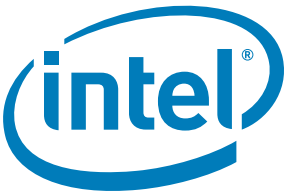


Intel® Itanium® Processor Family System Abstraction Layer Specification

Revision 3.4

Document Number: 245359-009

Contents

Figures

Tables

§

1 Introduction

1.1 Objectives

This document describes the functionality of the System Abstraction Layer (SAL) for Itanium® architecture-based systems.

This document specifies requirements to develop platform firmware for Itanium architecture-based systems. A companion document, the *Unified Extensible Firmware Interface Specification*, describes additional interfaces that must be implemented to access devices on the platform. The *Unified Extensible Firmware Interface Specification* is a requirement for Itanium architecture-based firmware.

This document is intended for firmware designers, system designers, and writers of diagnostic and low level operating system software. This document is an architectural specification describing the platform-dependent firmware interfaces needed to support the objectives listed below. It does not require a specific implementation, nor is it intended to document PC infrastructure specifications.

The primary objectives of Itanium architecture-based firmware are to:

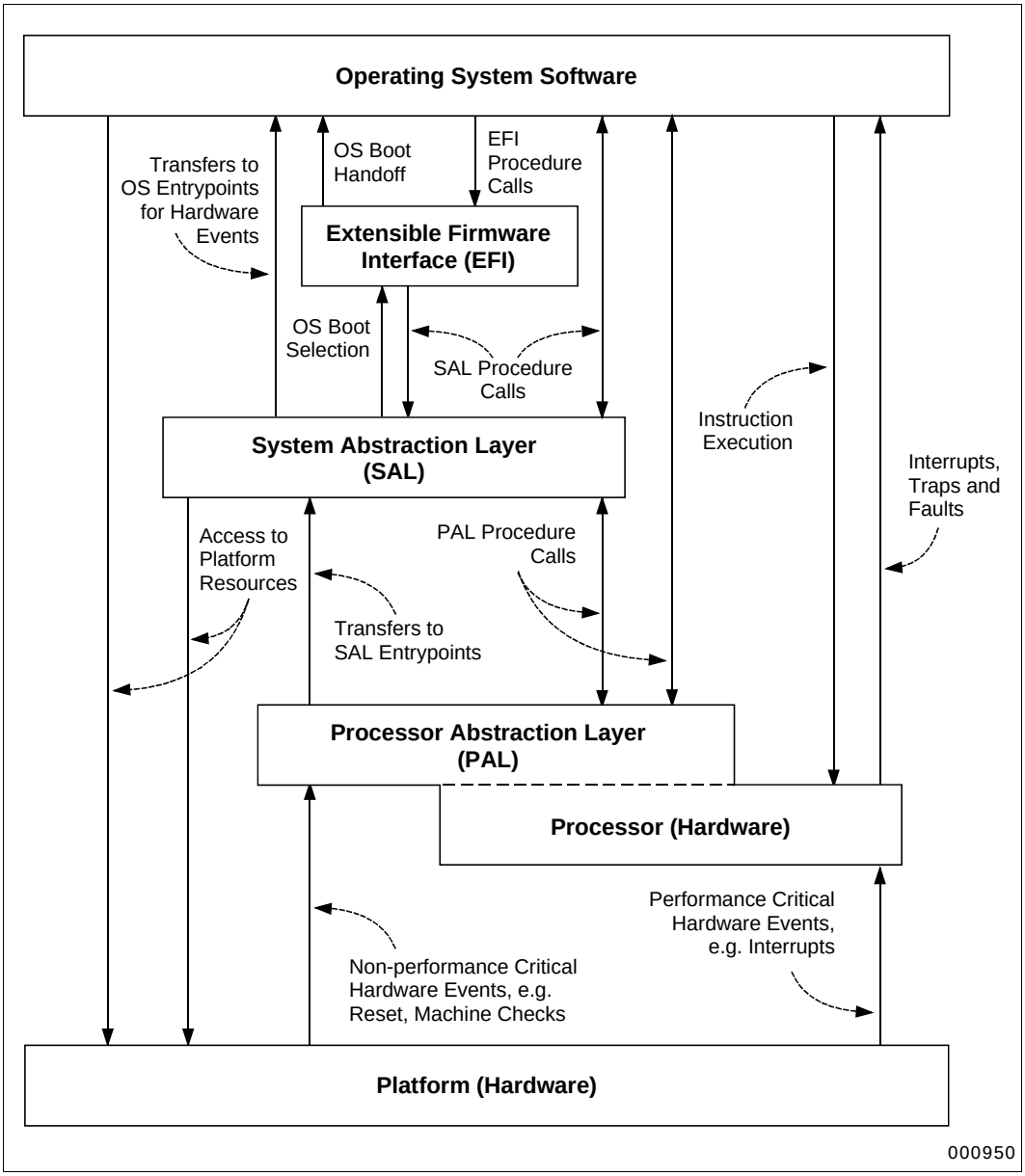
- Enable boot of Itanium architecture-based operating systems.
- Ensure that the firmware interfaces encapsulate the platform implementation differences within the hardware abstraction layers and device driver layers of operating systems.
- Separate the platform abstraction from the processor abstraction.
- Enable platform differentiation, hardware innovation, and optimization of Itanium architecture-based platforms.
- Support the scaling of systems from the low-end to the high-end including servers, workstations, mainframe alternatives, and supercomputers. Features supported will include high availability, error logging and recovery, large memory support, multiprocessing, and broader and deeper I/O hierarchies (possibly greater than 100 I/O cards).
- While using Itanium instructions is preferred, IA-32 BIOS code can be used in SAL. The extent of the IA-32 BIOS reuse is implementation-dependent, but all SAL entrypoints from the Processor Abstraction Layer (PAL) will use the Itanium system environment.
- Optionally, enable the use of legacy PC peripherals, option ROMs, and PCI cards with IA-32 Plug-and-Play expansion ROMs.

1.2 Firmware Model

As shown in Figure 1-1, Itanium architecture-based firmware has three components:

1. Processor Abstraction Layer
2. System Abstraction Layer
3. Extensible Firmware Interface

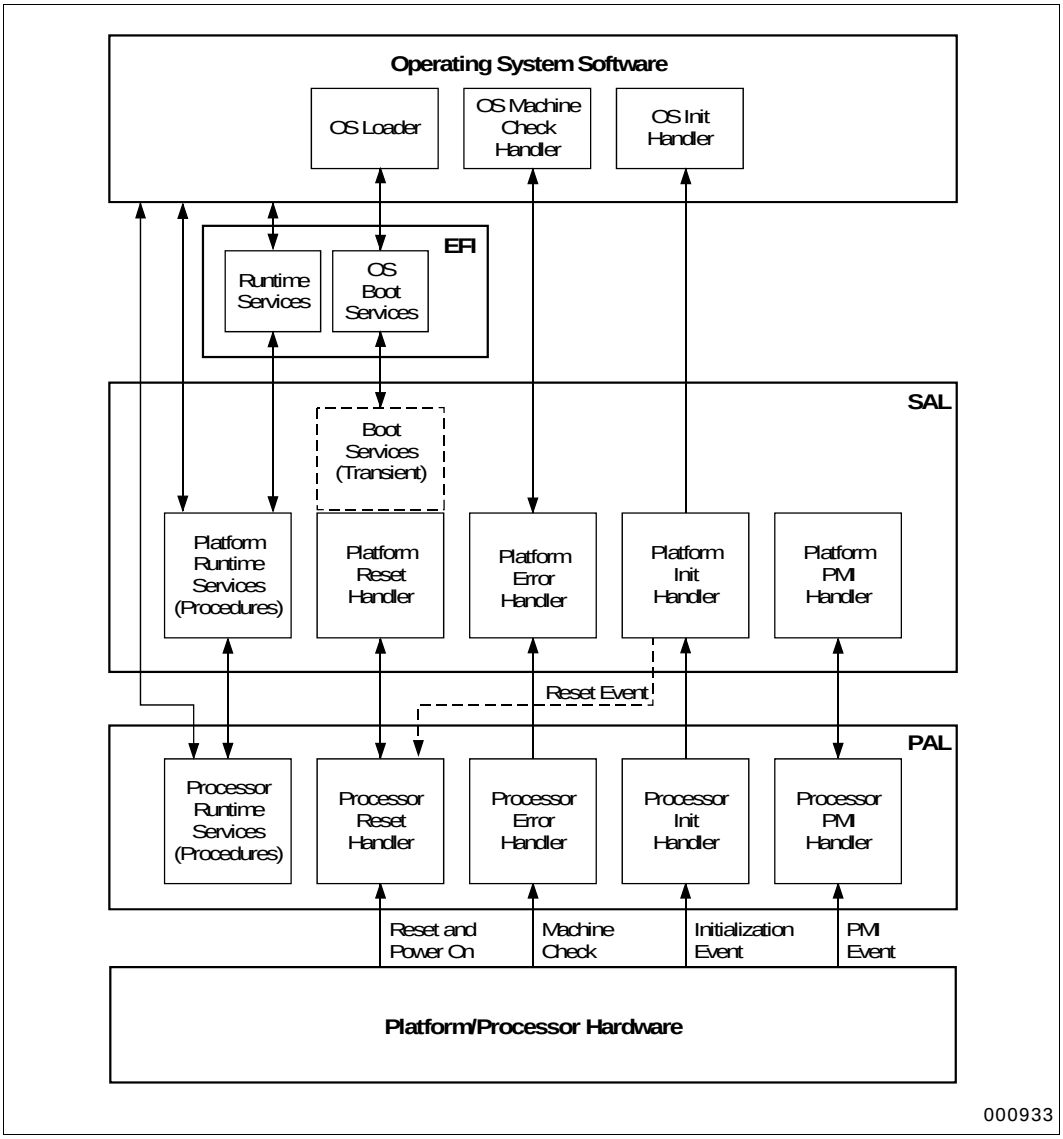
Figure 1-1. Firmware Model



PAL encapsulates processor implementation-specific features and is required in the Itanium architecture. PAL is not multiprocessor (MP) aware but is thread-aware for Itanium architecture processors that support multi-threading. SAL is the platform-specific firmware component that isolates operating systems and other higher level software from implementation differences in the platform. EFI provides a legacy free API interface to the operating system loader.

PAL, SAL, and EFI together provide system initialization and boot, Machine Check Abort (MCA) handling, Platform Management Interrupt (PMI) handling, and other processor and system functions which would vary between implementations. The interaction of the various functional firmware blocks is shown in Figure 1-2.

Figure 1-2. Firmware Services Model



1.3 System Abstraction Layer Overview

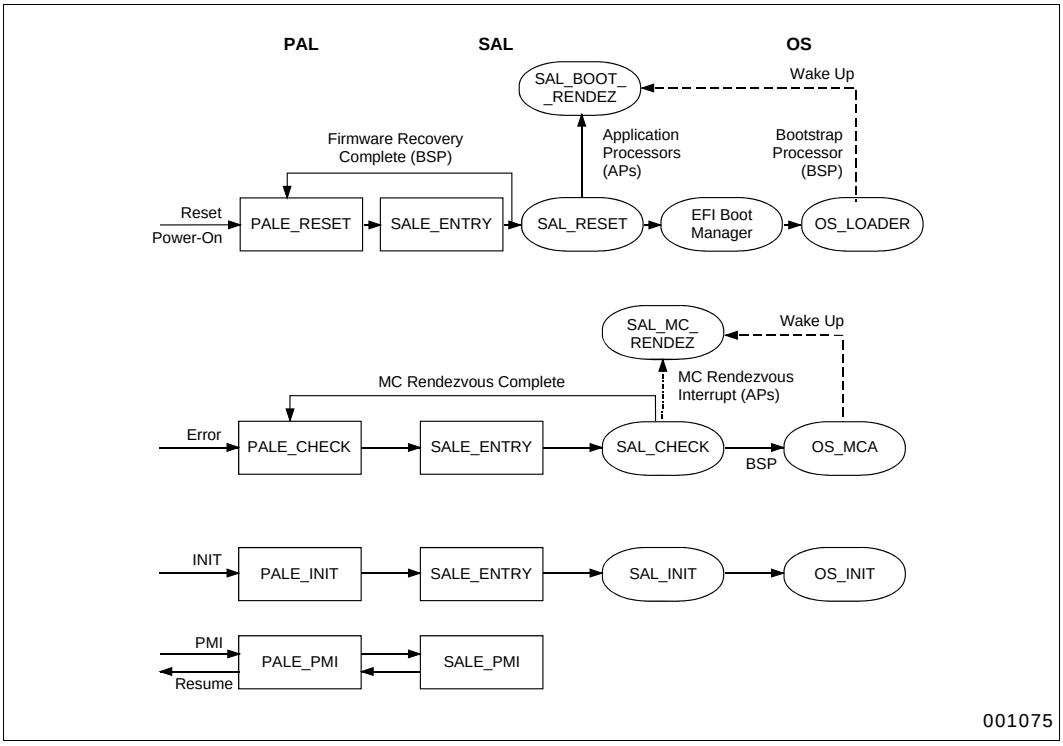
SAL provides the following functionality for an Itanium architecture-based platform:

- Initialize, configure, and test the platform hardware. This includes the memory and I/O subsystems, the necessary boot devices, and platform-specific hardware.
- Select the bootstrap processor (BSP) in a MP platform and set the configurable processor features. An Itanium architecture processor has its own PAL firmware for initialization and test. PAL has no knowledge of the platform, so further platform-specific action is necessary to integrate the processor with the rest of the system. For example, SAL must configure, test, and initialize memory before the processor cache to memory interface can be established and tested.
- Optionally encapsulate and provide the environment to run IA-32 BIOS and plug-in cards containing IA-32 Option ROMs.

- Provide low level service routines to aid EFI and the operating system loader in establishing the environment necessary for the operating system.
- Provide common data structures to the operating system to convey initialization and configuration information.
- Provide the necessary services and common infrastructure to support MP configurations.
- Provide runtime service routines to encapsulate those functions of the platform for EFI and the operating system while they are running.
- Provide the functions to aid in the logging and recovery from Machine Check conditions (SAL_CHECK and OS_MCA interface).
- Provide the functions necessary to aid in the logging and recovery from INIT conditions (SAL_INIT and OS_INIT interface).
- Provide the functions necessary to handle the platform management events (SALE_PMI interface).
- Optionally, provide the functions to aid in the recovery from a corrupted boot ROM.
- Optionally, provide a user interface to aid in system configuration, information passing, and troubleshooting.

1.4 Firmware Entrypoints

Figure 1-3. Firmware Entrypoints Logical Model



1.4.1 Processor Abstraction Layer Entrypoints

The following hardware events can trigger the execution of a PAL entrypoint:

- Power-on/reset
- Hardware errors (both correctable and uncorrectable)
- Initialization request
- PMIs

These hardware events trigger the execution of one of the following PAL entrypoints (as shown in Figure 1-1 and Figure 1-3):

1. PALE_RESET initializes the processor following power-on or reset. This PAL entrypoint calls the SALE_ENTRY entrypoint in the SAL to test for firmware recovery. SALE_ENTRY, in turn, calls SAL_RECOVERY_CHECK to perform recovery if the firmware recovery indication is present on the platform, otherwise it returns to PAL via SALE_ENTRY. If firmware recovery is required, the SAL recovery code will accomplish the firmware recovery function, reset the recovery indication, and then trigger a system wide reset, causing re-entry into the PALE_RESET. If SAL reports to PAL that a firmware recovery condition does not exist, PAL conducts additional processor tests and then branches to SALE_ENTRY. SALE_ENTRY then branches to a procedure within SAL called SAL_RESET to initialize the system.
2. PALE_CHECK saves the minimal processor state, determines if errors are processor related, saves processor-related error information, and corrects errors where possible (for example, by flushing a corrupted instruction cache line and marking the cache line as unusable). PALE_CHECK then branches to the SALE_ENTRY entrypoint. SALE_ENTRY, in turn, branches to SAL_CHECK to complete error logging, correction, and reporting. PALE_CHECK is entered as a response to processor or platform errors.
3. PALE_INIT is entered as a response to an initialization event. PALE_INIT saves minimal processor state and branches to SALE_ENTRY. SALE_ENTRY, in turn, branches to SAL_INIT.
4. PALE_PMI is entered as a response to a platform management event. PALE_PMI determines the type of platform management event and branches to SALE_PMI for certain conditions.

1.4.2 System Abstraction Layer Entrypoints

Following are the entrypoints from PAL into SAL:

1. SALE_ENTRY is the entrypoint PAL branches to after a power-on, reset, machine check, or initialization event. The code at this entrypoint uses the hand-off value in a general register to jump to SAL for reset, firmware recovery, machine check, and INIT events.

   These functions are available from SALE_ENTRY:

   - SAL_RESET within SAL is entered for system initialization after PAL has initialized the processor. SAL_RESET functionality is described in Chapter 3.
   - SAL_RECOVERY_CHECK within SAL is entered after a power-on reset from PAL to test if a recovery condition is present. Only SAL has enough knowledge of platform resources to determine if a firmware recovery condition is present.
   - SAL_CHECK within SAL is entered for logging errors and correcting platform related errors where possible. SAL_CHECK functionality is described in Chapter 4.

- SAL_INIT within SAL is entered for saving the state of the system and performing additional functions as defined in Chapter 5.

2. SALE_PMI is the entrypoint for handling platform management events in an implementation-dependent manner.

1.4.3 Operating System Handlers

There are several entrypoints from SAL into an operating system (or equivalent software):

- OS_LOADER is the entrypoint the BSP enters from SAL_RESET after the system has been initialized and the operating system loader image has been loaded by the EFI component from the boot device. Refer to the *Unified Extensible Firmware Interface Specification* for details.
- OS_BOOT_RENDEZ is the operating system MP rendezvous handler. It is entered from SAL when operating system loader on the BSP wakes up the application processors (APs), to permit synchronization of APs in an MP environment.
- OS_MCA is the operating system MCA handler that is called from SAL_CHECK to allow the OS to handle the machine checks that are not corrected by hardware, PAL, or SAL.
- OS_INIT is the operating system handler that is called from SAL_INIT to handle a valid INIT event.

1.5 Related Documents

The following documents contain additional material related to Itanium architecture-based platforms:

- *Advanced Configuration and Power Interface Specification* – Intel/Microsoft/Toshiba
- BIOS Boot Specification, 1996 – Compaq/Phoenix/Intel
- BIOS Enhanced Disk Drive Specification, Version 3.0 – Phoenix
- Bootable CD-ROM Format Specification, 1994 – Phoenix/IBM
- CBIOS for IBM Computers and Compatibles – Phoenix
- *Unified Extensible Firmware Interface Specification* – UEFI Forum
- *Intelligent Platform Management Initiative Specification* – Intel, NEC, HP, Dell
- *Intel® Itanium® Architecture Software Developer's Manual* – Intel
- *Intel® Itanium® Processor Family Error Handling Guide* – Intel
- *Itanium® Software Conventions and Runtime Architecture Guide* – Intel
- PCI BIOS Specification – PCI SIG
- PCI Local Bus Specification – PCI SIG
- Plug and Play ISA Specification, 1994 – Microsoft

1.6 Revision History

The revision number of the SAL specification supported by the SAL implementation is specified in the SAL System Table Header (refer to Table 3-3, "SAL System Table Header").

Date	Description
May 2009	Revision 3.4. In Section B.2.4.8, the PCIe 1.1/2.0 error section PCIe_OEM_DATA_STRUCT has been aligned to an 8-byte boundary. The error sections for processor machine check, deconfigured processor self-test errors and PCIe 1.1/2.0 VARIABLE_LENGTH_DATA_OFFSET values have been corrected to reflect the offset from the beginning of the error section to the variable-length data portion of the error section.
March 2008	Revision 3.3. Added Error Section clarification for latent error handling. Added clarification for PAL relocation. Removed non-mapped resource runtime access section. Added SAL br0 loop changes. Added SAL arg0 usage clarification. Added OS_MCA handling clarifications. Incorporated changes from SAL Specification Update June 2004 and SAL Specification Update August 2005. Incorporated changes from SAL Specification Update November 2006, but dropped Document Change 1 and 2.
December 2003	Removed references to IA-32 Operating System boot. Clarified OS_BOOT_RENDEZ usage and handoff requirements. Added SAL_PHYSICAL_ID_INFO call. Added extensions to SAL procedures to address PCI Express. Added clarifications to address multithreaded processors. Clarified use of SAL_SET_VECTORS checksum. Added example of GUID memory ordering. Added additional information on SAL procedure error return values. Clarified usage of fields in the SAL Error Record Header and Error Section Header. Included notes on SAL use of Translation Registers (TRs). Added Error Record alignment requirements. Updated glossary definitions. Clarified memory state on firmware to OS handoff. Clarified SAL Revision numbering. Incorporated changes from SAL Specification Update January 2003.
November 2002	Split PAL_A information added. Enhancements and clarifications to SAL_CACHE_FLUSH, SAL_MC_RENDEZ, and SAL_GET_STATE_INFO calls. Entrypoint descriptor field and memory attribute aliasing attributes added. IVA requirements in virtual addressing mode specified, and SAL/PAL flow for PAL firmware-corrected error modified. Error record revision value update and usage requirements defined. Clarifications and extensions to the error record and section headers, memory error section, PCI bus sections and PCI component sections.
July 2001	Platform requirement clarifications, Boot sequence clarifications, Additions to OS restrictions for boot sequence, Changes to MCA SAL_CHECK, Platform Errors, and OS_MCA sections, Added SAL procedures callable by OS_INIT, Clarification to Interface Conventions to SAL Procedures, Added changes regarding re-entrancy of SAL Runtime Services, Clarifications to SAL procedure definitions, Added terms to the glossary
January 2001	MCA related changes, Platform Error definition.
July 2000	Reflected changes in MCA handling due to PAL MCA changes.
January 2000	Changes to some SAL procedure definitions.
June 1999	Defined hand-off to EFI, Removed NVRAM functionality.
August 1998	Defined NVRAM record formats, changes to SAL procedures.
February 1998	Initial definition.

§

2 Platform Requirements

2.1 Firmware Address Space

The firmware address space occupies the 16 MB region below 4 GB (addresses 0xFF00_0000 through 0xFFFF_FFFF). This address space is shown in Table 2-1.

Table 2-1. Firmware Address Space

0xFFFF_FFFF	PAL/SAL ROM
	SAL Resources
0xFF00_0000	

The firmware address space is logically partitioned into two major functional blocks: the ROM area (shared by the SAL and PAL) and the SAL resources area. The ROM area is placed in the address space such that its ending address is 0xFFFF_FFFF. The SAL resources area occupies the portion of 16 MB firmware address space not occupied by the ROM area. SAL code can use the special hardware resources that the platform has implemented in the SAL Resources area. The hardware resources may include scratch RAM, non-volatile memory (NVRAM), environment control, and status registers. The location of the hardware resources within the SAL resources area is platform-dependent.

2.2 PAL/SAL ROM Space

The PAL/SAL ROM space within the firmware address space must contain the PAL and SAL code areas and a table called the Firmware Interface Table (FIT). See Section 2.5.

PAL code is broken into two subcomponents:

- PAL_A, which is independent of processor stepping.
- PAL_B, which is processor stepping-dependent.

These two subcomponents are required. The PAL_A block contains a limited subset of PAL procedures that can be invoked by SAL while performing a firmware recovery. (Refer to Volume 2 of the *Intel® Itanium® Architecture Software Developer's Manual* for details.) The PAL_B block contains the PAL procedures that can be invoked by SAL and the operating system.

In a similar fashion, SAL code can be broken into two subcomponents. Unlike the PAL, the SAL subcomponents need not be separate components:

- SAL_A which contains the SALE_ENTRY entrypoint and code needed for firmware recovery.
- SAL_B which contains code to test and initialize the platform.

The PAL_A, PAL_B, SAL, and FIT components are architecturally required.

PAL_A code can transition to:

- Code in the PAL_B using the FIT. First, the beginning address of the PAL_B block is determined from the FIT. Then, the entrypoints within the PAL_B block (for example, PAL_RESET) are determined in a PAL implementation-dependent manner.
- Code in the SAL address space at SALE_ENTRY, which serves as the entrypoint for reset, recovery, machine check and INIT events.

In order to conserve space in the firmware ROM, portions of the SAL code may be compressed. SAL code that is executed out of ROM such as early stages of the Reset sequence and code for handling Machine check and INIT cannot be compressed.

2.3 Simplified Firmware Address Map

A simplified example of the firmware address map that shows the *minimum* architectural components is shown in Figure 2-1. Refer to Section 2.4.1 for description of the fields. This layout cannot be used with a protected boot block.

2.4 Example Firmware Organization Using a Protected Boot Block

This section describes a typical firmware organization using flash ROM that contains a protected boot block.

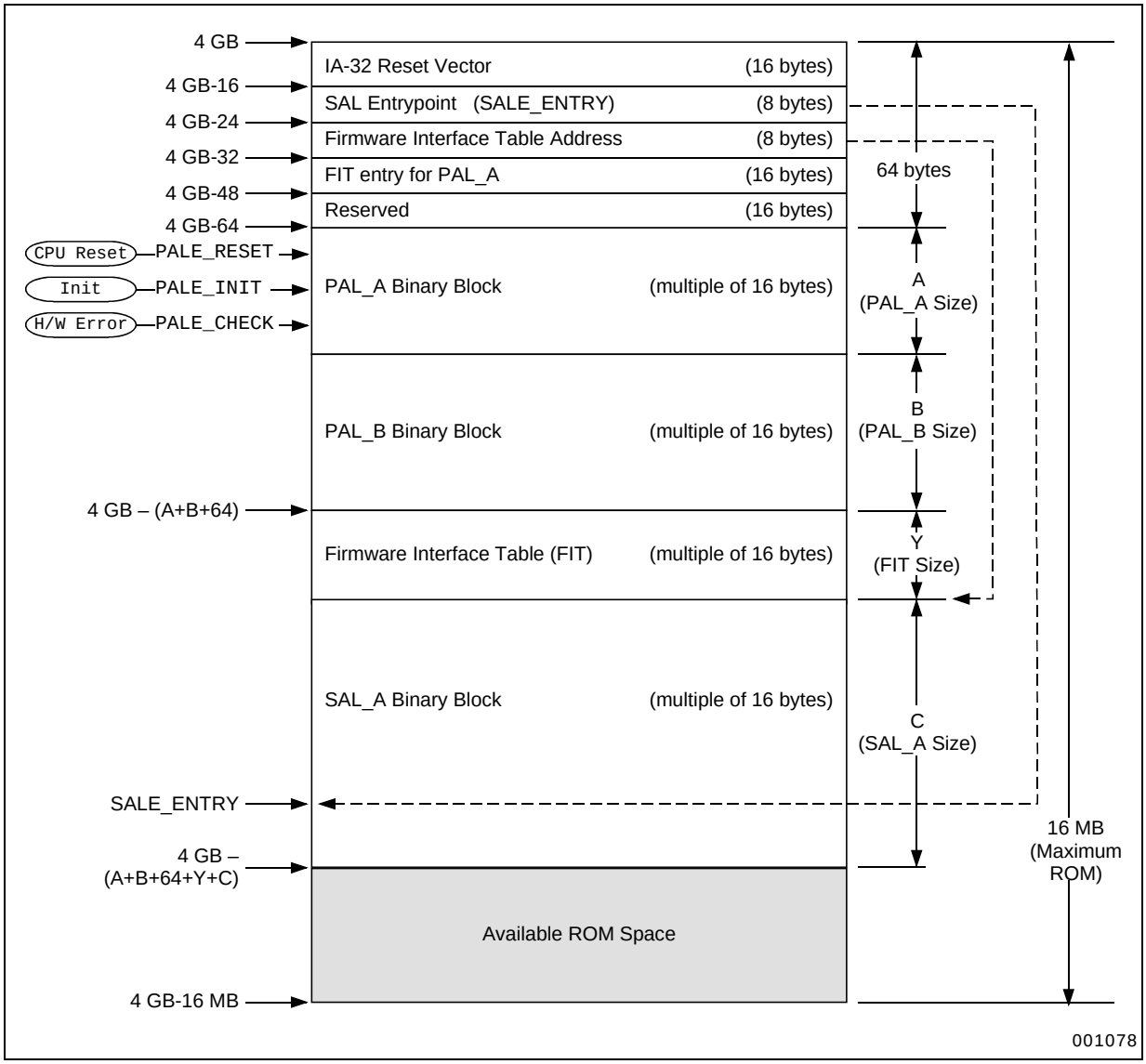
A protected boot block refers to a block of the flash ROM that the hardware protects from modification. Code in this block can contain logic to restore PAL/SAL code in the erasable portion of the flash part after a previous flash programming attempt has been accidentally aborted. Firmware using a protected boot block requires some data structures in addition to the minimum architectural requirements discussed earlier.

There are two primary layouts of the firmware address space that support a protected boot block. The first layout (shown in Figure 2-2) has one PAL_A component. In this layout, the PAL_A component must be within the protected boot block.

The second layout (shown in Figure 2-3) splits the PAL_A block into two components. The first component is referred to as the generic PAL_A and the second component is the processor-specific PAL_A. The generic PAL_A resides in the protected boot block and will work across processor generations for a given platform. The processor-specific PAL_A resides outside the protected boot block and is particular to processor generation.

In both layouts, the SALE_ENTRY entrypoint and the code needed for firmware recovery (located in SAL_A) must be located in the protected boot block.

Figure 2-1. Simplified Firmware Address Map



2.4.1 Firmware Components

The firmware address space is shared by the SAL and the PAL. Some of the SAL/PAL boundaries are implementation-dependent. The firmware address space contains several regions and locations as shown in Figure 2-2 and Figure 2-3 below for a typical implementation.

The firmware address space contains the following regions and locations:

- The 16 bytes at (4 GB – 16) contains the IA-32 reset vector for legacy compatibility.

Figure 2-2. Firmware Address Map

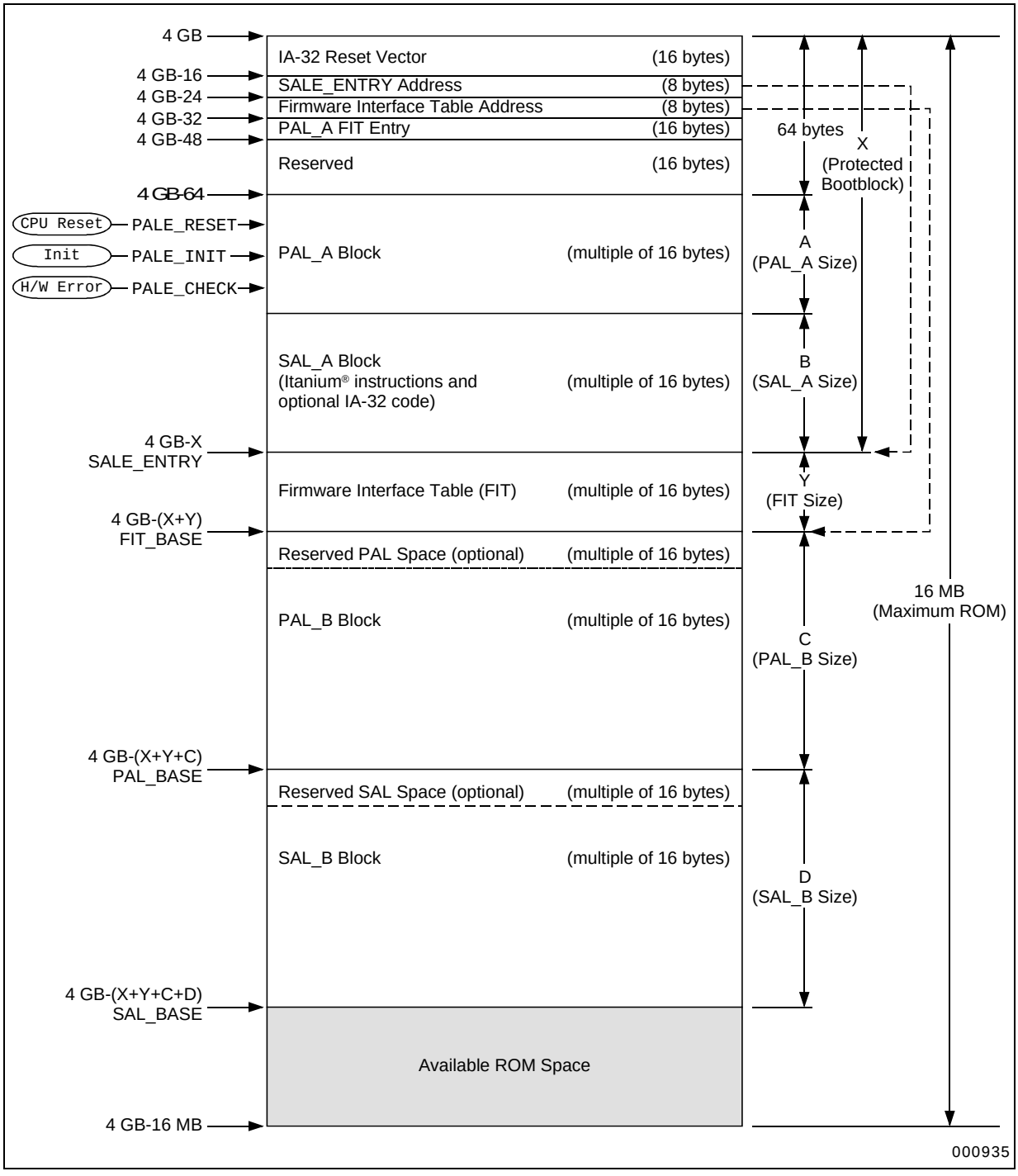
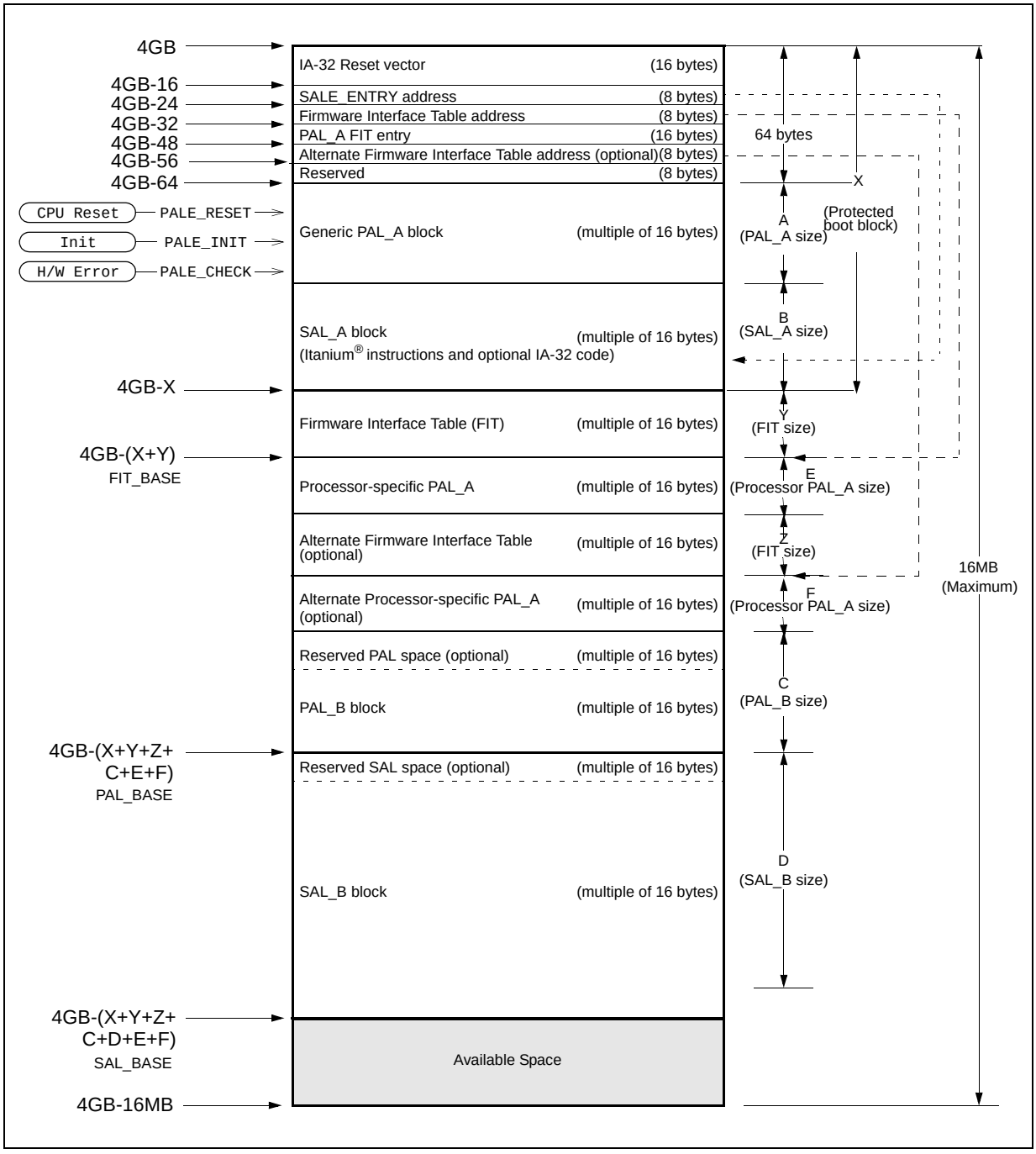


Figure 2-3. Firmware Address Map with Split PAL_A Components



- The 8 bytes at (4 GB – 24) contain the address of the SALE_ENTRY entrypoint. Bit 63 of this address must be set to 1 to specify the uncacheable memory attribute in physical addressing mode.
- The 8 bytes at (4 GB – 32) contain the pointer to the FIT. Bit 63 of this address must be set to 1. The FIT need not be located immediately before the protected boot block. However, the FIT cannot be moved to a different location since its address is contained in the protected boot block.
- The 16 bytes at (4 GB – 48) describe the characteristics of the PAL_A component (or generic PAL_A in the split PAL_A model) in the ROM: base address, size, version number, type, and so on. This is represented in the FIT entry format. Bit 63 of the *address* field within this FIT entry must be set to 1 and the *type* field must have a value of 0x0F.
- The 8 bytes at 0xFFFF_FFC8 (4 GB-56) contains the physical address of the Alternate FIT. This pointer is optional and is only needed if the firmware contains an alternate FIT table. If no alternate FIT table is provided, a value of 0x0 should be encoded in this entry.
- The 8 bytes at (4 GB – 64) are reserved.
- The PAL_A code (also known as the generic PAL_A code in the split PAL_A model) resides below the (4GB – 64) address. This variable size area contains the hardware-triggered entrypoints (PALE_RESET, PALE_INIT, and PALE_CHECK). In the model where PAL_A is not split, the PAL_A code will perform minimum processor initialization. In the split PAL_A model, the generic PAL_A will search the FIT table(s) to find the processor-specific PAL_A code. It will then branch to this code to perform the processor-specific initialization:
  - The PAL_A code block must be a multiple of 16 bytes in length. PAL_A uses the FIT entry of the PAL_B to reach continuation entrypoints in PAL_B for reset, machine check, and initialization.
  - The code in the PAL_A block(s) contains enough capability to initialize the processor, invoke the SALE_ENTRY procedure for test of the recovery indication, and continue with normal PAL execution in the PAL_B code area.
- SAL_A code occupies the bottom of the protected boot block. To provide maximum flexibility and to conserve space in the protected boot block, this area will primarily contain code for firmware recovery. When entered for other conditions such as normal reset, machine check, or initialization, the code in this block will find the continuation entrypoints in the SAL_B block (using the FIT or other means) and jump to the same. The method by which SALE_ENTRY code reaches continuation entrypoints in SAL_B for reset, machine check, and initialization is SAL implementation-dependent.

Note: The sizes of the PAL_A (generic PAL_A in the split PAL_A model) and SAL_A code blocks shown in Figure 2-2 and Figure 2-3 are not needed during firmware execution but may be needed by the utility that merges these components to format the protected boot block portion of the flash ROM.

- Below the protected boot block is the FIT. It consists of 16-byte entries containing starting address and size information of the remaining firmware components. Optionally, an alternate FIT may be included in the firmware. The alternate FIT will only be used if the primary FIT failed its checksum. This feature allows hand-off to the SAL recovery code, even if there is a primary FIT checksum failure. Refer to Section 2.5 for FIT details.
- Below the FIT(s) is the processor-specific PAL_A. This component is only available on processors that support a split PAL_A firmware model. One processor-specific PAL_A is architecturally required in this model. The firmware may optionally contain two or more processor-specific PAL_A components.

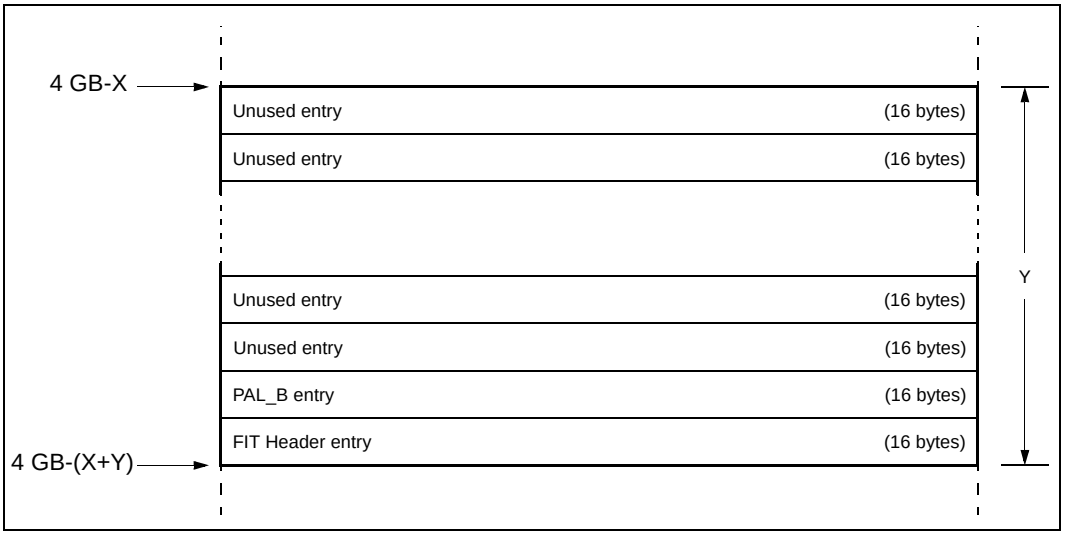
- Below the FIT is the code for the IA-32 BIOS, EFI, SAL_B, and PAL_B components. There are no ordering requirements for the firmware components within the flash ROM.
- The PAL_B binary block contains PAL code that is not required for firmware recovery. The PAL_B code area is a multiple of 16 bytes in length and must be aligned on a 32K-byte boundary. PAL_B's FIT entry contains the address and size of the PAL_B binary block.
- The remainder of the SAL/PAL ROM area is occupied by the SAL_B code. SAL_B's FIT entry (if present in the FIT) contains the address and size of the SAL_B binary block.

Note: Code within SAL (SAL_A and SAL_B) may include IA-32 code. The location of the SAL_B and IA-32 BIOS code within the SAL/PAL ROM area is implementation- dependent. Some SAL implementations may separate the code containing Itanium instructions and IA-32 instructions as separate firmware blocks with unique FIT entry types. In a similar fashion, the SAL_B component may include the EFI component or a separate FIT entry may point to the EFI component.

2.5 Firmware Interface Table

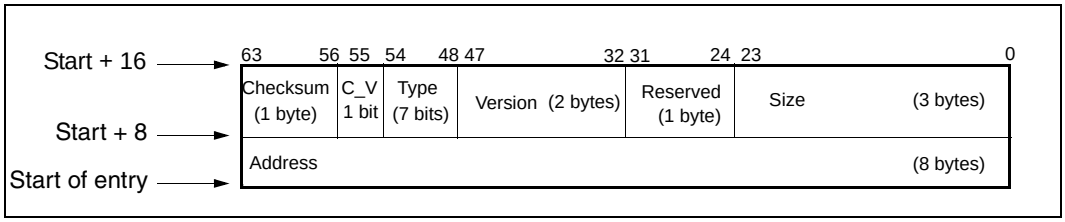
The FIT contains starting addresses and sizes for the firmware components that are outside the protected boot block. Because these code blocks may be compiled at different times and places, code in one block (such as PAL_A) cannot branch to code in another block (such as PAL_B) directly. The FIT allows code in one block to find entrypoints in another. Figure 2-4 shows the FIT layout.

Figure 2-4. Firmware Interface Table



Each active FIT entry contains information for the corresponding firmware component. The first two entries are used to describe the FIT table itself and the PAL_B block respectively and these two entries are architecturally required. FIT entries shall be in ascending order of entry types, otherwise firmware behavior is unpredictable. The FIT entry format is shown in Figure 2-5.

Figure 2-5. Firmware Interface Table Entry



Address is the base address of the component and it must be aligned on a 16-byte boundary. For the FIT Header entry, this field contains the ASCII value of '_FIT_<sp><sp> <sp>' where <sp> represents the space character. For the processor-specific PAL_A and PAL_B entries, bit 63 of the address field must be set to 1 to indicate the uncacheable memory attribute in physical addressing mode. The PAL_B component must be aligned on a 32K-byte boundary.

Size is the size of the component in paragraphs of 16 bytes.

Version contains the component's version number. For the FIT Header Entry, the value in this field will indicate the revision number of the FIT data structure.

C_V is a one bit field that indicates whether the component has a valid checksum. If this bit is zero, the value in the *Chksum* field is not valid.

Type contains the seven-bit type code for the entry. Types are defined in Table 2-2.

Table 2-2. FIT Types

Type	Meaning
0x00	FIT Header entry
0x01	PAL_B
0x02-0x0D	Reserved
0x0E	Processor-specific Pal_A
0x0F	PAL_A (also generic PAL_A)
0x10-0x7E	OEM-defined
0x7F	Unused

The type code of 0x0F is used for PAL_A. Since PAL_A's binary image is located near the end of the 4 GB firmware address space (flash ROM organization with protected boot block), its FIT entry is also located within the protected boot block (at 4 GB – 48) and not in the FIT table. The OEM may define unique types for one or more blocks of SAL_B, EFI, IA-32 BIOS, and so on, within the OEM-defined type range of 0x10 to 0x7E.

Chksum contains the component's checksum. The modulo sum of all the bytes in the component and the value in this field (*Chksum*) must add up to zero. This field is only valid if the *C_V* field is non-zero. The checksum may be verified by firmware or software prior to its use. If the checksum option is selected for the FIT in the *FIT Header entry* (FIT type 0), the modulo sum of all the bytes in the FIT table must add up to zero. The PAL_A FIT entry is not part of the FIT table and hence not included in the checksum computation of the FIT.

With this address layout, when one of the firmware components changes, only that component's flash portion requires changes. This address layout can also support multiple ROMs for the firmware components, and such ROMs are not restricted to reside below 4 GB.

2.6 Resources Required for Legacy Compatibility

All platforms shall implement a minimum of 64 MB of memory. The area of memory below 1 MB is defined as the compatibility area and is used by firmware when initializing and executing IA-32 BIOS (refer to Table 2-3). The requirements specified below need not be implemented on the platform if legacy compatibility is not required.

Table 2-3. 1 MB Compatibility Memory Address Space

0x000F_FFFF 0x000F_0000	Shadowed IA-32 System BIOS
0x000E_FFFF 0x000E_0000	Shadowed IA-32 Extended System BIOS/Option ROM/Memory Area
0x000D_FFFF 0x000C_0000	Shadowed IA-32 Option ROM BIOS
0x000B_FFFF 0x000A_0000	VGA Frame Buffer
0x0009_FFFF 0x0000_0500	Memory
0x0000_04FF 0x0000_0400	IA-32 BIOS RAM Data Area
0x0000_03FF 0x0000_0000	IA-32 Interrupt Vector Area

Within the 1 MB compatibility memory address space, empty spaces can be mapped to system memory. For example, a server platform may choose to implement the system console on a serial port and eliminate the VGA frame buffer and the VGA BIOS components. IA-32 stack should be allocated in the memory region (0x0000_0500 to 0x0009_FFFF) for use by the real mode IA-32 BIOS code.

Itanium architecture-based platforms may optionally use I/O adapter cards containing IA-32 option ROMs during the boot process. A portion of the SAL code may also contain IA-32 code. Such IA-32 code as well as IA-32 operating systems may rely on the existence of legacy components. If it is necessary to support such IA-32 code, Itanium architecture-based platforms may implement the I/O ports specified in the Table 2-4 or alternatively, the SAL can trap some or all IA-32 I/O instructions and emulate the I/O ports that are not present on the platform. Refer to Section 7.2.4, “IA-32 Support Environment” for more details.

Table 2-4. IA-32 Compatibility I/O Ports

Port	Description
0x20-0x21	Programmable Interrupt Controller (Master)
0x40-0x43	Programmable Interval Timer
0x70-0x71	CMOS NVRAM Address, Data Ports
0xA0-0xA1	Programmable Interrupt Controller (Slave)

2.7 Chipset and Shadowing Requirements

Chipset implementations have the following SAL requirements:

- The firmware code and data within the firmware address range must be accessible from the processor without any special system fabric initialization sequence. This implies that the system fabric is implicitly initialized at power-on for accessing the firmware address space.
- Firmware may copy ROM-based code and data structures to RAM to increase performance and to allow for updates of ROM based data structures by initialization firmware. Platforms may implement any write protection for these shadowed areas. Since hardware events such as reset, machine check and initialization enter architected PAL entrypoints in the ROM around the 4 GB address, chipsets shall not disable accesses (by aliasing or other means) to the PAL/SAL ROM area subsequent to the shadowing of firmware code.

  Itanium instructions provide the necessary memory management features to prevent writes to the shadowed RAM areas while executing IA-32 code. The Itanium instruction set provides instructions to synchronize the instruction and data caches in the presence of self-modifying code.
- Chipsets need not implement in-line shadowing (read cycles going to ROM, write cycles going to RAM) for copying IA-32 code segments to memory addresses in the range of 0xE0000 to 0xFFFFF.

2.8 Platform Support for Variant Architectural Features

Platform implementations may vary in the features they implement and remain architecturally compliant. As an example, some platforms will implement bus lock while other platforms will not. This has implications for software running on these platforms, and therefore this information must be communicated to software. SAL firmware is responsible for knowing the architecture implementation variations and correctly communicating the information to software. How SAL knows about the architectural variant is implementation-dependent. The following lists the features which fall into this category and describe the method of abstraction to software.

- Bus Lock: If the processor supports the bus lock signal and the platform implements bus lock, then SAL shall set the Default Control Register Lock Check Enable bit to 0 (DCR.lc = 0), otherwise the DCR.lc shall be set to 1. The operating system shall not alter DCR.lc bit setting if it is set to 1. Refer to the PAL call PAL_BUS_ SET_FEATURES in the *Intel® Itanium® Architecture Software Developer's Manual* for information on masking bus lock signal and executing the locked transaction as a series of non-atomic transactions.
- Lowest Priority Interrupt: SAL shall communicate to the operating system, through the SAL System Table (Table 3-6), whether this feature is supported by the platform.
- Address Space Attributes: SAL shall communicate to software the supportable access attributes for all valid address space mappings. This information is provided to the operating system by the EFI component. As an example of this architectural implementation options, consider two memory controllers where one supports sub-cache line writes to memory and another which does not. The first case would be described as write-through or write-back cacheable, whereas the second case would be described as supporting only write-back cacheable. Similarly, the UCE memory attribute indicates whether the address space permits the exporting of the *fetchadd* operation outside the processor. Memory attribute features for address

spaces are fully described in the *Intel® Itanium® Architecture Software Developer's Manual.*

2.9 Platform Considerations Related to Processor Physical Location

Following are the SAL requirements from the platform pertaining to the physical locations of processors in an MP configuration:

- The platforms shall provide a mechanism to generate unique geographic identifiers for those physical components that have software visibility. As an example, imagine a complex MP implementation that has more than one main system bus to which processors are attached. Each logical processor returns its logical address on the bus via a call to PAL_FIXED_ADDR, but this PAL call does not reflect the multinode configuration of the platform. It is therefore required that the platform provide some mechanism for SAL to ascertain which bus a processor is attached to. SAL will use this value to load the Streamlined Advanced Programmable Interrupt Controller (SAPIC) EID field in the Local ID register (CR.LID) of the processor(s). This is necessary for supporting interprocessor interrupts. The above example is not meant to limit this requirement to processors, as multiple host I/O bridges and multiple memory controllers, and so on, may also have a similar requirement.

  Platforms may implement unique ways of providing the SAPIC EID value. For example, in a single-node system, SAL may use the hardcoded value of zero for this field. Another example is a node controller that provides different EID values for processors connected to different buses in the system. It is expected that these mechanisms will be very simple, to facilitate exchange of interprocessor interrupts between processors (if needed), to determine the BSP node and the BSP processor in an MP environment. The BSP selection needs to be done very early in the boot sequence and during firmware recovery. Since multiple processors may be attempting to read the EID, a scheme that involves writing an index followed by reading the value from a node controller I/O port or the CMOS NVRAM I/O port may be prone to errors.
- A multi-Translation Lookaside Buffer (TLB) coherence domain platform must provide a mechanism for detecting which TLB coherence domain the processor is located in.

2.10 Non-Volatile Memory Requirements

Itanium architecture-based platform hardware must provide a minimum of 32KB of NVRAM to hold the error log captured during uncorrected machine check events. There may be additional NVRAM requirements to hold information on the operating systems that can be booted from the platform, the platform configuration, and so on. Refer to the *Extensible Firmware Interface Specification* for requirement details as well as the interfaces to the NVRAM space.

The NVRAM must preserve memory contents when the system power is off. Some possible NVRAM implementations are battery-backed SRAM and flash memory. The physical address and size of each NVRAM object in the system will be specified in Table 3-5, "Entrypoint Descriptor Entry Format" with:

- *Memory type* classification of *Regular Memory* and *Memory Usage* classification of *Firmware Reserved Memory* for battery-backed SRAM implementation, and
- *Memory type* classification of *Firmware Address Space* when NVRAM is implemented as part of the firmware flash ROM.

2.11 Miscellaneous Platform Requirements

Following are the additional platform requirements for SAL:

- If firmware recovery is supported in SAL, Itanium architecture-based platforms must provide a mechanism for user-requested firmware recovery.
- Itanium architecture-based platforms must support simple hardware or software implementations for BSP selection, for example, write once port. This is necessary since only the BSP is allowed to execute the firmware recovery code.
- Itanium architecture-based platforms must provide mechanisms to determine the base frequency of the platform (clock input to the processor).
- Itanium architecture-based platform hardware must provide a mechanism for firmware to reset all components within the platform.
- Itanium architecture-based platform hardware must provide a switch or other mechanism that produces an INIT signal. This feature, generally known as the CrashDump switch, may be used to effect a crash dump on a “hung system.”
- Itanium architecture-based platform hardware must provide user friendly mechanisms for displaying the progress of the boot and firmware recovery, for example, LCD display.

§

3 Boot Sequence

3.1 Overview of the Code Flow after Hard Reset

This chapter describes the firmware execution sequence from reset to operating system launch.

PALE_RESET is an entry point within the PAL_A code area near 4 GB in the firmware address space. All processors begin execution at this point on system reset. The exact implementation PALE_RESET is implementation dependent. PALE_RESET initializes and tests the processor using stepping-independent code. It will then call SALE_ENTRY with the *Recovery Check* function to verify if the user has requested firmware recovery in a platform-dependent manner.

SALE_ENTRY is the shared SAL_A entrypoint from code in the PAL_A and PAL_B blocks for reset, recovery, machine check, and initialization events. PAL code obtains the SALE_ENTRY entrypoint from the 8-byte pointer at location 4 GB – 24. The state of the processor on entry into SALE_ENTRY is described in the *Intel® Itanium® Architecture Software Developer's Manual.* A general register (GR20) indicates the event causing entry into SALE_ENTRY – reset, recovery check, machine check, or initialization. SALE_ENTRY uses this argument to jump to internal entrypoints within SAL – SAL_RESET, SAL_RECOVERY_CHECK, SAL_CHECK, or SAL_INIT.

PAL_A passes status information to SALE_ENTRY on the health of the processor and whether the version of the PAL_B in the firmware is compatible with the processor's stepping. Table 3-1 shows the recommended SAL actions based on the self-test state parameter provided by PAL_A.

Table 3-1. SAL Actions Based on Processor Self-Test State

Processor Health	SAL Handling
Catastrophic Failure	None. PAL disables interrupts and Machine Checks, then keeps the processor in a spin loop in PAL or in a halt state.
Healthy	Proceed with SAL Reset.
Performance Restricted	Proceed with SAL Reset if this is the only processor in the system. Else, try to inform the user. The processor may be an attached processor in a MP configuration.
Functionally Restricted	Try to inform the user. Disable interrupts and Machine Checks, then go into a spin loop. Operating systems may not boot successfully if key processor functionality is unavailable.

The code in SAL_A will initiate recovery and update the firmware if:

- The platform indicates a user-requested recovery.
- The PAL_A code reports an authentication failure on the PAL_B component in the firmware.
- The PAL_A code reports checksum or other errors in the FIT or the PAL_B component.
- The PAL_A code reports on all the processors that the version of the PAL_B in the firmware is incompatible with the stepping level of the processors in the system.

3.1.1 Code Flow During Recovery

If firmware recovery is required, the SAL recovery code shall authenticate a new firmware image using a PAL_A procedure. The SAL code will then accomplish the firmware recovery function, reset the recovery indication, and trigger a system wide reset causing re-entry into PALE_RESET. SAL recovery code contains the logic to update one or more of the firmware components from OEM supported media.

Note: The firmware recovery code in SAL_A must be independent of processor stepping and must not invoke code in the PAL_B block.

In a multiprocessing environment, the recovery code will first select a BSP. SAL shall not select a processor as the Bootstrap processor (BSP) unless it is reported as healthy or performance restricted by PAL and the version of PAL_B in the system is compatible with the processor stepping. The BSP will rendezvous the APs and then proceed with the recovery of firmware. Note that the processors that are incompatible with the version of PAL_B in the system must not be woken up until the PAL_B component is updated.

Since PAL_B functionality cannot be invoked during recovery, only a limited set of PAL procedures in the PAL_A are available for use by the SAL recovery code. (Refer to the *Intel® Itanium® Architecture Software Developer's Manual* for details.) Furthermore, if SAL_A invokes the IA-32 BIOS, the floating-point transcendental instructions listed below cannot be executed from the IA-32 instruction set:

```
F2XM1, FCOS, FPATAN, FPTAN, FPREM, FPREM1,
|FSIN, FSINCOS, FYL2X, FYL2XP1
```

3.1.2 Boot Flow

If a recovery condition does not exist, SALE_ENTRY shall return to PALE_RESET on all the processors that are compatible with the version of PAL_B in the system, using the return address provided by PALE_RESET to begin the second stage of processor test and initialization. If SAL_A did not result in such a return, the processor may run in a degraded (functionally restricted) mode. The PAL_PROC address provided to SALE_ENTRY at the time of *Recovery Check* supports only a subset of the PAL procedures. (See the *Intel® Itanium® Architecture Software Developer's Manual* for details.)

On return from SALE_ENTRY, the PALE_RESET code obtains the address of the FIT from location (4 GB – 32) and then uses the FIT to get the address of the PAL_B component in the non-recovery portion of the flash ROM. PAL_A code will locate the address of the PAL_RESET in the PAL_B block and jump to it. The processor stepping-dependent code in the PAL_B block will then perform the complete processor testing (processor late self-test) and initialization and then re-enter the SALE_ENTRY with the function value of *Normal Reset.* Code at SALE_ENTRY will jump to the code in the SAL_B block to continue the boot sequence and will eventually boot the machine to the operating system.

3.2 SAL_RESET

SAL_RESET is responsible for performing platform test and initialization and invoking the EFI environment, which then invokes the operating system loader. SAL_RESET may also be entered from SAL_INIT if an OS_INIT handler is not registered with SAL. One of

the parameters passed into SAL_RESET (zero value in GR32) indicates that SAL_RESET was entered from PALE_RESET. GR32 must be non-zero if SALE_ENTRY is entered from locations other than PALE_RESET.

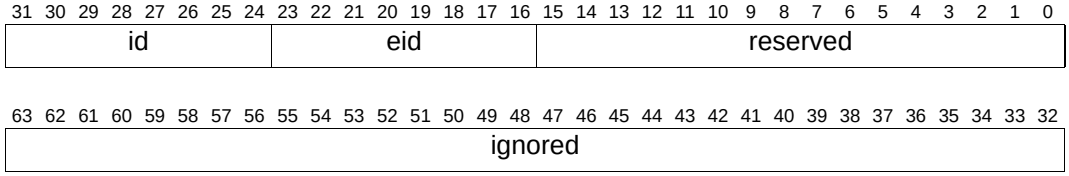
SAL_RESET functionality can be subdivided into the following phases:

- Initialization
- BSP identification
- Platform initialization
- Operating system boot

3.2.1 Initialization Phase

This phase begins execution at SAL_RESET and is performed on all the processors in the system. The Local ID (LID register) is described in the *Intel® Itanium® Architecture Software Developer's Manual*. It is the SAL's responsibility to uniquely initialize this register in each logical processor prior to performing BSP selection and enabling interrupts in an MP system. For uniprocessor (UP) systems, SAL must initialize this register prior to enabling interrupts. The operating system must not change the value that SAL stored into this register. Otherwise, routing of interrupts to the correct processor may not function correctly. The LID register's format is shown in Figure 3-1.

Figure 3-1. Local ID Register Format



SAL must invoke the PAL_PLATFORM_ADDR procedure on all processors to set the physical address of the SAPIC Interrupt block memory and the IA-32 I/O port space if the default address values are not used. The default address for the SAPIC Interrupt block memory is 0x00000000_FEE00000 and the default address for the IA-32 I/O port space is the 64 MB region below the highest physical address supported by the processor implementation. SAL will use a value that does not conflict with other devices on the platform. The operating system shall not change either of these address values. SAL will set up the IOBASE register (AR.k0) that provides the high order bits of the virtual address of the IA-32 I/O port block, to the same value as its physical address, to maintain identity mapping.

3.2.2 Bootstrap Processor Identification Phase in a Multiprocessor Configuration

Bootstrap processor selection is executed on all processors. The PAL_FIXED_ADDR procedure will be called to obtain a unique address on the bus to which the processor is connected. SAL will use this address and bus identification information to derive a unique geographical address for the processor and use the same in the selection of the boot processor. The determination of the unique geographical address is implementation-dependent. SAL shall not select a processor as the BSP unless it is reported as healthy by PAL and the version of PAL_B in the system is compatible with the processor stepping.

Refer to Figure 3-2 for SAL processing steps in a MP configuration. The APs will set up processor-specific resources such as the Interrupt Vector Address (IVA) and wait in the rendezvous state (Rendezvous_1 in Figure 3-2) until the SAL on the BSP wakes them

up for further processing. Processors in the rendezvous state will disable external interrupts and poll for the rendezvous interrupt vector which the BSP will utilize to wake up the sleeping APs. The BSP will continue with platform initialization. When sufficient memory has been tested, the BSP will wake the APs with a rendezvous interrupt so that they can run late self-test. After the APs have finished the late self-test, they will return to the rendezvous state (Rendezvous_2).

The BSP continues with platform initialization by loading the EFI firmware, which searches for bootable devices, loads the operating system loader, and transfers control to it. These steps are described in later sections of this document and the *Extensible Firmware Interface Specification*.

3.2.2.1 Rendezvous Functionality

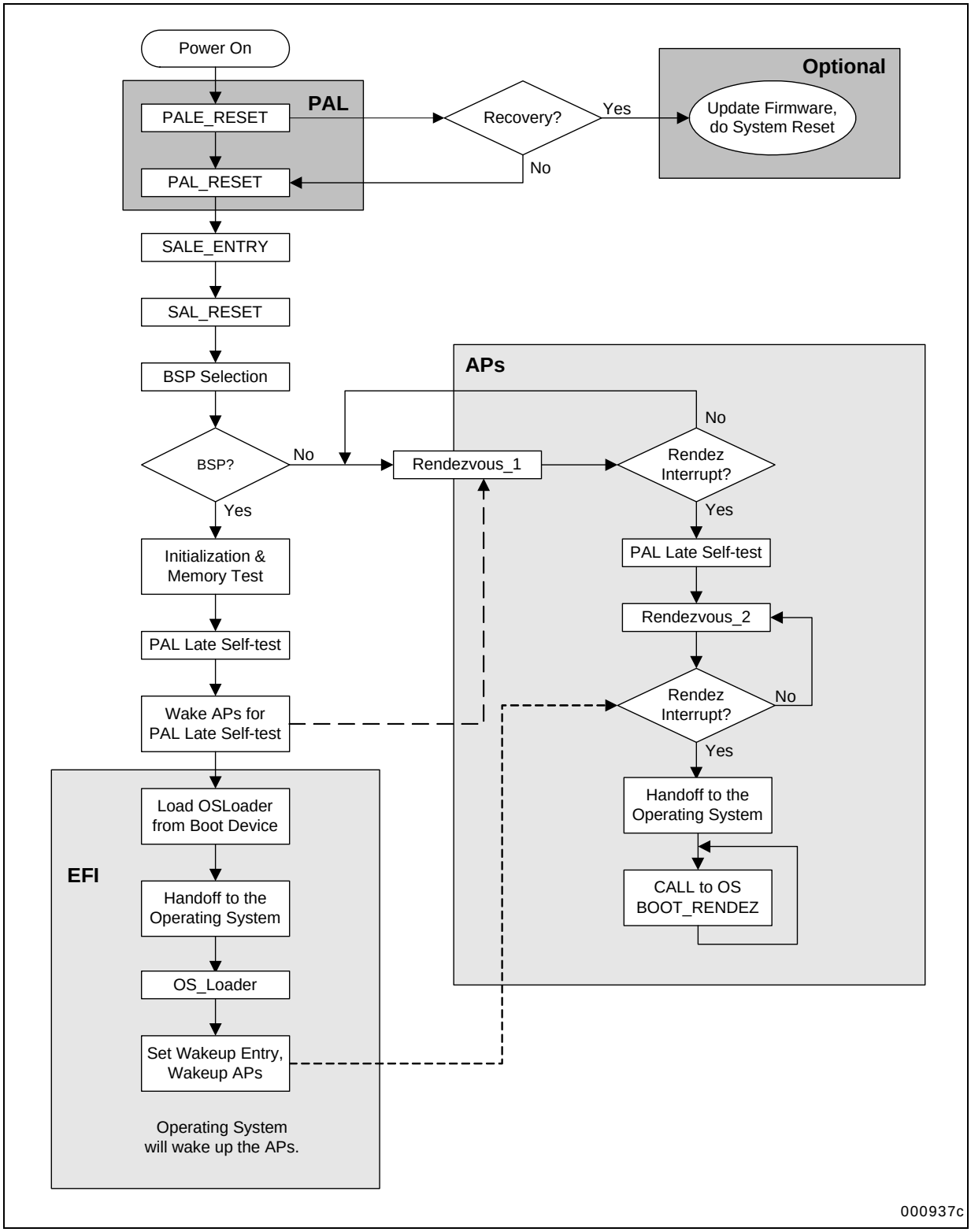
Rendezvous functionality is required only in MP environments and is utilized in two different ways:

- To wake up the APs during boot: The APs stay in a loop until woken up by the SAL layer on the BSP. The BSP wakes up the APs at various stages of booting to conduct processor and platform tests. Once these tests are completed, the APs return to the wait loop within SAL. Also, once the operating system kernel takes over, it will wake up the APs based on the wake up information provided by the SAL (refer to Section 3.2.5 and Table 3-10).
- To bring the APs to a spin loop during machine check rendezvous and to wake up the APs after machine check processing is completed: The operating system specifies the external interrupt vector to be used by SAL to bring the APs to a spin loop as well as the external interrupt vector/memory variable to be used for the wake up. Refer to "SAL_MC_SET_PARAMS" on page 104 for details.

For the wake up functionality, the mechanism could be an external interrupt vector in the range of 0x10 to 0xFF or a memory variable.

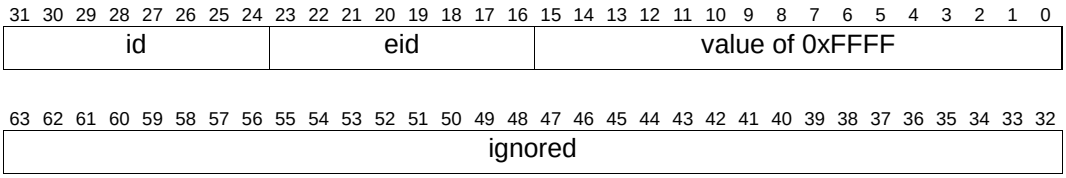
If external interrupt mechanism is chosen, APs will disable interrupts and poll the local SAPIC IRR register for the bit corresponding to the selected rendezvous interrupt to be set. The Task Priority Register (TPR) must be set such that a read of the IVR register will return the rendezvous interrupt vector (instead of the spurious interrupt), if one is pending. On receipt of the interrupt, the AP will read the IVR register and issue an End of Interrupt (EOI) to the local SAPIC to clear the interrupt bit. The AP will execute the next phase of SAL code and, if necessary, return to the wait loop.

Figure 3-2. Control Flow of Boot Process in a Multiprocessor Configuration



If a memory variable wake-up mechanism is chosen, the APs will disable interrupts and poll the memory variable for the unique value that matches the contents of their Local ID Register in bits 16-31 and a value of 0xFFFF in bits 0-15 (refer to Figure 3-3). The BSP will set this value to wake up one AP at a time. The AP will clear the memory variable to zero, execute the next phase of SAL code and, if necessary, return to the wait loop.

Figure 3-3. Wake-up Memory Variable Format



SAL exports details of the wake-up mechanism to the operating system through the SAL System Table (refer to Table 3-3) so that the operating system kernel code on the BSP may wake up the APs when appropriate. While memory variable mechanism may be used by the BSP and APs during the platform initialization phase, SAL shall indicate only the external interrupt wake-up mechanism to the operating system. The operating system shall not use the indicated external interrupt vector until it takes over the IVA. The operating system on the BSP will invoke the SAL_SET_VECTORS procedure to set the continuation point for the APs within the operating system kernel (OS_BOOT_RENDEZ) and then trigger the wake up of the APs. SAL will transition the APs to the registered OS_BOOT_RENDEZ entrypoint.

3.2.3 Platform Initialization Phase

This phase is primarily executed on the BSP. The APs will execute some of the steps described below. This phase will perform the following functions, the ordering of which is implementation- dependent:

1. Initialize the IVA to point to a 32 KB Interrupt Vector Table (IVT) in the firmware address space. Some SAL implementations may choose to build the IVT in RAM after finding the first 64 MB of memory. This step must be accomplished on all the processors in an MP environment.
2. Initialize the system fabric and chipsets. The method of handling the initialization is implementation-dependent.
3. On a cold boot, SAL will initialize at least the first 4 MB of memory for BSP late self-test. This self-test is done by calling the PAL_TEST_PROC procedure which returns information on whether the processor is healthy or not. This PAL procedure tests the path from the processor to the memory through the caches and returns information on whether the processor is fully functional. This PAL procedure will not return to the SAL if the processor under test experiences a catastrophic failure. SAL must contain logic to select a new BSP if necessary. SAL shall shut down the system if there are no healthy or performance-restricted processors in the system.

   After this point, the memory stack and RSE can be tested and enabled in the Itanium system environment.
4. Issue a rendezvous interrupt to wake up APs for a late self-test using the PAL_TEST_PROC procedure. The SAL code on the BSP must contain sufficient logic to detect APs that experience a catastrophic failure during the late self-test. On completion of late self-test, the BSP will set the APs back to the rendezvous state (Rendezvous_2 in Figure 3-2). After this stage, caches have been fully tested.

APs entering the rendezvous state at this point are required to execute the following steps to ensure that their caches only contain prefetches for firmware code and data. This restriction allows the processors to be safely removed during an on-line deletion operation without OS intervention.

1. Call PAL_PTCE_INFO to get information needed to purge translation caches, then use the parameters returned to execute the loop describing the ptc.e (purge translation cache entry) instruction in the Intel Itanium Architecture Software Developers Manual, Revision 2.2, Volume 3, Section 2.2.
2. Call PAL_PREFETCH_VISIBILITY with trans_type=1
3. Call PAL_CACHE_FLUSH with the inv parameter set to 1.
4. Call PAL_MC_DRAIN

After this sequence the processor ma enter into the rendezvous state.

Note: In multithreaded processors, PAL_TEST_PROC will disable the other thread while running, for up to several seconds. In addition, the architectural requirement that PAL_TEST_PROC cannot be called with the same memory buffer on multiple processors also applies to threads. SAL must not call this procedure with the same buffer on separate threads at the same time.

5. Search for console using implementation-dependent algorithms. If found, initialize the console so that the progress of the boot may be displayed.
6. Map and initialize memory.The memory test is implementation-dependent. The memory test includes testing of refresh logic and testing all the address lines for shorts.
7. Initialize the interrupt controllers to all interrupts disabled.
8. Allocate memory for use by PAL and SAL near the top of physical memory. This area should be below 4 GB if IA-32 code needs to call the SAL code with Itanium instructions, since IA-32 code can only address memory up to 4 GB.
9. Copy the PAL_B into memory using the PAL_COPY_PAL procedure. The PAL code in memory must be aligned such that the entire PAL space in memory may be covered by one Instruction Translation Register (ITR). It is recommended to copy PAL code and SAL code to contiguous locations in order that the operating system may cover the entire space using the same ITR. Refer to the *IIntel® Itanium® Architecture Software Developer's Manual* for PAL's requirements on ITR/DTR.
   Note: Until this step, the following floating-point transcendental instructions from the IA-32 instruction set cannot be executed:

   ```
   F2XM1, FCOS, FPATAN, FPTAN, FPREM, FPREM1,
          FSIN, FSINCOS, FYL2X, FYL2XP1
   ```

10. Copy SAL, PMI and IA-32 code to memory. The IA-32 BIOS code will be copied to the appropriate addresses in the address of 0x000C_0000 to 0x000F_FFFF. The portion of the SAL code containing Itanium instructions will be copied to a high memory address which must be above 1 MB. Copying code to RAM speeds up the boot sequence and additionally permits some portions of the code to be held in compressed format in the firmware address space. Firmware code may then be write-protected using the TLB or chipset features.
11. Set up an IVT in memory aligned on a 32 KB boundary and point the IVA register to it. This step must be accomplished on all the processors in an MP environment.
12. Register the SAL_PMI entrypoint in RAM with PAL. This step must be accomplished on all the processors in an MP environment.

13. Call the PAL_MC_REGISTER_MEM procedure on all the processors and specify PAL Min-State Save areas. These areas provide sufficient resources for the PAL code to perform the necessary machine check or INIT processing. Enable the BERR# and BINIT# sampling and signaling by invoking the PAL_BUS_SET_FEATURES procedure. Set the CMCI, MCA, and BERR# promotion strategy by invoking the PAL_PROC_SET_FEATURES procedure. These steps must be accomplished on all the processors in an MP environment.
14. Process configuration information in NVRAM and perform full chipset configuration. If NVRAM information is invalid, initialize NVRAM to default configuration values. Refer to the *Extensible Firmware Interface Specification* for details.
15. Initialize and configure I/O buses. Walk all buses, identify all resource requirements and set necessary range registers of chipsets. At this point, the complete system topology and addresses of all fabric segments are known.
16. Construct the ACPI Tables, SAL System Table and other shared data structures.
17. Execute the option ROMs as needed. If these contain IA-32 code, some of the IA-32 instructions may cause traps into the Itanium instruction set and suitable support needs to be provided by the trap/fault handler code. These interactions are more fully described in Volume 2 of the *Intel® Itanium® Architecture Software Developer's Manual,* and Chapter 7 of this specification. As a side effect of supporting IA-32 Option ROMs, it is possible to have some of the SAL code implemented in the IA-32 instruction set.
18. Copy the EFI code into memory and transfer control to it. Branch register BR0 shall be set up to point to the instruction following the call to the EFI code. The EFI firmware will search for bootable devices, load the operating system loader image and transfer control to it. EFI may utilize the underlying SAL and IA-32 BIOS layers for accesses to platform devices. Refer to the *Extensible Firmware Interface Specification* for interface description.

3.2.4 Firmware to Operating System Loader Hand-off State

The hand-off from firmware to Itanium architecture-based operating system loaders is fully described in the *Extensible Firmware Interface Specification*. Included in the hand-off are:

- The pointer to the SAL System Table (Section 3.2.6).
- The pointer to the Root System Description Pointer as described in the*Advanced Configuration and Power Interface Specification.*

All processor register state at the time of hand-off to the operating system loader is SAL implementation-dependent, except as follows:

- ARs:
  - — The backing store shall contain a minimum of 8 KB of available storage space in memory claimed by SAL.
  - — RSC will indicate enforced lazy mode, little-endian.
  - — IOBASE (AR.k0) will contain the virtual address of the IA-32 I/O port block.

Note: Only SAL implementations that execute IA32 BIOS code set IOBASE (AR.k0) to contain the virtual address of the IA-32 I/O port block. The AR.k0 virtual address is identity-mapped and only usable by the SAL environment.

- GRs:
  - — GR12 = Stack pointer with a minimum of 8 KB of available storage space in memory claimed by SAL.

- PSR:
  - PSR.ac = 1 (alignment check enabled).
  - PSR.ic = 1, PSR.i = 0 (interrupt collection on, interrupts off). There may be some pending interrupts.
  - PSR.it, PSR.dt, PSR.rt = 0 (instruction translation, data translation and RSE translation off).
  - PSR.bn = 1 (register bank 1 selected).
  - PSR.dfl, PSR.dfh = same values as on entry from PALE_RESET.
  - All other bits = 0.
- CRs:
  - DCR: Bus lock setting (DCR.lc) is platform implementation-dependent, all other bits of DCR = 0.
  - IVA = physical address of a SAL implementation-dependent IVT.
  - PTA.ve = 0 (if the virtual hash page table (VHPT) is disabled).
  - LID = the unique id/eid value for this processor.
- Data Breakpoint Registers – DBRs: Same as on entry to SALE_ENTRY.
- Instruction Breakpoint Registers – IBRs: Same as on entry to SALE_ENTRY.
- RRs

  Region Register 0 will contain an ID of 0x1000. Other Region Registers will have implementation-dependent values except that RRs 1-3, if non-zero, will contain Region ID values of 0x1001-0x1003 respectively.
- Protection Key Registers – PKRs, are set to 0.
- TLB:
  - TRs: ITR(0) will map an area that includes the SAL's IVT and PAL code. All other TR entries are invalidated.
  - TCs: These are implementation-dependent and no expectation can be assumed. SAL invalidates all TC entries on APs before entering the rendezvous state.
- Caches

  Enabled, coherent and consistent with the contents of memory.

3.2.5 OS_BOOT_RENDEZ

OS_BOOT_RENDEZ is the entrypoint for operating system MP rendezvous code. The operating system code on the BSP registers this entrypoint by invoking SAL_SET_VECTORS, supplying the 16-byte aligned physical address of the operating system code. SAL exports details of the wake-up mechanism to the operating system through the SAL System Table (refer to Table 3-10) so that the operating system kernel code on the BSP may wake up the APs when appropriate. When SAL on the APs receives the wake-up, it will call the registered OS_BOOT_RENDEZ entrypoint. Refer to Section 3.2.2.1, "Rendezvous Functionality" for additional details.

The OS_BOOT_RENDEZ hand-off requirements specified in Section 3.2.4, "Firmware to Operating System Loader Hand-Off State" are the same for the BSP and APs (each BSP's and AP's SP points to a separate memory buffer, and each AP's backing store shall contain a minimum of 8KB of available storage space of its own in memory claimed by SAL) with the following exception:

BR0 = Return address into the SAL boot rendezvous routine. If OS_BOOT_RENDEZ returns a processor to the SAL using the Branch register BR0, SAL will re-enter the spin loop awaiting a wake-up by the BSP.

When returning a processor to SAL boot rendezvous using branch register BR0, the OS must return the processor system registers to the state specified in Section 3.2.5.1, “OS_BOOT_RENDEZ to SAL Return State”.

SAL must implement the following behavior for processors in SAL boot rendezvous during boot time or runtime and upon entry or return to SAL boot rendezvous:

- Processors in boot rendezvous are required to execute the following steps to ensure that their caches only contain prefetches for firmware code and data. This restriction allows the processors to be safely removed during an on-line deletion operation without OS intervention.
    1. Call PAL_PTCE_INFO to get information needed to purge translation caches, then use the parameters returned to execute the loop describing the ptc.e (purge translation cache entry) instruction in the Intel Itanium Architecture Software Developers Manual, Revision 2.2, Volume 3, Section 2.2.
    2. Call PAL_PREFETCH_VISIBILITY with trans_type=1
    3. Call PAL_CACHE_FLUSH with the inv parameter set to 1.
    4. Call PAL_MC_DRAIN
- Processors in SAL boot rendezvous should not hand off to the OS on MCA or INIT events. Note: MCA or INIT events may occur in the time the OS returns the processor through BR0 to the time SAL sets itself up to avoid handing off the INIT or MCA events.
- If a fatal, non-maskable MCA event is detected (for example, BINIT), SAL should not hand the processor to the OS. Possible implementations include:
    - Resume to the interrupted context (BR0 for the rendezvoused processor) before clearing the respective processor error log.
    - Spin in a tight loop within the SAL MCA handler and wait for a system reset.
- SAL and the OS should avoid sending any MC Rendezvous or MC Wakeup IPIs to processors in SAL boot rendezvous.

3.2.5.1 OS_BOOT_RENDEZ to SAL Return State

The following conventions, compatible with the *Itanium® Software Conventions and Runtime Architecture Guide*, govern the transition from OS_BOOT_RENDEZ to SAL using the BR0 return address. From the perspective of the Software Conventions, SAL is considered the caller and OS_BOOT_RENDEZ is the callee.

Floating-point, Predicate, and Branch Registers use the standard calling convention.

General Registers use the standard calling convention, except that:

- GR1 (gp): Preserved.
- GR12 (sp): Preserved.
- GR13 (tp): Preserved.
- Bank 0 GR16 - GR31: Scratch.

Application Registers use the standard calling convention, except that:

- FPSR:
  - — trap disable bits: Preserved
  - — sf0-sf3 control bits: Preserved
  - — sf0-sf3 flag bits: Scratch
- RNAT: Preserved.
- BSPSTORE: OS_BOOT_RENDEZ must restore the backing store to the state described in Section 3.2.4.
- RSC: OS_BOOT_RENDEZ must restore RSC to the state described in Section 3.2.4.
- ITC: Scratch.
- KR0 - KR7: Scratch.
- All defined ARs not mentioned in the convention (for example, the IA-32 ARs) are scratch.

The system register conventions are described in Table 3-1.

Table 3-2. OS_BOOT_RENDEZ to SAL System Register Conventions

Name	Description	Class
PSR	Processor Status Register	Preserved[1]
DCR	Default Control Register	Preserved
ITM	Interval Timer Match Register	Scratch
IVA	Interruption Vector Address	Preserved
PTA	Page Table Address	Preserved
GPTA	Reserved IA-32 Resource	Unchanged
IPSR	Interruption Processor Status Register	Scratch
ISR	Interruption Status Register	Scratch
IIP	Interruption Instruction Bundle Pointer	Scratch
IFA	Interruption Faulting Address	Scratch
ITIR	Interruption TLB Insertion Register	Scratch
IIPA	Interruption Instruction Previous Address	Scratch
IFS	Interruption Function State	Scratch
IIM	Interruption Immediate Register	Scratch
IHA	Interruption Hash Address	Scratch
LID	Local Interrupt ID	Unchanged
IVR	Interrupt Vector Register (Read Only)	Scratch
TPR	Task Priority Register	Scratch
EOI	End of Interrupt	Scratch
IRR0-IRR3	Interrupt Request Registers 0-3 (Read Only)	Scratch
ITV	Interval Timer Vector	Preserved
PMV	Performance Monitoring Vector	Preserved
CMCV	Corrected Machine Check Vector	Preserved
LRR0-LRR1	Local Redirection Registers 0-1	Preserved
RR	Region Registers	Preserved
PKR	Protection Key Registers	Preserved[1]

Table 3-2. OS_BOOT_RENDEZ to SAL System Register Conventions (Continued)

Name	Description	Class
TR	Translation Registers	Invalidated[2]
TC	Translation Cache	Scratch
IBR/DBR	Breakpoint Registers	Scratch
PMC	Performance Monitor Control Registers	Scratch
PMD	Performance Monitor Data Registers	Scratch

Notes:

1. As described in Section 3.2.4, when OS_BOOT_RENDEZ is invoked, these registers have known constant values, so the OS needn't actually preserve the values, but can simply recreate the known values.
2. All TRs must be invalidated, including ITR[0]. SAL must set-up ITR[0] again, before reinvoking OS_BOOT_RENDEZ on a subsequent rendezvous interrupt.

3.2.5.2 Boot Strap Processor Return to SAL

Section 3.2.5.1 describes how to return Application Processors to firmware. This interface is based on the standard calling conventions, where SAL is considered the Caller and OS_BOOT_RENDEZ the Callee. The OS saves the register context of the firmware when it launches a new CPU, and restores it when it returns a hot removed CPU to the firmware. Since the Boot Strap Processor (BSP) is handed-off from the bootloader, this model cannot be applied to the BSP.

The following rules allow the OS to return the BSP:

- Operating Systems shall use the BR0 Return address of one of the Application Processors (AP) to return the BSP.
- The OS shall initialize the IVA using the value preserved for one of the AP.
- The following registers shall be preserved (i.e., OS to populate them based on the firmware to OS loader handoff state value as defined in section 3.2.4)
  - PSR (physical mode with interrupts turned off)
  - RSC (i.e., enforced lazy mode, little endian)
  - DCR
- The following registers shall be unchanged
  - LID
- System firmware shall reinitialize the following registers upon handoff:
  - GRs (including the gp and sp)
  - BSPSTORE
  - ITV
  - DCR
  - PMV
  - CMCV

3.2.6 SAL System Table

SAL uses the SAL System Table to export a variety of information to the operating system loader. The pointer to the SAL System Table is provided by EFI to the operating system loader. Refer to the *Extensible Firmware Interface Specification* for hand-off details. If a recovery condition is present, the SAL System Table is not built, and a pointer value of 0 is provided.

The SAL System Table begins with a header which is described in Table 3-3. The SAL System Table header will be followed by a variable number of variable length entries. The first byte of each entry will identify the entry type and the entries shall be in ascending order by the entry type. Each entry type will have a known fixed length. The total length of this table depends upon the configuration of the system. Operating system software must step through each entry until it reaches the ENTRY_COUNT. The entries are sorted on entry type in ascending order. Table 3-4 describes each entry type. Unless otherwise stated, there is one entry per entry type.

Table 3-3. SAL System Table Header

Field	Offset (Bytes)	Length (Bytes)	Description
SIGNATURE	0	4	The ASCII string representation of “SST_”, which confirms the presence of the table.
TOTAL_TABLE_ LENGTH	4	4	The length of the entire table in bytes, starting from offset zero and including the header and all entries indicated by the ENTRY_COUNT field.This field aids in calculation of the checksum.
SAL_REV	8	2	The revision number of the *Itanium® Processor Family System Abstraction Layer Specification* supported by the SAL implementation in binary coded decimal (BCD) format. Byte 8 – Minor[1] Byte 9 – Major [2] SAL Revision 3.3 (0x0330)[3] corresponds to SAL Spec, March 2008. SAL Revision 3.2 (0x0320) corresponds to SAL Spec, December 2003. SAL Revision 3.1 corresponds to SAL Spec, November 2002. SAL Revision 3.0 corresponds to SAL Spec, January 2001 or July 2001. SAL Revision 2.9 corresponds to SAL Spec, July 2000. SAL Revision 2.8 corresponds to SAL Spec, January 2000.
ENTRY_COUNT	10	2	The number of entries in the variable portion of the table. This field helps software in identifying the end of the table when stepping through the entries.
CHECKSUM	12	1	A modulo checksum of the entire table and the entries following this table. All bytes including the Checksum bytes must add up to zero.
RESERVED	13	7	Unused, must be zero.
SAL_A_VERSION	20	2	Version Number of the SAL_A firmware implementation in BCD format. Byte 20 – Minor Byte 21 – Major
SAL_B_VERSION	22	2	Version Number of the SAL_B firmware implementation in BCD format. Byte 22 – Minor Byte 23 – Major

Table 3-3. SAL System Table Header (Continued)

Field	Offset (Bytes)	Length (Bytes)	Description
OEM_ID	24	32	An ASCII identification string which uniquely identifies the manufacturer of the system hardware. This string can be exactly 32 bytes in length or shorter if null terminated. Compliance with the SAL specification requires that this string be unique with respect to all other manufacturers. It is forbidden to use another manufacturer's identification even if the system is otherwise identical.
PRODUCT_ID	56	32	An ASCII identification string which uniquely identifies a family of compatible products from the manufacturer. This string can be exactly 32 bytes in length or shorter if null terminated.
RESERVED	88	8	Unused, must be zero.

Notes:

1. An increase in the minor revision value indicates that changes are compatible with software based on earlier revisions. This includes, but is not limited to, errata, expansion of functionality of existing APIs through the use of reserved fields, and the addition of new APIs.
2. An increase in the major revision is required when changes may not be compatible with software that is based on the previous major revisions. For example, modifying the behavior of an API for pre-existing arguments would be a change that is not compatible.
3. The format 0x1234 conveys the major number encoded in the first two hex digits and the minor in the last two, with a fixed point assumed in between.

Table 3-4. SAL System Table Entry Types

Entry Type[1]	Entry Length (in Bytes)	Description
0	48	Entrypoint Descriptor.
1	32	Memory descriptor (one entry for each contiguous block with similar attributes).[2]
2	16	Platform Features Descriptor.
3	32	Translation Register Descriptor (one entry for each TR used by SAL at the time of hand-off to the operating system).
4	16	Purge Translation Cache (PTC) Coherence Descriptor.
5	16	AP Wake-up Descriptor.

Notes:

1. All other types are reserved.
2. Not required for Itanium® architecture-based operating systems.

3.2.6.1 Entrypoint Descriptor Entry

The Entrypoint Descriptor Entry (refer to Table 3-5) provides the addresses in memory of PAL_PROC and SAL_PROC that may be used by the operating system to invoke the procedures within the PAL and the SAL. When the operating system calls SAL_PROC, the GP register must contain the physical or virtual address of the SAL's GP value specified in the Entrypoint Descriptor, depending on the mode in which the SAL_PROC procedure is called.

Table 3-5. Entrypoint Descriptor Entry Format

Offset (in Bytes)	Length (in Bytes)	Description
0	1	Entry type = 0 denoting Entrypoint Descriptor type.
1	7	Reserved (must be zero).
8	8	Physical address of the PAL_PROC entrypoint in memory.

Table 3-5. Entrypoint Descriptor Entry Format (Continued)

Offset (in Bytes)	Length (in Bytes)	Description
16	8	Physical address of the SAL_PROC entrypoint in memory.
24	8	Global Data Pointer (physical address value) for SAL procedures.
32	16	Reserved (must be zero).

3.2.6.2 Platform Features Descriptor Entry

The Platform Features Descriptor Entry (refer to Table 3-6) describes the features implemented on the platform.

Table 3-6. Platform Features Descriptor Entry

Offset (in Bytes)	Length (in Bytes)	Description
0	1	Entry type = 2 denoting Platform Features type.
1	1	Platform Feature List: Bit 0: 1 if Bus Lock is implemented on the processor as well as the platform. Bit 1: 1 if the chipset supports redirection hint for interrupt messages originating from the platform (lowest priority interrupt). Bit 2: 1 if the chipset supports redirection hint for IPI messages originating from the processors. Bit 3: 1 if Interval Time Counters (ITCs) among processors in the system may drift from each other. 0 if the processor ITCs will not drift from each other once synchronized. Bits 4-7 = Reserved.
2	14	Reserved.

3.2.6.3 Translation Register Descriptor Entry

The Translation Register Descriptor entries (refer to Table 3-7) describe the parameters used by the SAL during insertion of the TRs. These entries will be used by the operating system to purge SAL's TRs after the operating system takes over the IVA. As specified in Section 3.3.2.1, "TLB Resource Partition" SAL is only allowed to use ITR(0). This table will only contain the ITR(0) mapping.

Table 3-7. Translation Register Descriptor Entry

Offset (in bytes)	Length in bytes)	Description
0	1	Entry type = 3 denoting the Translation Register Descriptor type.
1	1	Type of Translation Register: 0: Instruction Translation Register 1: Data Translation Register Other values: Reserved
2	1	Translation Register number.
3	5	Reserved.
8	8	Virtual address of the area covered by the Translation Register. Bits 61-63 of this field indicate the Region Register number.
16	8	Encoded value of the page size covered by the Translation Register. Refer to the *Intel® Itanium® Architecture Software Developer's Manual, Addressing and Protection* chapter for the format of this field.
24	8	Reserved.

3.2.6.4 Purge Translation Cache Coherence Domain Entry (Optional)

The purge translation cache (PTC) Coherence Domain Entry (refer to Table 3-8) describes the number of coherence domains and the scope of PTC instruction propagation for each domain.

Table 3-8. Purge Translation Cache Coherence Domain Entry

Offset (in Bytes)	Length (in Bytes)	Description
0	1	Entry type = 4 denoting PTC Coherence Domain Entry type.
1	3	Reserved (must be zero).
4	4	Number of coherence domains for the platform.
8	8	64-bit memory address of the coherence domain information.

Platforms must provide a mechanism for detecting which TLB coherence domain a processor lives in. SAL captures this information in an implementation-dependent manner and passes the same to the operating system.

The coherence domain information is an array of length of (16 • Number of coherence domains). As shown in Table 3-9, for each coherence domain, there will be two information fields:

1. Number of processors in the TLB coherence domain.
2. 64-bit memory address of a list of Local ID register values for the processors within the TLB coherence domain. Each logical processor will require two bytes of memory (*id* field in low order byte and *eid* field in high order byte) to represent the Local ID information.

Table 3-9. Coherence Domain Information

Offset (in Bytes)	Length (in Bytes)	Description
0	8	Number of processors in TLB coherence domain #1.
8	8	64-bit memory address of a list of Local ID register values for the processors within the TLB coherence domain #1.
16	8	Number of processors in TLB coherence domain #2.
24	8	64-bit memory address of a list of Local ID register values for the processors within the TLB coherence domain #2.
–	–	–
–	–	–
16•(N-1)	8	Number of processors in TLB coherence domain #N.
8+16•(N-1)	8	64-bit memory address of a list of Local ID register values for the processors within the TLB coherence domain #N.

3.2.6.5 Application Processor Wake-Up Descriptor Entry (Optional)

The AP Wake-up Descriptor Entry (refer to Table 3-10) describes the mechanism for waking up APs in an MP environment. Refer to Section 3.2.2.1, “Rendezvous Functionality” for details on operating system usage of this entry. This entry is required for MP configurations.

Table 3-10. Application Processor Wake-up Descriptor Entry

Offset (in bytes)	Length (in bytes)	Description
0	1	Entry type = 5 denoting AP Wake-up Descriptor Entry type.
1	1	Wake-up Mechanism type: 0: External interrupt Other values: Reserved
2	6	Reserved (must be zero).
8	8	External Interrupt vector in the range of 0x10 to 0xFF.

3.3 Itanium® Architecture-based Operating System Loader Requirements

The firmware will jump to the Itanium architecture-based operating system loader with the hand-off state described in the *Extensible Firmware Interface Specification*. Included in this state information is a pointer to the SAL procedures that the operating system can invoke. These procedures are described in Chapter 9.

This section describes the requirements on the operating system loader while operating under the SAL execution environment.

3.3.1 Fault Handling

This section describes the fault-handling guidelines for the operating system loader.

After the operating system is completely loaded[1], it will take control of the IVA and replace the SAL environment with its own memory management. Until then, the operating system shall use SAL's virtual memory environment — IVA, Interrupt controller mode, TC mappings, and so on, and it shall not change any of these resources. The operating system is not permitted to replace the fault handler entries within the SAL's Interrupt Vector Table (IVT).

The operating system loader code may be executed in physical mode with interrupts disabled, or in virtual mode with Instruction, Data and RSE translation on (PSR.it = 1, PSR.dt = 1, PSR.rt = 1). While executing in virtual mode, the operating system loader code is permitted to cause TLB faults for which SAL shall provide the appropriate fault handlers. These TLB faults are:

- Alternate Instruction TLB fault: This TLB fault occurs during instruction fetches if SAL does not implement the Virtual Hash Page Table (VHPT). If VHPT is not used, the Page Table Address (PTA) need not be initialized and the SAL will turn off the PTA.ve bit to disable the processor walking the VHPT. VHPT is an optional feature of the Itanium architecture. Avoiding VHPT usage also permits the IA-32 support code to operate out of the firmware address space.
- Alternate Data TLB fault: This TLB fault occurs during data accesses if SAL does not implement the VHPT. The SAL's fault handler shall test whether the TLB fault surfaced during speculative load accesses (LDx.s). Such an access is indicated if the ISR.sp bit is set. If this bit is set, the SAL shall return to the faulting instruction with the IPSR.ed bit thereby turning on the NaT bit of the target register for the load.

1. The OS is considered loaded at the successful completion of the EFI ExitBootServices() call.

- VHPT-related faults: VHPT translation fault, Data TLB fault and Nested TLB fault, if SAL implements VHPT.
- Instruction and Data Access Rights faults: SAL shall install TCs with the page privilege level set to 0 and execute code with the PSR.cpl value to 0. On processor implementations with unified TLBs, Access Rights faults may surface if the TC is present but the required page permissions are not present, for example, TC is present with RW page access rights but RX page access rights is needed for instruction execution.
- External interrupt: Hardware interrupts will be received by SAL in the Itanium system environment. This code will read the IVR register. If the vector read is 0, it signifies an interrupt from the 8259 interrupt controller and SAL must issue a load to the architected INTA_address (default address 0xFEFE_0000) in the processor interrupt delivery block to issue an interrupt acknowledge (INTA) bus cycle and obtain the interrupt vector from the 8259. SAL will then jump to the appropriate interrupt handler using its internal tables. If the interrupt needs to be reflected to IA-32 code, the address will be derived from the IA-32 Interrupt Descriptor Table. The operating system loader is restricted from sending IPI messages (that is, causing bits in the SAPIC IRR registers to be set) with vector values other than the one specified in the AP Wake-up Descriptor Entry (refer to Table 3-10).
- SAL may install TC entries with the Present, Dirty, and Accessed bits on and thereby avoid Page not present, Data Dirty bit, and Data Access bit faults.
- SAL may disable Protection Key checking (PSR.pk = 0) and thereby avoid Instruction Key miss, Data Key miss, and Key Permission faults.
- Speculation fault: Speculation faults are caused by CHK.s, CHK.a, and FCHK instructions. SAL will provide the transition mechanism to the recovery code.
- Unaligned fault: The operating system loader shall not make data references to misaligned data. However, this fault may arise during speculative load accesses. Such an access is indicated if the ISR.sp bit is set. If this bit is set, the SAL shall return to the faulting instruction with the IPSR.ed bit thereby turning on the NaT bit of the target register for the load. A similar logic must be incorporated in SAL's Alternate Data TLB fault handler.
- SAL shall not use advanced load (LD.a) or check load (LD.c) instructions, hence ALAT entries created by operating system loader code are preserved across SAL calls and SAL's fault handlers.
- Divide by zero: SAL shall display an error message for the Break interrupts caused by the run-time checking of integer divide by zero. Refer to the *Itanium® Software Conventions and Runtime Architecture Guide*.

The operating system must not rely on any other fault handlers installed by SAL. SAL will display an error message if an unsupported fault is encountered. SAL will not provide support for the following faults:

- Nested TLB fault: ITR(0) will map the SAL's IVT and the code areas covering SAL's fault handlers. All fault handlers in SAL shall run with PSR.dt, PSR.rt turned off to avoid the nested TLB fault that can occur while accessing the fault handler's local variables and data structures.
- NaT Consumption fault: NaT Consumption faults are generated by non-speculative operations (for example, load, store, control register access, instruction fetch, and so on) that use a source register containing a NaT value or reference a NaTPage. Properly constructed code should never generate this fault.

- General Exception fault: The operating system loader shall not cause the general exception fault by executing illegal operations, invoking SAL procedures in physical/ virtual mode with arguments specifying unimplemented data addresses.
- Floating-point faults: The operating system loader shall not disable accesses to the floating-point register sets by setting PSR.dfl or PSR.dfh bits or cause any floating-point exceptions.
- Other traps/faults: The operating system loader must not cause other traps or faults such as debug, single step, taken branch, and so on. Normally, the operating system kernel provides these services after it takes over the IVA.

Additional fault handlers to support IA-32 execution are described in Chapter 7.

3.3.2 Memory Management Resources Usage

This section describes SAL's usage of various memory management resources and provides guidelines for their use by the operating system loader code.

3.3.2.1 TLB Resource Partition

SAL will use only TCs and the ITR(0). Use of multiple Translation Registers (TRs) by SAL may cause problems with booting of Itanium architecture-based operating systems. The operating system loader is free to use TRs other than ITR(0).

Note: When setting up new TRs, the OS loader must not overlap its own TRs with the ITR[0] set up by system firmware. In addition, OS loaders should not assume that memory is contiguous when mapping its TRs.

The advantage of this resource partition is that hardware interrupts which cause a transition to SAL will not affect the TRs set up by the operating system loader. Ideally, the operating system loader will set up the TRs for its memory mappings and not cause TLB faults. However, should the operating system loader code cause a TLB miss, the TLB Miss handler in SAL would automatically install a TC with identity mapping. The restriction on ITR(0) is not relevant after the operating system takes over the memory management and the IVA.

Use of TCs in SAL code should not cause any performance problems since SAL is not performance critical. Most of the SAL code will write and read back memory addresses traversing the entire physical address space. Use of additional TRs will not provide improved performance. SAL will primarily be limited by memory and I/O speeds.

SAL will use TC entries with length of 4KB by default and will try to coalesce contiguous entries with similar attributes into larger page sizes.

3.3.2.2 Identity Mapping Usage

The Itanium processor virtual address range is 85 bits wide and the Itanium processor physical address range is 63 bits wide. Bits 0 to 60 of the virtual address provide the virtual page number and offset. Bits 61 to 63 of the virtual address are used as an index into the Region Registers which supplies a Region ID value that can be up to 24 bits wide. Thus the 85-bit virtual address comprises the low order 61 bits of the virtual address and the 24-bit Region ID. This 85-bit virtual address is transformed into a 63-bit physical address by the Itanium processor's TLB mechanism as described in the *Intel® Itanium® Architecture Software Developer's Manual*.

SAL will use identity mappings in which virtual addresses are equal to physical addresses. The advantage of identity mapping is that the same pointer can be used to access the same memory location regardless of the state of the PSR.dt bit.

3.3.2.3 Unique Region IDs for SAL

The firmware will load the operating system loader and jump to it. The operating system loader will load the rest of the operating system using the firmware boot services procedures. While SAL can operate with identity mapping, there may be a need for the operating system loader to use a non-identity mapping. As an example, there may be an I/O device at physical address 2.5 GB for which SAL would have established an identity mapping with uncacheable memory attribute. The operating system loader may need to load additional layers of software and fix up address relocations using virtual addressing. The operating system loader may need to load software at physical address 0.5 GB mapped to virtual address of 2.5 GB. When operating system refers to the virtual address 2.5 GB, it is referring to RAM at 0.5 GB and when SAL refers to 2.5 GB virtual address, it is referring to the I/O device at 2.5 GB physical address. Clearly, the operating system loader cannot use the TLB mapping set up by SAL for this case.

This problem can be solved by using unique Region registers and Region ID values for the SAL and the operating system. Differing Region ID values ensure that earlier TC/TR entries with a different Region ID value no longer cause TLB hits.

Since SAL uses 64-bit addressing, if the physical address space is less than or equal to 2^{61} bytes, SAL will be capable of addressing the entire physical address space using Region Register 0. In general, the SAL would need only Region Register 0, leaving the other Region Registers for operating system use. SAL shall set up the Region Register 0 with a Region ID value of 0x1000, if physical address space is less than or equal to 2^{61} bytes. If the physical memory is larger, it shall load the Region Registers 1 to 3 with Region ID values of 0x1001 to 0x1003, respectively.

The operating system loader shall not change the contents of Region Registers that are in use by SAL. If the value in Region Register 0 is changed, access to the IVT is lost and the system will crash. Similarly, the operating system loader shall be restricted from using Region ID values of 0x1000 to 0x1003 until operating system is ready to take over the memory management and the IVA. If this restriction is not followed by the operating system loader, a MCA might result when SAL attempts to insert a TC entry using the ITC.i or ITC.d instruction. Should the operating system loader set up any of the Region Registers unused by SAL, it shall:

- Set the ve bit in the Region Register to 0 to disable the VHPT.
- Set the ps bit's value to indicate preferred page size of 4 KB.

The operating system loader will need to refer to the data structures common to SAL and operating system in the process of loading the operating system kernel. Similarly, the operating system will need to pass parameters to SAL through pointers in Memory Stack Pointer (SP) and Global Data Pointer (GP) registers. The SAL and the operating system must refer to these common data structures using Region Register 0, that is, the virtual addresses used to address the common data structures must have bits 61-63 set to 0.

3.3.2.4 Memory Attribute Aliasing Guidelines

Several memory resources are used by the PAL/SAL and the OS. The memory attributes between these need to be consistent in order to avoid memory attribute aliasing (that is, the same page cannot be accessed in both UC and WB modes).

Memory attribute aliasing may result in stale data left in the processor cache and may cause a processor machine check event. This section describes the steps that can be taken to ensure consistent access to common memory areas by the PAL, SAL, and OS.

The Min-State Save area is used by the PAL to dump registers and processor state during MCA and INIT. The PAL requires that the Min-State Save area is UC. To ensure consistent access to the Min-State Save area, the SAL must:

- Define the Min-State Save area as UC separated from other data areas by 16K (this is to avoid any prefetching side effects).
- Report the Min-State Save area to the OS as EfiMemoryMappedIO (UC).
- All SAL accesses to Min-State Save area performed in physical addressing mode, uncached only.

The firmware address space contains the PAL/SAL entrypoints which are accessed in physical address mode, uncached. These are the PAL entrypoints for reset, init, and machine check, and the SALE_ENTRY entrypoint for the SAL layer. In addition, both the SAL and PAL access the FIT to get the address of other firmware components and the flash read/write utilities access the firmware address space. To avoid memory attribute aliasing, the firmware address space should be reported to the OS as EfiMemoryMappedIO (UC).

When calling SAL procedures, the caller must set bit 63 of arguments which are physical addresses, according to the argument's memory attribute.

3.3.3 Other Restrictions on the Operating System

1. The operating system shall not change the values of the following system resources:
    - LID register, the unique id/eid value for this processor.
    - DCR.lc, the Bus lock setting for the platform, if the same is set to 1. Note that the PAL_BUS_SET_FEATURES procedure may be invoked to execute the locked transactions as a series of non-atomic transactions. Refer to the *Intel® Itanium® Architecture Software Developer's Manual* for details.
    - Physical address of the Processor Interrupt Block Address.
    - Physical address of the IA-32 I/O Port Block.
    - Physical location of the PAL procedures within memory. SAL copies the PAL procedures into memory using the PAL_COPY_PAL procedure during the system boot operation.
    - The value in the IOBASE register (AR.k0) until the OS takes over the IVA.
2. The operating system shall not change the Min-State Save area which was registered by the SAL using the PAL_REGISTER_MEM procedure.
3. The operating system shall not change the location of the PAL procedures within memory. SAL copies the PAL procedures into memory using the PAL_COPY_PAL procedure.
4. The operating system creates virtual address mappings for the PAL and the SAL procedures and registers them with the firmware using interfaces provided by the *EFI specification*. The operating system shall not alter the virtual address mappings after such a registration, as this is not permitted by the *EFI specification.*
5. The operating system may lower the CMCI, MCA, and BERR# promotion strategy set by SAL by invoking the PAL_PROC_SET_FEATURES procedure, but this is not recommended.
6. Refer to Table 9-2 for restrictions on the OS from calling certain PAL procedures.

7. In addition to the handlers for virtual memory management (if in virtual addressing mode), the OS must provide these handlers when invoking SAL runtime services in virtual addressing mode:
   - Floating Point Fault Vector (0x5C00): This vector provides the floating point software assists.
   - Speculation Vector (0x5700) if check branching is not implemented on the processor. This vector provides the software assists for speculative check instructions.
   - External Interrupt Vector (0x3000) if external interrupts are enabled (PSR.i == 1).

§

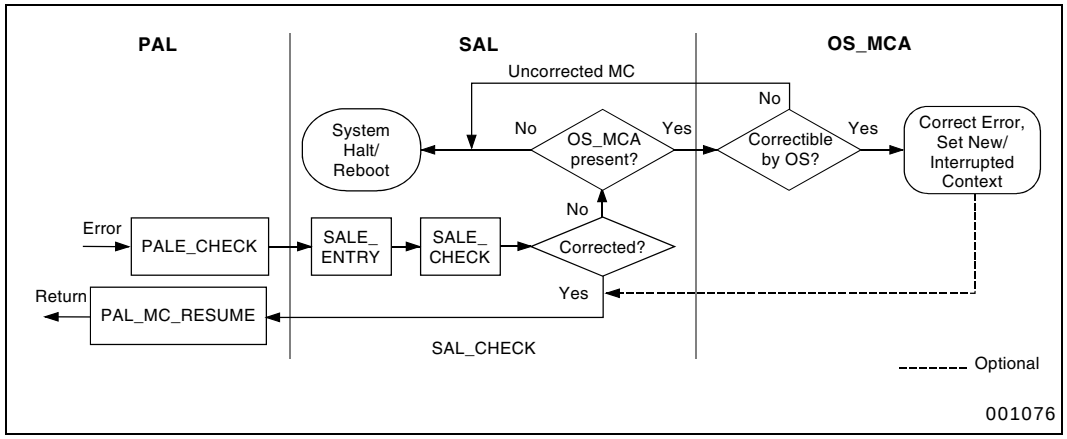
4 Machine Checks

Machine checks, including MCAs, and expected machine checks cause the processor to jump to the PALE_CHECK entrypoint in PAL. Please refer to Volume 2, Chapter 11 in the *Intel® Itanium® Architecture Software Developer's Manual* for details regarding PALE_CHECK processing. Also refer to the *Intel® Itanium® Processor Family Error Handling Guide* for error handling from a system software perspective.

When PALE_CHECK has finished processing, it will pass control to SALE_ENTRY entrypoint in SAL, which in turn branches to the SAL MCA handler. The entry conditions for SALE_ENTRY are described in the *Intel® Itanium® Architecture Software Developer's Manual.*

This chapter defines the actions required of SAL_CHECK as well as optional considerations. Figure 4-1 shows a simplified control flow of Machine Check processing.

Figure 4-1. Overview of Machine Check Flow



Uncorrected machine checks refer to errors that cannot be corrected at PAL and SAL layers. These may still be fully or partially recoverable at the operating system layer. The control flow differs between corrected and uncorrected machine checks. For corrected machine checks, the operating system-corrected error interrupt handlers will be invoked some time after returning to the interrupted process. Section 4.1 describes the functionality and processing steps for the uncorrected machine checks and Section 4.2 describes the corrected machine checks.

4.1 SAL_CHECK

SAL_CHECK has the basic responsibility for the following:

- Record processor and platform error information.
- Save the processor and platform state information.
- Perform any platform hardware-specific corrections.
- For uncorrected machine checks, validate the OS_MCA entrypoint and branch to it.
- Clear the error record resources and re-enable future information collection.
- Halt the processor or platform as necessary.

- Handle MP situations.
- Ensure that processors in SAL boot rendezvous do not hand off to the OS on MCA events. This includes fatal, non-maskable MCA events. SAL should also avoid sending MC Rendezvous or MC Wakeup IPS to these processors. See Section 3.2.5, “OS_BOOT_RENDEZ” for more details.

In addition, it is useful to note that where hardware/firmware cannot fix a machine check condition, SAL_CHECK should provide the necessary information and conditions to allow the operating system to recover whenever possible. It is expected that most of the error recovery is performed at the OS_MCA layer. The amount of state information saved by SAL is implementation-dependent and the SAL_GET_STATE_INFO procedure provides validation bits indicating the saved state information.

4.1.1 SAL_CHECK Processing Details

During boot, SAL_RESET code will call PAL_MC_REGISTER_MEM to assign PAL a Min-State Save area used to deposit minimal processor state information, when PAL performs PALE_CHECK processing. This step is performed on all the processors in the system.

During the platform test and initialization stage, SAL may invoke the PAL_MC_EXPECTED procedure to notify PAL that a machine check may surface and that PAL must not attempt to correct the error. If the machine check was expected by SAL, SAL will check the results of the operation, invoke PAL_MC_EXPECTED to notify PAL that machine check is no longer expected, and resume execution by calling PAL_MC_RESUME.

When an unexpected machine check event has occurred and SAL_CHECK is entered, it is the responsibility of SAL_CHECK to call back to PAL (PAL_MC_ERROR_INFO), in order to retrieve processor-specific error information pertaining to the machine check. In addition, SAL_CHECK should interrogate the platform (through error logging registers) for any platform-specific information which pertains to the machine check condition. Calls to PAL_MC_DYNAMIC_STATE can be made to obtain processor state information for Intel to use to assist in debug. Check the PSP.dy bit to ensure the dynamic state information is valid and available. Once the processor error record information is retrieved, SAL_CHECK will call PAL_MC_CLEAR_LOG to make the processor error logging resources available for capturing future machine check error information. A similar task is necessary to make platform error logging resources available for future events.

An error due to an MCA event, when corrected by firmware becomes a Processor Corrected Machine Check or a Platform Corrected Error event condition. A hand-off to the OS_MCA is also not required during this event type transformation.

When multiple processors experience machine checks simultaneously, SAL selects a “monarch” machine check processor to accumulate all the error records at the platform level and continue with the machine check processing. “Monarch” status is relevant only for the current MCA error event.

SAL is responsible for reporting the state information to the operating system via the SAL_GET_STATE_INFO call so that the operating system can make the determination to:

- Fix the error and return to interrupted or new context through the SAL MCA handler, or
- Request the SAL MCA handler to reset the platform.

SAL_CHECK shall not hide any architectural state from the OS_MCA layer. This permits the OS_MCA layer to run unencumbered. OS_MCA can save the processor and platform state and re-enable future machine checks as soon as possible. Otherwise, OS_MCA would be constrained to operating with machine checks disabled in order to preserve the architectural information at the PAL and SAL layers.

When the operating system registers the OS_MCA entrypoint with SAL, it also supplies the length of the code (or at least the length of the first level OS_MCA handler). The operating system may optionally supply a modulo checksum of the code area (all bytes of the code area including the checksum byte must add up to zero). The SAL saves the checksum for this code area. Prior to entering the OS_MCA, it is SAL_CHECK's responsibility to ensure that the OS_MCA vector is valid by verifying the checksum of the OS_MCA code. The SAL code that verifies the integrity of the OS_MCA code shall respect the cacheability attribute of the OS_MCA code. Thus, if the operating system had provided an uncacheable address for the OS_MCA entrypoint (bit 63 of physical address = 1), the SAL code shall not make cacheable accesses to the OS_MCA code areas while verifying the checksum.

There may be some platform-specific issues that render the OS_MCA handler invalid. For example since the OS_MCA handler is in memory, if the memory controller which handles that portion of memory is no longer functional, it does not make sense to attempt to branch to that code. If either the OS_MCA handler was not registered prior to the machine check event or if the OS_MCA handler is otherwise invalid, SAL_CHECK may halt or reboot the system. This action is SAL implementation-dependent. When the OS_MCA returns to the SAL indicating that the error has been corrected by the operating system layer, SAL will call the PAL_MC_RESUME procedure to resume execution. See Section 4.8.1 for other options.

Figure 4-2 depicts the control flow during corrected and uncorrected machine checks.

4.2 Corrected Machine Checks

There are different categories of Itanium architecture corrected machine checks:

- Corrected internally by the processor hardware, for example, single bit data ECC error on a processor cache.
- Corrected by PAL, for example, double bit data ECC error on a clean processor cache line (E, I, S), during an instruction fetch operation.
- Corrected by the platform hardware, for example, single bit data ECC error in system memory.
- Corrected by SAL. These are primarily platform errors that can be corrected by SAL without immediate involvement of the operating system.

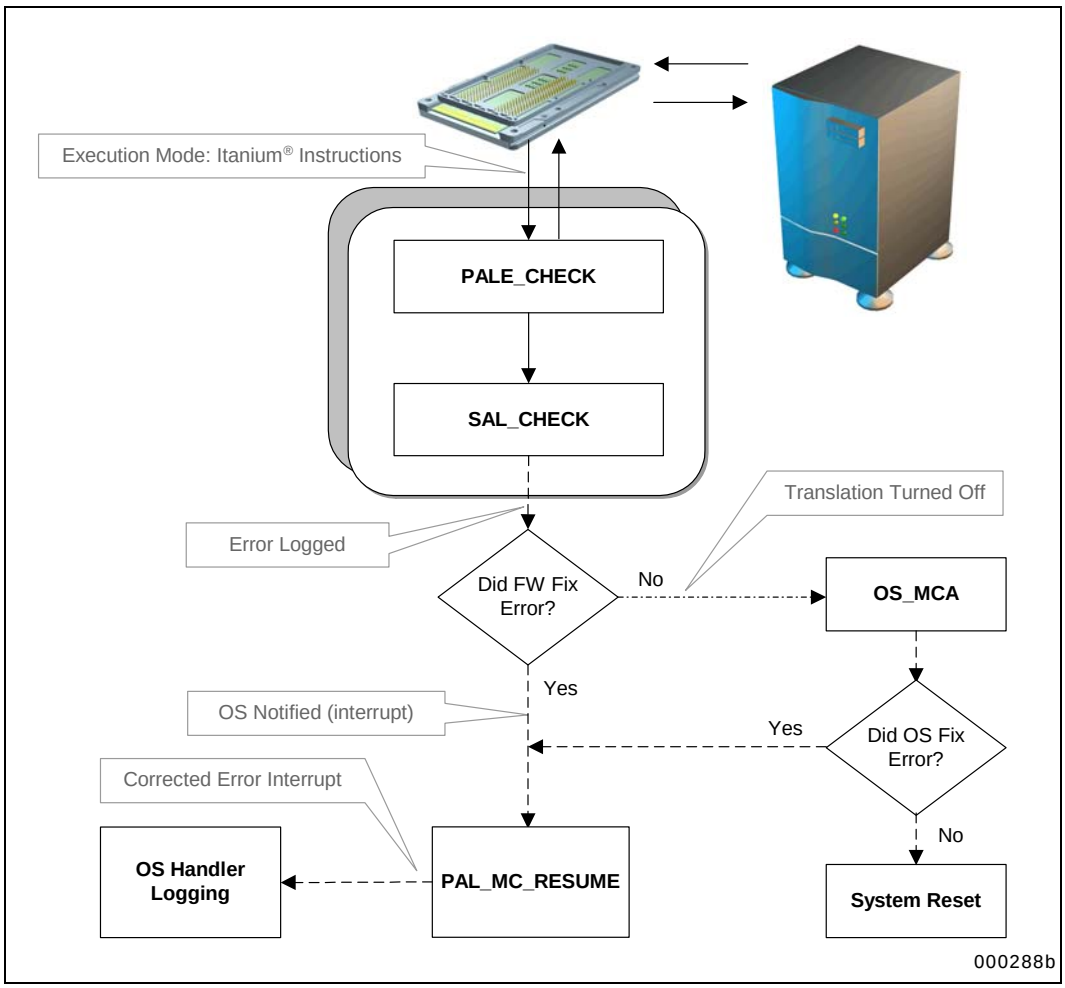
None of these categories will require a processor rendezvous.

The SAL_CHECK processing steps for corrected machine checks are similar to the steps for the uncorrected machine checks. SAL will maintain the processor and platform error information and save the state of the processor and platform. In the subsequent steps, SAL may do one of the following:

- If the error was corrected by PAL, SAL returns to the interrupted context by calling PAL_MC_RESUME. PAL_MC_RESUME procedure provides an option for generating a Corrected Machine Check (CMC) interrupt to the operating system for the processor CMC events. The CMCV register specifies the CMC interrupt vector and its mask status.

- SAL will perform any platform hardware-specific correction as described in Section 4.3, “Platform Errors,” send a Corrected Platform Error Interrupt (CPEI) to the operating system and then call PAL_MC_RESUME to return to the interrupted context.

Figure 4-2. Machine Check Code Flow



For corrected machine checks, SAL does not call the OS_MCA layer immediately but the operating system CMC interrupt handler or the operating system Corrected Platform Error interrupt handler will be invoked some time after returning to the interrupted process, assuming that the CMC or Corrected Platform Error interrupt is enabled in hardware. Some operating systems may choose to poll for corrected processor and/or platform errors instead of relying on the CMCI/CPEI interrupts. Refer to Section 4.4 for details.

The operating system-corrected error handler shall run with interrupts enabled[1] and would invoke the SAL_GET_STATE_INFO and the SAL_CLEAR_STATE_INFO procedures to process the error information associated with the event(s). The operating system must ensure that the entire CMC or Corrected Platform Error handler executes on the same processor on which it was signalled.

1. It is required that the operating system handlers operate with interrupts enabled, so that system firmware can manage its resources (like NVRAM based error records) without impacting the system performance.

The amount of state information saved by SAL is implementation-dependent and SAL provides validation bits indicating the saved state information. Thus, a particular SAL implementation may choose not to save ARs, CRs, or floating-point registers during a corrected machine check, for performance reasons.

4.3 Platform Errors

Platform errors refer to errors signalled by system components other than processors, for example, memory, I/O buses, chipsets, devices, and so on.

Uncorrected platform errors may be signalled by asserting pins such as BERR# or BINIT# or by generating a 2 x ECC or a synchronous Hard Fail response on the processor system bus.

Corrected platform errors are usually signalled using an interrupt line. An example of a corrected error is a single bit error corrected by the memory controller. An interrupt will be signalled by the platform when the data from the memory location is consumed.

Some platforms may use interrupts to signal a potential uncorrected error. An example of this situation is poisoned data stored into memory. A CPEI is signalled to the processor at the time of the store and if the poisoned data is consumed later by a processor, that processor will incur a local MCA.

4.3.1 Scope of Platform Errors

The scope of platform errors is platform and firmware implementation-dependent. Depending upon the platform topology, a single physical platform may have multiple nodes, each with a set of processors and its own error event generation and notification. There may be requirements for routing the interrupt signals to specific processors as processors may not have visibility to all the platform components in a system. The SAL shall provide details of the interrupt input line(s) and the interrupt routing requirements, including the ID and EID of the processor to receive the CPEI interrupt to the operating system through the ACPI tables. The scope of platform error logs is implicitly indicated by the SAL by providing multiple entries for Corrected Platform Error interrupts in the ACPI tables. Platforms that do not support interrupts on corrected errors may report the scope through an ACPI structure. Refer to the Advanced Configuration and Power Interface Specification for additional details.

4.3.2 Processing of Corrected Platform Errors

When the operating system wants to be notified of the platform error events through an interrupt, it will select a corrected platform error vector (CPEV) and arm the interrupt line(s) to deliver interrupt(s) to the processor. The operating system is also required to register the chosen interrupt vector number with SAL through the SAL_MC_SET_PARAMS procedure.

The system component responsible for the corrected error (hardware or firmware) sends event notification to the operating system. For hardware-corrected platform errors, the hardware device sends the Corrected Platform Error Event notification to the operating system by asserting an interrupt. For firmware-corrected errors, SAL reports the platform-corrected error event to the operating system by sending an interprocessor interrupt to the processor with the CPEV that is registered by the operating system through the SAL_MC_SET_PARAMS procedure.

On the processor on which the CPEI was signalled, the operating system shall invoke the SAL_GET_STATE_INFO and the SAL_CLEAR_STATE_INFO procedures with argument type of CPE to retrieve and process the corrected platform error information.

4.3.3 Processing of Uncorrected Platform Errors

Uncorrected platform errors may result in a local or a global MCA. The operating system shall invoke the SAL_GET_STATE_INFO and the SAL_CLEAR_STATE_INFO procedures with the argument type of MCA on all the processors on which the MCA condition is signalled to retrieve and process the uncorrected platform error information.

The SAL shall return an error record on each of the processors that experienced the MCA condition. Some error records may have a processor error section and one or more platform error sections, while some error records may have only the processor error section. The platform section(s) would provide the error information for the node associated with the processor on which the SAL call is made. If a SAL implementation is capable of accessing error information for the entire multi-node system from one processor, it is permitted to aggregate all the platform error sections within one error record.

4.4 Polling for Corrected Errors

Some operating systems may choose to poll for corrected processor and platform error events. For corrected processor events, the operating system must periodically invoke the SAL_GET_STATE_INFO and the SAL_CLEAR_STATE_INFO procedures on each logical processor in the system.

For corrected platform events, the operating system must periodically invoke the SAL_GET_STATE_INFO and the SAL_CLEAR_STATE_INFO procedures from a processor on each node within the system since some platform errors may only be visible on the node of occurrence.

If the operating system chooses to employ polling for the corrected platform error events, it must neither program the interrupt redirection table entry for the interrupt line on which the Corrected Platform Error is signalled nor register the CPEV vector with the SAL. Instead, it should periodically call the SAL_GET_STATE_INFO procedure on the same processor(s) for which it would have programmed the interrupt. All other processing steps are the same as for the interrupt driven approach.

The OS will use the SAL_MC_SET_PARAMS to register and de-register a CPE vector with SAL. This mechanism is used by the OS when it wants to be notified of a corrected error event either through an interrupt signaling or polling. When a non-zero value is registered by the OS, SAL is allowed to send an IPI to the OS with this vector value. When a NULL value is registered by the OS, SAL shall not send an IPI with this vector to the OS. The firmware sets the default value for this vector as NULL.

4.5 OS_MCA

When the operating system is ready to handle machine check events, it should call SAL_SET_VECTORS to register the physical address, length, and the GP of the OS_MCA handler. It is highly recommended that a non-zero length and checksum be supplied by the operating system to the SAL so that the SAL can ensure the integrity of the OS_MCA code. The operating system must use the SAL_SET_VECTORS function if it expects to be able to recover from any machine check conditions in which it may have

to be involved, or to retrieve error records and state information and dumping such information for subsequent debug analysis. After registering the OS_MCA address, the operating system can re-enable machine checks by clearing the PSR.mc bit. The operating system must call the SAL_GET_STATE_INFO_SIZE procedure to obtain the maximum size of machine check state information that SAL would return for the MCA events.

When the machine check event occurs, SAL_CHECK will invoke OS_MCA. OS_MCA functionality is implementation-dependent. At a minimum, OS_MCA must call SAL_GET_STATE_INFO to retrieve the error records and state information. When it has finished this task, it must call SAL_CLEAR_STATE_INFO[1] to release the SAL resources used for logging MCA events and state save. The OS_MCA can then re-enable machine checks by clearing the PSR.mc bit to 0. Once the operating system has consumed and cleared an error record, it will no longer be available from the SAL. SAL error records are always associated with a particular MCA or Corrected error event and shall contain all the relevant information packaged together as a record, and may contain error information from just the processor or platform or both. This information is presented in an error record structure with a Record Header and multiple sections. Each section has an associated globally unique ID (GUID) to identify the section type as being processor, memory, bus, controller or platform-specific hardware. Refer to Appendix B for details.

The operating system may perform any corrections on the operating system controlled hardware resources. The operating system makes the decision whether it wants to recover the interrupted context or not, but it must take into account the state information retrieved from the SAL_GET_STATE_INFO call. This information describes the continuability of the processor/system. Thus, even if the operating system could correct the error, if PAL reports that it did not capture the entire processor context, (for example, processor state parameter states that the GRs are invalid), resumption of the interrupted context will not be possible. The operating system must also determine from values in the Min-State Save area whether the machine check occurred while operating with PSR.ic set to 0 and whether the processor implements the XIP, XPSR and XFS registers necessary for the recovery.

When OS_MCA returns to SAL or PAL, it is permitted to set new values for the registers that are passed by PAL in the Min-State Save area. This is achieved by constructing a data structure with the Min-State Save area format and returning it to SAL. Refer to the *Intel® Itanium® Architecture Software Developer's Manual* for the layout of this structure.

OS_MCA may select one of the following actions:

- Correct the error and return to SAL_CHECK with the status of "corrected." The operating system may set a new context in the Min-State Save area, and SAL will then invoke PAL_MC_RESUME to return to the interrupted or the new context. If the interrupted context was in the firmware address range and the operating system decides to set a new context, the operating system must take steps for resumption of the firmware code eventually, otherwise the system may become unstable.
- In the event of an uncorrected error, return to SAL_CHECK with the uncorrected status value and an indication to the SAL to halt or reboot the system.

Figure 4-3 shows the flow of control through SAL_CHECK on the monarch processor.

1. The error records maintained by firmware are returned one at a time to the operating system. It is necessary for the operating system to clear the current error record to be able to retrieve the next unread record.

Any SAL runtime service that operates with interrupts disabled for an extended period of time may cause an OS or driver time-out and system shutdown to occur. It is strongly recommended that SAL handle machine check aborts promptly and hand off to the OS as soon as possible after storing error records in non-volatile memory and completing any PAL-specified actions, such as processor rendezvous.

4.5.1 Unconsumed Error Records across Reboots

There may be situations where the OS_MCA layer could not be invoked or the OS_MCA layer could not invoke the SAL_CLEAR_STATE_INFO procedure to clear a pending error record. If the SAL implementation had logged the error to NVRAM, it should provide the unconsumed error information to the operating system following the next reboot of the system. To support this capability, following the next boot of the operating system, the operating system is required to call SAL_GET_STATE_INFO and SAL_CLEAR_STATE_INFO procedures on the BSP and APs with type argument of MCA (and optionally with type arguments of CMC, CPE, INIT) to retrieve unconsumed error records and log them to operating system persistent storage.

For SAL revisions 3.3 or higher, the operating system may optionally call SAL_GET_STATE_INFO and SAL_CLEAR_STATE_INFO with type = "deconfigured processor" on the BSP and APs to collect unconsumed deconfigured processor error records.

For additional details, refer to the *Intel® Itanium® Processor Family Error Handling Guide.*

If the operating system fails to clear the log before another MCA surfaces, the SAL may overwrite the unconsumed NVRAM log, if there is not space for another record. The SAL implementation may additionally escalate the error severity (Section B.2.1, “Record Header”) when the error information is subsequently provided to the operating system.

4.6 Procedures Used in Machine Check Handling

PAL_CHECK and SAL_CHECK execute out of the firmware address space. SAL_CHECK may, however, invoke the PAL procedures in memory after ensuring that the memory area containing the PAL procedures is intact.

Following are typical PAL procedures that may be invoked by SAL_CHECK:

- PAL_MC_ERROR_INFO
- PAL_MC_RESUME
- PAL_MC_CLEAR_LOG
- PAL_MC_DYNAMIC_STATE

The following procedures may be called by SAL_RESET to control handling of machine checks:

- PAL_BUS_GET_FEATURES
- PAL_BUS_SET_FEATURES
- PAL_PROC_GET_FEATURES
- PAL_PROC_SET_FEATURES
- PAL_MC_REGISTER_MEM[1]
- PAL_MC_EXPECTED

SAL may call the following procedure to ensure that all outstanding instructions within a processor are completed and any potential machine checks due to these transactions get serviced.

- PAL_MC_DRAIN

Following are the SAL procedures that may be invoked by operating system to register its machine check layer interfaces:

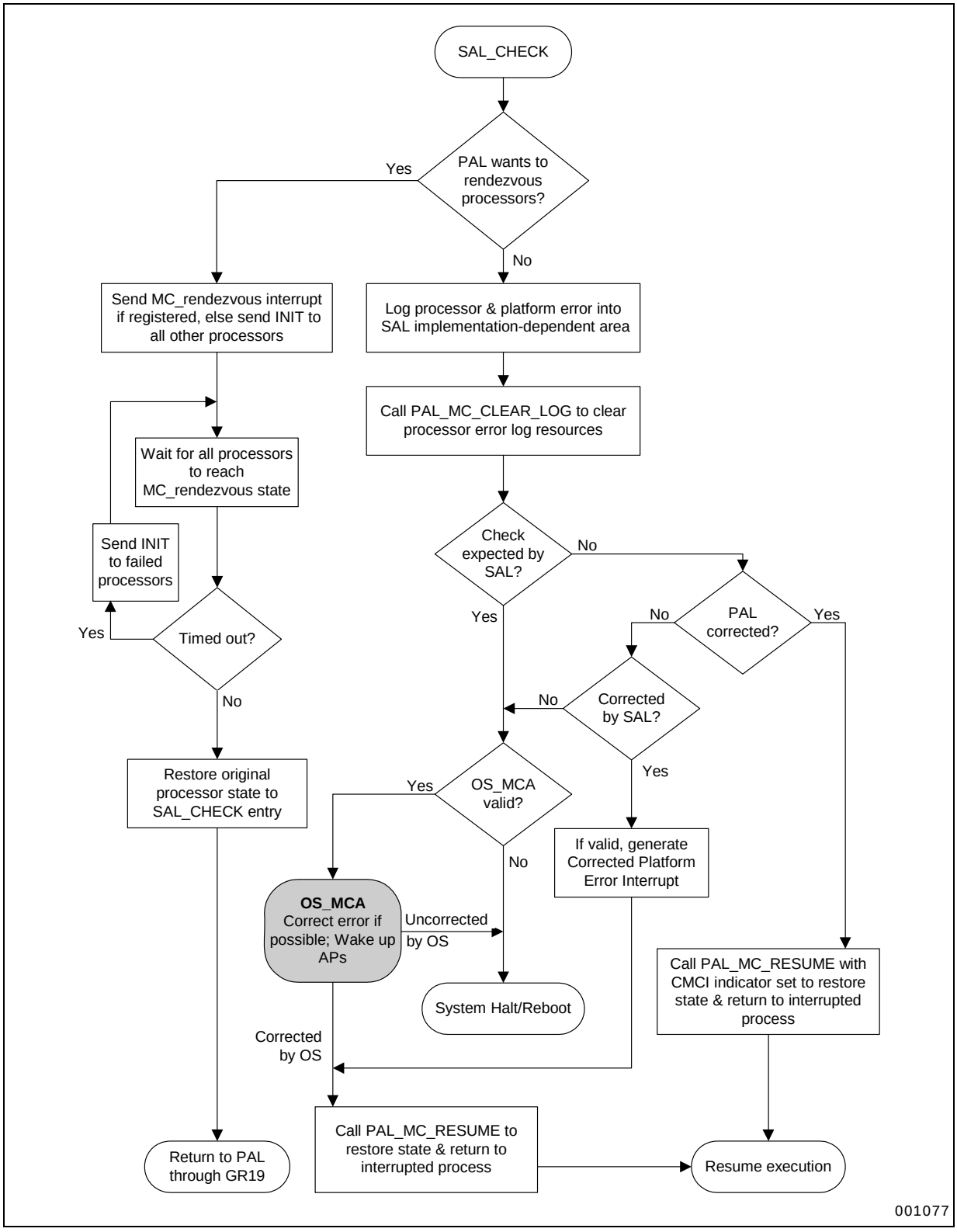
- SAL_MC_SET_PARAMS
- SAL_SET_VECTORS

Following are the typical SAL procedures that may be invoked by the operating system during machine check processing:

- SAL_MC_RENDEZ
- SAL_GET_STATE_INFO
- SAL_GET_STATE_INFO_SIZE
- SAL_CLEAR_STATE_INFO

1. This procedure is intended for use during firmware initialization. It shall not be invoked by the operating system during runtime as this might affect firmware functionality.

Figure 4-3. SAL_CHECK Detailed Flow on the Monarch Processor



4.7 Machine Checks in MP Configurations

There are certain machine check scenarios that require additional actions and considerations in MP configurations. A local MCA on one or more processors may require the system to be in a quiescent state for error handling. This is accomplished by bringing all the processors in the system that are not already in MCA to an idle state. The MCA architecture has defined a mechanism for processor rendezvous through firmware and operating system coordination.

4.7.1 Rendezvous Requirements

In MP configurations, a coordination between processors is performed through processor rendezvous. Refer to Section 3.2.2.1, "Rendezvous Functionality" for details of how the rendezvous mechanism works.

Rendezvous of processors is done for one of the following reasons:

- When PAL initiates a rendezvous request during an MCA.
- When SAL determines that the platform error needs rendezvous.
- When the operating system sets a flag requesting firmware to perform rendezvous for all MCA errors.

PAL-Initiated Rendezvous: If the PAL machine check layer determines that other processors must be rendezvoused for error containment, it passes an indication to SAL_CHECK to perform the rendezvous and supplies a return address within PAL in GR19. Upon return, PALE_CHECK performs the appropriate action and then calls SAL_CHECK again in the normal manner (with no rendezvous indicator). The SAL must determine the state of other processors and bring all processors not already in MCA to a spinloop by generating SAPIC interrupt messages. The interrupt vector used by SAL to request for rendezvous is the one already registered by the operating system during the OS_MCA handler initialization

SAL-Initiated Rendezvous: Additionally, there may be platform related machine check situations which require SAL firmware to rendezvous processors. For example, if platform hardware were to stop forwarding transactions in order to maintain error containment, the other processors in the system must be rendezvoused before that platform hardware can be corrected to resume forwarding transactions.

Operating System-Initiated Rendezvous: If the operating system sets the *rz_always* flag during invocation of the SAL_MC_SET_PARAMS procedure, the SAL is required to rendezvous all the processors in the system for all detected processor and platform MCA conditions, when such errors are not corrected by the firmware. If this flag is not set, then rendezvous is done only during the PAL or SAL initiated rendezvous conditions described above.

4.7.2 Flow of Control during MCA in MP Configurations

The high level flow of control during MCAs in MP configurations is depicted in Figure 4-4 and Figure 4-5. The flow for a normal MCA rendezvous is as outlined below:

1. Processor detects an MCA event. PAL takes control and attempts an error recovery.
2. PAL may ask SAL to rendezvous for certain errors. SAL may decide to do a rendezvous on its own accord or if the operating system has registered a configuration option to rendezvous for all MCA errors, if it is not already done at PAL's request. If rendezvous does not occur, then steps 3, 4, 5, and 6 are skipped.

3. SAL sends SAPIC interrupt messages to all the slave processors except those in SAL boot rendezvous.
4. Interrupted slave processors enter a spin loop by calling SAL_MC_RENDEZ.
5. SAL selects a monarch for handling the error. All slaves processors in SAL_MC_RENDEZ check in their status with the SAL on the monarch.
6. After all the slaves check in with SAL, the monarch SAL returns to PAL.
7. PAL starts the actual error handling process with subsequent hand-off to SAL.
8. SAL finishes the MCA handling on all the processors that are in MCA and waits for all the processors in MCA to synchronize before branching to OS MCA for further processing. Note that the hand-off to OS MCA from SAL MCA occurs simultaneously on all processors executing in SAL MCA handler.
9. OS_MCA may choose a monarch processor to continue with error handling. After OS_MCA completes the error handling, the monarch processor wakes up all the slaves through a wake-up message as shown by (9) in Figure 4-4.

Figure 4-4. Normal SAL Rendezvous Flow

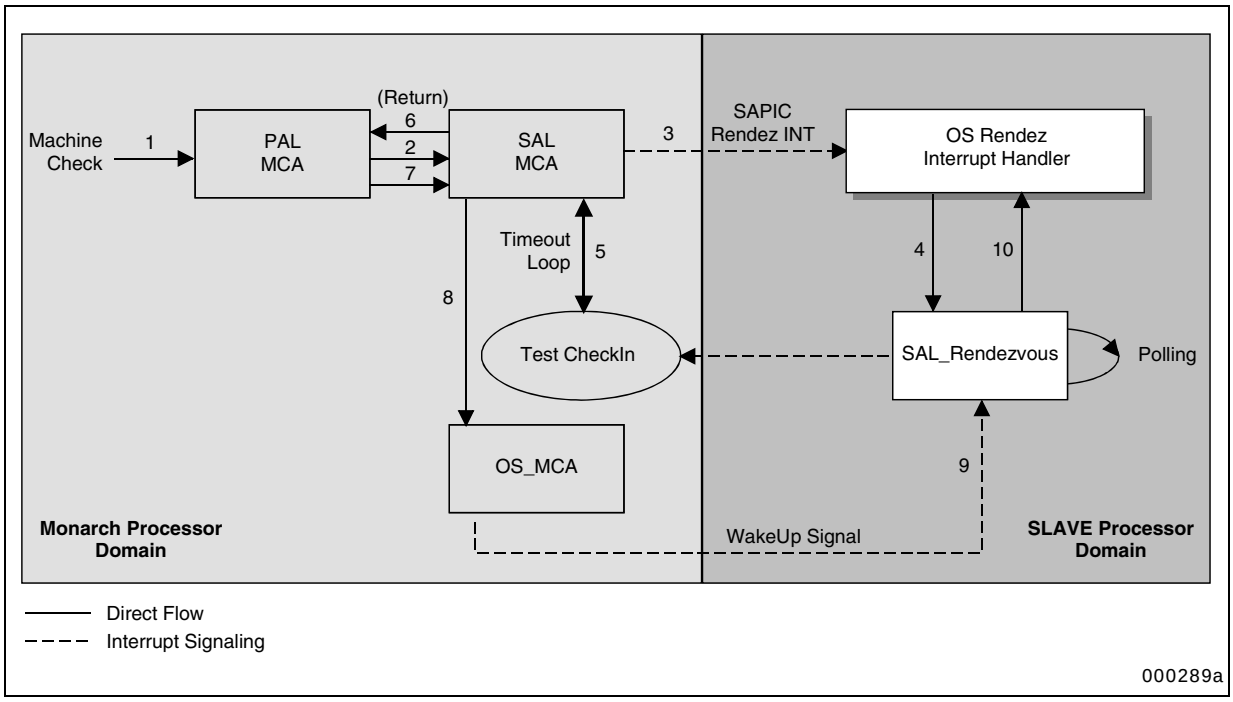
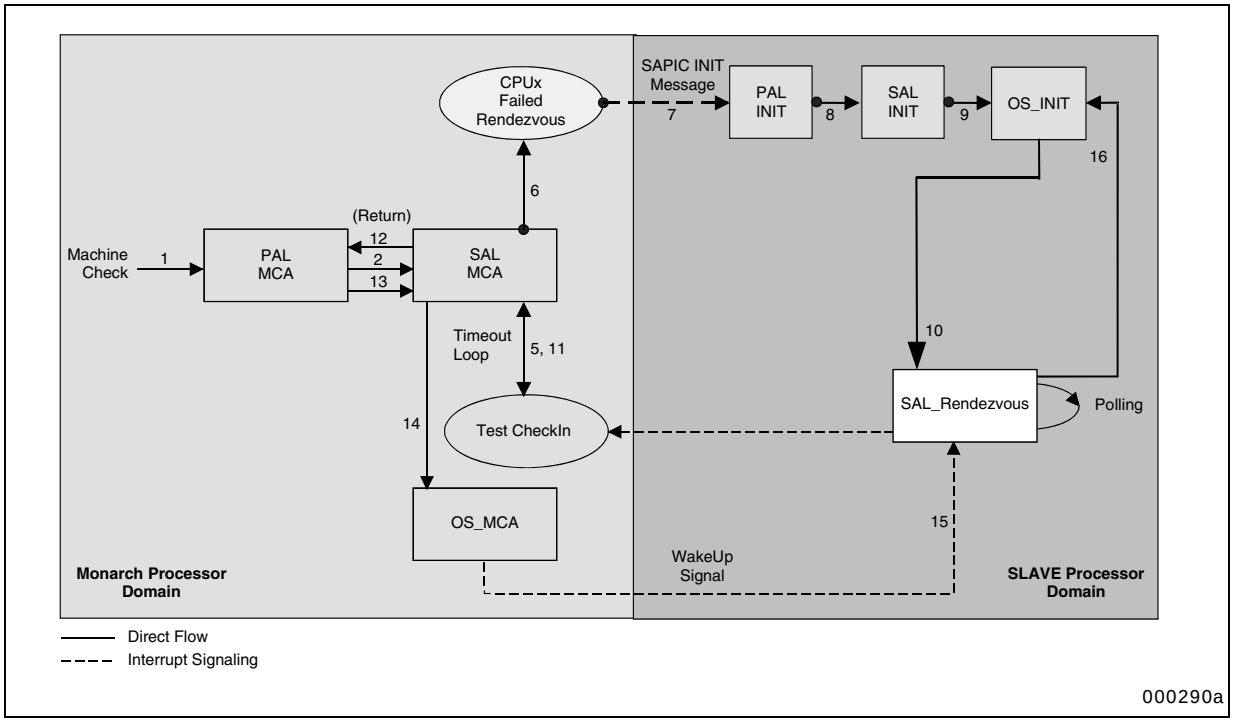


Figure 4-5. Failed SAL Rendezvous Flow



During the initial attempt to rendezvous, some processors may fail to respond to the interrupt for an extended period of time. The monarch processor SAL forces the failed processors to respond by sending an SAPIC INIT message as shown in Figure 4-5. Once all the processors are in the spin loop, then the monarch processor that received the MCA will attempt to recover from the error. The flow of bringing the processors to a rendezvous state is the same as in Figure 4-4, except for the additional steps 6., 7., 8., and 9.

4.7.3 OS_MCA Responsibilities

In order to support the MCA events in MP configurations, the operating system does the following:

- Register the address of OS_MCA entrypoint and its GP value using the SAL_SET_VECTORS function.
- Invoke the SAL_MC_SET_PARAMS procedure specifying an interrupt vector on which SAL firmware can signal the non-monarch processors and the mechanism that the operating system will employ to wake up the non-monarch processors at the end of machine check processing.
- Invoke the SAL_MC_SET_PARAMS procedure specifying if a rendezvous is always required for an MCA and whether MCAs should be escalated to BINIT# while machine checks are masked.

On receipt of the MC_rendezvous interrupt or the INIT for MC_rendezvous, the operating system on the non-monarch processors will:

- Disable further interrupts.
- Set an OS implementation specific variable to indicate that a rendezvous interrupt was received. Such a variable may be used by the OS_MCA layer on the monarch

processor to identify the processors that need to be woken up at the end of MCA processing.

- Call SAL_MC_RENDEZ. This procedure will call PAL_MC_DRAIN to complete all outstanding transactions within the processor and then enter a spin loop within SAL. This SAL procedure shall be MP-safe. If the processor in rendezvous takes a machine check while waiting for wake-up, the SAL should delay the handling of this subsequent machine check event until completion of the current machine check (that is, return from the OS_MCA layer). SAL implementations that do not provide this capability may mask further machine checks and escalate future MCA events to BINIT# using the PAL_PROC_SET_FEATURES procedure.

SAL on the monarch processor will wait a specified amount of time for the signalled processors to enter the SAL_MC_RENDEZ procedure. The wait time is specified as a parameter to the SAL_MC_SET_PARAMS procedure. Assuming all processors report in as expected, the PAL and SAL will perform the appropriate state save functions and proceed to the OS_MCA entrypoint to allow the operating system to take the appropriate error recovery actions. Refer to Figure 4-4 for more details on the control flow between the PAL, the SAL, and the operating system.

In situations where either the operating system has not registered an interrupt vector via the SAL_MC_SET_PARAMS call or where the specified time to wait has elapsed and the signalled processor did not respond, the SAL firmware on the monarch processor will send an INIT to the remaining processors in order that the machine check handlers in PAL and SAL can proceed. This scenario is depicted in Figure 4-5. While sending an INIT to the other processors may not create an inherently unrecoverable situation, it increases the risk for successful recovery. This is the rationale for registering the MC_rendezvous interrupt vector using the SAL_MC_SET_PARAMS procedure. The monarch processor must allow sufficient time for the INIT IPI processing and rendezvous on the targeted processors.

Note: The PAL_INIT and the SAL_INIT firmware code executes out of the firmware address space and contends for firmware accesses with the processors that experienced the machine check events.

If the PAL requests rendezvous of all the processors and SAL is unable to do so, SAL will return to PAL with a non-zero value in GR19. Refer to the *Intel® Itanium® Architecture Software Developer's Manual* for details regarding PALE_CHECK processing.

After the error is corrected by the OS MCA handler, OS_MCA on the monarch processor will wake up the rendezvoused processors using the wake up mechanism specified in the SAL_MC_SET_PARAMS call. For the processors rendezvoused using the MC_rendezvous interrupt or the INIT, the continuation point is merely a return from the SAL_MC_RENDEZ procedure. It is the responsibility of the operating system to clear the IRR bits for the MC_rendezvous interrupt and the wake up interrupt, if any. The operating system must re-enable future interrupts and machine checks.

It should be noted that under certain machine check circumstances some platform implementations will cause multiple processors to enter PALE_CHECK and SAL_CHECK. PAL code will be generally unaware of this, but SAL code should make every effort to take such situations into account. SAL code must implement methods of detecting which processors have entered the SAL_CHECK entrypoint and avoid steps to rendezvous such processors (using MC_rendezvous interrupt or INIT). Some examples of situations when multiple processors experiencing machine checks simultaneously are as follows:

- Broadcast machine check (BERR signal) from the platform.
- Error during a cast out of a cache line in response to an incoming snoop cycle from another processor.

When multiple processors experience machine checks simultaneously, SAL selects a monarch machine check processor to accumulate all the error records at the platform level. Once this is done, the OS_MCA procedure will take control of further error handling on all the processors that experienced the machine checks. The OS_MCA layer may need to implement a similar monarch processor selection for the error recovery phase. The operating system will be aware of which processors invoked the SAL_MC_RENDEZ procedure in response to the MC_rendezvous interrupt or the INIT signal and shall wake up those processors.

4.7.4 Machine Check Processing Steps within Firmware and Operating System

Figure 4-6 depicts the typical flow of machine check processing steps from various firmware and software layers in an MP configuration. This figure illustrates the example of two processors (1 and 2) experiencing a machine check within a four processor system. The error requires the other processors to be rendezvoused.

On entry into SAL_CHECK, processor 1 promotes further MCAs to BINIT# for better error containment. This is based on an argument supplied by the operating system as part of the SAL_MC_SET_PARAMS procedure. The SAL on processor 1 is not aware of any other processors having experienced machine check and hence sends the MC_rendezvous interrupt to all the other processors including processor 2. It also sets a memory semaphore (MCA_In_Prog) to indicate that a machine check is in progress. By setting such a semaphore, processor 1 gains the monarch status for this machine check incidence at the SAL layer. Semaphore operations such as XCHG, CMPXCHG can only be made to cacheable locations. If the platform provides an equivalent mechanism such as a read/write-once port, the same may be employed in lieu of a cacheable memory semaphore.

The operating system on the processor 3 receives the MC_rendezvous interrupt, sets a flag for being rendezvoused in the operating system data structures and then calls the SAL_MC_RENDEZ procedure. The processor 4 is running with interrupts masked and does not recognize the MC_rendezvous interrupt in a timely manner. Hence, the processor 1 sends an INIT IPI to the processor 4. This causes the processor 4 to enter the OS_INIT layer, which records the fact of being rendezvoused in the operating system data structures and then calls the SAL_MC_RENDEZ procedure.

The SAL on processor 1, using SAL data structures, recognizes that processor 2 has reached the SAL_CHECK layer and that processors 3 and 4 have reached the SAL_MC_RENDEZ procedure. It clears the MCA_In_Prog semaphore, instructs the processor 2 to proceed to the OS_MCA layer, and then proceeds to the OS_MCA layer itself.

At the OS_MCA layer, the operating system, using its data structures, determines that only processors 1 and 2 will reach the OS_MCA layer. The operating system elects a monarch to handle the machine check (processor 2 in Figure 4-6). The operating system makes necessary SAL calls to retrieve and clear the processor and platform error information. The operating system on processor 2 then instructs processors 1, 3 and 4 to return to the interrupted contexts. The processor 1 returns via SAL and the PAL_MC_RESUME procedure while processors 3 and 4 return to the procedure that invoked the SAL_MC_RENDEZ procedure.

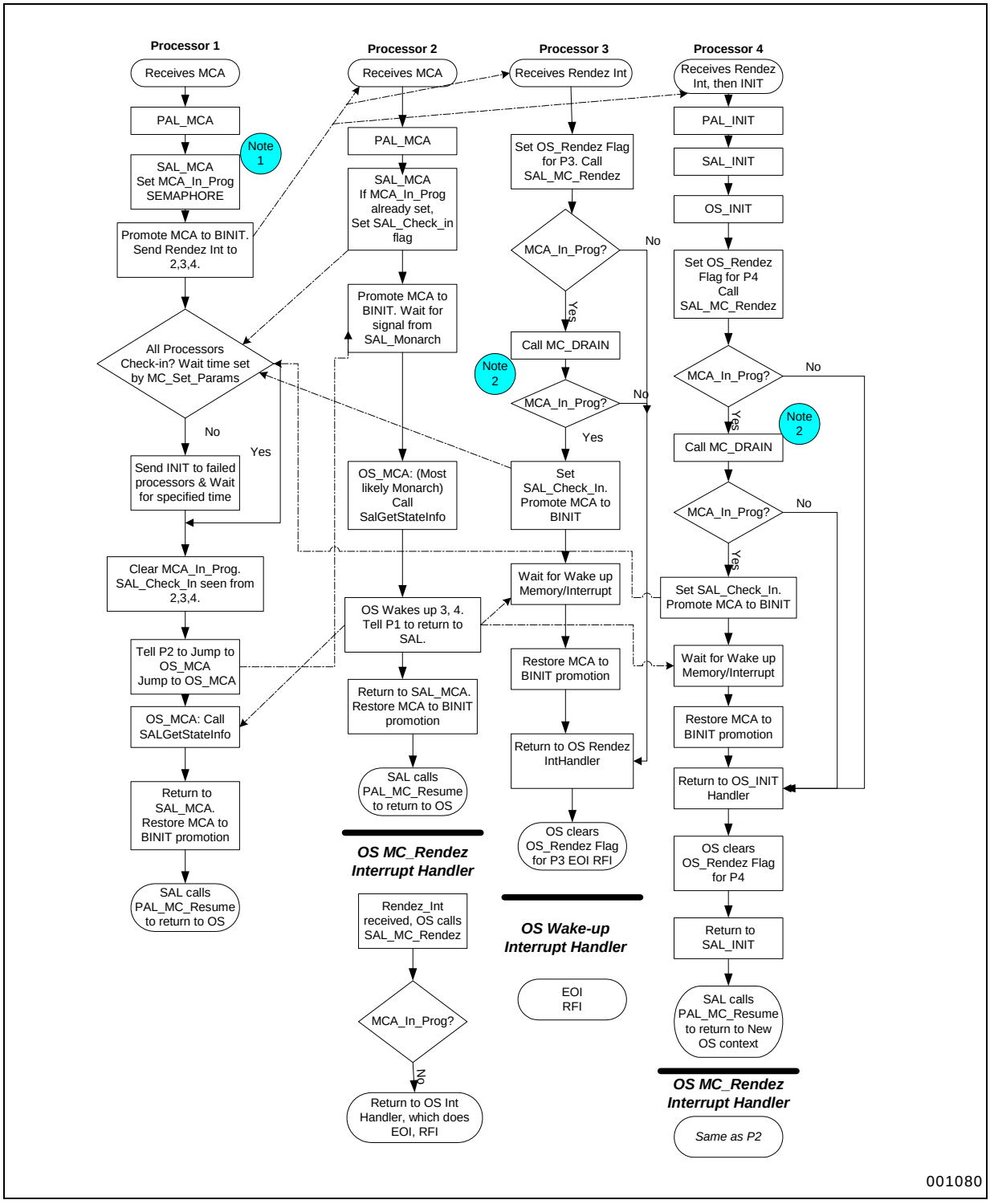
Once interrupts are re-enabled, the operating system on the processor 2 services a spurious MC_Rendezvous interrupt and invokes the SAL_MC_RENDEZ procedure. The SAL finds that no machine check is in progress and hence returns to the operating system immediately. If the operating system chosen wake-up mechanism is an

interrupt, the operating system on the processors 3 and 4 will service the wake-up interrupt. As part of servicing these interrupts, the operating system reads the CR.IVR register and issues an EOI to the local SAPIC thereby clearing the interrupt.

Notes for Figure 4-6:

1. This platform could provide a mechanism such as an I/O port to activate the need for a memory semaphore. Memory semaphore operations can only be made to cacheable space.
2. If MCA occurs here, control flow is the same as for P2. At the end of the MCA, follow path for NO since MCA is no longer present. OS will not send wake-up for P3 although OS Rendez flag is set.

Figure 4-6. Machine Check Handling in a Typical MP Configuration



4.8 OS_MCA Hand-off State

The OS_MCA interface defines the boundary between SAL_CHECK and the operating system machine check handler (OS MCA handler). The contents of non-banked and banked general registers at the time of the interruption are saved by PAL in the Min-State Save area and are available for use by SAL and the OS MCA handler. The following register contents define the OS MCA hand-off state.

The state of the processor is the same as on exiting PALE_CHECK (refer to the *Intel® Itanium® Architecture Software Developer's Manual*) except as below:

GR1 = OS_MCA Global Pointer (GP) registered by the operating system (the operating system's GP).

GRs2-7= Unspecified.

GR8 = Physical address of the PAL_PROC entrypoint.

GR9 = Physical address of the SAL_PROC entrypoint.

GR10 = GP (Physical address value) for SAL.

GR11 = Rendezvous state information:

- 0 = Rendezvous of other processors was not required by firmware and hence not done.
- 1 = All other processors in the system were successfully rendezvoused using MC_rendezvous interrupt.
- 2 = All other processors in the system were successfully rendezvoused using a combination of MC_rendezvous interrupt and INIT.
- –1 = Rendezvous of other processors was required but was unsuccessful on one or more processors.

GR12 = Return address to a location within the SAL_CHECK procedure.

GRs13-31 = Refer to the *Intel® Itanium® Architecture Software Developer's Manual.*

BR0 = Unspecified.

Note: On entry into SAL_CHECK, the RSE has been set to enforced lazy mode configuration. The operating system shall not make cacheable accesses to the Min-State Save area, otherwise unexpected behavior will occur.

For all SAL to OS MCA handoffs, the OS is expected to be able to execute the OS MCA handler from memory at minimum. If a platform cannot guarantee the integrity of the system memory, the platform firmware shall not hand off to the OS MCA handler, but shall cause an immediate system reset. Error information shall then be reported to the OS during the next system boot and initialization. Platforms that expect the OS MCA handler to have I/O support (display, disk logging, and so on) must additionally guarantee the availability of critical I/O devices before firmware hands off to the OS MCA handler.

4.8.1 Return from the OS_MCA Procedure

The OS_MCA procedure shall return to the SAL_CHECK at the end of its MCA processing using a br instruction and the SAL_CHECK return address passed in GR12.. When the OS_MCA procedure returns to the SAL, it must set appropriate values in the Min-State Save area pointed to by GR22, for continuing execution at the interrupted or a new context. The operating system must restore the processor state to the same as on entry to OS_MCA except as follows:

GRs1-7 = Unspecified.

GR8 = 0 if error has been corrected by OS_MCA:
–1 if error was not corrected by OS_MCA and SAL must warm boot the system.
–2 if error was not corrected by OS_MCA and SAL must cold boot the system.
–3 if error was not corrected by OS_MCA and SAL must halt the system.

GR9 = GP (Physical address value) for SAL.

GR10 =	0 if return will be to the same context. 1 if return will be to a new context.
GRs11-21 =	Unspecified.
GR22=	Pointer to a structure containing new values of registers in the Min-State Save area; PAL_MC_RESUME procedure will restore the register values from this structure; OS_MCA must supply this parameter even if it does not change the register values in the Min-State Save area.
GRs23-31 =	Unspecified.
PSR =	Same as on entry from SAL_CHECK except that PSR.mc may be either 0 or 1.
BR0 =	Unspecified.

§

5 Initialization Event

INIT is an event generated by the platform or by software (through a SAPIC message). The INIT event, which is typically used for crash dump reporting, causes the processor to branch to the processor-dependent INIT handler (PALE_INIT). PALE_INIT saves minimum register state and branches to SALE_ENTRY which, in turn, passes control to the SAL INIT handler (SAL_INIT). The state of the processor on exiting PALE_INIT and entering SALE_ENTRY is defined in the *Intel® Itanium® Architecture Software Developer's Manual.*[1]

5.1 SAL_INIT

SAL_INIT is entered from PALE_INIT via SALE_ENTRY. SAL_INIT's purpose is to save the state of the processor to the platform-specific Processor State Information (PSI) area and either invoke an operating system INIT handler (OS_INIT) if the same has been registered through a SAL_SET_VECTORS call, or warm boot the system otherwise. The SAL_SET_VECTORS procedure permits the operating system to register separate entrypoints for the first processor (monarch) to enter the SAL_INIT layer and subsequent processors (non-monarchs).

SAL_INIT should ensure that processors in SAL boot rendezvous do not hand off to the OS on INIT events. Processors in SAL boot rendezvous should remain there until awakened by the BSP. See Section 3.2.5, "OS_BOOT_RENDEZ" for more details.

The warm boot mechanism is SAL implementation-dependent and can be done either by calling PAL_MC_RESUME with a new context to branch to SAL_RESET with a non-zero value in GR32 or by generating a reset event that will cause a system-wide warm boot. Note that during the transition from PALE_RESET to SAL_RESET via SALE_ENTRY, the value in GR32 will be zero.

The following defines the behavior of SAL_INIT:

- During boot, SAL_RESET code will call PAL_MC_REGISTER_MEM to tell PAL code where it may deposit some minimal processor state information so that PAL code has sufficient resources to perform the necessary machine check or INIT processing. This step is performed on all the processors in the system.

  SAL_INIT saves the minimal processor state information as well as some additional processor and platform state information in the SAL data area and provides the same to OS_INIT. PAL_INIT and SAL_INIT shall not hide any architectural state from the OS_INIT layer.
- Check if the OS_INIT handlers for the monarch and non-monarch processors are registered and that both of them are valid. When the OS_INIT procedures were registered with the SAL, the operating system may optionally supply the modulo checksum for the code areas (all bytes of the code area including the checksum byte must add up to zero). The SAL saves the checksums for the code areas. On receipt of the INIT condition, the SAL verifies the checksum of the code at the OS_INIT procedure addresses before jumping to it.
- If the code for the OS_INIT handlers are intact, call the OS_INIT handlers for the monarch and non-monarch processors.

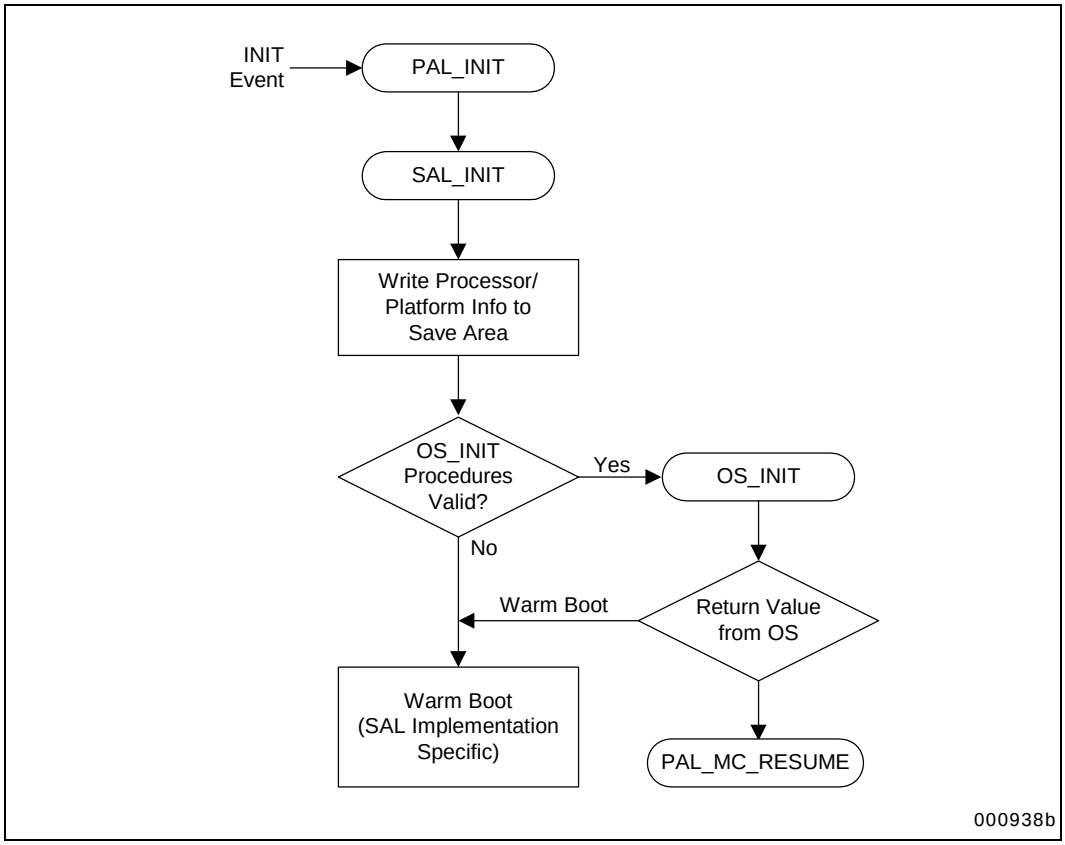
1. CR.itm and AR.itc must remain *unchanged* (see Table 8-1, "Definition of Terms").

- If the OS_INIT handler is not registered, set implementation-dependent SAL warm boot indicator and reboot the system either by calling SAL_RESET or by generating a reset event.

INITs are masked on entry to SAL_INIT and should remain masked (PSR.mc = 1) until the INIT processor state is logged at least. There is neither a requirement nor a way to clear a pending INIT condition.

Figure 5-1 shows a possible flow of control through SAL_INIT.

Figure 5-1. SAL_INIT Control Flow



5.2 OS_INIT

OS_INIT is an entrypoint into the operating system to deal with the initialization event. The exact definition of OS_INIT functionality is OS-dependent. SAL_SET_VECTORS is called by the operating system prior to the initialization event to register the physical addresses and the GP of the OS_INIT handlers for the monarch and non-monarch processors. If an operating system intends to make the monarch selection in the operating system layer, it could register the same OS_INIT entrypoint for both the monarch and non-monarch processors. From the SAL's perspective, there are no functionality differences between the two OS_INIT entrypoints and the hand-off state from the SAL to the OS_INIT layer are similar.

Following are the typical SAL procedures that may be invoked by the OS_INIT handler:

- SAL_MC_RENDEZ
- SAL_GET_STATE_INFO

- SAL_GET_STATE_INFO_SIZE
- SAL_CLEAR_STATE_INFO

When the OS_INIT layer is called by SAL_INIT, OS_INIT should call SAL_GET_STATE_INFO to get processor/platform state. When it has finished this task, it must call SAL_CLEAR_STATE_INFO to release these resources for future logging and state save. The OS_INIT can then re-enable further INITs and machine checks by clearing the PSR.mc bit to 0.

The OS_INIT handler shall return to the SAL with an indication to effect a warm reset or a return to the interrupted context. The OS_INIT may set new values for registers that are saved by PAL in the Min-State Save area. This is achieved by constructing a data structure with the format identical to the Min-State Save area and passing the same as an argument to the PAL_MC_RESUME procedure. Refer to the *Intel® Itanium® Architecture Software Developer's Manual* for the layout of this structure.

5.3 OS_INIT Hand-off State

The OS_INIT interface defines the boundary between SAL_INIT and the operating system code, OS_INIT. The contents of non-banked and bank zero general registers at the time of the interruption have been saved by PAL in the Min-State Save area and these are available for use by SAL and OS_INIT. The following register contents define the OS_INIT hand-off state.

The state of the processor is the same as on exiting PALE_INIT (refer to the *Intel® Itanium® Architecture Software Developer's Manual*) except as below:

GR1 =	Physical address of the OS_INIT Global Pointer (GP) registered by the operating system (the operating system's GP).
GRs2-7 =	Unspecified.
GR8 =	Physical address of the PAL_PROC entrypoint.
GR9 =	Physical address of the SAL_PROC entrypoint.
GR10=	GP value (Physical address) for SAL.
GR11 =	INIT reason code: 0 = Received INIT signal on this processor for reasons other than machine check rendezvous and CrashDump switch assertion. 1 = Received INIT signal on this processor during machine check rendezvous. 2 = Received INIT signal on this processor due to CrashDump switch assertion.
GR12 =	Return address to a location within the SAL_INIT procedure.
GRs13-31 =	Refer to the *Intel® Itanium® Architecture Software Developer's Manual.*
BR0 =	Unspecified.

Note: On entry into SAL_INIT, the RSE has been set to enforced lazy mode configuration. The operating system must not make cacheable accesses to the Min-State Save area, otherwise unexpected behavior will occur.

System state resources are:

- TLB –TCs and TRs are unchanged.
- Caches – Enabled, coherent and consistent in the absence of hardware failures.
- Memory – Unchanged, except for the updated Processor State Information (PSI) area.

Note: The RSE backing store must be restored to OS_INIT, such that the OS has the context to unwind the stack, if desired. This implies that ar.bspstore, and the RSE dirty register partition that existed at the time of the INIT must be restored on entry to OS_INIT.

5.4 Return from OS_INIT Procedure

The OS_INIT procedure shall return to the SAL_INIT using a br instruction and the SAL_INIT return address passed in GR12. When the OS_INIT procedure returns to the SAL, it must set appropriate values in the Min-State Save area pointed to by GR22, for continuing execution at the interrupted or a new context. The operating system must restore the processor state to the same as on entry to OS_INIT except as follows:

GRs1-7 =	Unspecified.
GR8 =	0 if SAL must return to interrupted context using PAL_MC_RESUME. –1 if SAL must warm boot the system.
GR9 =	GP (Physical address value) for SAL.
GR10 =	0 if return will be to the same context. 1 if return will be to a new context.
GRs11-21 =	Unspecified.
GR22=	Pointer to a structure containing new values of registers in the Min-State Save area; PAL_MC_RESUME procedure will restore the register values from this structure; OS_INIT must supply this parameter even if it does not change the register values in the Min-State Save area.
GRs23-31=	Unspecified.
PSR =	Same as on entry from SAL_INIT except that PSR.mc may be either 0 or 1.
BR0 =	Unspecified.

If OS_INIT requests SAL to reboot the system, it is SAL's responsibility to rendezvous all the processors in the system and then select a BSP for further system initialization.

5.5 MP INIT Support

In a MP configuration, the following should be observed:

- For monarch and non-monarch processors entering SAL_INIT, if there are no registered OS_INIT handlers or the OS_INIT checksum is incorrect, the system should reset the system and perform a warm boot. The first processor to observe this condition shall reset the system.
- Processors in SAL boot rendezvous should not enter SAL_INIT. They should remain in SAL boot rendezvous until awakened by the BSP or the system resets.

§

6 Platform Management Interruptions

Platform Management Interruptions (PMIs) provide an operating system-independent interrupt mechanism to support OEM and vendor-specific hardware events.

6.1 SALE_PMI Overview

PMI interrupts cause execution of code at PALE_PMI handler. This code saves key processor state in interruption resources and then calls the SALE_PMI handler. SALE_PMI shall return to the PALE_PMI layer which, in turn, will return to the interrupted context.

PALE_PMI calls SALE_PMI when the PMI pin is asserted, or on receipt of a SAPIC message with delivery type of PMI and interrupt vector value in the range reserved for SAL. Certain processor-specific events may also cause PMI interrupts. These are handled entirely within the PALE_PMI environment and the SAL layer is not notified. Refer to the *Intel® Itanium® Architecture Software Developer's Manual*[1] for details regarding PALE_PMI processing.

PMI is the highest priority external interrupt and it ranks after Reset, Machine Check and INIT in terms of priority. PMI is masked by setting the PSR.ic bit to 0 (interrupt collection disabled). The PSR.i bit (interrupt enable) has no effect on masking of PMI events.

The operating system can mask PMIs by setting PSR.ic bit to 0 (interrupt collection disabled). Also, PMI interrupt processing causes execution of PALE_PMI code before entering the SALE_PMI code. To minimize latency in entering code in the SALE_PMI layer, the operating system must avoid operating with PSR.ic bit set to 0 for long durations. Otherwise, some software in the SALE_PMI layer may fail. Note that some real time applications may have more stringent timing restrictions with regards to operating with interrupt collection disabled.

Operation with PSR.ic bit set to 0 compromises recovery from machine check and INIT events. It also causes special problems if multiple SAPIC messages of PMI delivery type are targeted to the same destination processor (see Section 6.4).

One method of software entry into the PMI environment is to send a SAPIC message to the same processor. Such a SAPIC message must use the interrupt vector value in the range reserved for SAL.

6.2 SALE_PMI Initialization

During power up, SAL copies the SALE_PMI handler to memory and then invokes the PAL procedure PAL_PMI_ENTRYPOINT to set the programmable entrypoint of the SALE_PMI procedure. In an MP environment, this step must be performed on all the processors. The SALE_PMI entrypoint can be different for various processors in an MP configuration.

1. CR.itm and AR.itc must remain *unchanged* (see Table 8-1, "Definition of Terms").

6.3 SALE_PMI Processing

On entry to SALE_PMI, one of the general registers contains the type of PMI interrupt and the interrupt vector value. The processor state at entry to SALE_PMI and the exit conditions from SALE_PMI to PALE_PMI are fully documented in the *Intel® Itanium® Architecture Software Developer's Manual*.

SALE_PMI is entered in physical mode with PSR.i and PSR.ic bits set to 0 (interrupt and interrupt collection bits disabled). SALE_PMI executes in the Itanium system environment regardless of the current processor state. The processing steps for various PMI events within the SAL layer are platform and SAL implementation-dependent. At the end of processing the PMI, SALE_PMI returns to PALE_PMI using branch register BR0. There is neither a requirement nor a way to clear a pending PMI interrupt.

It is possible for multiple SAPIC messages of PMI delivery type to be delivered to a processor simultaneously. In this situation, only one PMI interrupt will be recognized. This is analogous to sending edge triggered external interrupts using the same interrupt vector. To guard against loss of such PMI messages, SALE_PMI layer on the sending processor may communicate the reason for the PMI using memory data structures.

6.4 Special Considerations for Multiprocessor Configurations

Depending on the platform, SALE_PMI may determine whether to bring all the processors in the system to the SAL PMI environment. This can be achieved by sending a SAPIC message with delivery type of PMI. In an MP configuration, there could be conflicts between PMI and machine check. One of the processors could be in SAL_CHECK, trying to bring other processors to SAL_MC_RENDEZ using the MC_rendezvous external interrupt. If the latter were in SALE_PMI, the MC_rendezvous external interrupt would not be recognized immediately and this might necessitate the monarch processor to issue an INIT to the processor in the PMI environment. Since recoverability from INIT is minimized when PSR.ic is 0, it is recommended that SALE_PMI handler saves the interruption resources and set the PSR.ic bit to 1 as early as possible.

§

7 IA-32 Support (Optional)

7.1 IA-32 Support Model

This chapter describes the optional IA-32 support within SAL during the booting process. Additionally, it provides some guidelines on the choice of IA-32 instructions to SAL developers who plan to re-use existing IA-32 BIOS code.

For details on IA-32 instruction execution on Itanium architecture processors, refer to Volume 1, Chapter 6 and Volume 2, Chapter 10 of the *Intel® Itanium® Architecture Software Developer's Manual*.

IA-32 support code in SAL cannot be used after an operating system (IA-32 operating system or Itanium architecture-based operating system) has taken control of the translation resources. Most Itanium architecture-based operating systems will provide their own IA-32 support code and not use the code in SAL. If the user boots an IA-32 operating system, SAL would have invoked the PAL_ENTER_IA_32_ENV procedure which activates the PAL layer in support of IA-32 operating systems and this PAL firmware layer configures the processor to behave like a Pentium® III processor, obviating the need for SAL's IA-32 support code. For more details, refer to *Intel® Itanium® Processor Reference Manual for Software Development*.

During the platform initialization phase of the boot sequence, the IVA may point to a 32 KB IVT in the firmware address space. Some of the trap handlers in the IVT could support execution of IA-32 code. Thus, it is possible to execute IA-32 code early in the boot sequence, if needed. Refer to Chapter 3, for fault/trap handler support requirements in SAL.

7.2 IA-32 Support Requirements

Itanium architecture-based platforms may contain one or more IA-32 adapter cards containing IA-32 Option ROMs. If the adapter cards support boot devices, they will need to be initialized in the process of booting the operating system. The IA-32 support code in SAL will be exercised while executing the IA-32 code. Also, since SAL contains IA-32 support code for execution of the IA-32 Option cards, a portion of the SAL layer for Itanium architecture-based platforms may itself be coded in IA-32 ISA (that is, the traditional IA-32 System ROM BIOS may be reused).

7.2.1 Resources Supported by SAL

The following resources need to be supported by SAL for maintaining legacy compatibility.

- Legacy Memory Map:
  - Interrupt Vector Area 0 – 0x3FF: Contains entrypoints for software interrupts in offset:segment format.
  - BIOS RAM Data Area 0x400 – 0x4FF: Data variables stored by System BIOS and Option ROMs.
  - Option ROM Space: 0x000C_0000 – 0x000D_FFFF.
  - legacy Compatibility Entrypoints: Addresses in the 0x000F_E000 to 0x000F_FFFF range pointing to entrypoints and tables.

It is expected that SAL code would be designed to use identical virtual-to-physical memory mappings and not conflict with the IA-32 BIOS memory usage.

- Legacy I/O Map: Motherboard I/O ports are in the range of 00 to 0xFF and other IA-32 devices occupy the rest of the 64K I/O space. The most important I/O ports used by BIOS code are Interrupt controller (0x20, 0x21, 0xA0, 0xA1), Interval timer (0x40 to 0x43) and CMOS RAM (0x70, 0x71).

7.2.2 Overview of IA-32 Support Layer Functionality

IA-32 support layer is mainly required for the following areas:

- Memory mapped I/O: The processor needs to provide the uncacheable semantics for memory mapped I/O to devices such as VGA buffer. Also, the search for memory mapped devices need to be performed without caching artifacts. Caches within the processor are enabled by invoking the PAL_PROC_SET_FEATURES call. When processor caches are enabled, the uncacheable memory attribute required for I/O completion is specified by setting bit 63 of the memory address, in physical addressing mode. Bit 63 of the physical address has no effect while processor caches have been disabled using the PAL_PROC_SET_FEATURES call.

  Since it is not possible to generate an address with bit 63 set while operating in the 32-bit IA-32 ISA mode, IA-32 code needs to be executed with translations enabled and TLBs need to specify the uncacheable memory attribute. TLBs provide the same functionality as MTRRs on a Pentium Pro processor.
- Handle traps during IA-32 code execution.
- Virtualizing legacy peripherals: If some legacy devices are not present on the platform, SAL may provide the necessary virtualization during IA-32 code execution by setting up TLBs to trap the accesses.

7.2.3 IA-32 Instruction Usage Guidelines

IA-32 system BIOS code executing *within the SAL environment* must follow these guidelines in its usage of IA-32 instructions, in order to limit SAL's IA-32 support requirements. These restrictions do not affect operation of existing IA-32 *Option ROMs* which are restricted to operating in IA-32 real mode. Option ROM code on legacy compatible platforms are already compliant with the following guidelines:

- IA-32 code shall not use protected mode instructions of the IA-32 ISA. Only real mode and big real mode opcodes are permitted. The transitions between real mode and big real mode will occur using the SAL code that sets up the appropriate IA-32 segment descriptors, and not by use of the IA-32 LGDT instruction. The traditional IA-32 BIOS functions requiring protected mode usage, such as search for PCI Option ROMs near 4 GB address, can be done easily using the big real mode or in the Itanium system environment. SAL will provide support the Extended Memory Move function (IA-32 INT 0x15, sub function 0x87) for moving data to and from addresses above 1 MB.
- IA-32 code shall not alter the following bits of EFLAGS: TF, NT, RF, AC.
- IA-32 code shall not use code involving IA-32 privileged instructions such as LGDT, RDMSR, MOV to CRs, DRs, and so on. Such functionality must be replaced by equivalent Itanium instructions. Refer to the *Intel® Itanium® Architecture Software Developer's Manual* for a complete list of instructions that cause the IA-32 Instruction Intercepts.

- SAL shall provide necessary emulation support for the following instructions:

  CLI, CLTS, HLT, INT 3, INTO, INVD, INVLPG, IRET, IRETD,
  MOV SS, POP SS, POPF, POPFD, STI, WBINVD

- IA-32 code shall not use code involving IA-32 Call Gates.
- IA-32 stack must be aligned on an even byte boundary. The IA-32 support layer in SAL will need to retrieve or store values into the IA-32 stack in order to emulate instructions such as INT, IRET. If the IA-32 stack is aligned on an odd byte boundary, an unaligned data reference fault will result and SAL does not provide a handler for this exception.

The above restrictions are not applicable when the operating system kernel takes over. Thus, an IA-32 or Itanium architecture-based operating system may set up the environment for IA-32 protected mode and invoke protected mode functions of IA-32 BIOS.

7.2.4 IA-32 Support Environment

This section describes the execution environment for IA-32 code.

1. IA-32 BIOS code will be executed with Instruction translation on, Data translation on and RSE translation on (PSR.it = 1, PSR.dt = 1, PSR.rt = 1). The PSR.ac bit may be set to 0 to mask exceptions caused by unaligned memory references during execution of IA-32 code.
2. The following traps will be supported in the Interrupt Vector Table (IVT) for supporting IA-32 execution:
   - IA-32_Exception vector
   - IA-32_Intercept vector
   - IA-32_ Interrupt vector
   - External interrupt vector
3. SAL will set up CFLG register which maps to the IA-32 system registers CR0 and CR4. When SAL procedures are called by the operating system loader, SAL will set up the appropriate value in the CFLG register, if transition to IA-32 ISA mode is required.
4. The CFLG.io bit will be set to 0 to eliminate the need for Task State Segment (TSS) while executing IA-32 code. IA-32 EFLAG.iopl field should be set to 3 to permit IA-32 I/O instructions without causing any traps. IOBASE register and translation mechanisms within the processor will be set up to automatically convert the IA-32 I/O accesses to Itanium instructions for memory load or store operations with the uncacheable memory attribute. If some legacy devices are not present on the platform, TLBs may be set up to trap the accesses and SAL can either redirect the I/O to a different hardware on the platform or provide suitable software emulation.
5. The PSR.i bit may be set to 1 to enable interrupts in the Itanium system environment and the CFLG.if bit may be set to 1 to allow IA-32 code to control interrupt masking. With these settings, the IA-32 EFLAG.if bit will enable or disable external interrupts while executing IA-32 code. The EFLAG.if bit cannot mask/ unmask interrupts while executing the Itanium instruction set.
6. The CFLG.ii bit may be set to 0 if there is no need to intercept changes to interrupt enable flag.

7.2.5 IA-32 Interruption Handler Support

External interrupts, IA-32 defined exceptions and software interrupts are delivered to the interruption handlers in the Itanium system environment. All interruption handlers may run with PSR.dt, PSR.rt turned off to avoid the Nested TLB fault that can occur while accessing the fault handler's local variables and data structures. SAL will populate the following handlers in the IVT to handle interruption in its environment:

- IA-32_Exception vector: This handler will handle exceptions caused by IA-32 instructions such as Divide by zero fault. These interruptions should not occur while executing debugged IA-32 BIOS code. The exception should be reflected to IA-32 code using the IA-32 real mode Interrupt Descriptor Table (IDT) at locations 0 to 0x3FF. Typically, IA-32 code in the IDT will display an error message when such exceptions are encountered.
- IA-32_Intercept vector: This handler will handle several categories of intercepted instructions as described in the *Intel® Itanium® Architecture Software Developer's Manual*.
  - Instruction Intercept: Refer to Section 7.2.3 for a list of the IA-32 instructions that must be emulated by SAL.
  - Lock Intercept: This interruption handler will be invoked for the LOCK and the XCHG instructions. This intercept can be avoided by enabling the lock feature in the Itanium processor's Default Control Register (DCR.lc = 0), if the platform can support locked read modified writes. If the platform does not support the bus lock signal, PAL_BUS_SET_FEATURES may be invoked to execute the locked transactions as a series of non-atomic transactions. This, in effect, will mask the lock intercept. Refer to the *Intel® Itanium® Architecture Software Developer's Manual* for details.
  - Gate Intercept: Support is not needed for trapping privilege transitions using gates. IA-32 System BIOS code shall avoid this intercept and Option ROM code is not permitted to use privilege transitions using gates.
  - IA-32 System Flag Intercept: This intercept can be avoided for the STI, CLI, POPF and POPFD instructions by setting CFLG.if bit to 1, which allows the IA-32 code to control interrupt masking with the IA-32 EFLAG.if bit. To support the MOV SS and the POP SS instructions, SAL shall disable interrupts and execute the next IA-32 instruction with the PSR.ss set to 1. This will generate an IA-32_Exception (Debug). The handler for this exception will restore the previous value of PSR.i and return to the IA-32 code.
- IA-32_Interrupt vector: This handler supports the IA-32 INT instruction. SAL will provide the necessary emulation support for the Extended Memory Move function (INT 0x15, subvention 0x87) in order that real mode code may move data to and from addresses over 1 MB without requiring a transition to the Itanium system environment. The rest of the INT instructions will be emulated by jumping to the address pointed to by the IA-32 real mode IDT. Following is an example of pseudo code:
  - Get the Software interrupt number *nn* from ISR.vector.
  - Use *nn* as an index into the IA-32 real mode Interrupt Descriptor Table at location `0000h` and obtain the *segment:offset* of IA-32 code to be invoked.
  - Store the two byte FLAGS on IA-32 stack.
    - Store the *segment:offset* address of the IA-32 instruction following the *INT nn* on IA-32 stack. Store the IA-32 *segment:offset* addresses in the appropriate Itanium architecture processor registers corresponding to IP, CS selector, CS segment descriptor and transition to IA-32 code using *RFI* instruction.
    - The IA-32 code will terminate by issuing an *IRET* or a *RET 2* instruction and this will return to the IA-32 instruction following the *INT nn*.

- External interrupt vector: Hardware interrupts will be received by SAL in the Itanium system environment which will obtain the interrupt vector corresponding to the interrupting source. For more details, refer to Section 3.3.1. If the interrupts need to be reflected to IA-32 code, the address will be derived from the IA-32 Interrupt Descriptor Table.

§

8 Calling Conventions

8.1 SAL Calling Conventions

The following general rules govern the definition of the SAL procedure calling conventions.

8.1.1 Definition of Terms

The terms used in the definition of the requirements are defined in Table 8-1.

Table 8-1. Definition of Terms

Term	Description
Entry	Start of the first instruction of the SAL procedure.
Exit	Start of the first instruction after return to caller's code.
0	Must be zero at entry to or exit from the procedure.
1	Must be one at entry to or exit from the procedure.
C	The state of bits marked with C are defined by the caller. If the value at exit is also C, it must be the same as the value at entry.
Unchanged	The SAL procedure must not change these values from their entry values during execution of the procedure.
Scratch	There are no requirements on the state of these values during execution of the procedure. The SAL procedure may modify them as necessary during execution of the procedure.
Preserved	The SAL procedure may modify these values as necessary during execution of the procedure. However, they must be restored to their entry values prior to exit from the procedure.

8.1.2 Processor State

Table 8-2 defines the requirements for the Processor Status Register (PSR) at entry to and at exit from a SAL procedure call. The operating system loader must follow the state requirements for PSR shown below. SAL calls that invoke PAL procedures may impose additional requirements.

Table 8-2. State Requirements for PSR

PSR Bit	Description	Entry	Exit	Class
be	Big-endian memory access enable	0	0	Preserved
up	User performance monitor enable	C	C	Unchanged
ac	Alignment check	C	C	Preserved
mfl	Floating-point registers f2-f15 written	C	C	Preserved
mfh	Floating-point registers f16-f127 written	C	C	Preserved
ic	Interruption state collection enable	C 0	C 0	Preserved[1] Unchanged
i	Interrupt unmask	C	C	Preserved[2]
pk	Protection key validation enable	C	C	Unchanged
dt	Data address translation enable	C	C	Preserved[a]
dfl	Disabled FP register f2 to f15	C	C	Unchanged[3]
dfh	Disabled FP register f16 to f127	C	C	Unchanged$_c$

Table 8-2. State Requirements for PSR (Continued)

PSR Bit	Description	Entry	Exit	Class
sp	Secure performance monitors	C	C	Unchanged
pp	Privileged performance monitor enable	C	C	Unchanged
di	Disable ISA transition	C	C	Preserved
si	Secure interval timer	C	C	Unchanged
db	Debug breakpoint fault enable	C	C	Unchanged
lp	Lower-privilege transfer trap enable	C	C	Unchanged
tb	Taken branch trap enable	C	C	Unchanged
rt	Register stack translation enable	C	C	Preserved[a]
cpl	Current privilege level	0	0	Unchanged
is	Instruction set	0	0	Preserved
mc	Machine check abort mask	C 1	C 1	Preserved[4] Unchanged
it	Instruction address translation enable	C	C	Unchanged
id	Instruction debug fault disable	C	C	Unchanged
da	Disable Data access/dirty-bit faults	0	0	Unchanged
dd	Data debug fault disable	0	0	Unchanged
ss	Single step trap enable	0	0	Unchanged
ri	Restart instruction	0	0	Preserved
ed	Exception deferral	0	0	Preserved
bn	Register bank	1	1	Preserved
ia	Disable instruction access-bit faults	0	0	Unchanged

Notes:

1. If this bit is 0 on entry, the value of this bit shall be 0 on exit and it must be classified as unchanged.
2. SAL procedures shall not enable interrupts if interrupts are disabled on entry.
3. If this bit is 1 on entry, a Disabled FP-register vector fault may occur.
4. In general, this bit shall be 0 on entry, 0 on exit and of class preserved. If this bit is 1 on entry, the value on exit shall be 1 and must be classified as unchanged.

8.1.3 System Registers

Table 8-3. System Register Conventions

Name	Description	Class
DCR	Default Control Register	Unchanged
ITM	Interval Timer Match Register	Unchanged
IVA	Interruption Vector Address	Unchanged
PTA	Page Table Address	Unchanged
GPTA	Reserved IA-32 Resource	Unchanged
IPSR	Interruption Processor Status Register	Scratch
ISR	Interruption Status Register	Unchanged[1]
IIP	Interruption Instruction Bundle Pointer	Unchanged[a]
IFA	Interruption Faulting Address	Unchanged[a]
ITIR	Interruption TLB Insertion Register	Unchanged[a]
IIPA	Interruption Instruction Previous Address	Unchanged[a]
IFS	Interruption Function State	Unchanged[a]

Table 8-3. System Register Conventions (Continued)

Name	Description	Class
IIM	Interruption Immediate Register	Unchanged[a]
IHA	Interruption Hash Address	Unchanged[a]
LID	Local Interrupt ID	Unchanged
IVR	Interrupt Vector Register (read only)	Unchanged
TPR	Task Priority Register	Unchanged
EOI	End of Interrupt	Unchanged
IRR0-IRR3	Interrupt Request Registers 0-3 (read only)	Unchanged[a]
ITV	Interval Timer Vector	Unchanged
PMV	Performance Monitoring Vector	Unchanged
CMCV	Corrected Machine Check Vector	Unchanged
LRR0-LRR1	Local Redirection Registers 0-1	Unchanged
RR	Region Registers	Preserved
PKR	Protection Key Registers	Unchanged
TR	Translation Registers	Unchanged[2]
TC	Translation Cache	Scratch
IBR/DBR	Break Point Registers	Preserved
PMC	Performance Monitor Control Registers	Preserved
PMD	Performance Monitor Data Registers	Unchanged[3]

Notes:

1. SAL procedures may not update these registers, but the arrival of asynchronous interrupts may cause them to change.
2. If an implementation provides a means to read TRs through a PAL procedure call, this should be preserved.
3. No SAL procedure writes to the PMD. Depending on the PMC, the PMD may be kept counting performance monitor events during a procedure call.

8.1.4 General Registers

SAL will use the standard calling convention as described in the *Itanium® Software Conventions and Runtime Architecture Guide*. Routines written using this convention may be written either in assembly or C or other high level languages.

Table 8-4. General Registers – Standard Calling Conventions

Register	Conventions
GR0	Always 0.
GR1	Special; global data pointer (GP).
GR2 – GR3	Scratch; used with 22 bit immediate add.
GR4 – GR7	Preserved.
GR8 – GR11	Scratch, procedure return value.
GR12	Special, stack pointer. preserved.
GR13	Special, thread pointer. preserved.
GR14 – GR31	Scratch.

Table 8-4. General Registers – Standard Calling Conventions (Continued)

Register	Conventions
Bank 0 Registers (GR16 – GR23)	Preserved.
Bank 0 Registers (GR 24 – GR31)	Scratch.
GR32 – GR127	Stacked registers: in0 – in95: input arguments (SAL index must be in0) loc0 – loc95: local variables out0 – out95: output arguments

The GP for the SAL code should be known to system software as SAL passes it as one of the boot parameters. The caller must initialize the GP and SP prior to calling a SAL procedure. If the GP and SP values do not point to valid addresses, the SAL behavior is undefined. A minimum 16 KB bytes must be available for the stack space of the SAL procedure and a minimum of 16 KB bytes of RSE backing store must be available for SAL.

8.1.5 Floating-Point Registers

Although there is no SAL procedure that passes floating-point parameters, the floating-point register conventions are the similar to those specified by the *Itanium® Software Conventions and Runtime Architecture Guide*. SAL shall not use the floating-point registers 32 to 127, thus eliminating the need for the operating system to save these registers across SAL procedure calls. All the pending floating-point exceptions must be handled before calling SAL if the execution environment for calling SAL cannot handle any floating-point exceptions.

8.1.6 Predicate Registers

The conventions for these registers follow the *Itanium® Software Conventions and Runtime Architecture Guide*.

8.1.7 Branch Registers

The conventions for these registers follows the *Itanium® Software Conventions and Runtime Architecture Guide*. Note that the application register AR44 (ITC: Interval Time Counter) must remain *unchanged* (as in the definition presented in Table 8-1, “Definition of Terms”)

8.1.8 Application Special Registers

The application registers follow the *Itanium® Software Conventions and Runtime Architecture Guide*.

8.1.9 Parameter Buffers

The parameter buffers to SAL_PROC must be aligned to the greater of its data type size or 8-byte aligned. SAL may check for alignment and return a –2 error if unaligned. Addresses passed to SAL procedures as buffers for return parameters or input parameter may be physical or virtual and must be consistent with the PSR.dt value. The addressing mode of the parameter buffers depends on the execution environment of the caller. The following conventions are followed for the parameter buffers:

- Until the operating system takes over the IVT and translation faults, parameter buffers passed to SAL are identity mapped virtual addresses and are accessible by the region register 0 (RR0). In this environment, SAL can handle the access faults while accessing parameter buffers if the buffers are identity mapped.
- Parameter buffers passed to SAL runtime services can be either physical or virtual. If the parameter buffers are virtual, the operating system runtime execution environment must provide the proper mapping for the parameter buffers.

8.2 Software Interface Conventions for SAL Procedures

A generic interface is provided between the Itanium architecture-based operating system and SAL. An Itanium architecture-based operating system always follows the standard calling convention to call SAL functions. The parameters passed to the SAL interface are defined as follows:

SAL_PROC(*arg0, arg1,..., arg7*)

Where input parameters (maximum of eight 64-bit values) are:

arg0 – functional identifier. The upper 32 bits are ignored and only the lower 32 bits are used. The following functional identifiers are defined:

0x01XXXXXX – Architected SAL functional group.

0x02XXXXXX to 0x03XXXXXX – OEM SAL functional group. Each OEM is allowed to use the entire range in the 0x02XXXXXX to 0x03XXXXXX range.

0x04XXXXXX to 0xFFFFFFFF – Reserved.

arg1 – the first parameter of the architected/OEM specific SAL functions.

arg2 to *arg7* – additional parameters for architected/OEM specific SAL functions.

and return parameters (maximum of four 64-bit values) are:

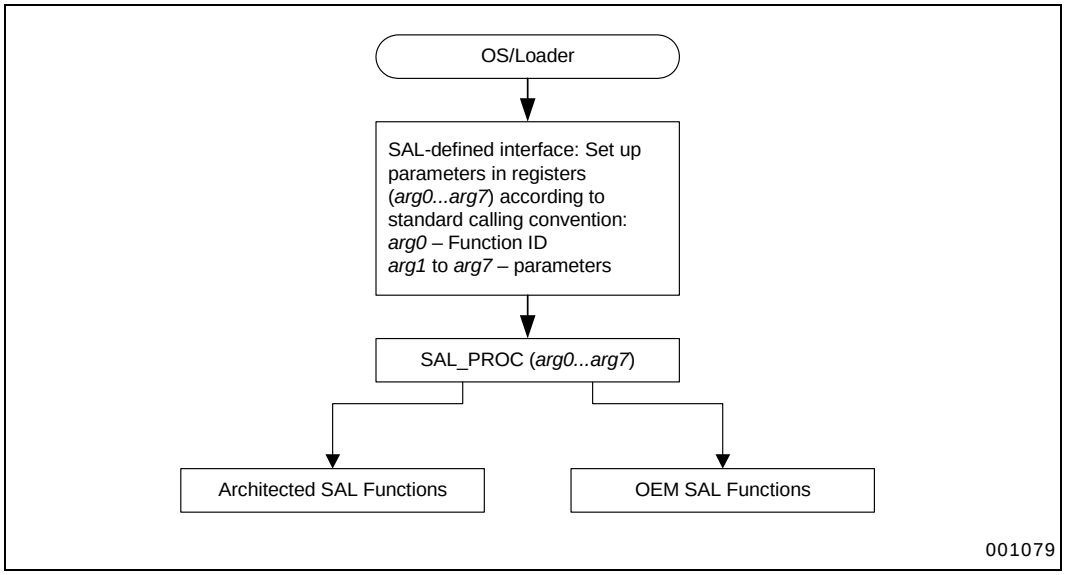
ret0 – return status: positive number indicates successful, negative number indicates failure.

ret1 to ret3 – other return parameters.

8.2.1 Control Flow of the SAL Interface

The operating system loader follows the standard calling convention to call both architected and OEM specific SAL functions. The operating system loader sets up the appropriate parameters in the Itanium architecture processor's general registers according to the calling convention and calls SAL_PROC. The first parameter passed to SAL_PROC specifies the functional identifier and based on the functional identifier, SAL dispatches the function to the appropriate functional block. Figure 8-1 shows the control flow of the SAL interface.

Figure 8-1. Control Flow of the SAL Procedure Interface



8.2.2 Calling Architected / OEM SAL Functions

To call an architected or OEM specific SAL function, the operating system loader sets up *arg0* to the appropriate architected SAL or OEM specific SAL functional identifier. It then sets up other parameters in *arg1* to *arg7* as specified by the SAL functional description and calls SAL_PROC. All reserved arguments shall contain the value of 0 else SAL shall return to the caller with the status of "Invalid argument." SAL_PROC dispatches this function to either the architected SAL function handler or the OEM specific SAL function handler based on the functional identifier. The SAL function returns the status in ret0 and the additional return parameters in ret1 to ret3. If the SAL function is not implemented, the SAL shall return with the *Not Implemented* return status.

8.2.2.1 SAL Return Status Value

SAL procedures return a 64-bit status value in the `ret0` parameter. Positive numbers indicate success and negative numbers indicate failure. Table 8-5 summarizes the error code.

Table 8-5. SAL Return Status

Register	Conventions
0	Call completed without error.
1	Call completed without error but some information was lost due to overflow.
2	Call completed without error; effect a warm boot of the system to complete the update.
3	More information is available for retrieval.
–1	Not implemented.
–2	Invalid Argument.
–3	Call completed with error due to hardware malfunction, firmware error, or if improperly called (for example, with PSR.cpl other than 0)
–4	Virtual address not registered.

Table 8-5. SAL Return Status (Continued)

Register	Conventions
–5	No information available.
–9	Scratch buffer required.
–15	Retry (CMC/CPE)

§

9 SAL Procedures

9.1 SAL Runtime Services Overview

SAL runtime services are the firmware procedures which provide abstractions to the operating system when it is executing. These services provide a platform-independent interface for hardware components. Runtime services contain procedures called by the operating system to access platform hardware features on behalf of the operating system. Runtime services should take no more time to perform an action than it would take the operating system to perform the same action.

The entire SAL runtime services code must be located in one contiguous memory area. Similarly, the SAL runtime services data area must be located in one contiguous memory area.

SAL runtime services are called from the following execution environment:

- Operating system runtime execution environment. The normal operating system execution environment is with translation on and interrupts enabled but the operating system may choose to call SAL runtime services in physical mode.
- Operating system machine check and initialization handler. The execution environment for these are provided by SAL and are in physical mode with interrupts disabled.
- SAL PMI handler. The execution environment is in physical mode with interrupts disabled.

The following general rules govern the operational characteristics of the SAL procedures:

- SAL runs in privilege level 0 and will return an error if called from other privilege levels.
- SAL runs little endian.
- SAL procedures follow the standard calling convention for the Itanium architecture processors. The SAL runtime services shall be implemented completely in the Itanium processor system environment.
- Bit 63 of arguments which are physical addresses must be set corresponding to the argument's memory attribute.
- Some SAL procedures are primarily intended for use during OS initialization and designed to be called on one processor. These are not required to be re-entrant. Some SAL procedures are required to be called on multiple processors simultaneously. These are required to be MP-safe but need not be re-entrant. Some SAL procedures may be re-invoked on the same processor, for example, the invocation of the SAL_GET_STATE_INFO procedure for a CPE event may be interrupted by the invocation of the same procedure for an MCA event on the same processor. Such procedures need to be re-entrant as well as MP-safe. These requirements are specified in Table 9-2. For the procedures that are not re-entrant, the operating system is required to enforce single threaded access.
- The operating system must ensure that SAL procedures run to completion on the same processor, that is, the SAL procedure cannot migrate to another processor due to OS context switching.

- Architected SAL runtime procedures are called either in virtual or physical mode under the operating system execution environment. Virtual mode means that PSR.it, PSR.dt, and PSR.rt are set to 1, while “Physical mode” means that all 3 bits are 0. If these 3 bits don't all match, SAL shall return a -3 error.
- OEM-specific SAL Runtime procedures may not support both virtual and physical modes of operation. These calls shall return a -3 error if called in an unsupported mode.
- All SAL procedures that don’t return the status of unimplemented procedure (–1), must be implemented.

9.1.1 Invoking SAL Runtime Services in Virtual Mode

SAL runtime services may be called either in virtual or physical mode. The normal operating system execution environment is with translation on and interrupts enabled but operating system may choose to call SAL runtime services in physical mode.

The parameters passed to SAL runtime services must be consistent with the addressing environment, that is, PSR.dt, PSR.rt setting. Additionally, the GP register must contain the physical or virtual address of the SAL’s GP value provided to the operating system in the Entrypoint Descriptor (refer to Table 3-5). SAL can compute the addresses of code and data objects within SAL using offsets relative to the ip and GP. In other words, SAL code will be position independent.

The hand-off state from the EFI to the operating system loader will indicate the SAL’s requirements for virtual address mappings. (Refer to the *Extensible Firmware Interface Specification* for details.) In an MP configuration, the virtual addresses registered by the operating system must be valid globally on all the processors in the system. The *Extensible Firmware Interface Specification* also provides the interfaces for the operating system to register the virtual address mappings. Some typical requirements for virtual address mappings are described below:

1. The code and data areas of PAL and SAL in memory must be mapped contiguously in virtual address space.
2. Some of the SAL runtime services, for example, SAL_CACHE_FLUSH, will need to invoke PAL procedures in memory. Prior to invoking the SAL procedures in virtual mode, the operating system must register the virtual address of the PAL code space in memory. If SAL needs to invoke a PAL procedure, SAL shall do so in the same mode in which it was called by the operating system (that is, without changing the PSR.dt, PSR.rt and PSR.it bits). While invoking these SAL procedures, the operating system must provide the appropriate translation resources required by PAL (that is, ITR and DTC covering the PAL code area).
3. The SAL_UPDATE_PAL procedure will invoke some PAL procedures in the firmware address space. The operating system must register the virtual address of the firmware address space (ending at 4 GB). The operating system must provide a contiguous virtual address mapping for the entire firmware address space. If the SAL_UPDATE_PAL procedure is called in the virtual mode, SAL will compute the virtual addresses of the relevant PAL procedures in the firmware address space and invoke them in the virtual addressing mode.
4. The operating system shall register the virtual addresses of the Firmware Reserved Memory if requested by the SAL (refer to Table 3-5). Such registration must be done prior to making SAL calls in virtual mode and the operating system must provide a contiguous virtual address mapping for each of the data areas.

9.2 SAL Procedures that Invoke PAL Procedures

Some of the SAL procedures incorporate both processor and platform functionality. To perform the processor functionality, these SAL procedures invoke the underlying PAL procedures. These dependencies are listed in Table 9-1. The operating system is required to call the SAL procedures instead of directly calling the PAL procedures.

Table 9-1. SAL Procedures Invoking PAL Procedures

SAL Procedure	PAL Procedure
SAL_CACHE_FLUSH	PAL_CACHE_FLUSH
SAL_CLEAR_STATE_INFO	PAL_MC_CLEAR_LOG
SAL_GET_STATE_INFO	PAL_MC_ERROR_INFO
Return to SAL at the end of OS_MCA, OS_INIT	PAL_MC_RESUME

9.3 SAL Procedure Summary

Table 9-2. SAL Procedures

Procedure	Function ID (hex)	Description	MP-Safe	Re-entrant
SAL_SET_VECTORS	0x01000000	Register software code locations with SAL.		
SAL_GET_STATE_INFO	0x01000001	Return Machine State information obtained by SAL.	X	X
SAL_GET_STATE_INFO_SIZE	0x01000002	Obtain size of Machine State information.	X	X
SAL_CLEAR_STATE_INFO	0x01000003	Clear Machine State information.	X	X
SAL_MC_RENDEZ	0x01000004	Cause the processor to go into a spin loop within SAL.	X	
SAL_MC_SET_PARAMS	0x01000005	Register the machine check interface layer with SAL.		
SAL_REGISTER_PHYSICAL_ADDR	0x01000006	Register the physical addresses of locations needed by SAL.		
SAL_CACHE_FLUSH	0x01000008	Flush the instruction or data caches.	X	
SAL_CACHE_INIT	0x01000009	Initialize the instruction and data caches.	X	
SAL_PCI_CONFIG_READ	0x01000010	Read from the PCI configuration space.	X	X
SAL_PCI_CONFIG_WRITE	0x01000011	Write to the PCI configuration space.	X	X
SAL_FREQ_BASE	0x01000012	Return the base frequency of the platform.	X	
SAL_PHYSICAL_ID_INFO	0x01000013	Returns information on the physical processor mapping within the platform.	X	
SAL_UPDATE_PAL	0x01000020	Update the contents of firmware blocks.		

SAL_CACHE_FLUSH

Purpose: To flush the instruction or data caches on the current processor as well as the platform.

Calling Conventions: Standard. Callable by the operating system in virtual or physical mode.

Arguments:

Argument	Description
func_id	Function ID of SAL_CACHE_FLUSH within the list of SAL procedures
i_or_d	Unsigned 64-bit integer denoting type of cache flush operation: 1 = Flush instruction cache 2 = Flush data cache 3 = Flush instruction and data cache 4 = Make local instruction caches coherent with the data caches Other values are reserved
Reserved	0
Reserved	0
Reserved	0
Reserved	0
Reserved	0

Returns:

Return Value	Description
status	Return status of SAL_CACHE_FLUSH procedure
Reserved	0
Reserved	0
Reserved	0

Status:

Status Value	Description
0	Call completed without error
–2	Invalid Argument
–3	Call completed with error
–4	Virtual address not registered

Description: Flushes the instruction and/or data caches to memory from all levels of cache hierarchy, controlled by the platform and the processor on which this procedure is invoked. This SAL procedure must be invoked on at least one processor within a cache hierarchy (that is, if platform caches are node specific, this SAL procedure must be invoked on each node). If cache hierarchy information is not known, then it must be invoked on each logical processor.

The *i_or_d* parameter specifies the instruction and/or data caches. Unified caches are flushed with both instruction and data caches. This procedure has the effect of invalidating all instruction cache lines, or causing a write back and then invalidating all data cache lines.

With the *i_or_d* parameter value of 4, the caller specifies SAL to make the local instruction caches coherent with the data caches. This has the effect of ensuring that the local instruction caches see the effects of earlier stores of instruction code done by the local processor.

This SAL procedure invokes the corresponding PAL procedure, PAL_CACHE_FLUSH. Refer to the *Intel® Itanium® Architecture Software Developer's Manual* for details. This PAL procedure may return to SAL without completing the flush operation should there be an intervening interrupt. The PAL procedure also returns the external interrupt vector as a return parameter. In order to execute the associated external interrupt handler, SAL shall:

- Write to the EOI register (CR.eoi);
- Repost the interrupt by issuing an IPI message to self with the vector;

- Re-enable interrupts; and
- On return from the external interrupt handler, re-invoke the PAL_CACHE_FLUSH procedure specifying the continuation point for the cache flush.

If interrupts need to be handled on a timely basis, this SAL procedure must be invoked with interrupts enabled, that is, PSR.i set to 1.

This SAL procedure is required to be MP-safe to permit the operating system on the various processors to invoke this SAL procedure simultaneously.

Platform Requirements: None

SAL_CACHE_INIT

Purpose: To initialize the instruction and data caches on the platform.

Calling Conventions: Standard. Callable by the operating system in virtual or physical mode.

Arguments:

Argument	Description
func_id	Function ID of SAL_CACHE_INIT within the list of SAL procedures
Reserved	0
Reserved	0
Reserved	0
Reserved	0
Reserved	0
Reserved	0
Reserved	0

Returns:

Return Value	Description
status	Return status of SAL_CACHE_INIT procedure
Reserved	0
Reserved	0
Reserved	0

Status:

Status Value	Description
0	Call completed without error
–2	Invalid Argument
–3	Call completed with error
–4	Virtual address not registered

Initializes the instruction and data caches controlled by the *platform only*. The operating system is required to invoke the PAL_CACHE_INIT procedure to initialize the instruction and data caches within the processor. All cache lines will be invalidated without causing a write back.

This SAL procedure must be invoked on at least one processor within a cache hierarchy (that is, if platform caches are node specific, this SAL procedure must be invoked on each node). If cache hierarchy information is not known, then it must be invoked on each logical processor.

This SAL procedure is required to be MP-safe to permit the operating system on the various processors to invoke this SAL procedure simultaneously.

Platform Requirements: None

SAL_CLEAR_STATE_INFO

Purpose: This procedure is used to invalidate the error record logged by SAL with respect to the machine state at the time of MCAs, INITs, CMCs, CPEs, or deconfigured processor error events.

Calling Conventions: Standard. Callable by the operating system in virtual or physical mode.

Arguments:

Argument	Description
func_id	Function ID of SAL_CLEAR_STATE_INFO call within the list of SAL procedures.
type	The type of information being invalidated: 0 – MCA event information 1 – INIT event information 2 – Processor CMC event information 3 – Corrected Platform event information 4 – Deconfigured processor information Other values are reserved
Reserved	0
Reserved	0
Reserved	0
Reserved	0
Reserved	0
Reserved	0

Returns:

Return Value	Description
status	Return status of SAL_CLEAR_STATE_INFO
Reserved	0
Reserved	0
Reserved	0

Status:

Status Value	Description
0	Call completed without error
3	More Error Records of the type are available to be retrieved and cleared
–2	Invalid Argument
–3	Call completed with error
–4	Virtual address not registered

Description: This call will invalidate an error record that is logged by SAL for the specified event type. Once the record has been invalidated, any subsequent calls to SAL_GET_STATE_INFO will get a –5 return value (no information available).

When called with argument type = MCA, INIT, or CMC, SAL_CLEAR_STATE_INFO clears record information for the processor on which the call is executed.
When called with argument type = CPE, SAL_CLEAR_STATE_INFO clears platform record information.

When called with "type" = "deconfigured processor", SAL_CLEAR_STATE_INFO clears deconfigured processor record information. When clearing deconfigured processor error records, the OS must call SAL_CLEAR_STATE_INFO on the same processor that the corresponding SAL_GET_STATE_INFO call was made on.

By calling this procedure, the operating system indicates that the resources used by the SAL to record the event are available for re-use.

If an MCA has been logged and the operating system fails to invalidate the record prior to another MCA, then SAL may save the additional error records and would consider this to be a fatal condition with a halt or

reboot of the system. This means that the error record information should be read as part of the OS_MCA handler or the operating system boot loader and then followed by an explicit clear operation.

SAL returns one error record at a time through the SAL_GET_STATE_INFO procedure. In certain cases, SAL may have multiple pending error records, to be retrieved. A return status value of 3 from this call indicates that SAL can be called to get more error records. Unless the current error record is cleared, further error records shall not be provided by the SAL.

Platform Requirements: None

SAL_FREQ_BASE

Purpose: This call returns the base frequency of the platform and other clock related information.

Calling Conventions: Standard. Callable by the operating system in physical or virtual mode.

Arguments:

Argument	Description
func_id	Function ID of SAL_FREQ_BASE within the list of SAL procedures
clock_type	Unsigned 64-bit integer specifying the type of clock source: 0 = Platform base clock frequency (clock input to the processor) 1 = Input frequency to the Interval Timer on the platform (optional) 2 = Input frequency to the Real time clock on the platform (optional) Other values are reserved
Reserved	0
Reserved	0
Reserved	0
Reserved	0
Reserved	0
Reserved	0

Returns:

Return Value	Description
status	Return status of SAL_FREQ_BASE procedure
clock_freq	Frequency information in ticks per second
drift_info	Drift value in parts per million clock ticks (optional)
Reserved	0

Status:

Status Value	Description
0	Call completed without error
–2	Invalid Argument
–3	Call completed with error
–4	Virtual address not registered

Description: This procedure is a runtime interface to determine the platform clock frequencies and to facilitate the operating system in selecting the most accurate clock source. This call could, in turn, use the services of PAL_FREQ_BASE if the processor implementation provides an output that is used as the platform clock.

This call is used in determining the frequencies of the processor, the system bus and the interval timer within the processor. First, the platform base clock frequency is determined by invoking this SAL procedure with the *clock_type* value of 0. The *clock_freq* return parameter provides the platform base clock frequency which is also the frequency of the clock input to the processor. The next step is for the operating system to invoke the PAL_FREQ_RATIOS and this procedure supplies the ratios of processor frequency, bus frequency and the interval timer frequency relative to the clock input to the processor. The products of the *clock_freq* return parameter and the various ratios provide the frequencies of the processor, the system bus and the interval timer within the processor.

This procedure must supply the correct value for the platform base clock frequency (*clock_type* of 0) and this value returned cannot be –1. Support for the other clock types and drift information is optional. The value in the *clock_freq* and *drift_info* fields is set to –1 if the requested information is not available.

Platform Requirements: Itanium architecture-based platforms must provide mechanisms to determine the base frequency of the platform.

SAL_GET_STATE_INFO

Purpose: Provide a programmatic interface to the processor and platform information logged by SAL with respect to the machine state at the time of the MCAs, INITs, CMCs, CPEs, or deconfigured processor error events.

Calling Conventions: Standard. Callable by the operating system in virtual or physical mode.

Arguments:

Argument	Description
func_id	Function ID of SAL_GET_STATE_INFO call within the list of SAL procedures.
type	The type of information being requested: 0 – MCA event information 1 – INIT event information 2 – Processor CMC event information 3 – Corrected Platform Event information 4 – Deconfigured processor information Other values are reserved
Reserved	0
memaddr	Memory address of the buffer where the requested information should be written
Reserved	0
Reserved	0
Reserved	0
Reserved	0

Returns:

Return Value	Description
status	Return status of SAL_GET_STATE_INFO
total_len	Size in bytes of the error information returned to the caller
Reserved	0
Reserved	0

Status:

Status Value	Description
0	Call completed without error
1	Call completed without error but some information was lost due to overflow
–2	Invalid Argument
–3	Call completed with error
–4	Virtual address not registered
–5	No information available
-15	Retry (CMC/CPE), invalid response for MCA/INIT events

Description: This procedure enables the operating system (and diagnostic software) to gather information obtained by SAL with respect to the machine state at the time of MCAs, INITs, Processor CMCs, or Corrected Platform events.

This call will return any information logged by SAL for the specified event *type*. In response to the MCA, Processor CMC, or Corrected Platform event, the operating system must call this procedure to obtain all the pending processor and platform error information that triggered the event.

The operating system is expected to call this procedure to retrieve the error record related to an event. The record is cleared by the operating system calling SAL_CLEAR_STATE_INFO. Once all the records have been cleared, any subsequent calls will get a –5 return value (no information available). The operating system must be prepared to handle the –5 return value.

For MCA and INIT events, SAL_GET_STATE_INFO must return the state information for the event in progress when called with the appropriate type unless the event record has been cleared. Unconsumed MCA or INIT

records that do not pertain to the current event may be returned in any order. For CMC and CPE events, records may be returned in any order.

A return status of –15 indicates that the SAL was unable to create a valid error record containing processor and/or platform error information due to a resource conflict. No error record is retrieved. The OS may call SAL_GET_STATE info at a later time to get the error information. Note that CMC/CPE interrupts may remain asserted (if these interrupts are enabled).

A return status of –15 is not a valid response for MCA/INIT events. For MCA/INIT events, if the SAL is unable to query the processor or platform to obtain error information due to a resource conflict, then the severity of the error most be reported as fatal in the record header.

The maximum length of the buffer required to hold the requested record information is obtained by calling the SAL_GET_STATE_INFO_SIZE procedure. The operating system is expected to allocate the memory buffer according to the returned size and provide the same for the *memaddr* argument. SAL returns only one error record at a time in the memory buffer area provided by the *memaddr* argument. SAL may indicate the existence of more than one error record through an appropriate return status during the call to the SAL_CLEAR_STATE_INFO procedure.

When called with argument type = MCA, INIT, or CMC, SAL_GET_STATE_INFO returns record information for the processor on which the call is executed.

When called with argument type = CPE, SAL_GET_STATE_INFO returns platform record information.

When called with "type" = "deconfigured processor", SAL_GET_STATE_INFO returns records pertaining to processors not available to the OS, which may include processors deconfigured on reboot, processors that failed self-test, or processors in SAL rendezvous. When the OS calls SAL_GET_STATE_INFO with "type" = "deconfigured processor," it should check the "PROC_CR_LID" field of the error info record to identify which processor an error records pertains to.

When called with "type" = "deconfigured processor," SAL_GET_STATE_INFO returns each error section in a separate error record. To clear a deconfigured processor error record, the OS must call SAL_CLEAR_STATE_INFO on the same processor that the corresponding SAL_GET_STATE_INFO call was made on. If SAL_CLEAR_STATE_INFO returns status = 3 (More Error Records of the type are available to be retrieved and cleared), the OS is expected to continue calling SAL_GET_STATE_INFO and SAL_CLEAR_STATE_INFO until all error records are exhausted.

SAL_GET_STATE_INFO implementations may choose to return all "deconfigured processor" error records from a single processor. Alternatively, SAL may distribute the return of deconfigured processor error records among the BSP and APs, but must avoid returning the same deconfigured processor record from different processors when the OS calls SAL_GET_STATE_INFO on the BSP and APs.

The information returned in the *memaddr* argument will contain the error information logged for an event for all the error devices like the called processor, memory controller, and I/O devices (including host bridges) in the system. The exact format of the records will be implementation-

dependent but the record for each type of device will follow an architected structure to allow the operating system to parse the records and extract the information. Refer to Chapter , “Error Record Structures” for format of the error record information returned in the *memaddr* argument.

Some categories of CMCs are entirely corrected by processor hardware. When this procedure is invoked for CMC information on a particular processor, SAL will obtain all of the processor error information, by invoking the PAL_MC_ERROR_INFO procedure. This procedure will then return to the caller both the information buffered by SAL and the information collected from the PAL.

If an MCA has been logged and the operating system fails to clear the log prior to another MCA, then SAL may save the additional error records and would consider this to be a fatal condition with a halt or reboot of the system. Hence, the MCA log information should be read as part of the OS_MCA handler or the operating system boot loader. On the other hand, if a CMC occurs prior to the operating system clearing the CMC error log, the same shall not be fatal. If SAL's internal buffers are not sufficient to log multiple errors of the same *type*, SAL may overwrite the event logs for lower priority events or older events.

An error record for an MCA event shall be available across reboots if the operating system has not cleared it already. SAL shall have an implementation specific NVRAM storage for backing up the MCA error records. The SAL is not required to log CMC or CPE error records to the NVRAM storage. An operating system is expected to retrieve and clear all pending error records during system boot time. If the operating system fails to clear the log before another MCA surfaces, the SAL may overwrite the unconsumed NVRAM log, if there is not space for another record.

Platform Requirements: None

SAL_GET_STATE_INFO_SIZE

Purpose: This procedure is used to obtain the maximum size of the information that could be logged by SAL with respect to the machine state at the time of MCAs, INITs, CMCs, CPEs, or deconfigured processor error events.

Calling Conventions: Standard. Callable by the operating system in virtual or physical mode.

Arguments:

Argument	Description
func_id	Function ID of SAL_GET_STATE_INFO_SIZE call within the list of SAL procedures.
type	The type of information being requested: 0 – MCA event information 1 – INIT event information 2 – Processor CMC event information 3 – Corrected Platform Event information 4 – Deconfigured processor information Other values are reserved
Reserved	0
Reserved	0
Reserved	0
Reserved	0
Reserved	0
Reserved	0

Returns:

Return Value	Description
status	Return status of SAL_GET_STATE_INFO_SIZE
size	The maximum size of the information logged for the specified type
Reserved	0
Reserved	0

Status:

Status Value	Description
0	Call completed without error
–2	Invalid Argument
–3	Call completed with error
–4	Virtual address not registered

Description: This call will return the maximum size of the processor or platform information logged by SAL for the specified event *type*. The operating system must make this call to determine the maximum size of data logged by SAL for each *type* of record. The operating system may then allocate suitable buffers, and provide the pre-allocated buffers as argument to subsequent calls to the SAL_GET_STATE_INFO procedure.

Platform Requirements: None

SAL_MC_RENDEZ

Purpose: This procedure causes the processor to go into a spin loop within SAL where SAL awaits a wake up from the monarch processor.

Calling Conventions: Standard. Callable by the operating system in virtual or physical mode.

Arguments:

Argument	Description
func_id	Function ID of SAL_MC_RENDEZ call within the list of SAL procedures
Reserved	0
Reserved	0
Reserved	0
Reserved	0
Reserved	0
Reserved	0
Reserved	0

Returns:

Return Value	Description
status	Return status of SAL_MC_RENDEZ procedure
Reserved	0
Reserved	0
Reserved	0

Status:

Status Value	Description
0	Call completed without error
–1	Not implemented
–3	Call completed with error
–4	Virtual address not registered

Description: This procedure is invoked on non-monarch processors during machine check processing. In some instances, this procedure may not be invoked on a processor until machine check processing has completed. To support these spurious MC_Rendezvous interrupts, the SAL first determines if a machine check is in progress. If not, then it returns immediately.

Once SAL has determined that a machine check is in progress, this procedure is invoked on non-monarch processors. This procedure will disable interrupts and set an implementation-specific check-in flag within the SAL data area to indicate to the monarch processor that the non-monarch processor has reached SAL. Next, it will call the PAL_MC_DRAIN procedure to complete all outstanding transactions within the processor. The non-monarch processor will then go into a spin loop awaiting a wakeup signal from the monarch processor. The wakeup mechanism may be an external interrupt or a memory variable as set up by the SAL_MC_SET_PARAMS procedure. SAL will return an error if a wakeup mechanism has not been registered.

If the external interrupt wake up mechanism is chosen, SAL spin loop routine will poll the local SAPIC IRR register for the bit corresponding to the selected wakeup interrupt to be set.

If a memory variable mechanism is chosen, SAL spin loop routine will poll the memory variable for the unique value that includes the contents of the Local ID Register (refer to Figure 3-1). The monarch processor will set this value to wake up one non-monarch processor at a time. SAL on the non-monarch processor will clear the memory variable to zero and return. This procedure may be called in virtual or physical mode but when memory variable mechanism is chosen, this procedure must be called in the same mode as the previous call to the SAL_MC_SET_PARAMS procedure that specified the memory variable.

If the rendezvoused processor takes a machine check while waiting for wake-up, the SAL should delay the handling of this subsequent machine check event until completion of the current machine check (that is, monarch processor returns from OS_MCA layer).

SAL implementations that do not provide this capability, may mask further machine checks and escalate future MCA events to BINIT# using the PAL_PROC_SET_FEATURES procedure. On receipt of the wake-up signal from the monarch, the SAL shall restore the original setting for error promotion and return to the operating system.

When this procedure returns, it is the responsibility of the operating system to clear the IRR bits for the MC_rendezvous interrupt and the wake up interrupt, if any.

This procedure is required for MP support. This SAL procedure is required to be MP-safe in order that operating system on the various non-monarch processors may enter the idle loop within the SAL simultaneously.

Platform Requirements: None

SAL_MC_SET_PARAMS

Purpose: This procedure allows the operating system to specify the interrupt number to be used by SAL to interrupt the operating system during the machine check rendezvous sequence as well as the mechanism to wake up the non-monarch processors at the end of machine check processing.

Calling Conventions: Standard. Callable by the operating system in virtual or physical mode.

Arguments:

Argument	Description
func_id	Function ID of SAL_MC_SET_PARAMS call within the list of SAL procedures
param_type	Unsigned 64-bit integer value for the parameter type of the machine check interface: 1 = rendezvous interrupt 2 = wake up 3 = Corrected Platform Error Vector Other values are reserved
i_or_m	Unsigned 64-bit integer value indicating whether interrupt vector or memory address is specified: 1 = interrupt vector 2 = memory address Other values are reserved
i_or_m_val	Unsigned 64-bit integer value specifying the interrupt vector or the memory address associated with the *i_or_m* parameter specified above.
time_out	Unsigned 64-bit integer value for rendezvous time out (in milliseconds).
mca_opt	Options set by the operating system for MCA handling within SAL.
Reserved	0
Reserved	0

Returns:

Return Value	Description
status	Return status of SAL_MC_SET_PARAMS procedure
time_out_min	Unsigned 64-bit integer value specifying the minimum rendezvous time out (in milliseconds)
Reserved	0
Reserved	0

Status:

Status Value	Description
0	Call completed without error
–1	Not implemented
–2	Invalid Argument
–3	Call completed with error
–4	Virtual address not registered

Description: This procedure allows the OS to specify parameters to the SAL for use during machine check processing. The parameters specified by the OS are applicable to all the processors within the system. This procedure is required for MP support. Section 3.2.2.1 provides details on how the rendezvous mechanism works in an MP configuration.

There are some machine check conditions which require the other processors in the system to be rendezvoused for error containment purposes and to recover from the error condition. This procedure allows the operating system to register the interrupt number it wishes to use for this purpose. Typically, when the operating system on the non-monarch processor receives the rendezvous interrupt, it will invoke the SAL_MC_RENDEZ procedure to go into a SAL spin loop routine. If the operating system does not register this interrupt, SAL_CHECK on the monarch processor will be forced to issue INIT and thereby compromise the recoverability from the machine check condition. This procedure must be called before MCAs can be handled by the operating system.

The *param_type* parameter indicates whether the rendezvous interrupt or wake up mechanism or Corrected platform Error Vector (CPEV) is being specified.

The *i_or_m* parameter specifies whether an interrupt or memory variable is used and this parameter is meaningful only for the *param_type* of 2. Interrupt is the only valid choice for the rendezvous function since the idea is to interrupt the non-monarch processor as quickly as possible and correct the error. Either interrupt or memory may be used for the wake up mechanism and this is operating system implementation-dependent.

The *i_or_m_val* parameter specifies the interrupt vector number or the memory address associated with the *i_or_m* parameter. If memory address is used for the wake up mechanism, the memory variable must be aligned on an 8-byte boundary and coherent across the system fabric. The operating system shall not change the physical address of the memory variable specified in the *i_or_m_val* parameter.

For the rendezvous interrupt vector, a value of 0 indicates use of PMI as the interrupt mechanism. The PMI interrupt mechanism shall not be employed by Itanium architecture-based operating systems as either the rendezvous or the wake-up interrupt. For the Corrected Platform Error Vector, a value of 0 de-registers the OS vector with the SAL and the SAL will not send Corrected Platform Error IPIs to the OS.

The PMI interrupt mechanism is needed for legacy operating system support. SAL may return an error status on platforms that do not support legacy operating systems.

The *mca_opt* argument specifies the options that the SAL MCA handler is required to follow during machine check handling. This parameter is valid only when the *param_type* is *rendezvous interrupt.* Following is the format of this argument:

Bit Positions	Length in Bits	Description
0	1	*rz_always* flag.
1	1	*binit_escalate* flag
2-63	61	Reserved, must be zero

If the *rz_always* flag is set to 1, the SAL is expected to rendezvous the system for all detected processor and platform MCA conditions. If this flag is set to zero, then rendezvous is done only when PAL initiates the rendezvous request during an MCA or if SAL decides to do it for certain platform MCA conditions.

During machine check processing, the SAL operates with machine checks masked and hence does not immediately recognize subsequent machine checks. If the operating system wishes to recognize subsequent machine checks in this condition, it will set the *binit_escalate* flag to 1. When the *binit_escalate* flag is set, the SAL shall escalate future MCAs and BERR events to BINIT# using the PAL_PROC_SET_FEATURES procedure. On return from the operating system, the SAL shall restore the original setting.

If the operating system intends to use interrupts for corrected platform events, it shall register the same interrupt vector number that is programmed into the I/O SAPIC redirection table entry for triggering platform-corrected error interrupts. If the operating system intends to

use polling to collect this information, it shall neither register an interrupt vector with the SAL nor program the I/O SAPIC redirection table entry.

Except for the PMI interrupt above, the external interrupt vector value must be in the range of 16 to 255 since these are the acceptable values that can be transferred using SAPIC IPI messages. A high value should be chosen for the rendezvous interrupt vector to facilitate prompt handling of machine checks. Even a higher value (close to 255) may need to be used for the wake up interrupt vector (if not using memory variable mechanism). This is because the operating system is responsible for clearing the IRR bit associated with the wake up interrupt vector by reading the IVR and issuing the EOI to the local SAPIC. If the wake up interrupt bit is not cleared promptly, a later call to the SAL_MC_RENDEZ procedure may return prematurely.

This procedure may be called in virtual or physical mode but when the *i_or_m* parameter specifies a memory address, subsequent calls to the SAL_MC_RENDEZ must be made in the same mode (virtual/physical) as this call.

The *time_out* field defines the rendezvous time out period in milliseconds. This parameter is only applicable to the *param_type* of rendezvous interrupt. If the non-monarch processor does not invoke the SAL_MC_RENDEZ procedure within the time out period, the monarch processor will generate an INIT signal to the non-monarch processor. The time out value must be sufficient to cover situations where other processors may be executing firmware code in local MCA and thus not be capable of servicing external interrupts or INIT. If the *time_out* input parameter is insufficient, the SAL shall return with a status of –2 and the *time_out_min* return argument shall specify the minimum time out interval required by the SAL.

Platform Requirements: None

SAL_PCI_CONFIG_READ

Purpose: This procedure is used to read from the PCI configuration space.

Calling Conventions: Standard. Callable by the operating system in virtual or physical mode.

Arguments:

Argument	Description
func_id	Function ID of SAL_PCI_CONFIG_READ within the list of SAL procedures
address	PCI configuration address: If *address type* = 0 Bits 0..7 – Register address Bits 8..10 – Function number Bits 11..15 – Device number Bits 16..23 – Bus number Bits 24..31 – PCI segment group Bits 32..63 – Reserved (0) If *address type* = 1 Bits 0..7 – Register address Bits 8..11 – Extended Register address Bits 12..14 – Function number Bits 15..19 – Device number Bits 20..27 – Bus number Bits 28..43 – PCI segment group Bits 44..63 – Reserved (0) Address must be naturally aligned with respect to the size of the read.
size	PCI config size (1, 2 or 4 bytes)
address type	The type of PCI configuration address 0 = PCI Compatible Address 1 = Extended Register Address Other values reserved
Reserved	0
Reserved	0
Reserved	0
Reserved	0
Reserved	0

Returns:

Return Value	Description
status	Return status of SAL_PCI_CONFIG_READ procedure
value	Value read from config space.
Reserved	0
Reserved	0

Status:

Status Value	Description
0	Call completed without error
–2	Invalid Argument
–3	Call completed with error
–4	Virtual address not registered

Description: This procedure is a runtime interface used to read from PCI configuration space. The mechanism for accessing PCI configuration space is abstracted by this procedure, thereby allowing host bridges to implement this mechanism in different ways.

A non-zero value in the segment field can be used to access devices on platforms with greater than 256 buses.

Platform Requirements: None

SAL_PCI_CONFIG_WRITE

Purpose: This procedure is used to write to the PCI configuration space.

Calling Conventions: Standard. Callable by the operating system in virtual or physical mode.

Arguments:

Argument	Description
func_id	Function ID of SAL_PCI_CONFIG_WRITE within the list of SAL procedures
address	PCI configuration address: If *address type* = 0 Bits 0..7 – Register address Bits 8..10 – Function number Bits 11..15 – Device number Bits 16..23 – Bus number Bits 24..31 – PCI segment group Bits 32..63 – Reserved (0) If *address type* = 1 Bits 0..7 – Register address Bits 8..11 – Extended Register address Bits 12..14 – Function number Bits 15..19 – Device number Bits 20..27 – Bus number Bits 28..43 – PCI segment group Bits 44..63 – Reserved (0) Address must be naturally aligned with respect to the size of the read.
size	PCI config size (1, 2 or 4 bytes)
value	Value to write to PCI config space
address type	The type of PCI configuration address 0 = PCI Compatible Address 1 = Extended Register Address Other values reserved
Reserved	0
Reserved	0
Reserved	0
Reserved	0

Returns:

Return Value	Description
status	Return status of SAL_PCI_CONFIG_WRITE procedure
Reserved	0
Reserved	0
Reserved	0

Status:

Status Value	Description
0	Call completed without error
–2	Invalid Argument
–3	Call completed with error
–4	Virtual address not registered

Description: This procedure is a runtime interface used to write to PCI configuration space. The mechanism for accessing PCI configuration space is abstracted by this procedure, thereby allowing host bridges to implement this mechanism in different ways. This procedure will guarantee the completion of the write to the caller.

A non-zero value in the segment field can be used to access devices on platforms with greater than 256 buses.

Platform Requirements: None

SAL_REGISTER_PHYSICAL_ADDR

Purpose: Provide a mechanism for software to register the physical addresses of locations needed by SAL

Calling Conventions: Standard. Callable by the operating system in virtual or physical mode.

Arguments:

Argument	Description
func_id	Function ID of SAL_REGISTER_PHYSICAL_ADDR call within the list of SAL procedures
phys_entity	The encoded value of the entity whose physical address is registered 0 = PAL_PROC Other values are reserved
p_addr	64-bit integer value denoting the physical address
Reserved	0
Reserved	0
Reserved	0
Reserved	0
Reserved	0

Returns:

Return Value	Description
status	Return status of SAL_REGISTER_PHYSICAL_ADDR procedure
Reserved	0
Reserved	0
Reserved	0

Status:

Status Value	Description
0	Call completed without error
–2	Invalid Argument
–3	Call completed with error
–4	Virtual address not registered

Description: The *phys_entity* argument specifies the entity whose physical address is being registered with the SAL and the *p_addr* argument provides its physical address.

Platform Requirements: None

SAL_SET_VECTORS

Purpose: Provide a mechanism for software to register software-dependent code locations with SAL. These locations are “handlers” or entrypoints where SAL will pass control for the specified event. The events handled are for the Boot Rendezvous, MCAs, and INIT scenarios.

Calling Conventions: Standard. Callable by the operating system in virtual or physical mode.

Arguments:

Argument	Description
func_id	Function ID of SAL_SET_VECTORS call within the list of SAL procedures
vector_type	Type of event handler: 0 = Machine Check 1 = INIT 2 = BOOT_RENDEZ 3–64 = Reserved other values are implementation-dependent
phys_addr_1	Physical address of the event handler. This field must be a 16-byte aligned address.
gp_1	Global pointer (GP) of the event handler.
length_cs_1	Size of the event handler procedure and its checksum information
phys_addr_2	Physical address of the event handler. This field must be a 16-byte aligned address.
gp_2	Global pointer (GP) of the event handler.
length_cs_2	Size of the event handler procedure and its checksum information

Returns:

Return Value	Description
status	Return status of SAL_SET_VECTORS procedure
Reserved	0
Reserved	0
Reserved	0

Status:

Status Value	Description
0	Call completed without error
–2	Invalid Argument
–3	Call completed with error
–4	Virtual address not registered

Description: This procedure enables the operating system (and diagnostic software) to inform firmware whether it is ready to handle the Machine Check, BOOT_RENDEZ, and INIT events and precisely where to vector for each case. Since all three events result in having processor execution being controlled by firmware, firmware requires these software addresses of the operating system or diagnostics in order to pass control. The operating system registers the *physical* address where the specific handler resides. SAL uses these addresses to vector to on occurrence of the event. The parameters specified by the OS are applicable to all the processors within the system.

For the INIT event in an MP configuration, separate arguments must be provided for the first processor (monarch) to enter the SAL_INIT layer and subsequent processors (non-monarchs). The *phys_addr_1, gp_1* and *length_cs_1* arguments specify the entrypoint, GP-value and the length details respectively of the OS_INIT procedure for the monarch and the *phys_addr_2, gp_2* and *length_cs_2* arguments respectively specify the entrypoint, GP-value and the length details of the OS_INIT procedure for the non-monarch processors. By having different entrypoints for the monarch and non-monarch processors, the operating system can easily put the non-monarch processors into a wait loop. It is permissible to have the same arguments for the monarch and non-monarch processors. In

this case, the operating system will need to perform the monarch selection on entry into the OS_INIT procedure.

The value in the *phys_addr_n* argument must be 16-byte aligned. The *phys_addr_n* argument may be checked as to whether it points into legal memory space (as opposed to I/O space or firmware space). Specifying a value of 0 in the *phys_addr_n* argument invalidates the event handler procedure. For the INIT event in an MP configuration, the values in the *phys_addr_1* and the *phys_addr_2* arguments must both be zeroes or non-zeroes, that is, it is not possible to invalidate only one of the two entrypoints. The *phys_addr_2, gp_2 and length_cs_2* arguments for the OS_MCA and the OS_BOOT_RENDEZ vector_type are reserved.

The *gp_n* field has the physical address of the GP for the event handler to be called by SAL.

The *length_cs_n* argument has the format shown below:

Bit Positions	Length in Bits	Description
0-31	32	Length of the operating system procedure in bytes (this field must be a multiple of 16).
32	1	0 = Checksum information not provided by the operating system. 1 = Checksum information provided by the operating system in bits 40-47.
33-39	7	Reserved
40-47	8	The modulo checksum of the operating system procedure code area. All bytes including the checksum byte must add up to zero.
48-63	16	Reserved.

The operating system has the option of registering the length and checksum of the operating system procedure (or at least the first level OS_MCA, OS_INIT, OS_BOOT_RENDEZ procedure). If *length_cs_n*.Bit32 is set, SAL saves the operating system provided checksum for the procedure, and before invoking that procedure, will authenticate the operating system code by verifying its checksum. If *length_cs_n*.Bit32 is not set, SAL will ignore the remaining *length_cs_n* bits and will not authenticate the checksum of the registered procedure before invoking it.

Platform Requirements: None

SAL_UPDATE_PAL

Purpose: This procedure is used to update the contents of the PAL block in the non-volatile storage device.

Calling Conventions: Standard. Callable by the operating system in virtual or physical mode.

Arguments:

Argument	Description
func_id	Function ID of the SAL_UPDATE_PAL within the list of SAL procedures
param_buf	Pointer to a buffer containing information about the new firmware block(s).
scratch_buf	Pointer to a scratch buffer.
scratch_buf_size	Unsigned 64-bit integer value for the size of the scratch buffer in bytes
Reserved	0
Reserved	0
Reserved	0
Reserved	0

Returns:

Return Value	Description
status	Return status of SAL_UPDATE_PAL procedure
error_code	Additional information pertaining to the error
scrbuf_size_req	Size of the scratch buffer needed
Reserved	0

Status:

Status Value	Description
0	Call completed without error
2	Effect a warm boot of the system to complete the update.
–2	Invalid Argument
–3	Call completed with error. See *error_code* for details
–4	Virtual address not registered
–9	Insufficient scratch buffer provided

Description: This procedure updates the contents of firmware blocks (for example, PAL_B) in the non-volatile storage device and revises the FIT entries pertaining to the firmware blocks. If checksum is implemented for the FIT table, this procedure will also revise the same. This procedure is capable of selecting the appropriate location in the storage device for the firmware components. In some flash ROM architectures, updates may not be possible until the following INIT. This scenario is described later.

Before performing update of PAL, this procedure will utilize resources within the processor and/or PAL to authenticate the contents of the new version of PAL provided by the caller. If the authentication is unsuccessful, the current PAL contents will be left intact.

The *param_buf* points to a 16-byte aligned data structure in memory with a length of 32 bytes that describes the new firmware. This information is organized in the form of a linked list with each element describing one firmware component. This procedure will update all the specified firmware components as well as their FIT entries if successful, and none of the firmware components if errors are encountered. The following table shows the format of each element of the data structure. Refer to Section 2.5, “Firmware Interface Table” for explanation of fields within the FIT.

Offset	Length	Description
0	8	64-bit pointer to the next element (0 if none present)
8	8	64-bit memory address of the *update_data_block* containing new firmware contents
16	1	Checksum flag: 0= Do not store checksum of this component in its FIT entry 1=Calculate and store checksum of this component in its FIT entry
17	15	Reserved

The *update_data_block* consists of a header of 64 bytes followed by the code for the firmware component. The following table shows the contents of the 64 byte header.

Offset	Length	Description
0	4	Size of the firmware component in bytes including the header (this field must be a multiple of 16)
4	4	Date of the firmware component in mmddyyyy format: month, day, year (for example, 07/18/99 stored as 0x07181999)
8	2	Version number of the firmware component to be stored in its FIT entry
10	1	Type of firmware component (Refer to Table 2-2 on page 22) 1 = PAL_B; 0x0F = PAL_A (also generic PAL_A); 0x0E = Processor-specific Pal_A
11	5	Reserved
16	8	Firmware Vendor ID
24	40	Reserved

This procedure will locate the PAL_B block on a 32K byte aligned boundary on the storage device.

If the scratch buffer size specified in the *scratch_buf_size* field is insufficient, the call will fail with a *status* of –9 and the *scrbuf_size_req* return parameter will specify the size of the scratch buffer required.

SAL reads the CPU identification registers on all the processors in the system and maintains the processor stepping information. If a split PAL architecture is supported (generic PAL_A, processor-specific PAL_A), then the SAL also maintains processor generation information. If the PAL_B component is being updated, SAL will ensure that the version number of the new PAL_B in the *update_data_block* is compatible with all the processors in the system else return an error *status.* If the processor-specific PAL_A component is being updated, SAL will ensure that the version number of the new processor-specific PAL_A in the *update_data_block* is compatible with all the processors in the system else return an error *status.*

The *error_code* return parameter provides additional information on the failure when the *status* field contains a value of –3. Following are the definitions for the *error_code* field.

Error Code	Description
–1	Version number of supplied PAL firmware is not suitable for one or more processors in the system
–2	Supplied version of PAL failed the authentication test
–3	Invalid firmware component type
–4	PAL_A firmware not erasable
–5 to –9	Reserved
–10	Write failure – inability to write to storage device
–11	Erase failure – inability to erase the storage device
–12	Read failure – inability to read the storage device

In some firmware architectures (for example, flash), writes to a chip or component containing firmware would prevent the same chip being available for code execution. For this reason, if the PAL or SAL firmware

code for handling machine checks were located on the chip being revised, machine checks must be masked on all the processors to avoid possible instruction fetch accesses to the firmware address space. In an MP environment, the operating system must rendezvous all the other processors on the node whose firmware is being updated. At the end of the firmware update, the operating system must invoke the PAL_MC_ERROR_INFO procedure to ascertain whether any machine checks occurred while they were masked and take corrective actions. The operating system must then wake up the rendezvoused processors and re-enable machine checks. In a multi-node system with multiple copies of firmware, it may be possible to redirect interrupts to nodes other than the one being updated.

In some flash architectures, writes to firmware address space may be prevented by the flash hardware except immediately following a Reset or INIT. The operating system may call this procedure in virtual mode but it is required to fix the pages containing the new firmware contents in memory, that is, the operating system must not change the contents of the corresponding physical pages until the firmware update is complete. SAL will be aware of flash architecture restrictions and will perform the usual authentication steps. If the authentication is successful, SAL will accumulate the physical addresses of the new firmware contents by executing the TPA instruction. (There may be several non-contiguous physical pages if the operating system had called this procedure in virtual mode.) SAL will then return to the operating system a status value of 1 requesting a warm reboot. When SAL regains control following the warm reboot, it will conduct the authentication steps again and, if successful, update the contents of firmware.

The firmware update is effective on the next reboot. However, after a successful update, firmware contents in the non-volatile storage device and memory will be inconsistent. The copy in ROM (new code) will be utilized by the machine check and INIT events while the copy in memory (old code) will be utilized by the operating system. The operating system may solve this problem either by rebooting the system following a firmware update, or by updating the memory copy of PAL procedures by invoking the PAL_COPY_PAL procedure.

If the operating system decides to update the memory copy of PAL procedures, there are additional considerations in an MP environment:

1. While the runtime copy of PAL is being revised (during execution of the PAL_COPY_PAL procedure), all the processors in the system must be prevented from executing PAL procedures in memory.
2. The monarch processor, after invoking the PAL_COPY_PAL procedure, must make the local instruction caches coherent with the data caches by invoking the SAL_CACHE_FLUSH procedure (with the *i_or_d* parameter value of 4).
3. The non-monarch processors on being woken up by the monarch processor must invoke the PAL_COPY_PAL procedure to register the new PAL entrypoints for PAL_PMI and PAL_FP. The non-monarch processors must do a SRLZ.I instruction to ensure that modifications to instruction prefetches are observed.
4. If the *physical address* of the PAL_PROC procedure changes, the operating system must register the new address with SAL by invoking the SAL_REGISTER_PHYSICAL_ADDR procedure.

Platform Requirements: Platform must provide non-volatile storage space to save firmware components.

SAL_PHYSICAL_ID_INFO

Purpose: Returns information on the physical processor die mapping in the platform.

Calling Conventions: Standard. Callable by the operating system in physical or virtual mode.

Arguments:

Argument	Description
func_id	Function ID of the SAL_PHYSICAL_ID_INFO within the list of SAL procedures
Reserved	0
Reserved	0
Reserved	0
Reserved	0
Reserved	0
Reserved	0
Reserved	0

Returns:

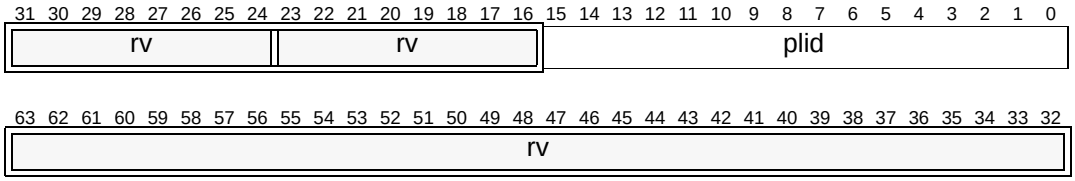
Return Value	Description
status	Return status of SAL_PHYSICAL_ID_INFO procedure
plat_log_info	The format of *plat_log_info* is shown in Figure 9-1.
Reserved	0
Reserved	0

Status:

Status Value	Description
0	Call completed without error
–1	Unimplemented procedure
–2	Invalid Argument
–3	Call completed with error.
–4	Virtual address not registered

Description: This API can be used in conjunction with the PAL_LOGICAL_TO_PHYSICAL API to uniquely identify a processor die/ package within a system that contains multiple host bus controllers. This procedure returns the value *plid*, which may be based on a platforms host bus number, node number, or cluster number. The values of *ppid* (returned by the PAL_LOGICAL_TO_PHYSICAL API) and *plid* combined uniquely identify each physical processor die in the platform. *Plid* values may not be contiguous between processor die nor within the platform.

Figure 9-1. Layout of *plat_log_info* Return Value



- *plid* - Platform ID. A value provided by the platform that may be based on host bus, node or cluster IDs. This value, along with the *ppid* (returned by the PAL_LOGICAL_TO_PHYSICAL API), must be unique for each physical processor die in the platform.
- *rv* - Reserved

Platform Requirements: Itanium architecture-based platforms must provide mechanisms to obtain the value used for the platform ID field. The platform ID along with the processor physical die ID must be a unique value across the platform.

§

A Glossary

ACPI
Advanced Configuration and Power

AP
Application Processor. One of the processors not responsible for system initialization.

API
Application Programming Interface

Bank
The memory modules on a card are organized into banks for better performance. The bank number identifies a bank on a memory card.

BIOS
Basic Input/Output System. A collection of routines that includes Power On Self-test (POST), system configuration and a software layer between the operating system and hardware. BIOS is written in IA-32 instruction set.

Boot Block Support
A hardware and/or software implementation that permits the end user to recover PAL/SAL layers of software into the flash part after the previous flash programming attempt was accidentally aborted.

BSP
Bootstrap Processor. The processor responsible for system initialization.

BSP
Backing Store Pointer (AR.BSP)

Card
The card number identifies the specific memory card attached to a memory controller. One or more memory cards may be attached to a memory controller. Each card consists of a number of memory modules organized in banks.

CMC
Corrected Machine Check

Cold Boot vs. Warm Boot
Cold Boot refers to a hardware/software event that sets all circuitry, including all processors, system components, add-in cards and control logic, to an initial state. Initial power-on of a system triggers a cold boot. Warm Boot refers to an event that sets some, but not all the circuitry of any or all of the processors and system components to an initial state. This capability is important to minimize system boot time. Warm boots can skip extensive memory testing, skip initialization of devices whose configuration registers are preserved across resets, and so on. Warm boots are not required to preserve any register or memory state. INIT or MCA events can trigger a warm boot.

Cold Reset vs. Hard Reset
Cold Reset refers to a hardware signal that sets all circuitry, including all processors, buses, system components, add-in cards and control logic, to an initial state. Hard Reset is triggered by a similar hardware signal. Hard Reset differs from Cold Reset in that some sticky error flags in some system components may not be cleared, thereby allowing determination of the

cause of the Reset. Both Cold Reset and Hard Reset signals operate without regard to cycle boundaries and are typically asserted by the RESET pin. Both Cold Reset and Hard Reset signals will include the functionality of the Cold Boot event.

Corrected Platform Error Interrupt (CPEI)
Interrupt generated by the platform following a hardware-corrected error. The interrupt vector is set by the operating system (e.g. in the vector field of an I/O SAPIC redirection table entry).

CPE
Corrected Platform Errors are the errors originating due to platform detected errors.

CPEV
Corrected Platform Error Interrupt Vector

Device Number
Each memory module consists of a number of DRAM devices. The device number identifies a specific device (h/w component or chip) on a module.

EFI
Extensible Firmware Interface. Firmware that provides a legacy free API interface to the operating system.

EOI
End of Interrupt

Error Categories
Corrected Error
All errors of this type are either corrected by the processor/platform hardware/firmware. This severity is for logging purposes only. There is no architectural damage to the detecting and reporting functions. Corrected errors require no operating system intervention to correct the error.

Fatal Error
An uncorrected error occurred which has corrupted state, and the state information may not be known. These type of errors cannot be corrected by the hardware, firmware, or the operating system. The integrity of the system, including the IO devices is not guaranteed and may require IO device initialization and a system reboot to continue. Fatal errors may or may not have been contained within the processor or memory hierarchy. If the error is not contained, it must be reported as fatal.

Recoverable Error
An uncorrected error occurred which had corrupted state, and the state information is known. Recoverable errors cannot be corrected by either the hardware or firmware. This type of errors requires operating system analysis and a corrective action to recover. System operation/state may be impacted.

FRU
Field Replaceable Unit

FT
Fault Tolerant

GP
Global Data Pointer. Every procedure that references statically-allocated data or calls another procedure requires a pointer to its data segment in the GP register so that it can access its static data and its linkage tables.

GUID
A 16 byte Globally Unique Identifier/Universally Unique Identifier representing an entity that needs to be uniquely identified.

Hardware-protected Flash Region
This term refers to a part of the flash storage that is hardware-protected against accidental erasure. Usually, this region is programmed by the OEM only. The hardware protection can either be on-chip and/or platform supported hardware.

IA-32 Architecture
The 32-bit and 16-bit Intel® architecture as described in the *Intel® Itanium® Architecture Software Developer's Manual.*

Itanium Architecture-based Operating System
An operating system written in the Itanium instruction set that can run Itanium architecture-based applications and, optionally, IA-32 applications.

INTA
Interrupt Acknowledge

IPI
Interprocessor interrupt signaling using the local SAPIC within the processor.

IPL
Initial Program Load

ISA
Instruction Set Architecture

IVT
Interrupt Vector Table

Logical Processor / Processor
A processor may provide the ability to support two or more logical processors, which are capable of executing independent instruction streams. The amount of processor hardware resources that may be shared between logical processors is implementation dependent. SAL should treat logical processors as if they are separate processing elements and dedicate resources accordingly.

MBR
Master Boot Record

MC_rendezvous Interrupt
An external interrupt vector provided to SAL by the Itanium architecture-based operating system for interrupting the operating system running on the APs.

MCA
Machine Check Abort

Minimal State Save Area
Area registered by SAL with PAL for saving minimal processor state during machine check and INIT processing. This area must be aligned on a 512-byte boundary and must be in uncacheable memory. See the *PAL EAS* for details.

Module or Rank
A module consists of a number of DRAM devices on a PCB board, which plugs into a socket. DIMM, RIMM are examples of memory modules. Module number identifies a module on a memory card (specifically, within a bank on the memory card). On smaller systems, the rank/module might match

the DIMM slot number. On larger systems, a particular DIMM might not be able to be called out and the module/rank number is the lowest FRU.

Monarch Processor
The processor selected by SAL to accumulate all the platform error logs and continue with the machine check processing, when multiple processors experience machine checks simultaneously.

MP
Multiprocessor

MP-Safe Procedure
A procedure that can be invoked concurrently by multiple processors. The caller is not required to enforce single-threaded access. If necessary, SAL performs synchronization between threads.

MPS
Multiprocessor Specification

Node
A node consists of processors, memory and, in some cases, I/O devices. A system may contain multiple nodes.

NTFS
Windows* NT File System

NVRAM
Non-volatile Random Access Memory

OS
Operating System

PAL
Processor Abstraction Layer. Firmware that abstracts processor implementation-specific features.

Plabel
Procedure label, a reference or pointer to a function. A plabel takes the form of a pointer to a special descriptor (a plabel descriptor) that uniquely identifies the function. The plabel descriptor contains the address of the function's actual entrypoint as well as its linkage table pointer.

PMI
Platform Management Interrupt

Re-entrant Procedure
A procedure that may be invoked multiple times concurrently from the same processor. That is, the procedure may be interrupted by an MCA and invoked again by OS_MCA during MCA processing. Note that an MCA may be promoted to Fatal severity if it occurs while executing a SAL procedure.

Row, Column
Memory cells (a cell may hold one more Bits of data) on a DRAM is organized as an array indexed by rows and columns. Row address and column address together uniquely identify a cell.

SAL
System Abstraction Layer. Firmware that abstracts system implementation differences.

SAL_REV
The revision number of the SAL specification supported by the SAL implementation. This information contains two one-byte fields for Major and Minor revision numbers and the same are represented in binary coded

decimal (BCD) format. For example, if this variable contains 02h, 06h, the SAL revision is 2.6.

SAPIC
Streamlined Advanced Programmable Interrupt Controller. The code name for the high performance interrupt architecture for the Itanium architecture. The **Local SAPIC** resides within the processor and accepts interrupts sent on the system bus. The **I/O SAPIC** resides on the I/O subsystem and provides the interrupt input pins on which I/O devices inject interrupts into the system.

Sector
This term refers to a logical block of 512 bytes.

SP
Memory Stack Pointer

TLB
Translation Lookaside Buffer

TSS
Task State Segment

USB
Universal Serial Bus

VHPT
Virtual Hash Page Table

Wake Up Interrupt
Interrupt sent by the operating system to wake up the APs from the SAL_MC_RENDEZ spin loop. This interrupt vector is registered by an Itanium architecture-based operating system with the SAL.

WBL
Write-back with Limited Speculation

§

B Error Record Structures

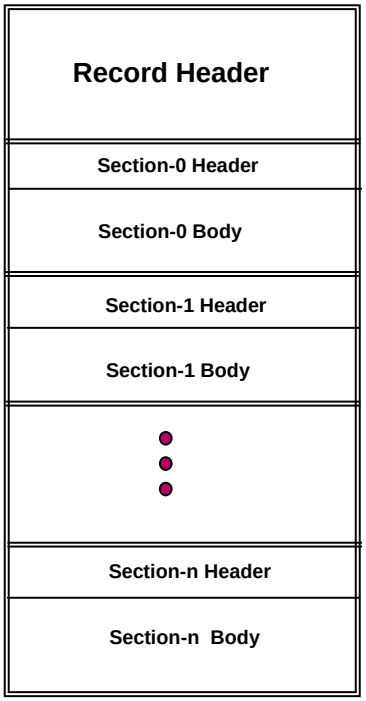
B.1 Overview

The goals of the Error Record structures is to keep it generic and flexible enough to be extensible and to abstract processor or platform implementation dependencies from the operating system layers, at the same time providing as much error information as possible to the operating system for error handling purposes.

B.2 Error Record Structure

The error record structure consist of many different components called sections. Each error record captures error information for one error event consisting of multiple sections. The size of the error record structure is as indicated by RECORD_LEN and is dynamically set based on the total size of all the section headers and section bodies combined. Each record must be 8-byte aligned and the size must be a multiple of 8 bytes.

An error record consists of a generic header followed by a list of sections with actual error information for the event. Each section relates to a particular error device (e.g. processor, platform memory, platform PCI Bus, platform ISA Bus, and so on), having a section header followed by section body.



B.2.1 Record Header

The format of the header for both the platform and processor error record is as shown below. For machine check events and INIT events, the ERR_SEVERITY information reflects the worst case error reported in the processor and platform sections. For these events, ERROR_SEVERITY may be corrected, recoverable or fatal. For corrected processor and platform errors (CMCs and CPEs), the error may be reported as corrected or recoverable. For CMC/CPE events, if the error severity is recoverable/ uncorrected, the OS may choose to take action based on the error. However, recovery from such CMCs and CPEs is not necessary for continued operation of the system.

The record header REVISION field consists of two fields: a major value and a minor value. The record header revision field denotes the version of the entire error record, not just the record header. (The section header revision field remains in the section header structure for backwards compatibility, but should not be used by operating systems.)

Changes to the major revision indicate major changes to the error record format. An increase in the major revision is required when changes to the headers and sections may not be compatible with software developed based on previous major revisions. For example, adding new fields, or changing the functionality of existing fields (excluding reserved bits) requires an increase in the major revision.

An increase in the minor revision indicates that changes to the headers and sections are compatible with software that use earlier revisions. This includes, but is not limited to, errata fixes, clarifications, the use of reserved fields, and the addition of new sections identified with new GUIDs. New error record sections and major updates to sections are introduced in a compatible way using the globally unique ID (GUID) of the error section header and incrementing the minor number of the record header revision field. See Section B.2.2 for details.

Offset	Length	Field	Description
0	8 bytes	RECORD_ID	An ID that is unique (system) wide that distinguishes between resets, and is increasing for MCA/INIT, CMC, and CPE events respectively[1]
8	2 bytes	REVISION	2-byte Major and Minor revision number of the Record in BCD format: Byte 0 - Minor (0x07) Byte 1 - Major (0x00)
10	1 byte	ERR_SEVERITY	This encoded field indicates error severity. See glossary section for details on the definition: 0 – Recoverable 1 – Fatal 2 – Corrected Others – Reserved
11	1byte	VALIDATION_BITS	Bit 0 = If 1, the OEM_PLATFORM_ID field below contains valid information. Bit 1 = If 1, the TIME_STAMP field below contains invalid information. Bits 2-7 – Reserved, must be zero
12	4 bytes	RECORD_LEN	Length of this error record in bytes, including the header.
16	8 bytes	TIME_STAMP	Timestamp recorded when MCA, INIT or CMC occurred. This contains the local time in BCD format: Byte 0 – Seconds Byte 1 – Minutes Byte 2 – Hours Byte 3 – Reserved Byte 4 – Day Byte 5 – Month Byte 6 – Year Byte 7 – Century
24	16 bytes	OEM_PLATFORM_ID	A unique identifier of the OEM platform.

Notes:

1. For example, an implementation that generates a RECORD_ID in the following manner would meet these requirements: RECORD_ID[63:24] contains a timestamp value that is obtained at reset, RECORD_ID[23:0] is divided into three fields (MCA/INIT, CMC, CPE), which are incremented for each type of error event respectively.

Refer to the *Intel® Itanium® Architecture Software Developer's Manual* for explanation of fields not described in this document.

B.2.2 Section Header

The Device specific error section follows the header. For processor errors, this field will contain an area that is architected for all Itanium architecture processors. For platform errors, this section will contain information specific to the platform devices.

A unique GUID is associated with each section for identification of the error device type (ex: processor, platform memory, platform PCI bus, and so on). The introduction of a new section or a major change to a section requires a new GUID and a corresponding increment of the minor number in the record header revision field.

Minor changes to the system abstraction layer (SAL) section header and the sections are also tracked by incrementing the minor number of the record header revision field. (The section header revision field remains in the section header structure for backwards compatibility, but should not be used by operating systems.)

The format of the section header for all error devices is as shown below:

Offset	Length	Field	Description
0	16 bytes	GUID	Unique 16-byte GUID for the error device. Refer to Table B-1 for the format.
16	2 bytes	REVISION	2-byte Major and Minor revision number of the Section in BCD format. The revision values are fixed as follows: Byte0 – Minor (03) Byte1 – Major (00) The section header revision field remains in the section header structure for backwards compatibility, but should not be used by operating systems.
18	1 byte	ERROR_RECOVERY_INFO	Bit 7 = If set, the remaining bits in this field and the corresponding error section contains information about the error(s) reported. Bit 6 = Reserved, must be 0 Bit 5 = Latent Error. If set, the reported error is considered as a not yet consumed latent error and could result in a more severe error when consumed. System software can take immediate proactive action on this class of error. This bit setting is qualified with bits 4-0. The severity associated with this flag is either corrected or uncorrected. It is to be noted that any error records returned in response to corrected event notification shall always report the severity as corrected. Bit 4 = Resource not accessible. If set, the resource could not be queried for error information due to conflicts with other system software or resources. Some fields of the section will be invalid. Bit 3 =Error threshold exceeded. If set, OS may choose to discontinue use of this resource. Bit 2 = Reset. If set, the component must be re-initialized or re-enabled by the operating system prior to use. Bit 1 = Containment Warning. If set, the error was not contained within the processor or memory hierarchy and the error may have propagated to persistent storage or network. Bit 0 = If set, the error has been corrected. If not set, the error was not corrected.
19	1 byte	RESERVED	Reserved.
20	4 bytes	SECTION_LEN	Length of this error device section in bytes, including the header.

The GUID structure is as follows:

Table B-1. GUID Format

Offset	Length	Field	Description
0	4 bytes	DATA1	Data1
4	2 bytes	DATA2	Data2
6	2 bytes	DATA3	Data3
8	8 bytes	DATA4	Data4

The first four bytes are treated as an integer, as are the next two two-byte entities. The last 8 bytes are treated as a byte stream. For example, the following GUID would follow the byte order in memory given in Table B-2:

GUID: `{0xdeadbeef, 0x1234, 0x5678, {0xde, 0xad, 0x12, 0x34, 0x56, 0x65, 0x43, 0x21}}`

Table B-2. GUID Ordering in Memory

Bytes															
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
ef	be	ad	de	34	12	78	56	de	ad	12	34	56	65	43	21

SAL may examine several platform hardware resources to collect information pertaining to the error and provide such information in various sections. *Not all sections may be present in each record but the SAL shall provide all the information significant for logging, identification of the errant component and recovery.* The section error information fields will have associated validation bit(s), as part of the section body.

Multiple sections with the same GUID may be present within a single error record. In this situation, the ordering of the sections does not imply the chronological sequence of the errors. The first error among the sections, if known to firmware, shall be indicated by setting the *First Error* bit (see Table B-5) in the error status field within the section.

The ERROR_RECOVERY_INFO field in some sections may indicate that the error has already been corrected. It is acceptable to provide corrected error information for some platform components as part of the MCA record, but the SAL must not provide uncorrected MCA information in response to the request for CMC or CPE errors.

If the Containment Warning bit is set in the ERROR_RECOVERY_INFO field, the SAL firmware may set the ERR_SEVERITY field in the Record Header (Section B.2.1) as “fatal”. Some operating systems or device drivers having a complete chronology of accesses to the platform component and knowledge of recovery capabilities within the device, may effect a recovery despite such a status.

The OS interpretation for the ERR_RECOVERY_INFO field is given in Table B-3.

Table B-3. Error Section Error_Recovery_Info Field Definition

Bit 7	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	OS Recovery Action
0	X	X	X	X	X	X	Error Recovery Field is not valid: Specialized MCA handlers with knowledge of the device reported by the section are required to recover from this error. OS Recovery is not possible, if specialized handlers are not available.
1	1	X	X	X	X	X	Latent Error: If set, the reported error is considered as not yet consumed latent error and could result in a more severe error when consumed. System software can take immediate proactive action for this error. This error type is a sub-category of uncorrected or corrected error. This bit setting is qualified with bits 4-0. Poisoned error in memory is a good example of errors reported with this flag set. This flag will also be set when a processor error is reported with its poisoned flag set.
1	X	1	X	X	X	X	Resource not accessible: Firmware must not return this status when reporting corrected and recoverable MCA.
1	X	X	X	X	1	X	Containment Warning: Corrupt data propagated to persistent storage or network. Generic MCA Handlers will treat this as a fatal error and reboot. Some operating systems or device drivers having a complete chronology of accesses to the platform component and knowledge of recovery capabilities within the device, may effect a recovery despite such a status.
1	X	0	0	0	0	0	Uncorrected error: Error recovery is necessary. Generic OS MCA handlers should be able to handle uncorrected Processor and Memory sections. For other platform uncorrected sections, platform OEMs must supply extended MCA handlers to recover from those errors.
1	X	0	0	1	0	0	Component Reset: Uncorrected error. Error recovery is necessary. The component must be re-initialized prior to use.
1	X	0	0	0	0	1	Corrected error: Only logging and reporting is required. OS recovery actions are not needed for the error in this section.
1	X	0	1	0	0	1	Corrected error: Error threshold exceeded. No immediate OS recovery actions are required for the error in this section. The resource in error should be deallocated.

B.2.3 Processor Errors

B.2.3.1 Processor Machine Check Errors

Refer to the *Intel® Itanium® Architecture Software Developer's Manual* for explanation of fields.

Offset	Length	Field	Description
0	16 bytes	GUID	{0xe429faf1, 0x3cb7, 0x11d4, {0xbc, 0xa7, 0x0, 0x80, 0xc7, 0x3c, 0x88, 0x81}}
16-23	8 bytes		See Section B.2.2 for details.

```
PROCESSOR_SPECIFIC_ERROR_RECORD SECTION BODY STRUCTURE

{
    VALIDATION_BITS[1]                              8 bytes
        PROC_ERROR_MAP_VALID_BIT                    Bit 0
        PROC_STATE_PARAMETER_VALID_BIT              Bit 1
        PROC_CR_LID_VALID_BIT                       Bit 2
        PROC_STATIC_STRUCT_VALID_BIT                Bit 3
        CACHE_CHECK_NUM                             Bit 4-7 (Cache errors 0 to 15)
        TLB_CHECK_NUM                               Bit 8-11 (TLB errors 0 to 15)
        BUS_CHECK_NUM                               Bit 12-15(BUS errors 0 to 15)
        REG_FILE_CHECK_NUM                          Bit 16-19 (REG errors 0 to 15)
        MS_CHECK_NUM                                Bit 20-23 (MS errors 0 to 15)
        CPUID_INFO_VALID_BIT                        Bit 24
        VARIABLE_LENGTH_DATA_OFFSET_VALID_BIT       Bit 25[2]
        PROC_DYNAMIC_STRUCT_VALID_BIT               Bit 26
        PROC_OEM_DATA_STRUCT_VALID_BIT              Bit 27
        RESERVED                                    Bits 28-63
    PROC_ERROR_MAP                                  8 bytes
    PROC_STATE_PARAMETER                            8 bytes
    PROC_CR_LID                                     8 bytes
    struct {                                        Nx48 max. bytes (cache errors 0 to 15)
        MOD_ERROR_INFO_STRUCT                       48 bytes each
```

1. The amount of information reported by SAL is implementation dependent. The validity of each fixed-length field is indicated by either a VALID_BIT or a NUM field. Data areas corresponding to invalid fixed-length fields will be padded. For CACHE, TLB, BUS, REG, and MS fields, the NUM field indicates the number of MOD_ERROR_INFO_STRUCTs for each category, ranging from 0-15; if the NUM field is zero, then the data area corresponding to that category will be absent. If their associated VALID_BIT is not set, variable-length fields are not padded and will not be present.
2. The VARIABLE_LENGTH_DATA_OFFSET must be valid if either (or both) PROC_DYNAMIC_STRUCT or PROC_OEM_DATA_STRUCT are valid.

} CACHE_ERROR_STRUCT[CACHE_CHECK_NUM]	
struct {	Nx48 max. bytes (TLB errors 0 to 15)
MOD_ERROR_INFO_STRUCT	48 bytes each
} TLB_ERROR_STRUCT[TLB_CHECK_NUM]	
struct {	Nx48 max. bytes (BUS errors 0 to 15)
MOD_ERROR_INFO_STRUCT	48 bytes each
} BUS_ERROR_STRUCT[BUS_CHECK_NUM]	
struct {	Nx48 max. bytes (Reg. errors 0 to 15)
MOD_ERROR_INFO_STRUCT	48 bytes each
} REG_FILE_CHECK_INFO[REG_FILE_CHECK_NUM]	
struct {	Nx48 max. bytes (MS errors 0 to 15)
MOD_ERROR_INFO_STRUCT	48 bytes each
} MS_CHECK_INFO[MS_CHECK_NUM]	
struct {	48 bytes
CPUID_INFO	40 bytes (CPUID registers 0 to 4)
RESERVED	8 bytes
} CPUID_INFO_STRUCT	
struct {	PROC_STATIC_STRUCT
VALID_FIELD_BITS[1]	8 bytes
MINSTATE_VALID_BIT	Bit 0
BR_VALID_BIT	Bit 1
CR_VALID_BIT	Bit 2
AR_VALID_BIT	Bit 3
RR_VALID_BIT	Bit 4
FR_VALID_BIT	Bit 5
RESERVED	Bit 6-63
Minimal State Save Info Structure[2]	1024 bytes
BRs 0-7	64 bytes
CRs 0-127	1024 bytes[3,4]
ARs 0-127	1024 bytes[3,4]

1. Data areas corresponding to invalid fixed-length fields will be padded.
2. This field is used to return the architected 1 Kbyte portion of the min-state save area. For more information about the contents of the structure, please refer to the *Intel® Itanium® Architecture Software Developer's Manual*.
3. The number of Control and Application registers on a processor is implementation dependent.
4. Some Application and Control registers (e.g. CR.IVR) are volatile and cannot be read without side effects. This information is returned by the PAL_REGISTER_INFO procedure. SAL shall not read and store such volatile registers in this data structure.

```
            RRs 0-7                                 64 bytes

            FRs 0-127                               2048 bytes

        } PROC_STATIC_STRUCT

        VARIABLE_LENGTH_DATA_OFFSET[1]              4 bytes

        ; This field provides the parser with an offset from the beginning of the
        ; error section to the variable-length data, which begins with the
        ; PROC_DYNAMIC_STRUCT.

        ; In the future, additional fixed-length sections may be added to this error
        ; section and would be inserted after this field and before the
        ; variable-length fields.

        ; To provide forwards compability with future error section revisions,
        ; error record parsers must dynamically read the
        ; VARIABLE_LENGTH_DATA_OFFSET value to determine where to start parsing
        ; the PROC_DYNAMIC_STRUCT, PROC_OEM_DATA_STRUCT, and future other
        ; variable length data.

        Reserved                                    4 bytes

        struct {                                    PROC_DYNAMIC_STRUCT[2]

            PROC_DYNAMIC_STATE_LENGTH               2 bytes

            ; This is the size of the returned PAL_MC_DYNAMIC_STATE data, which
            ; is reported in the Processor State Parameter dsize argument

            Reserved                                6 bytes

            ; Length of this structure is 8+M bytes
            ; The value of 8+M must align to an 8-byte boundary

            PROC_DYNAMIC_STATE                      M bytes

        } PROC_DYNAMIC_STRUCT

        struct {

            PROC_OEM_DATA_STRUCT                    N bytes[2]

            ; OEM specific data of variable length. See Table B-4 for the format of
            ; this structure.
            ; N = 0 if the PROC_OEM_DATA_STRUCT_VALID_BIT bit is not set.

        } PROC_OEM_DATA_STRUCT

    }


    The MOD_ERROR_INFO_STRUCT structure is defined as below:

    struct{                                         48 bytes[3](Mod)

           VALID_FIELD_BITS                         8 bytes

               CHECK_INFO_VALID_BIT                 Bit 0

               REQUESTOR_IDENTIFIER_VALID_BIT       Bit 1
```

1. Data areas corresponding to invalid fixed-length fields will be padded.
2. If their associated VALID_BIT is not set, variable-length fields are not padded and will not be present.
3. The size of this structure will always be 48 bytes, with invalid fields being padded with null values.

```
        RESPONDER_IDENTIFIER_VALID_BIT      Bit 2
        TARGET_IDENTIFIER_VALID_BIT         Bit 3
        PRECISE_IP_VALID_BIT                Bit 4
        RESERVED_VALID_BIT                  Bit 5-63
    MOD_CHECK_INFO                          8 bytes
    MOD_REQUESTOR_IDENTIFIER                8 bytes
    MOD_RESPONDER_IDENTIFIER                8 bytes
    MOD_TARGET_IDENTIFIER                   8 bytes
    MOD_PRECISE_IP                          8 bytes
} MOD_ERROR_INFO_STRUCT[1]
```

1. The MOD structure is common across CACHE, TLB, BUS, REGISTER_FILE and Microarchitectural structure error records.

B.2.3.2 Deconfigured Processor Machine Check Errors

When called with type = "deconfigured processor," SAL_GET_STATE_INFO can use the deconfigured processor device error info section to report machine check error information for processors not available to the OS.

Offset	Length	Field	Description
0	16 bytes	GUID	{ 0xde507947, 0x7720, 0x47c2 { 0x93, 0x11, 0x72, 0xdf, 0x02, 0xe6, 0x11, 0x1f} }
16-23	8 bytes		See Section B.2.2 for details.

The deconfigured processor error info section has the same format as the PROCESSOR_SPECIFIC_ERROR_RECORD specified in Section B.2.3.1, “Processor Machine Check Errors”.

B.2.3.3 Deconfigured Processor Self-Test Errors

When called with type = "deconfigured processor," SAL_GET_STATE_INFO can use the processor self-test error info section to report deconfigured processor self-test errors to the OS.

Offset	Length	Field	Description
0	16 bytes	GUID	{ 0xbd7614ef, 0x1109, 0x425b { 0x8f, 0x96, 0xf0, 0x5e, 0xfb, 0x18, 0xf9, 0x2f} }
16-23	8 bytes		See Section B.2.2 for details.

```
PROCESSOR_SELF_TEST_ERROR_RECORD SECTION

{

    VALIDATION_BITS                                 8 bytes

        PROC_CR_LID_VALID_BIT                       Bit 0

        EARLY_SELF_TEST_STATE_PARAM_VALID_BIT       Bit 1

        EARLY_SELF_TEST_CONTROL_WORD_VALID_BIT      Bit 2

        PAL_TEST_PROC_STATE_PARAM_VALID_BIT         Bit 3

        PAL_TEST_PROC_TEST_PARAM_VALID_BIT          Bit 4

        VARIABLE_LENGTH_DATA_OFFSET_VALID_BIT[1]    Bit 5

        PROC_OEM_DATA_STRUCT_VALID_BIT              Bit 6

        RESERVED                                    Bits 7-63

    PROC_CR_LID                                     8 bytes

    EARLY_SELF_TEST_STATE_PARAM                     8 bytes

    EARLY_SELF_TEST_CONTROL_WORD                    8 bytes

    PAL_TEST_PROC_STATE_PARAM                       8 bytes

    PAL_TEST_PROC_TEST_PARAM                        8 bytes

    VARIABLE_LENGTH_DATA_OFFSET                     4 bytes

    ; This field provides the parser with an offset from the beginning of the
    ; error section to the variable-length data, which begins with the
```

1. The VARIABLE_LENGTH_DATA_OFFSET must be valid if PROC_OEM_DATA_STRUCT is valid.

```
    ; PROC_OEM_DATA_STRUCT.

    ; SAL implementations for error records with major version = 0x00 and minor
    ; revision = 0x7 should return 0x50(80) for this field if the
    ; PROC_OEM_DATA_STRUCT is valid. If VARIABLE_LENGTH_DATA_OFFSET is invalid,
    ; it should be padded with 0's. If PROC_OEM_DATA_STRUCT isn't valid then
    ; VARIABLE_LENGTH_DATA_OFFSET should be invalid.

    ; In the future, additional fixed-length sections may be added to this error
    ; section and would be inserted after the existing fixed-length fields and
    ; before PROC_OEM_DATA_STRUCT

    ; To provide forwards compability with future error section revisions,
    ; error record parsers must dynamically read the
    ; VARIABLE_LENGTH_DATA_OFFSET value to determine where to start parsing
    ; the PROC_OEM_DATA_STRUCT and future variable length data.

    Reserved                                          4 bytes

    struct {

        PROC_OEM_DATA_STRUCT[1]                       N  bytes

        ; OEM specific data of variable length. See Table B-4 for the format
        ; of this structure. N = 0 if the PROCESSOR_OEM_DATA_VALID_BIT
        ; bit is not set.

    } PROC_OEM_DATA_STRUCT

}
```

B.2.4 Platform Errors

There are no standard platform errors defined in existing specifications. This section attempts to define some typical generic platform error information data structures. OEMs and platform vendors can define additional platform error sections with unique GUIDs customized to their platform topology.

B.2.4.1 Platform Memory Device Error Info

This section describes error information from the memory sub-system.

Offset	Length	Field	Description
0	16 bytes	GUID	{0xe429faf2, 0x3cb7, 0x11d4, {0xbc, 0xa7, 0x0, 0x80, 0xc7, 0x3c, 0x88, 0x81}}
16-23	8 bytes		See Section B.2.2 for details.

1. If their associated VALID_BIT is not set, variable-length fields are not padded and will not be present.

PLATFORM_MEMORY_ERROR_RECORD SECTION BODY STRUCTURE

Offset	Length	Field	Description
0	8	VALIDATION_BITS	Validation Bits to indicate the validity of each of the subsequent fields: Bit 0 – MEM_ERROR_STATUS_VALID_BIT Bit 1 – MEM_PHYSICAL_ADDR_VALID_BIT Bit 2– MEM_ADDR_MASK_BIT Bit 3 – MEM_NODE_VALID_BIT Bit 4 – MEM_CARD_VALID_BIT Bit 5 – MEM_MODULE_VALID_BIT Bit 6 – MEM_BANK_VALID_BIT Bit 7 – MEM_DEVICE_VALID_BIT Bit 8 – MEM_ROW_VALID_BIT Bit 9 – MEM_COLUMN_VALID_BIT Bit 10 – MEM_BIT_POSITION_VALID_BIT Bit 11 – MEM_PLATFORM_REQUESTOR_ID_VALID_BIT Bit 12 – MEM_PLATFORM_RESPONDER_ID_VALID_BIT Bit 13 – MEM_PLATFORM_TARGET_VALID_BIT Bit 14 – MEM_PLATFORM_BUS_SPECIFIC_DATA_VALID_BIT Bit 15 – MEM_PLATFORM_OEM_ID_VALID_BIT Bit 16 – MEM_PLATFORM_OEM_DATA_STRUCT_VALID_BIT Bit 17-63 – RESERVED
8	8 bytes	MEM_ERROR_STATUS	Memory device error status fields (see Table B-5).
16	8 bytes	MEM_PHYSICAL_ ADDR	64-Bit physical address of the memory error.
24	8 bytes	MEM_PHYSICAL_ ADDR_MASK	Defines the valid address Bits in the 64-Bit physical address of the memory error. The mask specifies the granularity of the physical address which is dependent on the h/w implementation factors such as interleaving.
32	2 bytes	MEM_NODE	In a multi-node system, this value identifies the node containing the memory in error.
34	2 bytes	MEM_CARD	The Card number of the memory error location.
36	2 bytes	MEM_MODULE	The Module or RANK number of the memory error location. (NODE, CARD, and MODULE should provide the information needed to identify the failing FRU)
38	2 bytes	MEM_BANK	The Bank number of the memory error location.
40	2 bytes	MEM_DEVICE	The Device number of the memory error location.
42	2 bytes	MEM_ROW	The Row number of the memory error location.
44	2 bytes	MEM_COLUMN	The Column number of the memory error location.
46	2 bytes	MEM_BIT_POSITION	Bit position specifies the Bit within the memory word that is in error.
48	8 bytes	REQUESTOR_ID	Hardware address of the device or component initiating transaction.
56	8 bytes	RESPONDER_ID	Hardware address of the responder to transaction.
64	8 bytes	TARGET_ID	Hardware address of intended target of transaction.
72	8 bytes	BUS_SPECIFIC_DATA	OEM specific bus-dependent data.
80	16 bytes	MEM_PLATFORM_ OEM_ID	OEM specific data containing identification information for the Memory Controller.
96	N bytes	MEM_PLATFORM_ OEM_DATA_STRUCT	OEM specific data of variable length. See Table B-4 for the format of this structure. N equals 0 if the MEM_PLATFORM_OEM_DATA_STRUCT_VALID_BIT is not set.

Table B-4. Format of Variable Length Info Structure

Offset	Length	Field	Description
0	2 bytes	LENGTH	Length of this structure in bytes. Length is 2 + *M* bytes. The value of 2 + *M* must align to an 8 byte boundary.
2	M bytes	VARIABLE_INFO	OEM defined variable size data.

B.2.4.2 Platform PCI Bus Error Info

This section describes the errors that occur on the PCI bus itself (e.g. parity error, target abort, and so on). Errors within a PCI component are described in Section B.2.4.3. An error within a PCI component that results in error signalling on the PCI bus will result in both sections being present in the error record.

Offset	Length	Field	Description
0	16 bytes	GUID	{0xe429faf4, 0x3cb7, 0x11d4, {0xbc, 0xa7, 0x0, 0x80, 0xc7, 0x3c, 0x88, 0x81}}
16-23	8 bytes		See Section B.2.2 for details.

PLATFORM_PCI_BUS_ERROR_RECORD SECTION BODY STRUCTURE

Offset	Length	Field	Description
0	8	VALIDATION_BITS	Validation Bits to indicate the validity of each of the subsequent fields: Bit 0 – PCI_BUS_ERROR_STATUS_VALID_BIT Bit 1 – PCI_BUS_ERROR_TYPE_VALID_BIT Bit 2 – PCI_BUS_ID_VALID_BIT Bit 3 – PCI_BUS_ADDRESS_VALID_BIT Bit 4 – PCI_BUS_DATA_VALID_BIT Bit 5 – PCI_BUS_CMD_VALID_BIT Bit 6 – PCI_BUS_REQUESTOR_ID_VALID_BIT Bit 7 – PCI_BUS_COMPLETER_ID_VALID_BIT Bit 8 – PCI_BUS_TARGET_ID_VALID_BIT Bit 9 – PCI_BUS_OEM_ID_VALID_BIT Bit 10 – PCI_BUS_OEM_DATA_STRUCT_VALID_BIT Bit 11..63– RESERVED
8	8 bytes	PCI_BUS_ERROR_ STATUS	PCI Bus error status fields (see Table B-5).
16	2 bytes	PCI_BUS_ERROR_ TYPE	PCI Bus error types Byte0: 0 – Unknown or OEM System Specific Error 1 – Data Parity Error 2 – System Error 3 – Master Abort 4 – Bus Time Out or No Device Present (No DEVSEL#) 5 – Master Data Parity Error 6 – Address Parity Error 7 – Command Parity Error Others – Reserved Byte1: Reserved
18	2 bytes	PCI_BUS_ID	Designated PCI Bus identifier encountering error. Bits 0..7 – Bus Number Bits 8..15 – Segment Number
20	4 bytes	Reserved	
24	8 bytes	PCI_BUS_ADDRESS	Memory or IO address on the PCI bus at the time of the event.
32	8 bytes	PCI_BUS_DATA	Data on the PCI bus at the time of the event.
40	8 bytes	PCI_BUS_CMD	Bus command or operation at the time of the event. Byte 7: Bits 7-1: Reserved (should be 0) Byte 7: Bit 0 = If 0, then the command is a PCI command. If 1, the command is a PCI-X command
48	8 bytes	PCI_BUS_ REQUESTOR_ID	PCI Bus Requestor ID at the time of the event[1].
56	8 bytes	PCI_BUS_ COMPLETER_ID	PCI Bus Responder ID at the time of the event.
64	8 bytes	PCI_BUS_TARGET_ID[2]	PCI Bus intended Target ID at the time of the event.
72	16 bytes	PCI_BUS_OEM_ID	OEM specific data containing identification information for the PCI Bus.
88	N bytes	PCI_BUS_OEM_DATA_ STRUCT	OEM specific data of variable length. See Table B-4 for the format of this structure. N equals 0 if the PCI_BUS_OEM_DATA_STRUCT_VALID_BIT is not set.

Notes:

1. As defined in the PCI-X Addendum to the *PCI Local Bus Specification* (a combination of the devices bus number, device number, and function number).
2. This could be a memory or I/O port address.

Refer to the PCI Specification (http://www.pcisig.com) for further details.

B.2.4.3 Platform PCI Component Error Info

Offset	Length	Field	Description
0	16 bytes	GUID	{0xe429faf6, 0x3cb7, 0x11d4, {0xbc, 0xa7, 0x0, 0x80, 0xc7, 0x3c, 0x88, 0x81}}
16-23	8 bytes		See Section B.2.2 for details.

PLATFORM_PCI_COMPONENT_ERROR_RECORD SECTION BODY STRUCTURE

Offset	Length	Field	Description
0	8	VALIDATION_BITS	Validation Bits to indicate the validity of each of the subsequent fields: Bit 0 – PCI_COMP_ERROR_STATUS_VALID_BIT Bit 1 – PCI_COMP_INFO_VALID_BIT Bit 2 – PCI_COMP_MEM_NUM_VALID_BIT Bit 3 – PCI_COMP_IO_NUM_VALID_BIT Bit 4 – PCI_COMP_REGS_DATA_PAIR_VALID_BIT Bit 5 – PCI_COMP_OEM_DATA_STRUCT_VALID_BIT Bit 6..63– RESERVED
8	8 bytes	PCI_COMP_ERROR_ STATUS	PCI Component error status fields (see Table B-5).
16	16 bytes	PCI_COMP_INFO	PCI Component Information to identify the device: Bytes 0-1 – Vendor ID Bytes 2-3 – Device ID Bytes 4-6 – Class Code Byte 7 – Function Number Byte 8 – Device Number Byte 9 – Bus Number Byte 10 – Segment Number Bytes 11-15 – Reserved (0)
38	4 bytes	PCI_COMP_MEM_NUM	Number of PCI Component Memory Mapped register address/data pair values present in this structure.
36	4 bytes	PCI_COMP_IO_NUM	Number of PCI Component Programmed IO register address/data pair values present in this structure.
40	2 x 8 x M bytes	PCI_COMP_REGS_ DATA_PAIR	An array of address/data pair values. The data may be 8 bytes in length. M = PCI_COMP_MEM_NUM + PCI_COMP_IO_NUM
40+2x 8xM	N bytes	PCI_COMP_OEM_ DATA_STRUCT	OEM specific data of variable length. See Table B-4 for the format of this structure. If PCI_COMP_OEM_DATA_STRUCT_VALID_BIT is not set, N = 0.

Refer to the PCI Bus Specification (http://www.pcisig.com) for further details. The above section definition does not specify which chipset registers are required in the error section. To decode the chipset errors completely, the error status registers may not be sufficient. Other implementation-dependent chipset configuration registers may be required to decode the error status information. The error handler is expected to have an intimate knowledge of the chipset and the platform to parse the error information.

Note that a multi-function device may require more than one section to report the error information (one for each function).

B.2.4.4 Platform SEL Device Error Info

Offset	Length	Field	Description
0	16 bytes	GUID	{0xe429faf3, 0x3cb7, 0x11d4, {0xbc, 0xa7, 0x0, 0x80, 0xc7, 0x3c, 0x88, 0x81}}
16-23	8 bytes		See Section B.2.2 for details.

PLATFORM_SYSTEM_EVENT_LOG_RECORD SECTION BODY STRUCTURE

Offset	Length	Field	Description
0	8	VALIDATION_BITS	Validation Bits to indicate the validity of each of the subsequent fields: Bit 0 – SEL_RECORD_ID_VALID_BIT Bit 1 – SEL_RECORD_TYPE_VALID_BIT Bit 2 – SEL_GENERATOR_ID_VALID_BIT Bit 3 – SEL_EVM_REV_VALID_BIT Bit 4 – SEL_SENSOR_TYPE_VALID_BIT Bit 5 – SEL_SENSOR_NUM_VALID_BIT Bit 6 – SEL_EVENT_DIR_TYPE_VALID_BIT Bit 7 – SEL_EVENT_DATA1_VALID_BIT Bit 8 – SEL_EVENT_DATA2_VALID_BIT Bit 9 – SEL_EVENT_DATA3_VALID_BIT Bit 10 – SEL_TIME_STAMP_VALID_BIT Bit 11-63 – RESERVED
8	2 bytes	SEL_RECORD_ID	Record ID used for SEL record access.
10	1 bytes	SEL_RECORD_TYPE	Indicates the record type: 0x02 – System Event Record 0xC0-0xDF – OEM time stamped, bytes 8-16 OEM defined 0xE0-0xFF – OEM non-time stamped, bytes 4-16 OEM defined
11	4 bytes	SEL_TIME_STAMP	Time stamp of the event log
15	2 bytes	SEL_GENERATOR_ID	Software ID if event was generated by software Byte1: Bit 7:1 – 7-Bit system software ID. Bit 0 – set to one (1) when using system software. Byte 2: Bit 7:2 – Reserved. Write as 0, ignore when read. Bit 1:0 – IPMB device LUN if byte 1 holds slave address, 0x0 otherwise.
17	1 bytes	SEL_EVM_REV	The error message format version.
18	1 bytes	SEL_SENSOR_TYPE	Sensor type code of the sensor that generated the event.
19	1 bytes	SEL_SENSOR_NUM	Number of the sensor that generated the event.
20	1 bytes	SEL_EVENT_DIR_ TYPE	Event Dir: Bit 7 – 0 for assertion; 1 for deassertion. Event Type: Type of trigger for the event, e.g. critical threshold going high, state asserted, and so on. Also indicates class of the event, e.g. discrete, threshold, or OEM. The Event Type field is encoded using the Event/Reading Type Code. See Section 30.1, Event/Reading Type Codes. Bit 6:0 – Event Type Code
21	1 bytes	SEL_DATA1	Event data field.
22	1 bytes	SEL_DATA2	Event data field.
23	1bytes	SEL_DATA3	Event data field.

Refer to the *Intelligent Platform Management Initiative Specification* (http:// developer.intel.com/ design/servers/ipmi) for further details.

B.2.4.5 Platform SMBIOS Device Error Info

Offset	Length	Field	Description
0	16 bytes	GUID	{0xe429faf5, 0x3cb7, 0x11d4, {0xbc, 0xa7, 0x0, 0x80, 0xc7, 0x3c, 0x88, 0x81}}
16-23	8 bytes		See Section B.2.2 for details.

PLATFORM_SMBIOS_ERROR_RECORD SECTION BODY STRUCTURE

Offset	Length	Field	Description
0	8	VALIDATION_BITS	Validation Bits to indicate the validity of each of the subsequent fields: Bit 0 – SMBIOS_EVENT_TYPE_VALID_BIT Bit 1 – SMBIOS_LENGTH_VALID_BIT Bit 2 – SMBIOS_TIME_STAMP_VALID_BIT Bit 3 – SMBIOS_DATA_VALID_BIT Bit 4-63 – RESERVED
8	1 bytes	SMBIOS_EVENT_TYPE	Event Type – enum see SMBIOS 2.3 – 3.3.16.6.1.
9	1 bytes	SMBIOS_LENGTH	Length of the error information in bytes.
10	6 bytes	SMBIOS_TIME_STAMP	Time stamp in BCD.
16	N bytes	SMBIOS_DATA	OEM specific data of variable length. See Table B-4 for the format of this structure. N equals 0 if SMBIOS_DATA_VALID_BIT is not set.

Refer to the SMBIOS Specification (http://www.dmtf.org/standards/bios.php) for further details.

B.2.4.6 Platform Specific Error Info

This section provides information on the OEM hardware errors that cannot be described by other sections. The operating system could handle the error in a generic way by examining the section GUID, the ERROR_RECOVERY_INFO, the PLATFORM_ERROR_STATUS, and the TARGET address fields. Refer to the respective platform document for further details.

Offset	Length	Field	Description
0	16 bytes	GUID	{0xe429faf7, 0x3cb7, 0x11d4, {0xbc, 0xa7, 0x0, 0x80, 0xc7, 0x3c, 0x88, 0x81}}
16-23	8 bytes		See Section B.2.2 for details.

PLATFORM_GENERIC_ERROR_RECORD SECTION BODY STRUCTURE

Offset	Length	Field	Description
0	8	VALIDATION_BITS	Validation Bits to indicate the validity of each of the subsequent fields: Bit 0 – PLATFORM_ERROR_STATUS_VALID_BIT Bit 1 – PLATFORM_REQUESTOR_ID_VALID_BIT Bit 2 – PLATFORM_RESPONDER_ID_VALID_BIT Bit 3 – PLATFORM_TARGET_VALID_BIT Bit 4 – PLATFORM_SPECIFIC_DATA_VALID_BIT Bit 5 – PLATFORM_OEM_ID_VALID_BIT Bit 6 – PLATFORM_OEM_DATA_STRUCT_VALID_BIT Bit 7 – PLATFORM_OEM_DEVICE_PATH_VALID_BIT Bit 8..63 – RESERVED
8	8 bytes	PLATFORM_ERROR_ STATUS	Platform generic error status fields (see Table B-5).
16	8 bytes	PLATFORM_ REQUESTOR_ID	Requestor ID at the time of the event.
24	8 bytes	PLATFORM_ RESPONDER_ID	Responder ID at the time of the event.
32	8 bytes	PLATFORM_TARGET_ID	Target ID at the time of the event.
40	8 bytes	PLATFORM_BUS_ SPECIFIC_DATA	OEM specific bus-dependent data.
48	16 bytes	OEM_COMPONENT_ID	A unique ID of the component reporting the error.
64	N bytes	PLATFORM_OEM_ DATA_STRUCT	OEM specific data of variable length. See Table B-4 for the format of this structure. N equals 0 if PLATFORM_OEM_DATA_STRUCT_VALID_BIT is not set
64+N bytes	X bytes	PLATFORM_OEM_ DEVICE_PATH	OEM specific Vendor Device Path. Please refer to the *Extensible Firmware Interface Specification* for the format of this field.X equals 0 if PLATFORM_OEM_DEVICE_PATH_VALID_BIT is not set.

B.2.4.7 Platform PCIe* 1.1 Error Info

This section describes the errors that occur on the PCI Express, for Root Port (RP), PCIe Bridge, Switch and End Point (EP) device. The error information for the PCIe RP is as reported by the RP bridge. An error within a PCIe component that results in error signalling on the PCIe bus will result in multiple sections being reported, one or more sections for each of the RP and one or more sections for the downstream PCIe devices.

In a system configured to enable AER, but with the presence of a downstream PCI/PCI-X bridge and its associated endpoint devices, the error record returned may consist of PCIe error sections with PCIe device types of 1, 7or 8. However, system firmware can optionally provide the PCI Bus and PCI Component error sections alongside PCIe error sections in a single error record.

The ownership of the AER resources is negotiated between SAL and OS through an ACPI interface defined in the PCI Firmware Specification version 3.0. SAL must not control or report on PCIe error resources, if the OS has taken native control of AER. SAL is allowed to provide the PCIe error record sections to the OS when SAL owns the AER resources. SAL owns the PCIe error resources on systems that boot an OS that is not ACPI 3.0 compliant or PCIe aware.

Offset	Length	Field	Description
0	16 bytes	GUID	{ 0x98d1e922, 0xd358, 0x4d22, { 0x98, 0xf3, 0xac, 0xad, 0xdc, 0x4c, 0x3b, 0x6 } }
16-23	8 bytes		See Section B.2.2 for details.

PLATFORM_PCIe_BUS_ERROR_RECORD SECTION BODY STRUCTURE

Offset	Length	Field	Description
0	8	VALIDATION_BITS	Validation Bits to indicate the validity of each of the subsequent fields: Bit 0- PCIe_DEVICE_PORT_TYPE_VALID_BIT Bit 1– PCIe_VERSION_INFO_VALID_BIT Bit 2– PCIe_CMD_STS_INFO_VALID_BIT Bit 3– PCIe_DEVICE_ID_INFO_VALID_BIT Bit 4– PCIe_DEVICE_SN_INFO_VALID_BIT Bit 5– PCIe_BRIDGE_CNTRL_STS_INFO_VALID_BIT Bit 6– PCIe_CAP_STRUCT_CNTRL_STS_INFO_VALID_BIT Bit 7– PCIe_AER_INFO_STRUCT_VALID_BIT Bit 8-15: Reserved Bit 16– PCIe_BUS_OEM_ID_VALID_BIT Bit 17 – PCIe_BUS_OEM_DATA_STRUCT_VALID_BIT Bit 18..63– RESERVED
8	4 bytes	PCIe_DEVICE_PORT_ TYPE	PCIe Device/Port Type as defined in the PCI Express capabilities register[1]. Refer to PCIe specification for more details on the following encoding. The encoding per the PCIe specification is as follows: 0: PCI Express End Point 1: Legacy PCI End Point Device 4: Root Port of root complex 5: Upstream port for PCI Express Switch 6: Downstream port for PCI Express Switch 7: PCI Express to PCI/PCI-X Bridge 8: PCI/PCI-X to PCI Express bridge 9: Root Complex Integrated Endpoint Device 10: Root Complex Event Collector
12	4 bytes	PCIe_VERSION_INFO	PCIe Spec. version number supported by the platform Byte 0-1: PCIe Spec. Version Number Byte 0: Minor Version in BCD Byte 1: Major Version in BCD Byte 2-3: Reserved
16	4 bytes	PCIe_CMD_STS_INFO	PCIe Root Port Bridge or Device PCI compatible Command & Status Register Information. Byte 0-1: PCI Command Register Byte 2-3: PCI Status Register
20	4 bytes	Reserved	Reserved
24	16 bytes	PCIe_DEVICE_ID_INFO	PCIe Root Port PCI/bridge PCI compatible device number and bus number information to uniquely identify the a root port or bridge. Default values for both the bus numbers is zero Byte 0-1: Vendor ID Byte 2-3: Device ID Byte 4-6: Class Code Byte 7: Function Number Byte 8: Device Number Byte 9-10: Segment Number Byte 11: Root Port/Bridge Primary Bus Number or device bus number Byte 12: Root Port/Bridge Secondary Bus Number Byte 13-14: • Bit 0-2: Reserved. • Bit 3:15: Slot number as defined in the PCIe Slot Capability register for "Physical Slot Number" field. Byte 15: Reserved

Offset	Length	Field	Description
40	8 bytes	PCIe_DEVICE_SN_ INFO	PCIe Root Port PCI/bridge or end point device serial number Byte 0-3: PCIe Device Serial Number Lower DW Byte 4-7: PCIe Device Serial Number Upper DW
48	4 bytes	PCIe_BRIDGE_CNTRL_ STS_INFO	PCIe Root Port/Bridge Secondary Status & Control Register Information. This field is valid for bridges only. Byte 0-1: Bridge Secondary Status Register Byte 2-3: Bridge Control Register
52	36 bytes	PCIe_CAP_STRUCT_ CNTRL_STS_INFO	PCIe Capability Structure The 36-byte structure containing device capabilities and status, as defined in the PCIe[2] Base Specification. The fields in the structure vary with different device types. The “Next CAP pointer” field should be considered invalid and any reserved fields of the structure are reserved for future use. Note that PCIe devices without AER (PCIe_AER_INFO_STRUCT_VALID_BIT=0) may report status using this structure.
88	96 bytes	PCIe_AER_INFO_ STRUCT	PCIe Advanced Error Reporting Extended Capability Structure corresponding to the device type specified in the capability register or the header. Refer to PCIe[3] and PCIe to PCI/PCI-X bridge[4] specification for details on the AER information for Root Port, End Point Device, PCIe bridge, PCI Switch, and so on. The fields in the structure will vary with different device types. The unused part of the structure is reserved for future use. For devices without AER capabilities this field is invalid and PCIe_AER_INFO_STRUCT_VALID_BIT should be set to 0.
184	16 bytes	PCIe_BUS_OEM_ID	OEM specific data containing identification information for the PCIe Bus.
200	N bytes	PCIe_BUS_OEM_ DATA_STRUCT	OEM specific data of variable length. See Table B-4 for the format of this structure. N equals 0 if the PCIe_BUS_OEM_DATA_STRUCT_VALID_BIT is not set.

Notes:

1. Refer to Section 7.8.2, Table 7-10 of PCI Express Base Spec., Rev. 1.1
2. Refer to Section 7.8, Figure 7-9 of PCI Express Base Spec., Rev. 1.1
3. Refer to Section 7.10, Figure 7-26 of PCI Express Base Spec., Rev. 1.1
4. Refer to Section 5.2.3, Figure 5-4 of PCI Express to PCI/PCI-X Bridge Spec., Rev. 1.0

Refer to the *PCIe* Specification (http://www.pcisig.com)* for further details.

B.2.4.8 Platform PCIe* 1.1/2.0 Error Info

This section describes the errors that occur on the PCI Express revisions 1.1 and 2.0 , for Root Port (RP), PCIe Bridge, Switch and End Point (EP) devices. The error information for the PCIe RP is as reported by the RP bridge. An error within a PCIe component that results in error signalling on the PCIe link will result in multiple sections being reported, one or more sections for each of the RP and one or more sections for the downstream PCIe devices.

In a system configured to enable AER, but with the presence of a downstream PCI/PCI-X bridge and its associated endpoint devices, the error record returned may consist of PCIe error sections with PCIe device types of 1, 7or 8. However, system firmware can optionally provide the PCI Bus and PCI Component error sections alongside PCIe error sections in a single error record.

The ownership of the AER resources is negotiated between SAL and OS through an ACPI interface defined in the PCI Firmware Specification version 3.0. SAL must not control or report on PCIe error resources, if the OS has taken native control of AER.

SAL is allowed to provide the PCIe error record sections to the OS when SAL owns the AER resources. SAL owns the PCIe error resources on systems that boot an OS that is not ACPI 3.0 compliant or PCIe aware.

Offset	Length	Field	Description
0	16 bytes	GUID	{ 0x09f42430, 0xd441, 0x11dc { 0x95, 0xff, 0x08, 0x00, 0x20, 0x0c, 0x9a, 0x66} }
16-23	8 bytes		See Section B.2.2 for details.

PLATFORM_PCIe_ERROR_RECORD Section Body Structure

Offset	Length	Field	Description
0	8	VALIDATION_BITS	Validation Bits to indicate the validity of each of the subsequent fields: Bit 0– PCIe_DEVICE_PORT_TYPE_VALID_BIT Bit 1– PCIe_VERSION_INFO_VALID_BIT Bit 2– PCIe_CMD_STS_INFO_VALID_BIT Bit 3– PCIe_DEVICE_ID_INFO_VALID_BIT Bit 4– PCIe_DEVICE_SN_INFO_VALID_BIT Bit 5– PCIe_BRIDGE_CNTRL_STS_INFO_VALID_BIT Bit 6– PCIE_CAP_STRUCT_CNTRL_STS_INFO_VALID_BIT Bit 7– PCIe_AER_INFO_STRUCT_VALID_BIT Bit 8– PCIe_OEM_ID_VALID_BIT Bit 9– VARIABLE_LENGTH_DATA_OFFSET_VALID_BIT Bit 10–31: Reserved Validation Bit 32 starts the variable-length data section of the error section. If this error section is extended with additional fixed-length fields, they will inserted before the variable-length fields, with the VARIABLE_LENGTH_DATA_OFFSET field updated to point to the start of the variable length data section. Bit 32– PCIe_OEM_DATA_STRUCT_VALID_BIT Bit 33– Bit 63 reserved
8	4 bytes	PCIe_DEVICE_PORT_TYPE	PCIe Device/Port Type as defined in the PCI Express capabilities register[1]. Refer to PCIe specification for more details on the following encoding. The encoding per the PCIe specification is as follows: 0: PCI Express End Point 1: Legacy PCI End Point Device 4: Root Port of root complex 5: Upstream port for PCI Express Switch 6: Downstream port for PCI Express Switch 7: PCI Express to PCI/PCI-X Bridge 8: PCI/PCI-X to PCI Express bridge 9. Root Complex Integrated Endpoint Device 10. Root Complex Event Collector
12	4 bytes	PCIe_VERSION_INFO	PCIe Spec. version number supported by the platform Byte 0-1: PCIe Spec. Version Number Byte 0: Minor Version in BCD Byte 1: Major Version in BCD Byte 2-3: Reserved
16	4 bytes	PCIe_CMD_STS_INFO	PCIe Root Port Bridge or Device PCI compatible Command & Status Register Information. Byte 0-1: PCI Command Register Byte 2-3: PCI Status Register
20	4 bytes	Reserved	Reserved

Offset	Length	Field	Description
24	16 bytes	PCIe_DEVICE_ID_INFO	PCIe Root Port PCI/bridge PCI compatible device number and bus number information to uniquely identify the a root port or bridge. Default values for both the bus numbers is zero Byte 0-1: Vendor ID Byte 2-3: Device ID Byte 4-6: Class Code Byte 7: Function Number Byte 8: Device Number Byte 9-10: Segment Number Byte 11: Root Port/Bridge Primary Bus Number or device bus number Byte 12: Root Port/Bridge Secondary Bus Number Byte 13-14: • Bit 0-2: Reserved. • Bit 3-15: Slot number as defined in the PCIe Slot Capability register for "Physical Slot Number" field Byte 15: Reserved
40	8 bytes	PCIe_DEVICE_SN_INFO	PCIe Root Port PCI/bridge or end point device serial number Byte 0-3: PCIe Device Serial Number Lower DW Byte 4-7: PCIe Device Serial Number Upper DW
48	4 bytes	PCIe_BRIDGE_CNTRL_STS_INFO	PCIe Root Port/Bridge Secondary Status & Control Register Information. This field is valid for bridges only. Byte 0-1: Bridge Secondary Status Register Byte 2-3: Bridge Control Register
52	60 bytes	PCIe_CAP_STRUCT_CNTRL_STS_INFO	PCIe Capability Structure The 60-byte structure is used to report device capabilities. This structure is used to report the 36-byte PCIe 1.1 Capability Structure (See Figure 7-9 of the PCI Express Base Specification, Rev 1.1) with the last 24 bytes padded. This structure is also used to report the 60-byte PCIe 2.0 Capability Structure (See Figure 7-9 of the PCI Express 2.0 Base Specification.) The fields in the structure vary with different device types. The “Next CAP pointer” field should be considered invalid and any reserved fields of the structure are reserved for future use. Note that PCIe devices without AER (PCIe_AER_INFO_STRUCT_VALID_BIT=0) may report status using this structure.
112	4 bytes	Reserved	Reserved
116	96 bytes	PCIe_AER_INFO_STRUCT	PCIe Advanced Error Reporting Extended Capability Structure corresponding to the device type specified in the capability register or the header. Refer to PCIe[2] and PCIe to PCI/PCI-X bridge[3] specification for details on the AER information for Root Port, End Point Device, PCIe bridge, PCI Switch and etc. The fields in the structure will vary with different device types. The unused part of the structure is reserved for future use. For devices without AER capabilities this field is invalid and PCIe_AER_INFO_STRUCT_VALID_BIT should be set to 0.
212	16 bytes	PCIe_OEM_ID	OEM specific data containing identification information for the PCIe Link.

Offset	Length	Field	Description
228	4 bytes	VARIABLE_LENGTH_DATA_OFFSET	This field provides the parser with an offset from the beginning of the error section to the variable-length data portion of the error section. SAL implementations for error records with major version = 0x00 and minor version = 0x07 should return 0x108 (264) for this field if PCIe_OEM_DATA_STRUCT is valid. If VARIABLE_LENGTH_DATA_OFFSET is invalid, it should be padded with 0's. If the PCIe_OEM_DATA_STRUCT isn't valid then VARIABLE_LENGTH_DATA_OFFSET should be invalid. In the future, additional fixed-length sections may be added to this error section and would be inserted before any variable-length fields. To provide forwards compatibility with future error section revisions, error record parsers must dynamically read the VARIABLE_LENGTH_DATA_OFFSET value to determine where to start parsing variable length data.
232	8 bytes	Reserved	Reserved
VARIABLE_LENGTH_DATA_OFFSET	N bytes	PCIe_OEM_DATA_STRUCT	OEM specific data of variable length. See Table B-4 for the format of this structure. N equals 0 if the PCIe_OEM_DATA_STRUCT_VALID_BIT is not set.

Notes:

1.Refer to Section 7.8.2, Table 7-10 of PCI Express Base Specification., Rev. 2.0
2.Refer to Section 7.10, Figure 7-26 of PCI Express Base Spec., Rev. 2.0
3.Refer to Section 5.2.3, Figure 5-4 of PCI Express to PCI/PCI-X Bridge Spec., Rev. 1.0

Refer to the *PCIe* Specification (http://www.pcisig.com)* for further details.

B.2.5 Error Status

The error status definition provides the capability to abstract information from implementation- specific error registers into generic error codes in order that the operating systems may deal with the errors without an intimate knowledge of the underlying platform.

Table B-5. Error Status Fields

Bit Position	Description
Bit 0-Bit7	Reserved.
Bit8 – Bit 15	Encoded value for the Error_Type[1] (see Table B-6).
Bit 16	Address: Error was detected on the address signals or on the address portion of the transaction.
Bit 17	Control: Error was detected on the control signals or in the control portion of the transaction.
Bit 18	Data: Error was detected on the data signals or in the data portion of the transaction.
Bit 19	Responder: Error was detected by the responder of the transaction.
Bit 20	Requestor: Error was detected by the requestor of the transaction.
Bit 21	First error: If multiple errors are logged for a section type, this is the first error in chronological sequence. Setting of this bit is optional.
Bit 22	Overflow: Additional errors occurred and were not logged due to lack of logging resources.
Bit 23..63	Reserved.

Notes:

1. Error_Type: Error_Type provides information about the type of error detected. If it is not possible to determine the exact cause of the error, the type may be promoted to one of the two values of 1 or 16 as described in Table B-6.

Table B-6. Error Types

Encoding	Description
1	ERR_INTERNAL Error detected internal to the component.
16	ERR_BUS Error detected in the bus.
Detailed Internal Errors	
4	ERR_MEM Storage error in memory (DRAM).
5	ERR_TLB Storage error in TLB.
6	ERR_CACHE Storage error in cache.
7	ERR_FUNCTION Error in one or more functional units.
8	ERR_SELFTEST component failed self test.
9	ERR_FLOW Overflow or Undervalue of internal queue.
Detailed Bus Errors	
17	ERR_MAP Virtual address not found on IO-TLB or IO-PDIR.
18	ERR_IMPROPER Improper access error.
19	ERR_UNIMPL Access to a memory address which is not mapped to any component.
20	ERR_LOL Loss Of Lockstep.
21	ERR_RESPONSE Response not associated with a request.
22	ERR_PARITY Bus parity error (must also set the A, C, or D Bits).
23	ERR_PROTOCOL Detection of a protocol error.
24	ERR_ERROR Detection of PATH_ERROR.
25	ERR_TIMEOUT Bus operation time-out.
26	ERR_POISONED A read was issued to data that has been poisoned.
27	ERR_BANDWIDTH Bus or Link is not operating at full bandwidth.
All Others	Reserved.

§