ExtendedSalEraseCustomBlockRange	This function is similar in functionality to the **`EraseBlocks()`** service of the EFI Firmware Volume Block Protocol except this function can specify a range of blocks with offsets into the starting and ending block. See Section 2.4 of the *Volume 3:Platform Initialization Specification, Shared Architectural Elements*. The function prototype for the **`EraseBlock()`** service is shown in Related Definitions.

ExtendedSalRead

Summary

This function is equivalent in functionality to the **Read()** service of the EFI Firmware Volume Block Protocol. See Section 2.4 of the *Volume 3:Platform Initialization Specification, Shared Architectural Elements*. The function prototype for the **Read()** service is shown in Related Definitions.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalRead (
  IN UINT64    FunctionId,
  IN UINT64    Arg2,
  IN UINT64    Arg3,
  IN UINT64    Arg4,
  IN UINT64    Arg5,
  IN UINT64    Arg6,
  IN UINT64    Arg7,
  IN UINT64    Arg8,
  IN BOOLEAN   VirtualMode,
  IN VOID      *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalFvbReadFunctionId**.

Arg2

This argument is interpreted as type **UINTN** that represents the Firmware Volume Block instance. This instance value is used to lookup a **EFI_FIRMWARE_VOLUME_BLOCK_PROTOCOL**. See the *This* parameter in Related Definitions.

Arg3

This argument is interpreted as type **EFI_LBA**. See the *Lba* parameter in Related Definitions.

Arg4

This argument is interpreted as type **UINTN**. See the *Offset* parameter in Related Definitions.

Arg5

This argument is interpreted as a pointer to type **UINTN**. See the *NumBytes* parameter in Related Definitions.

Arg6

This argument is interpreted as pointer to a buffer of type **VOID *** . See the *Buffer* parameter in Related Definitions.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Related Definitions

```
typedef
EFI_STATUS
(EFIAPI *EFI_FVB_READ) (
  IN EFI_FIRMWARE_VOLUME_BLOCK_PROTOCOL   *This,
  IN EFI_LBA                              Lba,
  IN UINTN                                Offset,
  IN OUT UINTN                            *NumBytes,
  OUT UINT8                               *Buffer
  );
```

Description

This function performs the equivalent operation as the **Read()** function of the EFI Firmware Volume Block Protocol. If this function is called in virtual mode before any required mapping have been converted to virtual addresses, then **EFI_SAL_VIRTUAL_ADDRESS_ERROR** is returned. Otherwise, the one of the status codes defined in the **Read()** function of the EFI Firmware Volume Block Protocol is returned.

Status Codes Returned

EFI_SAL_VIRTUAL_ADDRESS_ERROR	This function was called in virtual mode before virtual mappings for the specified Extended SAL Procedure are available.
Other	See the return status codes for the Read() function in the EFI Firmware Volume Block Protocol.

ExtendedSalWrite

Summary

This function is equivalent in functionality to the **Write()** service of the EFI Firmware Volume Block Protocol. See Section 2.4 of the *Volume 3:Platform Initialization Specification, Shared Architectural Elements*. The function prototype for the **Write()** service is shown in Related Definitions.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalWrite (
  IN UINT64    FunctionId,
  IN UINT64    Arg2,
  IN UINT64    Arg3,
  IN UINT64    Arg4,
  IN UINT64    Arg5,
  IN UINT64    Arg6,
  IN UINT64    Arg7,
  IN UINT64    Arg8,
  IN BOOLEAN   VirtualMode,
  IN VOID      *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalFvbWriteFunctionId**.

Arg2

This argument is interpreted as type **UINTN** that represents the Firmware Volume Block instance. This instance value is used to lookup a **EFI_FIRMWARE_VOLUME_BLOCK_PROTOCOL**. See the *This* parameter in Related Definitions.

Arg3

This argument is interpreted as type **EFI_LBA**. See the *Lba* parameter in Related Definitions.

Arg4

This argument is interpreted as type **UINTN**. See the *Offset* parameter in Related Definitions.

Arg5

This argument is interpreted as a pointer to type **UINTN**. See the *NumBytes* parameter in Related Definitions.

Arg6

This argument is interpreted as pointer to a buffer of type **VOID *** . See the *Buffer* parameter in Related Definitions.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Related Definitions

```
typedef
EFI_STATUS
(EFIAPI *EFI_FVB_WRITE) (
  IN EFI_FIRMWARE_VOLUME_BLOCK_PROTOCOL   *This,
  IN EFI_LBA                              Lba,
  IN UINTN                                Offset,
  IN OUT UINTN                            *NumBytes,
  IN UINT8                                *Buffer
  );
```

Description

This function performs the equivalent operation as the **Write()** function of the EFI Firmware Volume Block Protocol. If this function is called in virtual mode before any required mapping have been converted to virtual addresses, then **EFI_SAL_VIRTUAL_ADDRESS_ERROR** is returned. Otherwise, the one of the status codes defined in the **Write()** function of the EFI Firmware Volume Block Protocol is returned.

Status Codes Returned

EFI_SAL_VIRTUAL_ADDRESS_ERROR	This function was called in virtual mode before virtual mappings for the specified Extended SAL Procedure are available.
Other	See the return status codes for the Write() function in the EFI Firmware Volume Block Protocol.

ExtendedSalEraseBlock

Summary

This function is equivalent in functionality to the **EraseBlocks()** service of the EFI Firmware Volume Block Protocol except this function can only erase one block per request. See Section 2.4 of the *Volume 3:Platform Initialization Specification, Shared Architectural Elements*. The function prototype for the **EraseBlock()** service is shown in Related Definitions.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalEraseBlock (
  IN UINT64    FunctionId,
  IN UINT64    Arg2,
  IN UINT64    Arg3,
  IN UINT64    Arg4,
  IN UINT64    Arg5,
  IN UINT64    Arg6,
  IN UINT64    Arg7,
  IN UINT64    Arg8,
  IN BOOLEAN   VirtualMode,
  IN VOID      *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalFvbEraseBlockFunctionId**.

Arg2

This argument is interpreted as type **UINTN** that represents the Firmware Volume Block instance. This instance value is used to lookup a **EFI_FIRMWARE_VOLUME_BLOCK_PROTOCOL**. See the *This* parameter in Related Definitions.

Arg3

This argument is interpreted as type **EFI_LBA**. This is the logical block address in the firmware volume to erase. Only a single block can be specified with this Extended SAL Procedure. The **EraseBlocks()** function in the EFI Firmware Volume Block Protocol supports a variable number of arguments that allow one or more block ranges to be specified.

Arg4

Reserved. Must be zero.

Arg5

Reserved. Must be zero.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Related Definitions

```
typedef
EFI_STATUS
(EFIAPI *EFI_FVB_ERASE_BLOCKS) (
  IN EFI_FIRMWARE_VOLUME_BLOCK_PROTOCOL  *This,
  ...
  );
```

Description

This function performs the equivalent operation as the **EraseBlock()** function of the EFI Firmware Volume Block Protocol. If this function is called in virtual mode before any required mapping have been converted to virtual addresses, then **EFI_SAL_VIRTUAL_ADDRESS_ERROR** is returned. Otherwise, the one of the status codes defined in the **EraseBlock()** function of the EFI Firmware Volume Block Protocol is returned.

Status Codes Returned

EFI_SAL_VIRTUAL_ADDRESS_ERROR	This function was called in virtual mode before virtual mappings for the specified Extended SAL Procedure are available.
Other	See the return status codes for the EraseBlock() function in the EFI Firmware Volume Block Protocol.

ExtendedSalGetAttributes

Summary

This function is equivalent in functionality to the **GetAttributes()** service of the EFI Firmware Volume Block Protocol. See Section 2.4 of the *Volume 3:Platform Initialization Specification, Shared Architectural Elements*. The function prototype for the **GetAttributes()** service is shown in Related Definitions.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalGetAttributes (
  IN UINT64    FunctionId,
  IN UINT64    Arg2,
  IN UINT64    Arg3,
  IN UINT64    Arg4,
  IN UINT64    Arg5,
  IN UINT64    Arg6,
  IN UINT64    Arg7,
  IN UINT64    Arg8,
  IN BOOLEAN   VirtualMode,
  IN VOID      *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalFvbGetAttributesFunctionId**.

Arg2

This argument is interpreted as type **UINTN** that represents the Firmware Volume Block instance. This instance value is used to lookup a **EFI_FIRMWARE_VOLUME_BLOCK_PROTOCOL**. See the *This* parameter in Related Definitions.

Arg3

This argument is interpreted as pointer to a value of type **EFI_FVB_ATTRIBUTES**. See the *Attributes* parameter in Related Definitions.

Arg4

Reserved. Must be zero.

Arg5

Reserved. Must be zero.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Related Definitions

```
EFI_STATUS
(EFIAPI *EFI_FVB_GET_ATTRIBUTES) (
  IN EFI_FIRMWARE_VOLUME_BLOCK_PROTOCOL   *This,
  OUT EFI_FVB_ATTRIBUTES                  *Attributes
  );
```

Description

This function performs the equivalent operation as the **GetAttributes()** function of the EFI Firmware Volume Block Protocol. If this function is called in virtual mode before any required mapping have been converted to virtual addresses, then **EFI_SAL_VIRTUAL_ADDRESS_ERROR** is returned. Otherwise, the one of the status codes defined in the **GetAttributes()** function of the EFI Firmware Volume Block Protocol is returned.

Status Codes Returned

EFI_SAL_VIRTUAL_ADDRESS_ERROR	This function was called in virtual mode before virtual mappings for the specified Extended SAL Procedure are available.
Other	See the return status codes for the GetAttributes() function in the EFI Firmware Volume Block Protocol.

ExtendedSalSetAttributes

Summary

This function is equivalent in functionality to the **SetAttributes()** service of the EFI Firmware Volume Block Protocol. See Section 2.4 of the *Volume 3:Platform Initialization Specification, Shared Architectural Elements*. The function prototype for the **SetAttributes()** service is shown in Related Definitions.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalSetAttributes (
  IN UINT64    FunctionId,
  IN UINT64    Arg2,
  IN UINT64    Arg3,
  IN UINT64    Arg4,
  IN UINT64    Arg5,
  IN UINT64    Arg6,
  IN UINT64    Arg7,
  IN UINT64    Arg8,
  IN BOOLEAN   VirtualMode,
  IN VOID      *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalFvbSetAttributesFunctionId**.

Arg2

This argument is interpreted as type **UINTN** that represents the Firmware Volume Block instance. This instance value is used to lookup a **EFI_FIRMWARE_VOLUME_BLOCK_PROTOCOL**. See the *This* parameter in Related Definitions.

Arg3

This argument is interpreted as pointer to a value of type **EFI_FVB_ATTRIBUTES**. See the *Attributes* parameter in Related Definitions.

Arg4

Reserved. Must be zero.

Arg5

Reserved. Must be zero.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Related Definitions

```
typedef
EFI_STATUS
(EFIAPI *EFI_FVB_SET_ATTRIBUTES) (
  IN EFI_FIRMWARE_VOLUME_BLOCK_PROTOCOL   *This,
  IN OUT EFI_FVB_ATTRIBUTES               *Attributes
  );
```

Description

This function performs the equivalent operation as the **SetAttributes()** function of the EFI Firmware Volume Block Protocol. If this function is called in virtual mode before any required mapping have been converted to virtual addresses, then **EFI_SAL_VIRTUAL_ADDRESS_ERROR** is returned. Otherwise, the one of the status codes defined in the **SetAttributes()** function of the EFI Firmware Volume Block Protocol is returned.

Status Codes Returned

EFI_SAL_VIRTUAL_ADDRESS_ERROR	This function was called in virtual mode before virtual mappings for the specified Extended SAL Procedure are available.
Other	See the return status codes for the SetAttributes() function in the EFI Firmware Volume Block Protocol.

ExtendedSalGetPhysicalAddress

Summary

This function is equivalent in functionality to the **GetPhysicalAddress()** service of the EFI Firmware Volume Block Protocol. See Section 2.4 of the *Volume 3:Platform Initialization Specification, Shared Architectural Elements*. The function prototype for the **GetPhysicalAddress()** service is shown in Related Definitions.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalGetPhysicalAddress (
  IN UINT64    FunctionId,
  IN UINT64    Arg2,
  IN UINT64    Arg3,
  IN UINT64    Arg4,
  IN UINT64    Arg5,
  IN UINT64    Arg6,
  IN UINT64    Arg7,
  IN UINT64    Arg8,
  IN BOOLEAN   VirtualMode,
  IN VOID      *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalFvbGetPhysicalAddressFunctionId**.

Arg2

This argument is interpreted as type **UINTN** that represents the Firmware Volume Block instance. This instance value is used to lookup a **EFI_FIRMWARE_VOLUME_BLOCK_PROTOCOL**. See the *This* parameter in Related Definitions.

Arg3

This argument is interpreted as pointer to a value of type **EFI_PHYSICAL_ADDRESS**. See the *Address* parameter in Related Definitions.

Arg4

Reserved. Must be zero.

Arg5

Reserved. Must be zero.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Related Definitions

```
typedef
EFI_STATUS
(EFIAPI *EFI_FVB_GET_PHYSICAL_ADDRESS) (
  IN EFI_FIRMWARE_VOLUME_BLOCK_PROTOCOL  *This,
  OUT EFI_PHYSICAL_ADDRESS               *Address
  );
```

Description

This function performs the equivalent operation as the **GetPhysicalAddress()** function of the EFI Firmware Volume Block Protocol. If this function is called in virtual mode before any required mapping have been converted to virtual addresses, then **EFI_SAL_VIRTUAL_ADDRESS_ERROR** is returned. Otherwise, the one of the status codes defined in the **GetPhysicalAddress()** function of the EFI Firmware Volume Block Protocol is returned.

Status Codes Returned

EFI_SAL_VIRTUAL_ADDRESS_ERROR	This function was called in virtual mode before virtual mappings for the specified Extended SAL Procedure are available.
Other	See the return status codes for the GetPhysicalAddress() function in the EFI Firmware Volume Block Protocol.

ExtendedSalGetBlockSize

Summary

This function is equivalent in functionality to the **GetBlockSize()** service of the EFI Firmware Volume Block Protocol. See Section 2.4 of the *Volume 3:Platform Initialization Specification, Shared Architectural Elements*. The function prototype for the **GetBlockSize()** service is shown in Related Definitions.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalGetBlockSize (
  IN UINT64    FunctionId,
  IN UINT64    Arg2,
  IN UINT64    Arg3,
  IN UINT64    Arg4,
  IN UINT64    Arg5,
  IN UINT64    Arg6,
  IN UINT64    Arg7,
  IN UINT64    Arg8,
  IN BOOLEAN   VirtualMode,
  IN VOID      *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalFvbGetBlockSizeFunctionId**.

Arg2

This argument is interpreted as type **UINTN** that represents the Firmware Volume Block instance. This instance value is used to lookup a **EFI_FIRMWARE_VOLUME_BLOCK_PROTOCOL**.

Arg3

This argument is interpreted as type **EFI_LBA**. See *Lba* parameter in Related Definitions.

Arg4 *T*

his argument is interpreted as a pointer to a value of type **UINTN**. See *BlockSize* parameter in Related Definitions.

Arg5

This argument is interpreted as a pointer to a value of type **UINTN**. See *NumberOfBlocks* parameter in Related Definitions.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Related Definitions

```
typedef
EFI_STATUS
(EFIAPI *EFI_FVB_GET_BLOCK_SIZE) (
  IN EFI_FIRMWARE_VOLUME_BLOCK_PROTOCOL   *This,
  IN EFI_LBA                              Lba,
  OUT UINTN                               *BlockSize,
  OUT UINTN                               *NumberOfBlocks
  );
```

Description

This function performs the equivalent operation as the **GetBlockSize()** function of the EFI Firmware Volume Block Protocol. If this function is called in virtual mode before any required mapping have been converted to virtual addresses, then **EFI_SAL_VIRTUAL_ADDRESS_ERROR** is returned. Otherwise, the one of the status codes defined in the **GetBlockSize()** function of the EFI Firmware Volume Block Protocol is returned.

Status Codes Returned

EFI_SAL_VIRTUAL_ADDRESS_ERROR	This function was called in virtual mode before virtual mappings for the specified Extended SAL Procedure are available.
Other	See the return status codes for the GetBlockSize() function in the EFI Firmware Volume Block Protocol.

ExtendedSalEraseCustomBlockRange

Summary

This function is similar in functionality to the **EraseBlocks()** service of the EFI Firmware Volume Block Protocol except this function can specify a range of blocks with offsets into the starting and ending block. See Section 2.4 of the *Volume 3:Platform Initialization Specification, Shared Architectural Elements*. The function prototype for the **EraseBlock()** service is shown in Related Definitions.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalEraseCustomBlockRange (
  IN UINT64    FunctionId,
  IN UINT64    Arg2,
  IN UINT64    Arg3,
  IN UINT64    Arg4,
  IN UINT64    Arg5,
  IN UINT64    Arg6,
  IN UINT64    Arg7,
  IN UINT64    Arg8,
  IN BOOLEAN   VirtualMode,
  IN VOID      *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalFvbEraseCustomBlockRangeFunctionId**.

Arg2

This argument is interpreted as type **UINTN** that represents the Firmware Volume Block instance. This instance value is used to lookup a **EFI_FIRMWARE_VOLUME_BLOCK_PROTOCOL**. See the *This* parameter in Related Definitions.

Arg3

This argument is interpreted as type **EFI_LBA**. This is the starting logical block address in the firmware volume to erase.

Arg4

This argument is interpreted as type **UINTN**. This is the offset into the starting logical block to erase.

Arg5

This argument is interpreted as type **EFI_LBA**. This is the ending logical block address in the firmware volume to erase.

Arg6

This argument is interpreted as type **UINTN**. This is the offset into the ending logical block to erase.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Related Definitions

```
typedef
EFI_STATUS
(EFIAPI *EFI_FVB_ERASE_BLOCKS) (
  IN EFI_FIRMWARE_VOLUME_BLOCK_PROTOCOL  *This,
  ...
  );
```

Description

This function performs a similar operation as the **EraseBlock()** function of the EFI Firmware Volume Block Protocol. The main difference is that this function can perform a partial erase of the starting and ending blocks. The start of the erase operation is specified by *Arg3* and *Arg4*. The end of the erase operation is specified by *Arg5* and *Arg6*. If this function is called in virtual mode before any required mapping have been converted to virtual addresses, then **EFI_SAL_VIRTUAL_ADDRESS_ERROR** is returned. Otherwise, the one of the status codes defined in the **EraseBlock()** function of the EFI Firmware Volume Block Protocol is returned.

Status Codes Returned

EFI_SAL_VIRTUAL_ADDRESS_ERROR	This function was called in virtual mode before virtual mappings for the specified Extended SAL Procedure are available.
Other	See the return status codes for the EraseBlock() function in the EFI Firmware Volume Block Protocol.

10.4.10 Extended SAL MCA Log Services Class

Summary

The Extended SAL MCA Log Services Class provides logging services for MCA events.

GUID

```
#define EFI_EXTENDED_SAL_MCA_LOG_SERVICES_PROTOCOL_GUID_LO \
  0x4c0338a3cb3fd86e
#define EFI_EXTENDED_SAL_MCA_LOG_SERVICES_PROTOCOL_GUID_HI \
  0x7aaba2a3cf905c9a
#define EFI_EXTENDED_SAL_MCA_LOG_SERVICES_PROTOCOL_GUID \
  {0xcb3fd86e,0x38a3,0x4c03,\
  {0x9a,0x5c,0x90,0xcf,0xa3,0xa2,0xab,0x7a}}
```

Related Definitions

```
typedef enum {
  SalGetStateInfoFunctionId,
  SalGetStateInfoSizeFunctionId,
  SalClearStateInfoFunctionId,
  SalGetStateBufferFunctionId,
  SalSaveStateBufferFunctionId,
} EFI_EXTENDED_SAL_MCA_LOG_SERVICES_FUNC_ID;
```

ExtendedSalGetStateInfo

Summary

This function is equivalent in functionality to the SAL Procedure **SAL_GET_STATE_INFO**. See the *Intel Itanium Processor Family System Abstraction Layer Specification* Chapter 9.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalGetStateInfo (
  IN UINT64    FunctionId,
  IN UINT64    Arg2,
  IN UINT64    Arg3,
  IN UINT64    Arg4,
  IN UINT64    Arg5,
  IN UINT64    Arg6,
  IN UINT64    Arg7,
  IN UINT64    Arg8,
  IN BOOLEAN   VirtualMode,
  IN VOID      *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalGetStateInfoFunctionId**.

Arg2

type parameter to **SAL_GET_STATE_INFO**.

Arg3

Reserved. Must be zero.

Arg4

memaddr parameter to **SAL_GET_STATE_INFO**.

Arg5

Reserved. Must be zero.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

ExtendedSalGetStateInfoSize

Summary

This function is equivalent in functionality to the SAL Procedure **SAL_GET_STATE_INFO_SIZE**. See the *Intel Itanium Processor Family System Abstraction Layer Specification* Chapter 9.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalGetStateInfoSize (
  IN UINT64    FunctionId,
  IN UINT64    Arg2,
  IN UINT64    Arg3,
  IN UINT64    Arg4,
  IN UINT64    Arg5,
  IN UINT64    Arg6,
  IN UINT64    Arg7,
  IN UINT64    Arg8,
  IN BOOLEAN   VirtualMode,
  IN VOID      *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalGetStateInfoSizeFunctionId**.

Arg2

type parameter to **SAL_GET_STATE_INFO_SIZE**.

Arg3

Reserved. Must be zero.

Arg4

Reserved. Must be zero.

Arg5

Reserved. Must be zero.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

ExtendedSalClearStateInfo

Summary

This function is equivalent in functionality to the SAL Procedure **SAL_CLEAR_STATE_INFO**. See the *Intel Itanium Processor Family System Abstraction Layer Specification* Chapter 9.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalClearStateInfo (
  IN UINT64    FunctionId,
  IN UINT64    Arg2,
  IN UINT64    Arg3,
  IN UINT64    Arg4,
  IN UINT64    Arg5,
  IN UINT64    Arg6,
  IN UINT64    Arg7,
  IN UINT64    Arg8,
  IN BOOLEAN   VirtualMode,
  IN VOID      *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalGetStateInfoFunctionId**.

Arg2

type parameter to **SAL_CLEAR_STATE_INFO**.

Arg3

Reserved. Must be zero.

Arg4

Reserved. Must be zero.

Arg5

Reserved. Must be zero.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

ExtendedSalGetStateBuffer

Summary

Returns a memory buffer to store error records.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalGetStateBuffer (
  IN UINT64    FunctionId,
  IN UINT64    Arg2,
  IN UINT64    Arg3,
  IN UINT64    Arg4,
  IN UINT64    Arg5,
  IN UINT64    Arg6,
  IN UINT64    Arg7,
  IN UINT64    Arg8,
  IN BOOLEAN   VirtualMode,
  IN VOID      *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalGetStateBufferFunctionId**.

Arg2

Same as *type* parameter to **SAL_GET_STATE_INFO**.

Arg3

Reserved. Must be zero.

Arg4

Reserved. Must be zero.

Arg5

Reserved. Must be zero.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Description

This function returns a memory buffer to store error records. The base address of the buffer is returned in **SAL_RETURN_REGS.***r9*, and the size of the buffer, in bytes, is returned in **SAL_RETURN_REGS.***r10*. If a buffer is not available, then **EFI_OUT_OF_RESOURCES** is returned. Otherwise, **EFI_SUCCESS** is returned.

Status Codes Returned

EFI_SUCCESS	The memory buffer to store error records was returned in r9 and r10.
EFI_OUT_OF_RESOURCES	A memory buffer for string error records in not available.

ExtendedSalSaveStateBuffer

Summary

Saves a memory buffer containing an error records to nonvolatile storage.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalSaveStateBuffer (
  IN UINT64    FunctionId,
  IN UINT64    Arg2,
  IN UINT64    Arg3,
  IN UINT64    Arg4,
  IN UINT64    Arg5,
  IN UINT64    Arg6,
  IN UINT64    Arg7,
  IN UINT64    Arg8,
  IN BOOLEAN   VirtualMode,
  IN VOID      *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalSaveStateBufferFunctionId**.

Arg2

Same as *type* parameter to **SAL_GET_STATE_INFO**.

Arg3

Reserved. Must be zero.

Arg4

Reserved. Must be zero.

Arg5

Reserved. Must be zero.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Description

This function saved a memory buffer containing an error record to nonvolatile storage.

Status Codes Returned

EFI_SUCCESS	The memory buffer containing the error record was written to nonvolatile storage.
TBD	

10.4.11 Extended SAL Base Services Class

Summary

The Extended SAL Base Services Class provides base services that do not have any hardware dependencies including a number of SAL Procedures required by the *Intel Itanium Processor Family System Abstraction Layer Specification*.

GUID

```
#define EFI_EXTENDED_SAL_BASE_SERVICES_PROTOCOL_GUID_LO \
  0x41c30fe0d9e9fa06
#define EFI_EXTENDED_SAL_BASE_SERVICES_PROTOCOL_GUID_HI \
  0xf894335a4283fb96
#define EFI_EXTENDED_SAL_BASE_SERVICES_PROTOCOL_GUID \
  {0xd9e9fa06,0x0fe0,0x41c3,\
  {0x96,0xfb,0x83,0x42,0x5a,0x33,0x94,0xf8}}
```

Related Definitions

```
typedef enum {
  SalSetVectorsFunctionId,
  SalMcRendezFunctionId,
  SalMcSetParamsFunctionId,
  EsalGetVectorsFunctionId,
  EsalMcGetParamsFunctionId,
  EsalMcGetMcParamsFunctionId,
  EsalGetMcCheckinFlagsFunctionId,
  EsalGetPlatformBaseFreqFunctionId,
  EsalRegisterPhysicalAddrFunctionId,
  EsalBaseClassMaxFunctionId
} EFI_EXTENDED_SAL_BASE_SERVICES_FUNC_ID;
```

Description

Table 16. Extended SAL MP Services Class

Name	Description
ExtendedSalSetVectors	This function is equivalent in functionality to the SAL Procedure **SAL_SET_VECTORS**. See the *Intel Itanium Processor Family System Abstraction Layer Specification* Chapter 9.
ExtendedSalMcRendez	This function is equivalent in functionality to the SAL Procedure **SAL_MC_RENDEZ**. See the *Intel Itanium Processor Family System Abstraction Layer Specification* Chapter 9.
ExtendedSalMcSetParams	This function is equivalent in functionality to the SAL Procedure **SAL_MC_SET_PARAMS**. See the *Intel Itanium Processor Family System Abstraction Layer Specification* Chapter 9.
ExtendedSalGetVectors	Retrieves information that was previously registered with the SAL Procedure **SAL_SET_VECTORS**.
ExtendedSalMcGetParams	Retrieves information that was previously registered with the SAL Procedure **SAL_MC_SET_PARAMS**.
ExtendedSalMcGetMcParams	Retrieves information that was previously registered with the SAL Procedure **SAL_MC_SET_PARAMS**.
ExtendedSalGetMcCheckinFlags	Used to determine if a specific CPU has called the SAL Procedure **SAL_MC_RENDEZ**.
ExtendedSalGetPlatformBaseFreq	This function is equivalent in functionality to the SAL Procedure **SAL_FREQ_BASE** with a clock_type of 0. See the *Intel Itanium Processor Family System Abstraction Layer Specification* Chapter 9.
ExtendedSalRegisterPhysicalAddr	This function is equivalent in functionality to the SAL Procedure **SAL_REGISTER_PHYSICAL_ADDR**. See the *Intel Itanium Processor Family System Abstraction Layer Specification* Chapter 9.

ExtendedSalSetVectors

Summary

This function is equivalent in functionality to the SAL Procedure **SAL_SET_VECTORS**. See the *Intel Itanium Processor Family System Abstraction Layer Specification* Chapter 9.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalSetVectors (
  IN UINT64    FunctionId,
  IN UINT64    Arg2,
  IN UINT64    Arg3,
  IN UINT64    Arg4,
  IN UINT64    Arg5,
  IN UINT64    Arg6,
  IN UINT64    Arg7,
  IN UINT64    Arg8,
  IN BOOLEAN   VirtualMode,
  IN VOID      *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalSetVectorsFunctionId**.

Arg2

vector_type parameter to **SAL_SET_VECTORS**.

Arg3

phys_addr_1 parameter to **SAL_SET_VECTORS**.

Arg4

gp_1 parameter to **SAL_SET_VECTORS**.

Arg5

length_cs_1 parameter to **SAL_SET_VECTORS**.

Arg6

phys_addr_2 parameter to **SAL_SET_VECTORS**.

Arg7

gp_2 parameter to **SAL_SET_VECTORS**.

Arg8

length_cs_2 parameter to **SAL_SET_VECTORS**.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

ExtendedSalMcRendez

Summary

This function is equivalent in functionality to the SAL Procedure **SAL_MC_RENDEZ**. See the *Intel Itanium Processor Family System Abstraction Layer Specification* Chapter 9.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalMcRendez (
  IN UINT64    FunctionId,
  IN UINT64    Arg2,
  IN UINT64    Arg3,
  IN UINT64    Arg4,
  IN UINT64    Arg5,
  IN UINT64    Arg6,
  IN UINT64    Arg7,
  IN UINT64    Arg8,
  IN BOOLEAN   VirtualMode,
  IN VOID      *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalMcRendezFunctionId**.

Arg2

Reserved. Must be zero.

Arg3

Reserved. Must be zero.

Arg4

Reserved. Must be zero.

Arg5

Reserved. Must be zero.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

ExtendedSalMcSetParams

Summary

This function is equivalent in functionality to the SAL Procedure **SAL_MC_SET_PARAMS**. See the *Intel Itanium Processor Family System Abstraction Layer Specification* Chapter 9.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalMcSetParams (
  IN UINT64    FunctionId,
  IN UINT64    Arg2,
  IN UINT64    Arg3,
  IN UINT64    Arg4,
  IN UINT64    Arg5,
  IN UINT64    Arg6,
  IN UINT64    Arg7,
  IN UINT64    Arg8,
  IN BOOLEAN   VirtualMode,
  IN VOID      *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalMcSetParamsFunctionId**.

Arg2

param_type parameter to **SAL_MC_SET_PARAMS**.

Arg3

i_or_m parameter to **SAL_MC_SET_PARAMS**.

Arg4

i_or_m_val parameter to **SAL_MC_SET_PARAMS**.

Arg5

time_out parameter to **SAL_MC_SET_PARAMS**.

Arg6

mca_opt parameter to **SAL_MC_SET_PARAMS**.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

ExtendedSalGetVectors

Summary

Retrieves information that was previously registered with the SAL Procedure **SAL_SET_VECTORS**.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalGetVectors (
  IN UINT64    FunctionId,
  IN UINT64    Arg2,
  IN UINT64    Arg3,
  IN UINT64    Arg4,
  IN UINT64    Arg5,
  IN UINT64    Arg6,
  IN UINT64    Arg7,
  IN UINT64    Arg8,
  IN BOOLEAN   VirtualMode,
  IN VOID      *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalGetVectorsFunctionId**.

Arg2

The vector type to retrieve. 0 – MCA, 1-BSP INIT, 2 – BOOT_RENDEZ, 3 – AP INIT.

Arg3

Reserved. Must be zero.

Arg4

Reserved. Must be zero.

Arg5

Reserved. Must be zero.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Description

This function returns the vector information for the vector specified by *Arg2*. If the specified vector was not previously registered with the SAL Procedure **SAL_SET_VECTORS**, then **SAL_NO_INFORMATION_AVAILABLE** is returned. Otherwise, the physical address of the requested vector is returned in **SAL_RETURN_REGS**.*r9*, the global pointer(GP) value is returned in **SAL_RETURN_REGS**.*r10*, the length and checksum information is returned in **SAL_RETURN_REGS**.*r10*, and **EFI_SUCCESS** is returned.

Status Codes Returned

EFI_SUCCESS	The information for the requested vector was returned,
SAL_NO_INFORMATION_AVAILABLE	The requested vector has not been registered with the SAL Procedure SAL_SET_VECTORS.

ExtendedSalMcGetParams

Summary

Retrieves information that was previously registered with the SAL Procedure **SAL_MC_SET_PARAMS**.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalMcGetParams (
  IN UINT64    FunctionId,
  IN UINT64    Arg2,
  IN UINT64    Arg3,
  IN UINT64    Arg4,
  IN UINT64    Arg5,
  IN UINT64    Arg6,
  IN UINT64    Arg7,
  IN UINT64    Arg8,
  IN BOOLEAN   VirtualMode,
  IN VOID      *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalMcGetParamsFunctionId**.

Arg2

The parameter type to retrieve. 1 – rendezvous interrupt, 2 – wake up, 3 – Corrected Platform Error Vector.

Arg3

Reserved. Must be zero.

Arg4

Reserved. Must be zero.

Arg5

Reserved. Must be zero.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Description

This function returns information for the parameter type specified by *Arg2* that was previously registered with the SAL Procedure **SAL_MC_SET_PARAMS**. If the parameter type specified by *Arg2* was not previously registered with the SAL Procedure **SAL_MC_SET_PARAMS**, then **SAL_NO_INFORMATION_AVAILABLE** is returned. Otherwise, the **i_or_m** value is returned in **SAL_RETURN_REGS**.*r9*, the **i_or_m_val** value is returned in **SAL_RETURN_REGS**.*r10*, and **EFI_SUCCESS** is returned.

Status Codes Returned

EFI_SUCCESS	The information for the requested vector was returned,
SAL_NO_INFORMATION_AVAILABLE	The requested vector has not been registered with the SAL Procedure SAL_SET_VECTORS.

ExtendedSalMcGetMcParams

Summary

Retrieves information that was previously registered with the SAL Procedure **SAL_MC_SET_PARAMS**.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalMcGetMcParams (
  IN UINT64    FunctionId,
  IN UINT64    Arg2,
  IN UINT64    Arg3,
  IN UINT64    Arg4,
  IN UINT64    Arg5,
  IN UINT64    Arg6,
  IN UINT64    Arg7,
  IN UINT64    Arg8,
  IN BOOLEAN   VirtualMode,
  IN VOID      *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalMcGetMcParamsFunctionId**.

Arg2

Reserved. Must be zero.

Arg3

Reserved. Must be zero.

Arg4

Reserved. Must be zero.

Arg5

Reserved. Must be zero.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Description

This function returns information that was previously registered with the SAL Procedure **SAL_MC_SET_PARAMS**. If the information was not previously registered with the SAL Procedure **SAL_MC_SET_PARAMS**, then **SAL_NO_INFORMATION_AVAILABLE** is returned. Otherwise, the **rz_always** value is returned in **SAL_RETURN_REGS**.*r9*, **time_out** value is returned in **SAL_RETURN_REGS**.*r10*, **binit_escalate** value is returned in **SAL_RETURN_REGS.***r11*.

Status Codes Returned

EFI_SUCCESS	The information for the requested vector was returned,
SAL_NO_INFORMATION_AVAILABLE	The requested vector has not been registered with the SAL Procedure SAL_SET_VECTORS.

ExtendedSalGetMcCheckinFlags

Summary

Used to determine if a specific CPU has called the SAL Procedure **SAL_MC_RENDEZ**.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalMcGetMcCheckinFlags (
  IN UINT64    FunctionId,
  IN UINT64    Arg2,
  IN UINT64    Arg3,
  IN UINT64    Arg4,
  IN UINT64    Arg5,
  IN UINT64    Arg6,
  IN UINT64    Arg7,
  IN UINT64    Arg8,
  IN BOOLEAN   VirtualMode,
  IN VOID      *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalMcGetMcChckinFlagsFunctionId**.

Arg2

The index of the CPU in the set of enabled CPUs to check.

Arg3

Reserved. Must be zero.

Arg4

Reserved. Must be zero.

Arg5

Reserved. Must be zero.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Description

This function check to see if the CPU index specified by *Arg2* has called the SAL Procedure **SAL_MC_RENDEZ**. The CPU index values are assigned by the Extended SAL MP Services Class. If the CPU specified by *Arg2* has called the SAL Procedure **SAL_MC_RENDEZ**, then 1 is returned in **SAL_RETURN_REGS**.*r9*. Otherwise, **SAL_RETURN_REGS**.*r9* is set to 0. **EFI_SAL_SUCCESS** is always returned.

Status Codes Returned

EFI_SAL_SUCCESS	The checkin status of the requested CPU was returned.

ExtendedSalGetPlatformBaseFreq

Summary

This function is equivalent in functionality to the SAL Procedure **SAL_FREQ_BASE** with a clock_type of 0. See the *Intel Itanium Processor Family System Abstraction Layer Specification* Chapter 9.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalMcGetPlatformBaseFreq (
  IN UINT64    FunctionId,
  IN UINT64    Arg2,
  IN UINT64    Arg3,
  IN UINT64    Arg4,
  IN UINT64    Arg5,
  IN UINT64    Arg6,
  IN UINT64    Arg7,
  IN UINT64    Arg8,
  IN BOOLEAN   VirtualMode,
  IN VOID      *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalMcGetPlatformBaseFreqFunctionId**.

Arg2

Reserved. Must be zero.

Arg3

Reserved. Must be zero.

Arg4

Reserved. Must be zero.

Arg5

Reserved. Must be zero.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8 *Reserved. Must be zero.*

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended

SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

ExtendedSalRegisterPhysicalAddr

Summary

This function is equivalent in functionality to the SAL Procedure **SAL_REGISTER_PHYSICAL_ADDR**. See the *Intel Itanium Processor Family System Abstraction Layer Specification* Chapter 9.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalRegisterPhysicalAddr (
  IN UINT64    FunctionId,
  IN UINT64    Arg2,
  IN UINT64    Arg3,
  IN UINT64    Arg4,
  IN UINT64    Arg5,
  IN UINT64    Arg6,
  IN UINT64    Arg7,
  IN UINT64    Arg8,
  IN BOOLEAN   VirtualMode,
  IN VOID      *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalRegisterPhysicalAddrFunctionId**.

Arg2

phys_entity parameter to **SAL_REGISTER_PHYSICAL_ADDRESS**.

Arg3

paddr parameter to **SAL_REGISTER_PHYSICAL_ADDRESS**.

Arg4

Reserved. Must be zero.

Arg5

Reserved. Must be zero.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

10.4.12 Extended SAL MP Services Class

Summary

The Extended SAL MP Services Class provides services for managing multiple CPUs.

GUID

```
#define EFI_EXTENDED_SAL_MP_SERVICES_PROTOCOL_GUID_LO \
  0x4dc0cf18697d81a2
#define EFI_EXTENDED_SAL_MP_SERVICES_PROTOCOL_GUID_HI \
  0x3f8a613b11060d9e
#define EFI_EXTENDED_SAL_MP_SERVICES_PROTOCOL_GUID \
  {0x697d81a2,0xcf18,0x4dc0,\
  {0x9e,0x0d,0x06,0x11,0x3b,0x61,0x8a,0x3f}}
```

Related Definitions

```
typedef enum {
  AddCpuDataFunctionId,
  RemoveCpuDataFunctionId,
  ModifyCpuDataFunctionId,
  GetCpuDataByIdFunctionId,
  GetCpuDataByIndexFunctionId,
  SendIpiFunctionId,
  CurrentProcInfoFunctionId,
  NumProcessorsFunctionId,
  SetMinStateFunctionId,
  GetMinStateFunctionId,
  EsalPhysicalIdInfo,
} EFI_EXTENDED_SAL_MP_SERVICES_FUNC_ID;
```

Description

Table 17. Extended SAL MP Services Class

Name	Description
ExtendedSalAddCpuData	Add a CPU to the database of CPUs.
ExtendedSalRemoveCpuData	Add a CPU to the database of CPUs.
ExtendedSalModifyCpuData	Updates the data for a CPU that is already in the database of CPUs.
ExtendedSalGetCpuDataById	Returns the information on a CPU specified by a Global ID.

ExtendedSalGetCpuDataByIndex	Returns information on a CPU specified by an index.
ExtendedSalWhoAmI	Returns the Global ID for the calling CPU.
ExtendedSalNumProcessors	Returns the number of currently enabled CPUs, the total number of CPUs, and the maximum number of CPUs that the platform supports.
ExtendedSalSetMinState	Sets the MINSTATE pointer for the CPU specified by a Global ID.
ExtendedSalGetMinState	Retrieves the MINSTATE pointer for the CPU specified by a Global ID.
ExtendedSalPhysicalIdInfo	Retrieves the Physical ID of a CPU in the platform.

ExtendedSalAddCpuData

Summary

Add a CPU to the database of CPUs.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalAddCpuData (
  IN UINT64    FunctionId,
  IN UINT64    Arg2,
  IN UINT64    Arg3,
  IN UINT64    Arg4,
  IN UINT64    Arg5,
  IN UINT64    Arg6,
  IN UINT64    Arg7,
  IN UINT64    Arg8,
  IN BOOLEAN   VirtualMode,
  IN VOID      *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalAddCpuDataFunctionId**.

Arg2

The 64-bit Global ID of the CPU being added.

Arg3

The enable flag for the CPU being added. This value is interpreted as type **BOOLEAN**. **TRUE** means the CPU is enabled. **FALSE** means the CPU is disabled.

Arg4 *T*

he PAL Compatibility value for the CPU being added.

Arg5

The 16-bit Platform ID of the CPU being added.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Description

This function adds the CPU with a Global ID specified by *Arg2*, the enable flag specified by *Arg3*, and the PAL Compatibility value specified by *Arg4* to the database of CPUs in the platform. If there are not enough resource available to add the CPU, then **EFI_SAL_NOT_ENOUGH_SCRATCH** is returned. Otherwise, the CPU to added to the database, and **EFI_SAL_SUCCESS** is returned.

Status Codes Returned

EFI_SAL_SUCCESS	The CPU was added to the database.
EFI_SAL_NOT_ENOUGH_SCRATCH	There are not enough resource available to add the CPU.

ExtendedSalRemoveCpuData

Summary

Add a CPU to the database of CPUs.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalRemoveCpuData (
  IN UINT64    FunctionId,
  IN UINT64    Arg2,
  IN UINT64    Arg3,
  IN UINT64    Arg4,
  IN UINT64    Arg5,
  IN UINT64    Arg6,
  IN UINT64    Arg7,
  IN UINT64    Arg8,
  IN BOOLEAN   VirtualMode,
  IN VOID      *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalRemoveCpuDataFunctionId**.

Arg2

The 64-bit Global ID of the CPU being added.

Arg3

Reserved. Must be zero.

Arg4

Reserved. Must be zero.

Arg5

Reserved. Must be zero.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Description

This function removes the CPU with a Global ID specified by *Arg2* from the database of CPUs in the platform. If the CPU specified by *Arg2* is not present in the database, then **EFI_SAL_NO_INFORMATION** is returned. Otherwise, the CPU specified by *Arg2* is removed from the database of CPUs, and **EFI_SAL_SUCCESS** is returned.

Status Codes Returned

EFI_SAL_SUCCESS	The CPU was removed from the database.
EFI_SAL_NO_INFORMATION	The specified CPU is not in the database.

ExtendedSalModifyCpuData

Summary

Updates the data for a CPU that is already in the database of CPUs.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalModifyCpuData (
  IN UINT64    FunctionId,
  IN UINT64    Arg2,
  IN UINT64    Arg3,
  IN UINT64    Arg4,
  IN UINT64    Arg5,
  IN UINT64    Arg6,
  IN UINT64    Arg7,
  IN UINT64    Arg8,
  IN BOOLEAN   VirtualMode,
  IN VOID      *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalModifyCpuDataFunctionId**.

Arg2

The 64-bit Global ID of the CPU being updated.

Arg3

The enable flag for the CPU being updated. This value is interpreted as type **BOOLEAN**. **TRUE** means the CPU is enabled. **FALSE** means the CPU is disabled.

Arg4

The PAL Compatibility value for the CPU being updated.

Arg5

The 16-bit Platform ID of the CPU being updated.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Description

This function updates the CPU with a Global ID specified by *Arg2*, the enable flag specified by *Arg3*, and the PAL Compatibility value specified by *Arg4* in the database of CPUs in the platform. If the CPU specified by *Arg2* is not present in the database, then **EFI_SAL_NO_INFORMATION** is returned. Otherwise, the CPU specified by *Arg2* is updates with the enable flag specified by *Arg3* and the PAL Compatibility value specified by *Arg4*, and **EFI_SAL_SUCCESS** is returned.

Status Codes Returned

EFI_SAL_SUCCESS	The CPU database was updated.
EFI_SAL_NO_INFORMATION	The specified CPU is not in the database.

ExtendedSalGetCpuDataById

Summary

Returns the information on a CPU specified by a Global ID.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalGetCpuDataById (
  IN UINT64    FunctionId,
  IN UINT64    Arg2,
  IN UINT64    Arg3,
  IN UINT64    Arg4,
  IN UINT64    Arg5,
  IN UINT64    Arg6,
  IN UINT64    Arg7,
  IN UINT64    Arg8,
  IN BOOLEAN   VirtualMode,
  IN VOID      *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalGetCpuDataByIdFunctionId**.

Arg2

The 64-bit Global ID of the CPU to lookup.

Arg3 *T*

his parameter is interpreted as a **BOOLEAN** value. If **TRUE**, then the index in the set of enabled CPUs in the database is returned. If **FALSE**, then the index in the set of all CPUs in the database is returned.

Arg4

Reserved. Must be zero.

Arg5

Reserved. Must be zero.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Description

This function looks up the CPU specified by *Arg2* in the CPU database and returns the enable status and PAL Compatibility value. If the CPU specified by *Arg2* is not present in the database, then **EFI_SAL_NO_INFORMATION** is returned. Otherwise, the enable status is returned in **SAL_RETURN_REGS.***r9*, the PAL Compatibility value is returned in **SAL_RETURN_REGS**.*r10*, and **EFI_SAL_SUCCESS** is returned. If *Arg3* is **TRUE**, then the index of the CPU specified by *Arg2* in the set of enabled CPUs is returned in **SAL_RETURN_REGS**.*r11*. If *Arg3* is **FALSE**, then the index of the CPU specified by *Arg2* in the set of all CPUs is returned in **SAL_RETURN_REGS**.*r11*.

Status Codes Returned

EFI_SAL_SUCCESS	The information on the specified CPU was returned.
EFI_SAL_NO_INFORMATION	The specified CPU is not in the database.

ExtendedSalGetCpuDataByIndex

Summary

Returns information on a CPU specified by an index.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalGetCpuDataByIndex (
  IN UINT64    FunctionId,
  IN UINT64    Arg2,
  IN UINT64    Arg3,
  IN UINT64    Arg4,
  IN UINT64    Arg5,
  IN UINT64    Arg6,
  IN UINT64    Arg7,
  IN UINT64    Arg8,
  IN BOOLEAN   VirtualMode,
  IN VOID      *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalGetCpuDataByIndexFunctionId**.

Arg2

The index of the CPU to lookup.

Arg3

This parameter is interpreted as a **BOOLEAN** value. If **TRUE**, then the index in *Arg2* is the index in the set of enabled CPUs. If **FALSE**, then the index in *Arg2* is the index in the set of all CPUs.

Arg4

Reserved. Must be zero.

Arg5

Reserved. Must be zero.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Description

This function looks up the CPU specified by *Arg2* in the CPU database and returns the enable status and PAL Compatibility value. If the CPU specified by *Arg2* is not present in the database, then **EFI_SAL_NO_INFORMATION** is returned. Otherwise, the enable status is returned in **SAL_RETURN_REGS**.*r9*, the PAL Compatibility value is returned in **SAL_RETURN_REGS**.*r10*, the Global ID is returned in **SAL_RETURN_REGS**.*r11*, and **EFI_SAL_SUCCESS** is returned. If *Arg3* is **TRUE**, then *Arg2* is the index in the set of enabled CPUs. If *Arg3* is **FALSE**, then *Arg2* is the index in the set of all CPUs.

Status Codes Returned

EFI_SAL_SUCCESS	The information on the specified CPU was returned.
EFI_SAL_NO_INFORMATION	The specified CPU is not in the database.

ExtendedSalWhoiAml

Summary

Returns the Global ID for the calling CPU.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalWhoAmI (
  IN UINT64    FunctionId,
  IN UINT64    Arg2,
  IN UINT64    Arg3,
  IN UINT64    Arg4,
  IN UINT64    Arg5,
  IN UINT64    Arg6,
  IN UINT64    Arg7,
  IN UINT64    Arg8,
  IN BOOLEAN   VirtualMode,
  IN VOID      *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalWhoAmIFunctionId**.

Arg2 *T*

his parameter is interpreted as a **BOOLEAN** value. If **TRUE**, then the index in the set of enabled CPUs in the database is returned. If **FALSE**, then the index in the set of all CPUs in the database is returned.

Arg3

Reserved. Must be zero.

Arg4

Reserved. Must be zero.

Arg5

Reserved. Must be zero.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Description

This function looks up the Global ID of the calling CPU. If the calling CPU is not present in the database, then **EFI_SAL_NO_INFORMATION** is returned. Otherwise, the Global ID is returned in **SAL_RETURN_REGS**.*r9*, the PAL Compatibility value is returned in **SAL_RETURN_REGS**.*r10*, and **EFI_SAL_SUCCESS** is returned. If *Arg2* is **TRUE**, then the index of the calling CPU in the set of enabled CPUs is returned in **SAL_RETURN_REGS**.*r11*. If *Arg3* is **FALSE**, then the index of the calling CPU in the set of all CPUs is returned in **SAL_RETURN_REGS**.*r11*.

Status Codes Returned

EFI_SAL_SUCCESS	The Global ID for the calling CPU was returned.
EFI_SAL_NO_INFORMATION	The calling CPU is not in the database.

ExtendedSalNumProcessors

Summary

Returns the number of currently enabled CPUs, the total number of CPUs, and the maximum number of CPUs that the platform supports.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalNumProcessors (
  IN UINT64    FunctionId,
  IN UINT64    Arg2,
  IN UINT64    Arg3,
  IN UINT64    Arg4,
  IN UINT64    Arg5,
  IN UINT64    Arg6,
  IN UINT64    Arg7,
  IN UINT64    Arg8,
  IN BOOLEAN   VirtualMode,
  IN VOID      *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalNumProcessorsFunctionId**.

Arg2

Reserved. Must be zero.

Arg3

Reserved. Must be zero.

Arg4

Reserved. Must be zero.

Arg5

Reserved. Must be zero.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Description

This function returns the maximum number of CPUs that the platform supports in **SAL_RETURN_REGS**.*r9*, the total number of CPUs in **SAL_RETURN_REGS**.*r10*, and the number of enabled CPUs in **SAL_RETURN_REGS**.*r11*. **EFI_SAL_SUCCESS** is always returned.

Status Codes Returned

EFI_SAL_SUCCESS	The information on the number of CPUs in the platform was returned.

ExtendedSalSetMinState

Summary

Sets the MINSTATE pointer for the CPU specified by a Global ID.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalSetMinState (
  IN UINT64    FunctionId,
  IN UINT64    Arg2,
  IN UINT64    Arg3,
  IN UINT64    Arg4,
  IN UINT64    Arg5,
  IN UINT64    Arg6,
  IN UINT64    Arg7,
  IN UINT64    Arg8,
  IN BOOLEAN   VirtualMode,
  IN VOID      *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalSetMinStateFunctionId**.

Arg2

The 64-bit Global ID of the CPU to set the MINSTATE pointer.

Arg3

This parameter is interpreted as a pointer to the MINSTATE area for the CPU specified by Arg2.

Arg4

Reserved. Must be zero.

Arg5

Reserved. Must be zero.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Description

This function sets the MINSTATE pointer for the CPU specified by *Arg2* to the buffer specified by *Arg3*. If the CPU specified by *Arg2* is not present in the database, then **EFI_SAL_NO_INFORMATION** is returned. Otherwise, **EFI_SAL_SUCCESS** is returned.

Status Codes Returned

EFI_SAL_SUCCESS	The MINSTATE pointer was set for the specified CPU.
EFI_SAL_NO_INFORMATION	The specified CPU is not in the database.

ExtendedSalGetMinState

Summary

Retrieves the MINSTATE pointer for the CPU specified by a Global ID.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalSetMinState (
  IN UINT64    FunctionId,
  IN UINT64    Arg2,
  IN UINT64    Arg3,
  IN UINT64    Arg4,
  IN UINT64    Arg5,
  IN UINT64    Arg6,
  IN UINT64    Arg7,
  IN UINT64    Arg8,
  IN BOOLEAN   VirtualMode,
  IN VOID      *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalSetMinStateFunctionId**.

Arg2

The 64-bit Global ID of the CPU to get the MINSTATE pointer.

Arg3

Reserved. Must be zero.

Arg4

Reserved. Must be zero.

Arg5

Reserved. Must be zero.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Description

This function retrieves the MINSTATE pointer for the CPU specified by *Arg2*. If the CPU specified by *Arg2* is not present in the database, then **EFI_SAL_NO_INFORMATION** is returned. Otherwise, the MINSTATE pointer for the specified CPU is returned in **SAL_RETURN_REGS**.*r9*, and **EFI_SAL_SUCCESS** is returned.

Status Codes Returned

EFI_SAL_SUCCESS	The MINSTATE pointer for the specified CPU was retrieved.
EFI_SAL_NO_INFORMATION	The specified CPU is not in the database.

ExtendedSalPhysicalIdInfo

Summary

Returns the Physical ID for the calling CPU.

```
Prototype
SAL_RETURN_REGS
EFIAPI
ExtendedSalPhysicalIdInfo (
  IN UINT64    FunctionId,
  IN UINT64    Arg2,
  IN UINT64    Arg3,
  IN UINT64    Arg4,
  IN UINT64    Arg5,
  IN UINT64    Arg6,
  IN UINT64    Arg7,
  IN UINT64    Arg8,
  IN BOOLEAN   VirtualMode,
  IN VOID      *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalPhysicalIdInfo**.

Arg2

Reserved. Must be zero.

Arg3

Reserved. Must be zero.

Arg4

Reserved. Must be zero.

Arg5

Reserved. Must be zero.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Description

This function looks up the Physical ID of the calling CPU. If the calling CPU is not present in the database, then **EFI_SAL_NO_INFORMATION** is returned. Otherwise, the Physical ID is returned in **SAL_RETURN_REGS**.*r9*, and **EFI_SAL_SUCCESS** is returned.

Status Codes Returned

EFI_SAL_SUCCESS	The Physical ID for the calling CPU was returned.
EFI_SAL_NO_INFORMATION	The calling CPU is not in the database.

10.4.13 Extended SAL MCA Services Class

Summary

The Extended SAL MCA Services Class provides services to

GUID

```
#define EFI_EXTENDED_SAL_MCA_SERVICES_PROTOCOL_GUID_LO \
  0x42b16cc72a591128
#define EFI_EXTENDED_SAL_MCA_SERVICES_PROTOCOL_GUID_HI \
  0xbb2d683b9358f08a
#define EFI_EXTENDED_SAL_MCA_SERVICES_PROTOCOL_GUID \
  {0x2a591128,0x6cc7,0x42b1,\
  {0x8a,0xf0,0x58,0x93,0x3b,0x68,0x2d,0xbb}}
```

Related Definitions

```
typedef enum {
  McaGetStateInfoFunctionId,
  McaRegisterCpuFunctionId,
} EFI_EXTENDED_SAL_MCA_SERVICES_FUNC_ID;
```

Description

Table 18. Extended SAL MCA Services Class

Name	Description
ExtendedSalMcaGetStateInfo	Obtain the buffer corresponding to the Machine Check Abort state information.
ExtendedSalMcaRegisterCpu	Register the CPU instance for the Machine Check Abort handling.

ExtendedSalMcaGetStateInfo

Summary

Obtain the buffer corresponding to the Machine Check Abort state information.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalMcaGetStateInfo (
  IN UINT64 FunctionId,
  IN UINT64 Arg2,
  IN UINT64 Arg3,
  IN UINT64 Arg4,
  IN UINT64 Arg5,
  IN UINT64 Arg6,
  IN UINT64 Arg7,
  IN UINT64 Arg8,
  IN BOOLEAN VirtualMode,
  IN VOID   *ModuleGlobal OPTIONAL
  );
```

Parameters

FunctionId

Must be EsalMcaGetStateInfoFunctionId.

Arg2

The 64-bit Global ID of the CPU to get the MINSTATE pointer.

Arg3

Pointer to the state buffer for output.

Arg4

Pointer to the required buffer size for output

Arg5

Reserved. Must be zero.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Description

This function retrieves the MINSTATE pointer specified by *Arg3* for the CpuId specified by *Arg2*, and calculates required size specified by *Arg4*. If the CPU specified by *Arg2* was not registered in system, then **EFI_SAL_NO_INFORMATION** is returned. Otherwise, the CPU state buffer related information will be returned, and **EFI_SAL_SUCCESS** is returned.

Status Codes Returned

EFI_SAL_SUCCESS	MINSTATE successfully got and size calculated.
EFI_SAL_NO_INFORMATION	The CPU was not registered in system.

ExtendedSalMcaRegisterCpu

Summary

Register the CPU instance for the Machine Check Abort handling.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalMcaRegisterCpu (
IN UINT64 FunctionId,
IN UINT64 Arg2,
IN UINT64 Arg3,
IN UINT64 Arg4,
IN UINT64 Arg5,
IN UINT64 Arg6,
IN UINT64 Arg7,
IN UINT64 Arg8,
IN BOOLEAN VirtualMode,
IN VOID   *ModuleGlobal OPTIONAL
);
```

Parameters

FunctionId

Must be **EsalMcaRegisterCpuFunctionId**.

Arg2

The 64-bit Global ID of the CPU to register its MCA state buffer.

Arg3

The pointer of the CPU's state buffer.

Arg4

Reserved. Must be zero

Arg5

Reserved. Must be zero.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Description

This function registers MCA state buffer specified by *Agr3* for CPU specified by *Arg2*. If the CPU specified by *Arg2* was not registered in system, then **EFI_SAL_NO_INFORMATION** is returned. Otherwise, the CPU state buffer is registered for MCA handing, and **EFI_SAL_SUCCESS** is returned.

Status Codes Returned

EFI_SAL_SUCCESS	The CPU state buffer is registered for MCA handing successfully.
EFI_SAL_NO_INFORMATION	The CPU was not registered in system.