# Itanium™ Processor Family System Abstraction Layer Specification

*July 2001*

Document Number: 245359-005

# Contents

# Figures

# Tables

# Introduction 1

## 1.1 Objectives

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

This document specifies requirements to develop platform firmware for Itanium-based systems. A companion document, *The Extensible Firmware Interface (EFI) Specification*, describes additional interfaces that must be implemented to access devices on the platform. The *EFI Specification* is a platform binding specification and is also part of Itanium-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 and does not require a specific implementation.

The primary objectives of Itanium-based firmware are to:

- Enable boot of Itanium-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 abstraction for the platform hardware from the abstraction for the processor hardware.
- Enable platform differentiation, hardware innovation, and optimization of Itanium-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 & recovery, large memory support, multiprocessing, and broader and deeper I/O hierarchies (possibly greater than 100 I/O cards).
- Optionally enable shrink-wrapped versions of the IA-32 operating systems to boot. This will involve support of IA-32 industry standard calls and Application Programming Interfaces (APIs).
- Enable reuse of IA-32 BIOS code as part of 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 processor system environment.
- Optionally, enable the use of legacy PC peripherals, option ROMs, and PCI cards with IA-32 Plug-and-Play expansion ROMs.

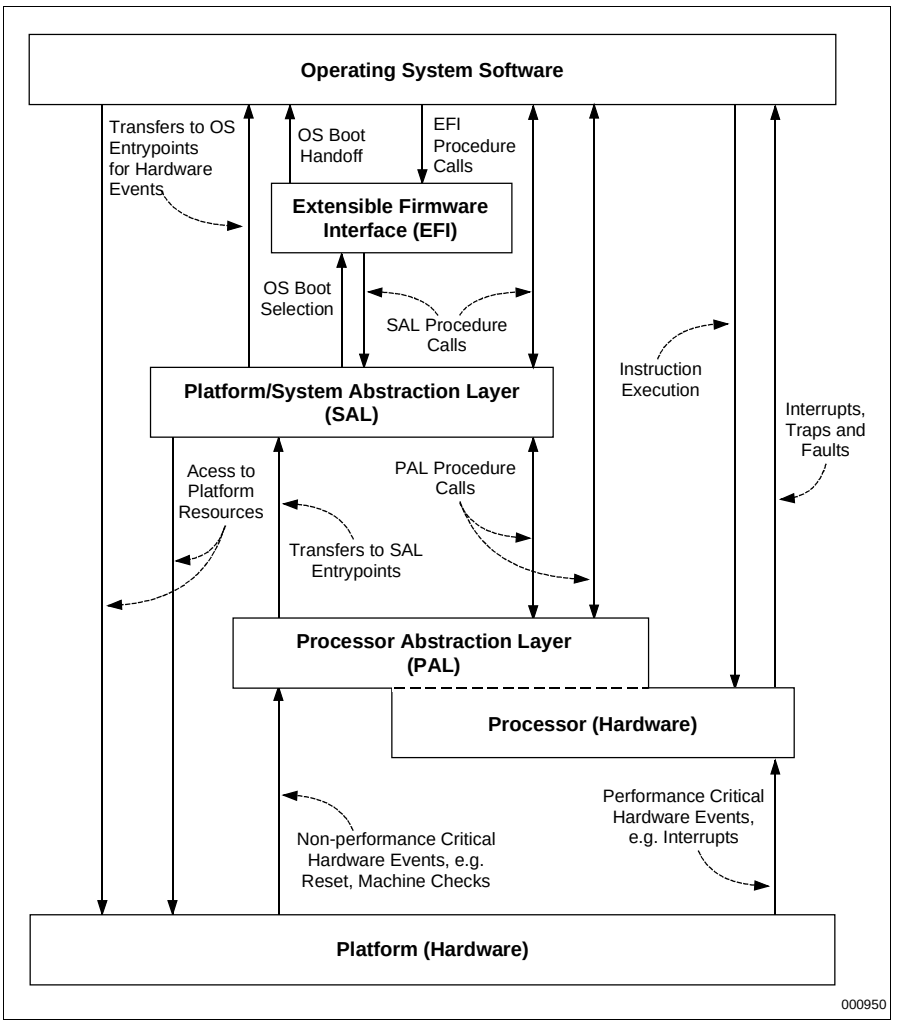
This document describes the platform dependent firmware interfaces needed to support these goals. However, this document is not intended to document PC infrastructure specifications.

## 1.2 Firmware Model

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

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

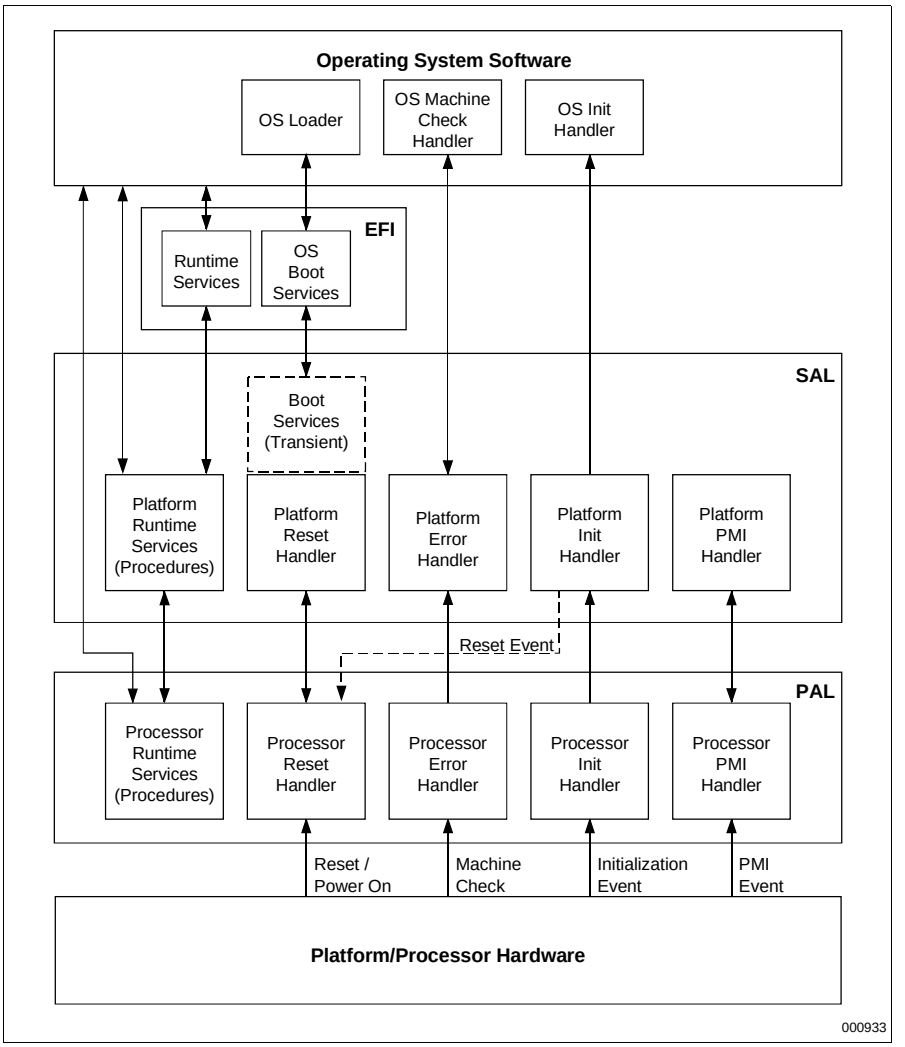
**Figure 1-1. Firmware Model**



PAL encapsulates processor implementation-specific features and is part of the Itanium processor architecture. PAL operates independently of the number of processors. SAL is the platform-specific firmware component that isolates operating systems and other higher level software from implementation differences in the platform. EFI is the platform binding specification layer that 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 major pieces of functionality for an Itanium-based platform:

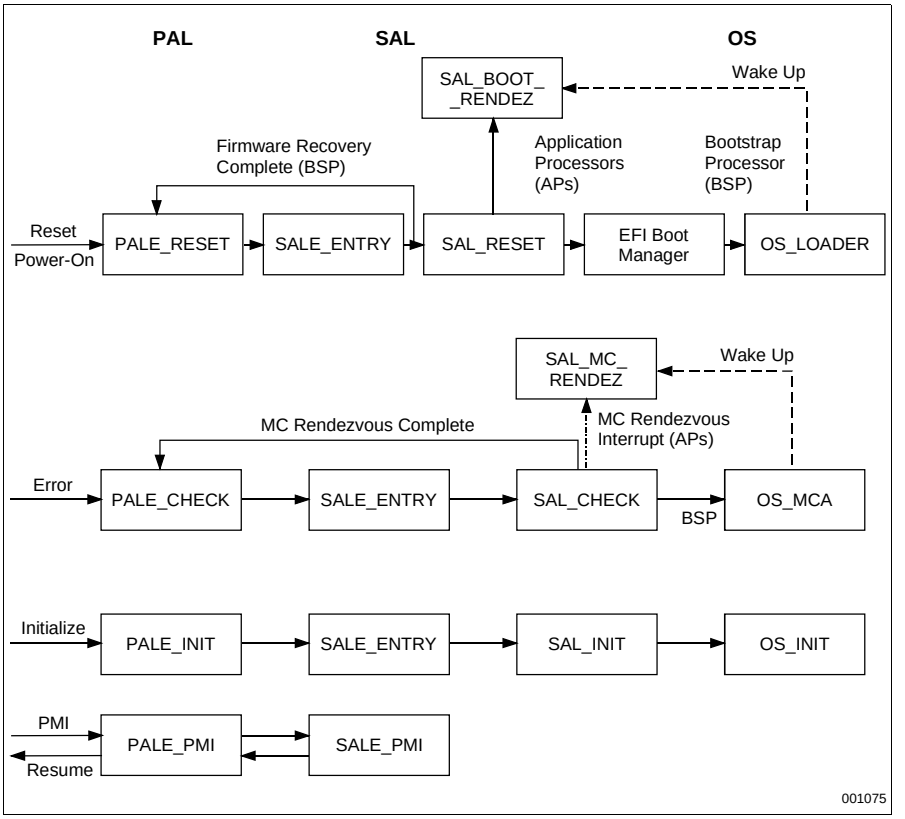
- Initialize, configure, and test the platform hardware. This includes the memory and I/O subsystems, the necessary boot devices, and platform specific hardware.
- Select the bootstrap processor (BSP) in a multiprocessor platform and set the configurable processor features. The Itanium processor provides its own PAL firmware for initialization and test, but this abstraction has no knowledge of the platform and 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 (SAL_RESET interface).
- Optionally encapsulate and provide the environment necessary 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 multiprocessor configurations.
- Provide runtime service routines to encapsulate those functions of the platform necessary for EFI and the operating system while they are running.
- Provide the functions necessary 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 necessary to aid in the recovery from a corrupted boot ROM.
- Optionally, provide an user interface to aid in system configuration, information passing and troubleshooting.

These SAL functions can be divided into the following interface categories:

- SAL entrypoints from PAL: SALE_ENTRY and SALE_PMI.
- Operating system entrypoints from SAL: OS_MCA, OS_INIT and OS_BOOT_RENDEZ.
- SAL runtime service routines.

## 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 1-2 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 the error logging, correction, and reporting. PALE_CHECK is entered as a response to processor or platform errors.
3. PALE_INIT saves the minimal processor state, initializes the processor, and branches to SALE_ENTRY. SALE_ENTRY, in turn, branches to SAL_INIT. PALE_INIT is entered as a response to an initialization event.
4. PALE_PMI determines the type of platform management event and branches to the SALE_PMI for certain conditions. PALE_PMI is entered as a response to a platform management event.

### 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 different entrypoints within the SAL for reset, firmware recovery, machine check and initialization events.

   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 firmware recovery condition is present. SAL is the only entity that has 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 PAL branches to for handling platform management events in an implementation-dependent manner.

### 1.4.3 Operating System Entrypoints

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 *EFI Specification* for details.
- OS_BOOT_RENDEZ is the operating system multiprocessor rendezvous handler. 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 machine check abort handler. Called from SAL_CHECK to allow the OS to handle the machine checks that are not corrected by hardware, PAL or SAL.
- OS_INIT – Operating system Initialization Handler. Called from SAL_INIT to handle a valid initialization event.

## 1.5 Related Documents

The following documents contain additional material related to Itanium-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
- *Extensible Firmware Interface Specification* – Intel
- *Itanium™ Software Conventions and Runtime Architecture Guide* – HP/Intel
- *Intel® Itanium™ Architecture Software Developer's Manual* – Intel
- *Itanium™ Processor Family Error Handling Guide* - Intel
- *PCI BIOS Specification*, 1994 – 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 3-2, "SAL System Table Header"

| Date of Revision | Description |
|---|---|
| February 1998 | Initial definition. |
| August 1998 | Defined NVM record formats, changes to SAL procedures. |
| June 1999 | Defined handoff to EFI, Removed NVM functionality. |
| January 2000 | Changes to some SAL procedure definitions. |
| July 2000 | Reflected changes in MCA handling due to PAL MCA changes. |
| January 2001 | MCA related changes, Platform Error definition. |
| 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. |

# Platform Requirements 2

## 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 |
| 0xFF00_0000 | SAL Resources |

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 (NVM), 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 processor stepping independent.
- PAL_B which is processor stepping dependent.

  These two subcomponents are required and must be separated logically even if they are physically located in contiguous spaces. 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:

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

  Unlike the PAL, the SAL subcomponents need not be separated from each other logically or physically.

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

Code in the PAL_A 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 (e.g. PAL_RESET) are determined in a PAL implementation-dependent manner.
- Code in the SAL_A 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 held in compressed format. 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 held in compressed format.

## 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.

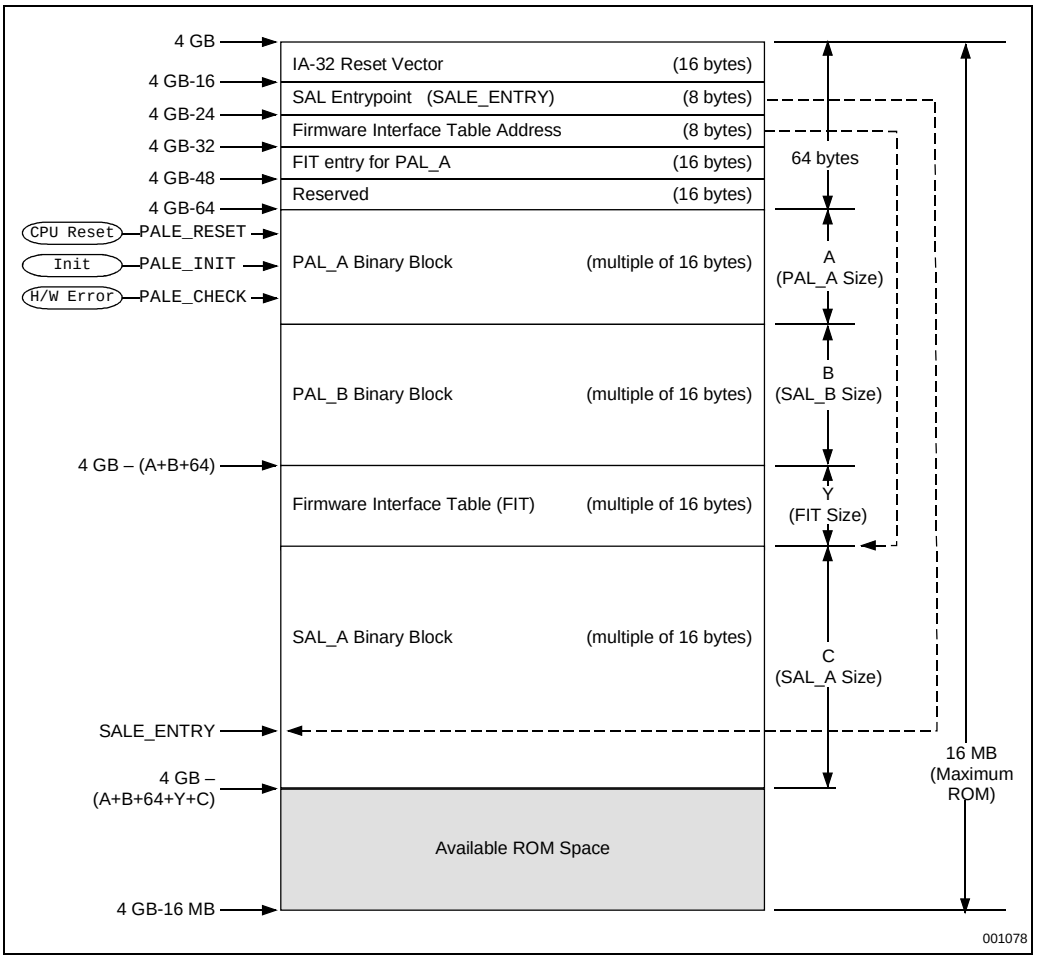
## 2.4 Firmware Organization using a Protected Boot Block

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.

To support the protected boot block, both the PAL_A code and SAL_A code must be within the protected boot block of the flash. The SALE_ENTRY entrypoint must be located in the SAL_A part of the protected boot block.

**Figure 2-1. Simplified Firmware Address Map**



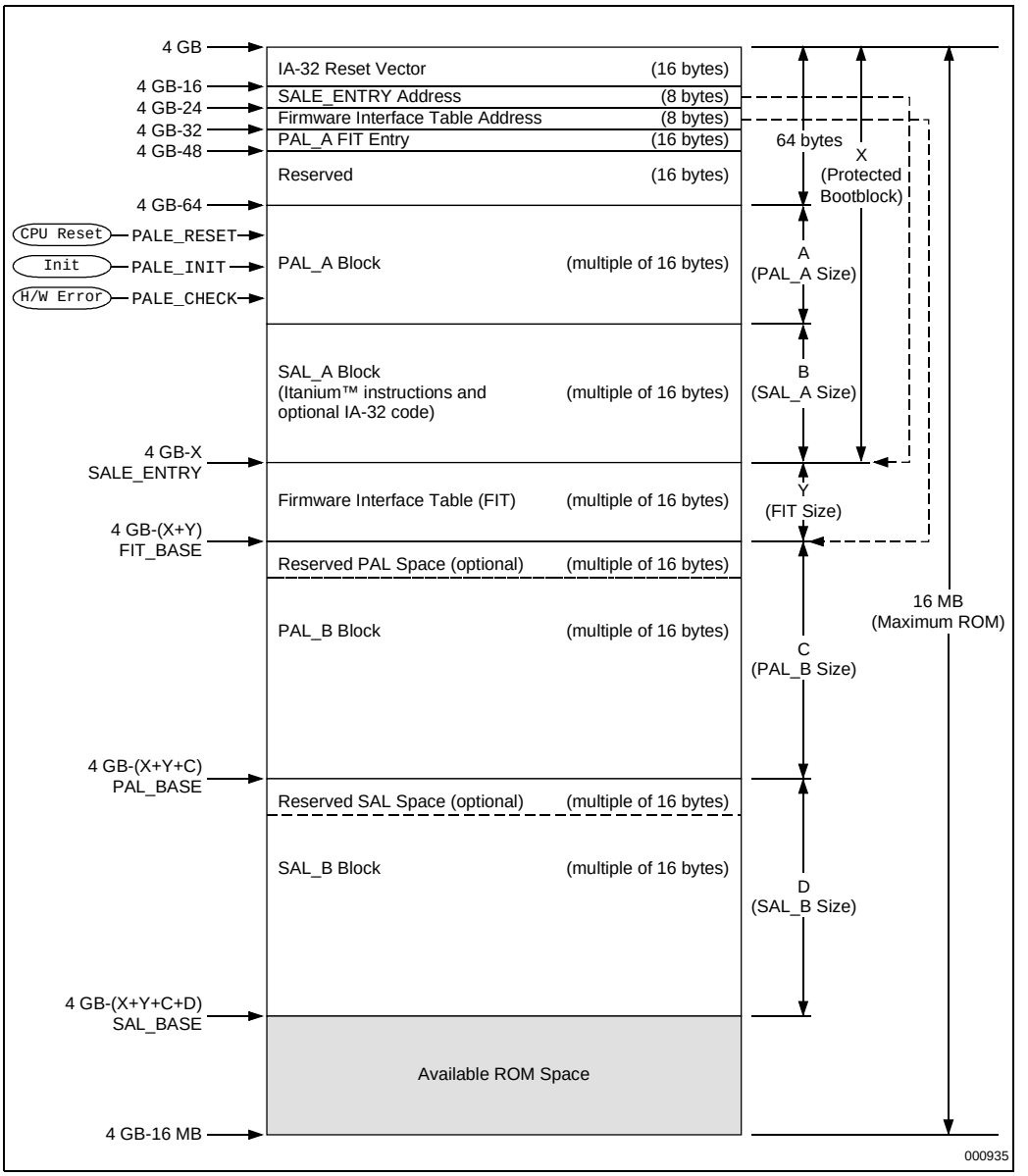

## 2.4.1 Firmware Components

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 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 PC-AT* compatibility. Some IA-32 operating systems may need the information in this area such as the date, the PC-AT model signature, etc.

## Figure 2-2. Firmware Address Map



- 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 in the ROM (base address, size, version number, type, etc.) This is represented in the FIT entry format for the sake of uniformity. 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 16 bytes at (4 GB – 64) are reserved for future use.
- The PAL_A code resides below the (4 GB – 64) address. This variable size area contains the hardware-triggered entrypoints (PALE_RESET, PALE_INIT, and PALE_CHECK) and minimal processor initialization code. This code area 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.

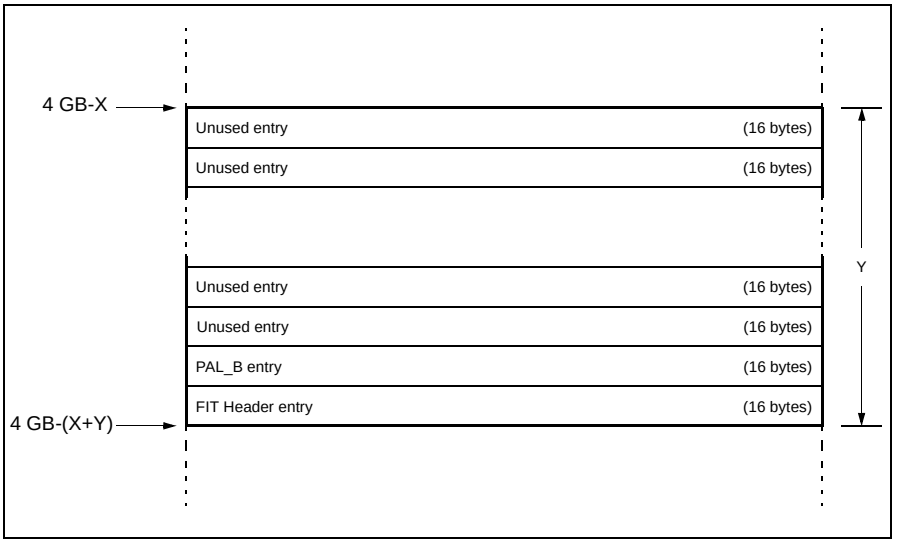
  The code in the PAL_A block contains enough capability to initialize the processor, invoke the SALE_ENTRY procedure for test of the recovery indication, and continue with normal PAL execution in the PAL_B code area. This code is processor stepping independent.
- SAL_A code occupies the bottom of the 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.

  The sizes of the PAL_A and SAL_A code blocks shown in Figure 2-2 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.
- Underneath the protected boot block is the FIT. It consists of 16-byte entries containing starting address and size information of the remaining firmware components in the non-recovery portion of the flash ROM – PAL_B, SAL_B, etc. Refer to Section 2.5 for FIT details.
- Underneath 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 the 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.
- Code within SAL (SAL_A & 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

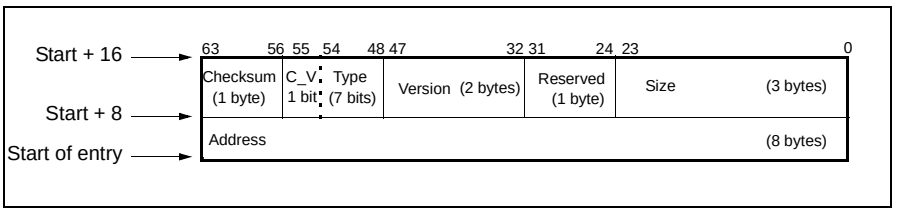
The Firmware Interface Table contains starting addresses and sizes for the firmware components that are outside the protected boot block. Because these code blocks may be compiled at different times and places, code in one block (such as PAL_A) cannot branch to code in another block (such as PAL_B) directly. The FIT allows code in one block to find entrypoints in another. The Figure 2-3 shows the FIT layout.

**Figure 2-3. 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 2-4.

**Figure 2-4. 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 PAL_B entry, 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 element. Types are defined in Table 2-2.

**Table 2-2. FIT Types**

| Type | Meaning |
|---|---|
| 0x00 | FIT Header entry |
| 0x01 | PAL_B |
| 0x02-0x0E | Reserved |
| 0x0F | 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, etc., 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 PC-AT* 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 PC-AT compatibility is not required.

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.

**Table 2-3. 1 MB Compatibility Memory Address Space**

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

Itanium-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 PC-AT compatible components. If it is necessary to support such IA-32 code, Itanium-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 or

the special hardware that contains the firmware code and data is implemented on the processor and not accessed across the system fabric.

- 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 are not required to 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 architecture 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

Different 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 3-7), 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 Geographic Location

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

- The platforms shall provide mechanisms to generate unique geographic identifiers for those components that have software visibility. As an example, imagine a complex multiprocessor implementation which has more than one main system bus to which processors are attached. A processor returns its location on the bus via a call to PAL_FIXED_ADDR, but this PAL call does not reflect the multi-bus 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, etc. may also have a similar requirement.

  Platforms may implement unique ways of providing the SAPIC EID value. For example, in a non-clustered environment, SAL may use the hardcoded value of 0 for this field. Another example is a cluster controller that provides different EID values for processors connected to different buses on 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 cluster 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-based platform hardware must provide a minimum of 32KB of NVM to hold the error log captured during uncorrected machine check events. There may be additional NVM requirements to hold information on the operating systems that can be booted from the platform, the platform configuration, etc. Refer to the *EFI Specification* for requirement details as well as the interfaces to the NVM space.

The NVM must preserve memory contents when the system power is off. Possible NVM implementations are battery-backed SRAM and flash memory. The physical address and size of each NVM object in the system will be specified in Table 3-5, “Memory Descriptor Entry” 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 NVM 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-based platforms must provide an implementation-specific hardware mechanism to reflect the user selected *firmware recovery condition* to all the processors on the platform.
- Itanium-based platforms must support simple hardware or software implementations for BSP selection, e.g. write once port. This is necessary since only the BSP is allowed to execute the firmware recovery code.
- Itanium-based platforms must provide mechanisms to determine the base frequency of the platform (clock input to the processor).
- Itanium-based platform hardware must provide a mechanism for firmware to reset all components within the platform.
- Itanium-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-based platform hardware must provide user friendly mechanisms for displaying the progress of the boot and firmware recovery, e.g. LCD display.

# Boot Sequence 3

## 3.1 Overview of the Code Flow after Hard Reset

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

On reset, the processor(s) begin execution at PALE_RESET, an entrypoint within the PAL_A code area near 4 GB in the firmware address space. The exact physical location of PALE_RESET is processor 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 selected firmware recovery in a platform dependent manner.

SALE_ENTRY is the common 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 4 GB – 24. The state of the processor on entry into SALE_ENTRY is described in the *Intel® Itanium™ Architecture Software Developer's Manual*. One of the general registers 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 within a spin loop in PAL. |
| Healthy | Proceed with SAL Reset. |
| Performance Restricted | Proceed with SAL Reset if this is the only processor on the system. Else, try to inform the user. The processor may be used as 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 missing. |

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

- The platform indicates a recovery condition.
- 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 the new binary using code in the PAL_A block. 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 processor stepping independent and must not invoke code in the PAL_B block.

In a multi-processing environment, the recovery code will first select a BSP. SAL shall not select a processor as the BSP unless it is reported as healthy or performance restricted by PAL and the version of PAL_B on 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 on the system must not be woken up until the PAL_B component is updated, otherwise the system behavior is unpredictable.

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). Further, if the 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 Normal Code 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 on the system, using the return address provided by PALE_RESET to effect the second stage of processor test and initialization. If SAL_A did not effect such a return, the processor may run in a degraded mode. In any case, the PAL_PROC address provided to SALE_ENTRY at the time of *Recovery Check* supports only a small 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 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 EFI firmware, which loads the first level of operating system loader. SAL_RESET may also be entered from SAL_INIT if an OS_INIT handler was 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. In other words, GR32 must be non-zero if SALE_ENTRY is entered from locations other than PALE_RESET.

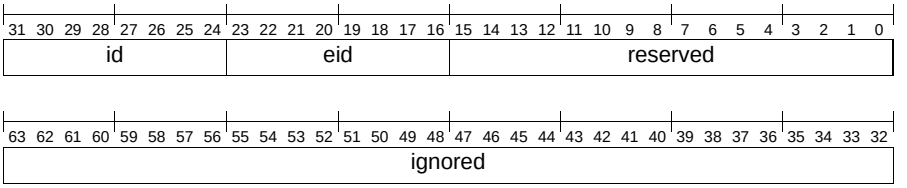
SAL_RESET functionality can be subdivided into the following phases:

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

### 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 architected in the *Intel® Itanium™ Architecture Software Developer's Manual*. It is the SAL's responsibility to uniquely initialize this register in each 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**




The *id* field is provided by the PAL during Reset handoff in a general register. This value is the *Bus Agent ID* which corresponds to the slot number on the front side bus that the processor is plugged into. For proper functioning of the lowest priority interrupt mechanism, the *id* field must match the *Bus Agent ID*. Otherwise, interrupts will be redirected to the wrong or non-existent processors.

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 space 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 both 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 an Multiprocessor Configuration

This phase is executed on all the processors. All processors may participate in the selection of the BSP. 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 derivation 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 on the system is compatible with the processor stepping.

Refer to Figure 3-2 for SAL processing steps in an multiprocessor configuration. The APs will set up processor-specific resources such as the Interrupt Vector Address (IVA) and wait in the rendezvous state (EM_Rendezvous_1 in Figure 3-2) until the SAL on the BSP wakes them up for further processing. Processors in 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 and when sufficient amount of memory has been tested, it will send a rendezvous interrupt to the APs to wake them up to run their late self-test (which requires memory to run). After the APs have finished the late self-test, they will return to the rendezvous state (EM_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 *EFI Specification*.

#### 3.2.2.1 Rendezvous Functionality

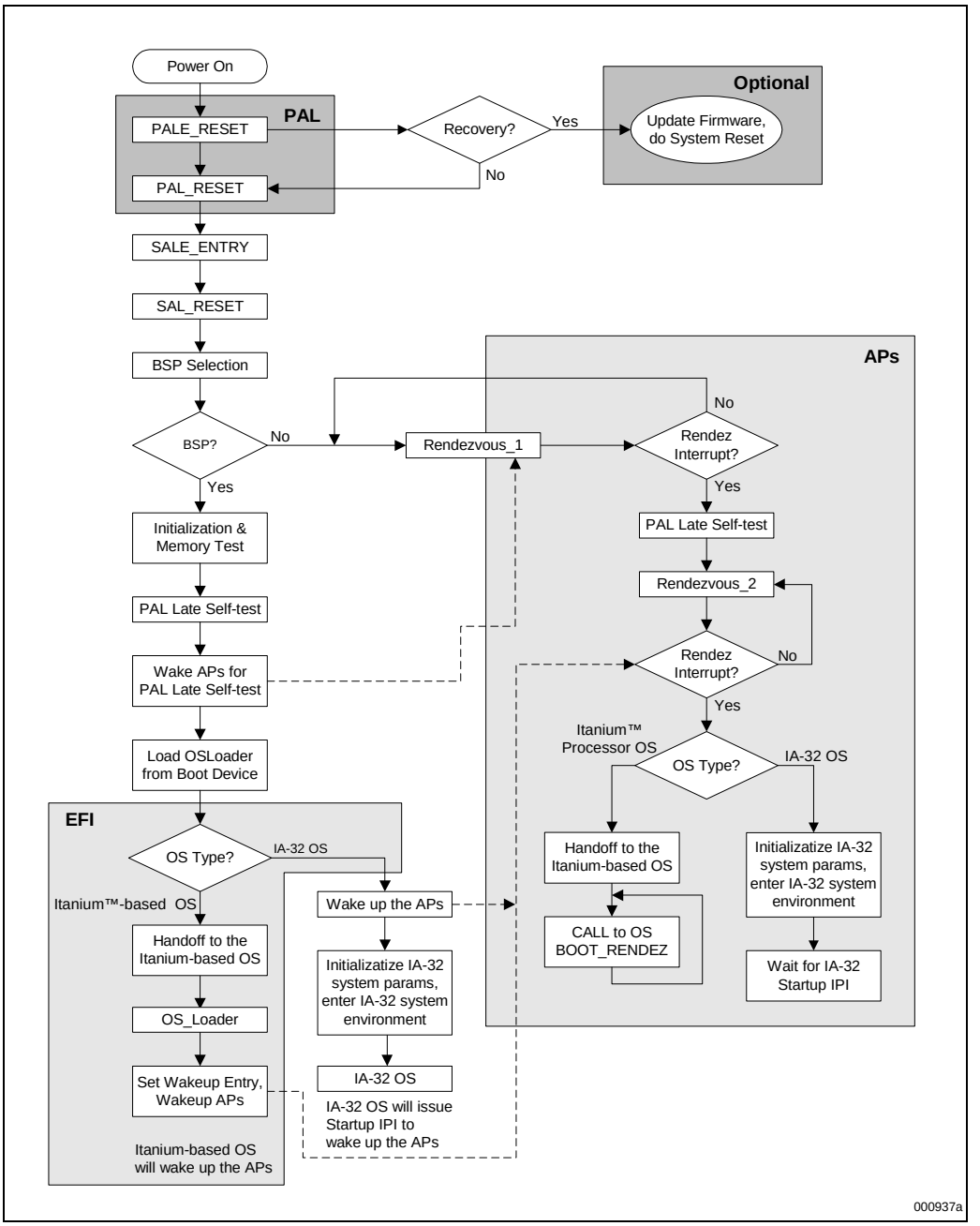
The rendezvous functionality is required only in multiprocessor environments and this functionality is utilized in two different situations:

- 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 3.2.6 and Table 3-11).
- 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 9-15 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.

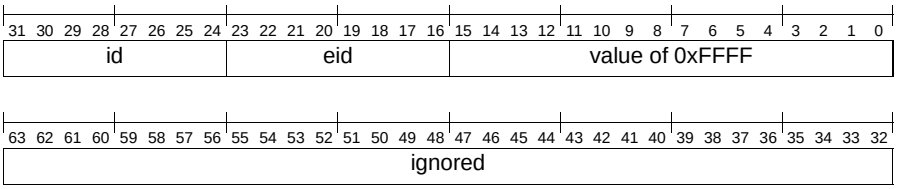
If external interrupt mechanism is chosen, APs will disable interrupts and poll the local SAPIC IRR register for the bit corresponding to the selected rendezvous interrupt to be set. The Task Priority Register (TPR) must be set such that a read of the IVR register will return the rendezvous interrupt 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 3-2) 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 f 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 as 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. If SAL_RESET was entered from SAL_INIT, memory shall not be re-initialized. 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 a performance restricted processors on the system.

   After this point, the memory stack and RSE can be tested and enabled in the Itanium processor 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 (EM_Rendezvous_2 in Figure 3-2). After this stage, caches may be relied upon.

5. Search for console using implementation-dependent algorithms. If found, initialize the console so that the progress of the boot may be displayed.
6. Determine and initialize memory. This step is not performed if SAL_RESET is entered from SAL_INIT. The memory test is implementation dependent. The memory test includes testing of refresh logic and testing all the address lines for shorts. On IA-32 systems, memory controllers alias the ROM at 0xE0000 to 0xFFFFF and thereby permit memory autoscan algorithm to be run from the aliased ROM at 0xE0000 to 0xFFFFF. Since memory aliasing is not a requirement for Itanium-based platforms, the autoscan function needs to be performed by the firmware SAL code in the ISA for Itanium instructions.
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 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 very desirable 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 *Intel® Itanium™ Architecture Software Developer's Manual* for PAL's requirements on ITR/DTR.
   ***Note:*** Until this step, the following floating-point transcendental instructions cannot be executed from the IA-32 instruction set:
   - 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 specifying unique memory areas where the PAL code may deposit some minimal processor state information. Additionally, these memory 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 *EFI 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 common 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, Chapter 10 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 ISA.
18. Copy the EFI code into memory and transfer control to it. Branch register B0 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 *EFI Specification* for interface description.

### 3.2.4 Operating System Boot Phase

This phase is executed only on the BSP. Refer to the *EFI Specification* for details of booting Itanium-based operating systems. If the selected operating system is a legacy IA-32 operating system, SAL does the following:

1. SAL will construct an MP Information Table that provides the mapping between the I/O SAPIC ID, EID values and the I/O APIC ID value for use by the legacy IA-32 operating system. This table is provided as a parameter to the PAL_ENTER_IA_32_ENV procedure.

   SAL will assign unique 4-bit *id* values for the Local APIC entries of the MP table based on the 16-bit *eid*, *id* fields of the corresponding Local SAPIC entries. The IDs assigned by SAL are suitable for the physical destination mode of the Local APIC. SAL will permit use of a maximum of 16 processors while booting a legacy IA-32 operating system. SAL will keep any additional processors in a loop within SAL and these processors shall not invoke the PAL_ENTER_IA_32_ENV procedure.

   SAL will assign unique 4-bit *id* values for the I/O APIC entries of the MP table based on the 16-bit *eid*, *id* fields of the corresponding SAPIC entries. The *id* values assigned by SAL for the Local APIC and the I/O APIC entries may overlap.

   SAL will provide the physical address of non-existent memory of a minimum of 4K bytes. This area will be specified in the Memory Descriptor Table (Table 3-5) with the *Memory type* classification of *Non-existent Memory.*
2. The PAL_ENTER_IA_32_ENV procedure also enables SAL to emulate some I/O ports not present on the platform. SAL conveys information on the emulated ports in the SAL I/O Intercept Table. Refer to Volume 2, Chapter 11 of the *Intel® Itanium™ Architecture Software Developer's Manual* for details.
3. Construct Memory Descriptor Table entries suitable for the platform.
4. Load one sector of the Master Boot Record (MBR) code from the boot device at address 0x7C00. Verify that the last two bytes of the sector end with 0x55 0xAA.

***Note:*** In this document, the term *sector* refers to a logical block of 512 bytes.

5. Determine the amount of memory needed by PAL in support of IA-32 operating systems by invoking PAL_COPY_INFO procedure and allocate the same with the requested alignment. Transition the processor to the IA-32 system environment and jump to the MBR code loaded at 0:7C00. This switch is effected by calling PAL_ENTER_IA_32_ENV procedure. (Refer to the *Intel® Itanium™ Architecture Software Developer's Manual*.) The return address in

SAL and the address of SAL_PROC are passed as a parameter to this call. SAL shall set the initial IA-32 stack to 0:0x7c00 (SS:ESP).

This PAL procedure will set up the appropriate memory attribute values based on the Memory Descriptor Table (Refer Table 3-5). If the IA-32 operating system exits by executing a JMPE instruction, PAL will return to the return address in SAL. When SAL regains control, it will de-allocate the memory allocated to PAL in support of IA-32 operating systems and attempt to boot a different operating system.

6. Some additional parameters are needed in an MP environment. The PAL_ENTER_IA_32_ENV procedure requires an input flag that indicates whether the call is being made on the BSP or APs and a count of the processors that have already been transitioned to the IA-32 system environment. Also, the PAL_ENTER_IA_32_ENV procedure requires that the first processor reach the IA-32 starting address before subsequent processors invoke the procedure.

   SAL implementation is simpler if the BSP transitions to the IA-32 system environment last. For example, the BSP can instruct APs to call the PAL_ENTER_IA_32_ENV procedure, one at a time. The APs will specify a starting address within the first MB of memory. The IA-32 code at this location will perform the check-in to inform the BSP that the transition to IA-32 system environment is completed, disable interrupts and go into a spin loop awaiting the Startup IPI from the BSP.

   Once all the APs have transitioned to the IA-32 system environment and checked in, SAL on the BSP will invoke the PAL_ENTER_IA_32_ENV procedure and specify the starting address as 0:7C00 where the MBR code from disk has been loaded. The PAL_ENTER_IA_32_ENV procedure will typically set the processor resources of the APs such that all processors have an identical view of the platform's memory attributes.

   The IA-32 operating system would be loaded eventually and this will send APIC INIT IPIs followed by APIC Startup IPIs to the APs. PAL's APIC emulation layer on the BSP will trap the APIC ICR writes and will eventually transition the APs to the starting address corresponding to the vector specified in the Startup IPI.

### 3.2.5 Firmware to Operating System Loader Handoff State

The handoff to an IA-32 operating system is compatible with the PC-AT industry standards. The handoff from firmware to Itanium-based operating system loaders is fully described in the *EFI Specification*. Included in the handoff are:

- The pointer to the SAL System Table (Section 3.2.7).
- The pointer to the Root System Description Pointer as described in the *Advanced Configuration and Power Interface Specification.*

The state of Itanium processor system registers at the time of handoff to the operating system loader is as follows:

- AR contents are SAL implementation-dependent except the following:
  - CFM: The backing store shall contain a minimum of 8 KB of available storage space defined in the SAL Boot Services data area.
  - RSC will indicate enforced lazy mode, little-endian.
  - IOBASE (AR.k0) will contain the virtual address of the IA-32 I/O port block.

- GR contents are SAL implementation-dependent except:
  - GR12 = Stack pointer with a minimum of 8 KB of available storage space defined in the SAL Boot Services data area.
- 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 but will likely contain identity mappings (virtual address to physical address).
- Caches
  Enabled, coherent and consistent with the contents of memory.

### 3.2.6 OS_BOOT_RENDEZ

OS_BOOT_RENDEZ is the entrypoint for operating system-dependent MP rendezvous code. The operating system code on the BSP registers this entrypoint by invoking SAL_SET_VECTORS, supplying the physical address of the operating system code that is 16-byte aligned. SAL exports details of the wake-up mechanism to the operating system through the SAL System Table (refer to Table 3-11) 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 state of the Itanium processor system registers at the time of handoff to the OS_BOOT_RENDEZ is similar to that for the BSP with the following exception:

- B0 = Return address into the SAL Boot_Rendezvous routine. If the OS_BOOT_RENDEZ returns to the SAL using the Branch register B0, the SAL will re-enter the spin loop awaiting a wake-up by the BSP.

## 3.2.7 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 *EFI Specification* for handoff 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 3-2. 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 3-3 describes each entry type.

**Table 3-2. SAL System Table Header**

| Field | Offset (in Bytes) | Length (in 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 SAL Specification* supported by the SAL implementation in binary coded decimal (BCD) format.<br>Byte 8 – Minor<br>Byte 9 – Major<br>SAL Revision 3.0 corresponds to SAL Specification, January 2001 or July 2001.<br>SAL Revision 2.9 corresponds to SAL Specification, July 2000.<br>SAL Revision 2.8 corresponds to SAL Specification, 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.<br>Byte 20 – Minor<br>Byte 21 – Major |
| SAL_B_VERSION | 22 | 2 | Version Number of the SAL_B firmware implementation in BCD format.<br>Byte 22 – Minor<br>Byte 23 – Major |

**Table 3-2. SAL System Table Header (Cont'd)**

| Field | Offset (in Bytes) | Length (in 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. |

Following are the entry types of entries that follow the SAL System Table Header. Unless otherwise stated, there is one entry per entry type.

**Table 3-3. SAL System Table Entry Types**

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

a. All other types are reserved.
b. Not required for Itanium-based operating systems.

### 3.2.7.1 Entrypoint Descriptor Entry

The Entrypoint Descriptor Entry (refer to Table 3-4) provides the addresses in memory of PAL_PROC, 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 3-4. 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. |
| 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). |

### 3.2.7.2 Memory Descriptor Table Entry

The Memory Descriptor Table (MDT) entries (refer to Table 3-5) are used only while booting an IA-32 operating system. Itanium-based operating systems obtain similar information from the EFI firmware component. The MDT entries describe all the main memory, firmware memory, memory mapped I/O, etc., in the system address space as well as the memory attributes currently set by SAL. Each contiguous block with similar memory attribute (WB, WC, UC or UCE) must be aligned on a 64 KB boundary as a minimum, for optimal TLB management. Note that memory usage values (byte 7 of the MDT entry) may change within a 64 KB memory block and hence it is legal to have more than one MDT entry describing a 64 KB memory region as long as the memory attribute (WB, WC, UC or UCE) does not change within that 64 K block.

SAL must provide entries that cover the entire system address space. The firmware must indicate its memory usage in order that the same may be not trampled by the operating system. Thus, if the SAL uses an underlying IA-32 BIOS layer for part of its functionality, it must report memory usage for the real mode interrupt vector table (0-0x3FF), the BIOS Data area (0x400-0x4FF) and the Extended BIOS Data area (downwards from 640 K) as Boot Services Data in the Memory Usage field of the MDT entries.

The EFI firmware component communicates the SAL's requirements for virtual address mappings to the operating system. Once the operating system takes control of the memory management and the IVA, it must provide TLB mappings for both the code and data accesses to the memory areas required by SAL, if those areas are accessed in virtual mode. The operating system must register these virtual addresses prior to invoking SAL procedures in virtual mode.

**Table 3-5. Memory Descriptor Entry**

| Offset (in Bytes) | Length (in Bytes) | Description[a] (Unsigned Integers) |
|---|---|---|
| 0 | 1 | Entry type = 1 denoting Memory Descriptor entry type. |
| 1 | 1 | Need virtual address registration for SAL operation in virtual mode:<br>0: No<br>1: Yes |
| 2 | 1 | Encoded value of current Memory Attribute[b] setting in bits 0-2:<br>000: WB<br>100: UC<br>101: UCE<br>110: WC |
| 3 | 1 | Page Access Rights set up by SAL for the memory range[b]: |
| 4 | 1 | Memory Attributes[b] supported:<br>Bit 0: WB<br>Bit 1: UC<br>Bit 2: UCE<br>Bit 3: WC |
| 5 | 1 | Reserved (must be zero). |

**Table 3-5. Memory Descriptor Entry (Cont'd)**

| Offset (in Bytes) | Length (in Bytes) | Description[a] (Unsigned Integers) | |
|---|---|---|---|
| 6 | 2 | Memory Type (byte 6)<br>0 = Regular Memory | Memory Usage (byte 7)<br>0 = Unspecified[c]<br>1 = PAL Code<br>2 = Boot Services Code<br>3 = Boot Services Data<br>4 = Runtime Services Code<br>5 = Runtime Services Data<br>6 = IA-32 Option ROM<br>7 = IA-32 System ROM<br>8 = ACPI Reclaim Memory[d]<br>9 = ACPI NVS Memory<br>10 = SAL PMI Code<br>11 = SAL PMI Data<br>12 = Firmware Reserved Memory[e]<br>128-255 = Reserved for OEM |
| | | 1 = Memory mapped I/O | 0 = Unspecified<br>1 = I2O Hidden space hole<br>2 = Video Memory<br>3-127 = Reserved<br>128-255 = Reserved for OEM |
| | | 2 = SAPIC IPI Block | 0 = Unspecified |
| | | 3 = IA-32 I/O Port space | 0 = Translated by processor to I/O cycles |
| | | 4 = Firmware address space | 0 = Unspecified |
| | | 9 = Bad Memory | 0 = Unspecified |
| | | 10 = Non-existent Memory (Black hole) | 0 = Unspecified |
| 8 | 8 | Physical Address of Memory | |
| 16 | 4 | Length (multiple of 4K pages) | |
| 20 | 4 | Reserved (must be zero) | |
| 24 | 8 | OEM Reserved | |

a. All unused values are reserved.
b. Refer to the *Intel® Itanium™ Architecture Software Developer's Manual,* for explanation of this field.
c. Refer to the EFI Specification for the usage description of this memory space.
d. This memory is available to the operating system after it reads the *Advanced Configuration and Power Interface Specification* tables.
e. This area is not visible in the IA-32 operating system environment.

The SAL also provides the memory type and usage information to the EFI. Refer to the *EFI Specification* for details. Table 3-6 specifies the mapping between MDT entries and the information provided by the SAL to the EFI.

**Table 3-6. Memory Type Information Provided to the EFI**

| Memory Type | Memory Usage | EFI Memory type |
|---|---|---|
| 0 = Regular Memory | 0 = Unspecified<br>1 = PAL Code<br>2 = Boot Services Code<br>3 = Boot Services Data<br>4 = Runtime Services Code<br>5 = Runtime Services Data<br>6 = IA-32 Option ROM<br>7 = IA-32 System ROM<br>8 = ACPI Reclaim Memory<br>9 = ACPI NVS Memory<br>10 = SAL PMI Code<br>11 = SAL PMI Data<br>12 = Firmware Reserved Memory<br>128-255 = Reserved for OEM | EfiConventionalMemory<br>EfiPalCode<br>EfiBootServicesCode<br>EfiBootServicesData<br>EfiRuntimeServicesCode<br>EfiRuntimeServicesData<br>EfiRuntimeServicesCode<br>EfiRuntimeServicesCode<br>EfiACPIReclaimMemory<br>EfiACPIMemoryNVS<br>EfiRuntimeServicesCode<br>EfiRuntimeServicesData<br>EfiRuntimeServicesData<br>EfiRuntimeServicesCode |
| 1 = Memory mapped I/O | <all values> | EfiMemoryMappedIO if virtual address registration is required, otherwise information not provided to the EFI. |
| 2 = SAPIC IPI Block | 0 = Unspecified | Information not provided to the EFI. |
| 3 = IA-32 I/O Port space | 0 = Translated by processor to I/O cycles | EfiMemoryMappedIOPortSpace. |
| 4 = Firmware address space | 0 = Unspecified | EfiRuntimeServicesData. |
| 9 = Bad Memory | 0 = Unspecified | EfiUnusableMemory. |
| 10 = Non-existent Memory (Black hole) | 0 = Unspecified | Information not provided to the EFI. |

### 3.2.7.3 Platform Features Descriptor Entry

The Platform Features Descriptor Entry (refer to Table 3-7) describes the features implemented on the platform. Refer to the *Itanium™ Platform Architecture Guide* for implementation considerations of these platform features.

**Table 3-7. 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:<br>Bit 0: 1 if Bus Lock is implemented on the processor as well as the platform.<br>Bit 1: 1 if the chipset supports redirection hint for interrupt messages originating from the platform (lowest priority interrupt).<br>Bit 2: 1 if the chipset supports redirection hint for IPI messages originating from the processors.<br>Bits 3-7 = Reserved. |
| 2 | 14 | Reserved. |

### 3.2.7.4 Translation Register Descriptor Entry

The Translation Register Descriptor entries (refer to Table 3-8) 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.

**Table 3-8. 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:<br>0: Instruction Translation Register<br>1: Data Translation Register<br>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. |

### 3.2.7.5 Purge Translation Cache Coherence Domain Entry (Optional)

The purge translation cache (PTC) Coherence Domain Entry (refer to Table 3-9) describes the number of coherence domains and the scope of PTC instruction propagation for each domain. This entry is optional. It is required only for MP systems that have multiple coherence domains.

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.

**Table 3-9. 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. |

The coherence domain information is an array of length of (16*Number of coherence domains). As shown in Table 3-10, 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 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.

This information is represented in Table 3-10.

**Table 3-10. Coherence Domain Information**

| Offset (in Bytes) | Length (in Bytes) | Description |
|---|---|---|
| 0 | 8 | Number of processors in TLB coherence #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 #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 #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. |

### 3.2.7.6 Application Processor Wake-up Descriptor Entry (Optional)

The AP Wake-up Descriptor Entry (refer to Table 3-11) 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 3-11. 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:<br>0: External interrupt<br>Other values: Reserved |
| 2 | 6 | Reserved (must be zero). |
| 8 | 8 | External Interrupt vector in the range of 0x10 to 0xFF. |

## 3.3 Itanium™-based Operating System Loader Requirements

The firmware will jump to the Itanium-based operating system loader with the handoff state described in the *EFI Specification*. Included in this state information is a pointer to the SAL procedures 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 guidelines to the operating system loader code as regards fault handling.

After the operating system is completely loaded, it will take over the IVA, and replace the SAL environment with its own memory management. Until that time, the operating system shall use SAL's virtual memory environment — IVA, Interrupt controller mode, TC mappings, etc., 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 processor 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.
- 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, e.g. 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 processor 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 (i.e. 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 3-11).
- 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 a load, store or move that uses a source register containing a NaT value or by accessing a NaTPage. This fault can be avoided by compiling the operating system loader code with speculation off.
- 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, etc. 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 several TRs by SAL may cause problems with booting of some Itanium-based operating systems. The operating system loader is free to use Translation Registers (TRs) other than ITR(0). 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 (virtual addresses = 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 code is 64-bit, 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 machine check abort 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* bits 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, i.e. the virtual addresses used to address the common data structures must have bits 61-63 set to 0.

### 3.3.3 Other Restrictions on the Operating System

1. The operating system shall not change the values of the following system resources:
   - LID, 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.
   - 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.

# Machine Checks 4

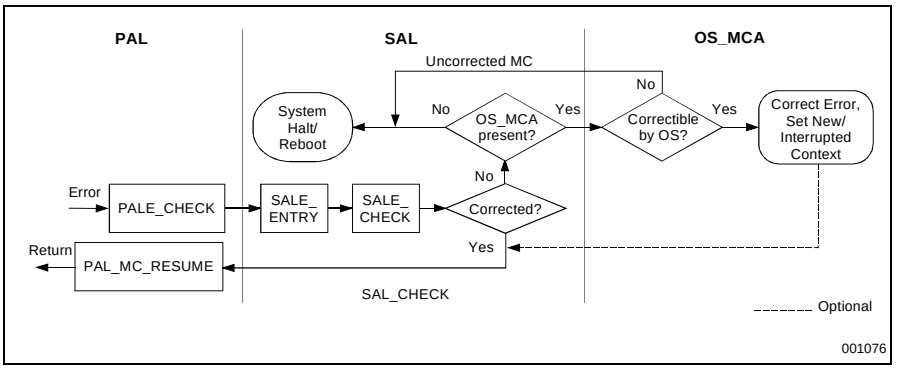
Machine checks, including Machine Check Aborts (MCAs), and expected machine checks cause processor execution to jump to the PALE_CHECK entrypoint in the Itanium architecture. Please refer to Volume 2, Chapter 11 in the *Intel® Itanium™ Architecture Software Developer's Manual* for details regarding PALE_CHECK processing. Also refer to the *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 the Itanium architecture, 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 some 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 multiprocessor (MP) situations.

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 tell PAL code where it may deposit some minimal processor state information so that PAL code has sufficient resources to perform the necessary 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 code (PAL_MC_ERROR_INFO), in order to retrieve processor-specific error information which pertains to the machine check taken. In addition, SAL_CHECK should interrogate the platform for any platform-specific information which pertains to the machine check condition. Once the processor error record information is retrieved, SAL_CHECK will call PAL_MC_CLEAR_LOG to enable the processor error logging resources for capturing future machine check error information. A similar task is necessary to enable platform error logging resources 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. Such a “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_PROC get state information calls so that the operating system can make the determination to:

- Fix the error and return to interrupted or new context through the SAL_MCA, or
- Request the SAL_MCA 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 honor the cacheability attribute of the OS_MCA code. Thus, if the operating system had provided an uncacheable address for the OS_MCA entry point (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 reasons which 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.

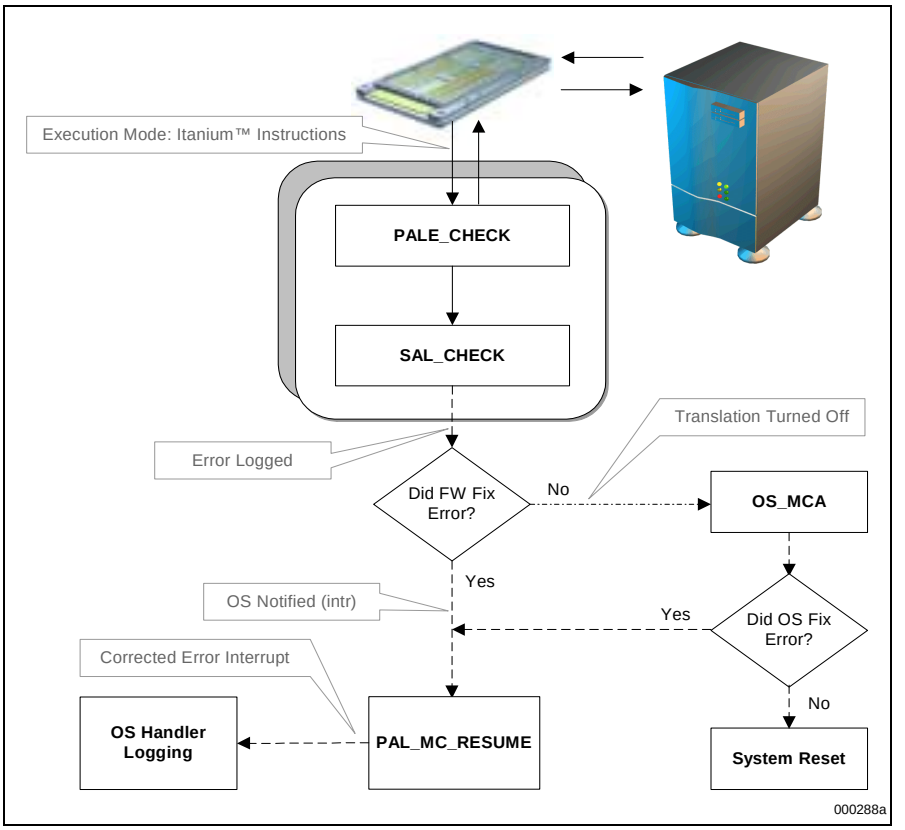
## 4.2 Corrected Machine Checks

There are different categories of corrected machine checks pertaining to Itanium processors:

- Corrected internally by the processor hardware, e.g. single bit data ECC error on a processor cache.
- Corrected by PAL, e.g. double bit data ECC error on a clean processor cache line, during an instruction fetch operation.
- Corrected by the platform hardware, e.g. single bit data ECC error on 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.

**Figure 4-2. Machine Check Code Flow**




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 is corrected by PAL, SAL would return to the interrupted context by calling PAL_MC_RESUME. PAL_MC_RESUME procedure provides an option for generating a Corrected Machine Check 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.

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 component that handles the CMC or Corrected Platform Errors 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.

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

## 4.3 Platform Errors

Platform errors refer to errors signalled by system components other than processors, e.g. memory, I/O busses, chipsets, devices, etc.

Uncorrected platform errors may be signalled by asserting pins such as BERR# or BINIT# or by generating a 2xECC[2] or a synchronous HardFail response on the processor front side bus (FSB).

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 & firmware implementation dependent. Depending upon the platform topology, a single physical platform may comprise of 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 number of nodes in a platform is implicitly indicated by the SAL by providing multiple entries for Corrected Platform Error interrupts in the ACPI tables. Refer to the *ACPI 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

1. It is required that the operating system handlers operate with interrupts enabled, so that system firmware can manage its resources (like NVM based error records) without impacting the system performance.
2. Also known as data poisoning.

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 the interrupt line of the IOSAPIC. For firmware-corrected errors, SAL reports the platform-corrected error event to the operating system by sending an inter-processor interrupt to the processor with the CPEV that is registered by the operating system through the SAL_MC_SET_PARAMS procedure.

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

### 4.3.3 Processing of Uncorrected Platform Errors

Uncorrected platform errors will 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 the corrected processor events, the operating system must periodically invoke the SAL_GET_STATE_INFO and the SAL_CLEAR_STATE_INFO procedures on each processor in the system. For the 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 IOSAPIC 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.

## 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 by verifying its checksum. 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 in order 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 the 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 contains relevant data with respect to 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, (e.g. 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 format identical to the Min-State Save area and returning the same 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

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

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.

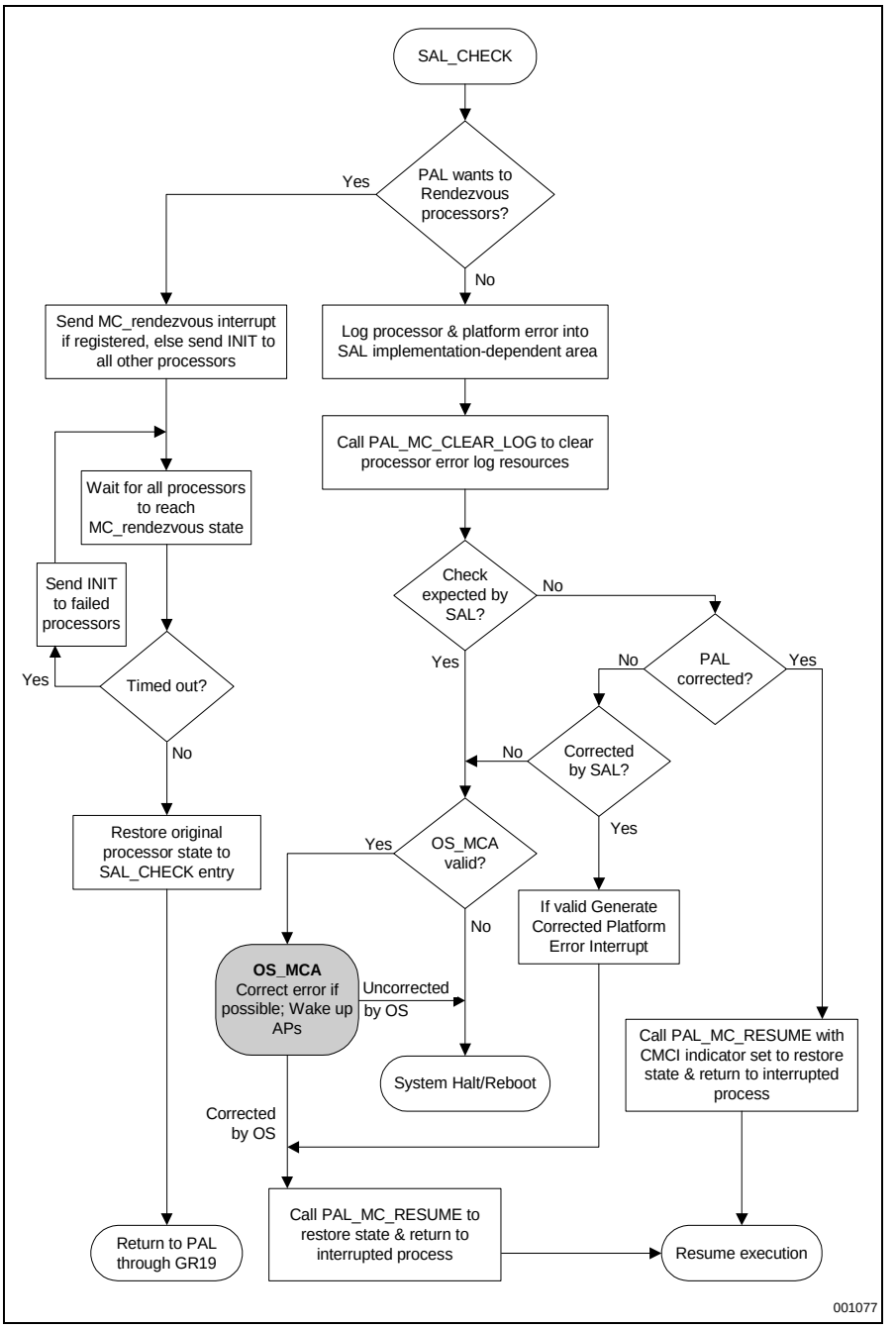
Figure 4-3 shows the flow of control through SAL_CHECK on the monarch processor.

### 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 NVM, it may be capable of providing the unconsumed error information to the operating system following the next reboot of the system. To support this capability, following the next reboot of the operating system, the operating system shall invoke the SAL_GET_STATE_INFO and the SAL_CLEAR_STATE_INFO procedures (with the *type* argument of MCA) to retrieve the pending error records and optionally log them to persistent storage under control of the operating system. These SAL calls to consume the pending error records may be made from any of the processors in the system. For additional details, refer to the *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 NVM 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.

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



## 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

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

## 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 graceful error handling. This is accomplished by bringing all the processors in

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

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 all processors is required by means of a 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 on its own accord 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 overall processing steps are as follows:

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.

4. Interrupt handlers of all slaves 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.

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.

**Figure 4-4. Normal SAL Rendezvous Flow**

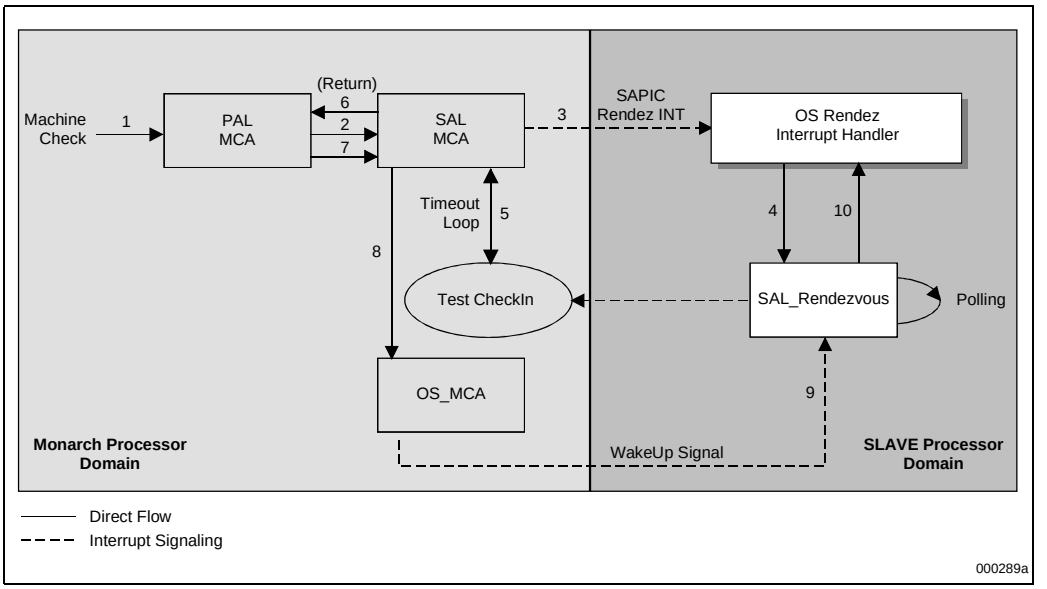
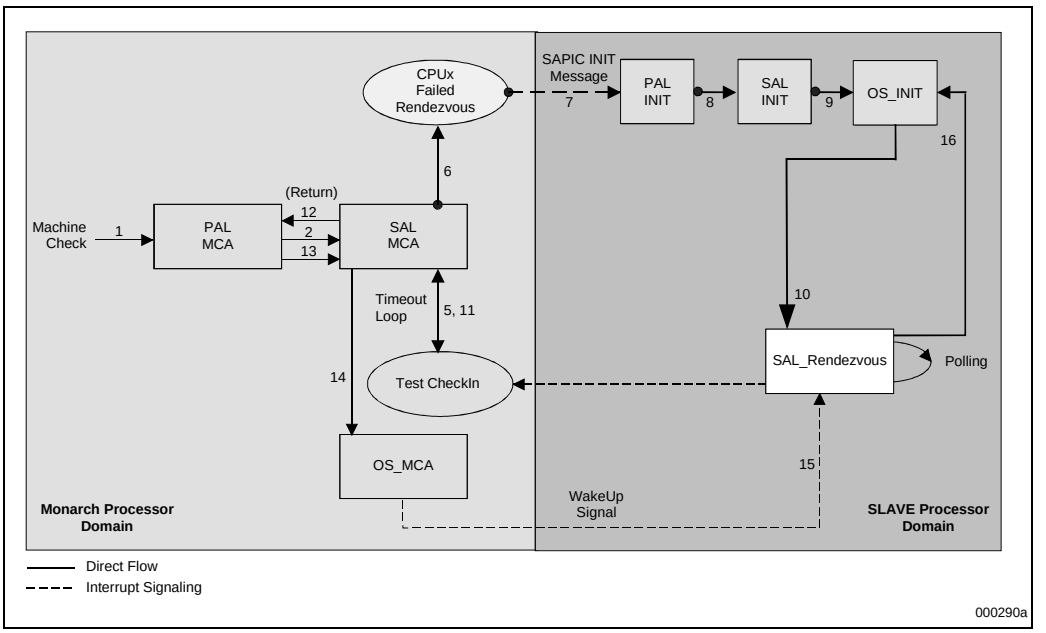


**Figure 4-5. Failed SAL Rendezvous Flow**




## 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. While waiting for the wake-up from the monarch processor, the SAL may mask further machine checks and escalate future MCA and BERR 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 the 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 the Figure 4-5.While sending an INIT to the other processors may not create an inherently unrecoverable situation, it certainly increases the risk for recoverability. 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 to be processed by the targeted processors and reach the rendezvous state.

***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 OS_MCA, 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 some platform implementations, under certain machine check circumstances, 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_MCA, 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.

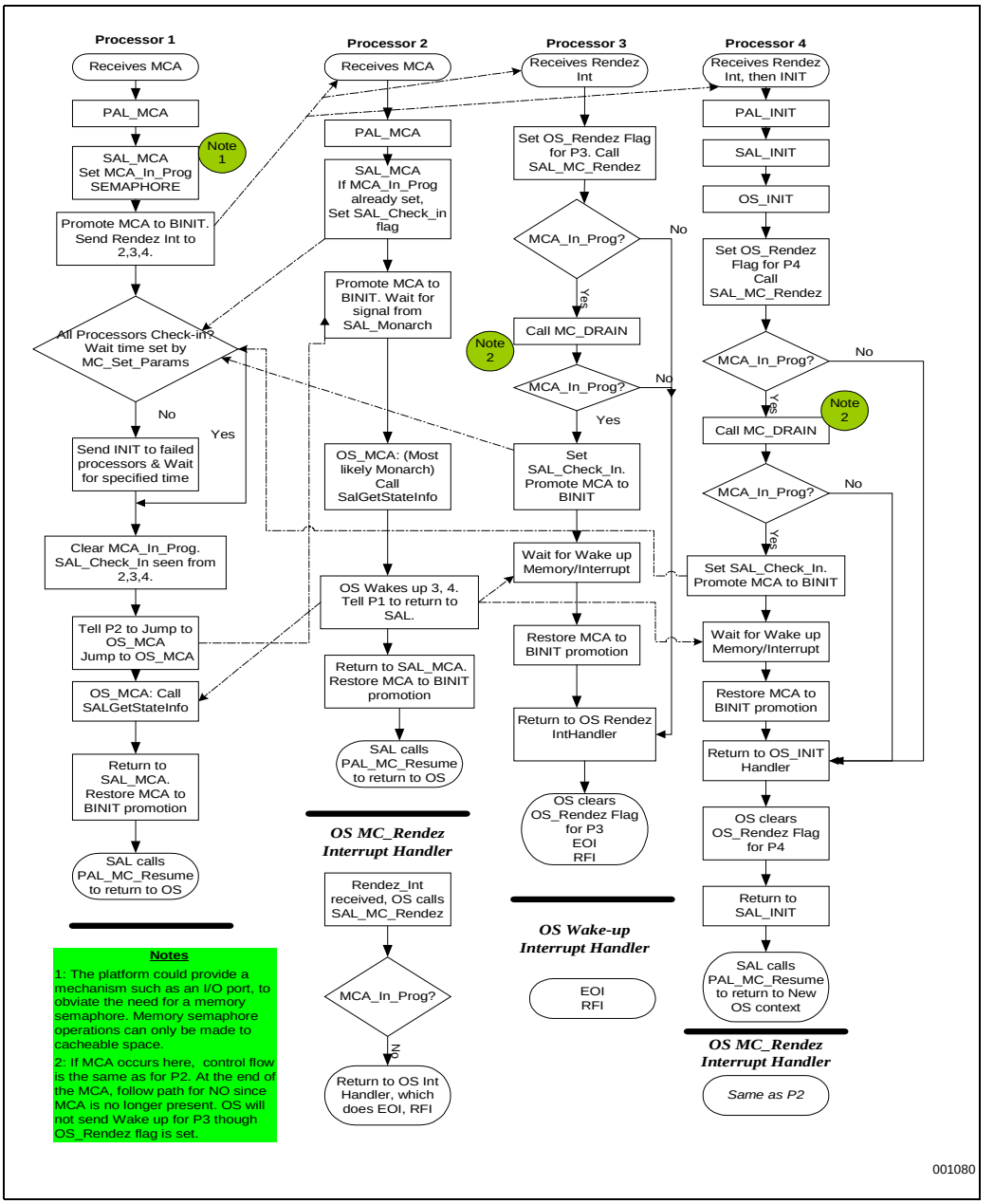
The operating system on the Processor 3 receives the MC_rendezvous interrupt, sets a flag for being rendezvoused in the operating system data structures and then calls the SAL_MC_RENDEZ procedure. The Processor 4 is running with interrupts masked and does not recognize the MC_rendezvous interrupt in 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.

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



## 4.8 OS_MCA Handoff State

The OS_MCA interface defines the boundary between SAL_CHECK and the operating system machine check handler, OS_MCA. The contents of non-banked and banked 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_MCA. The following register contents define the OS_MCA handoff 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 MinState area, otherwise unexpected behavior will occur.

### 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. 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.

# Initialization Event 5

INIT is an initialization event generated by the platform or by software through a SAPIC message. The INIT event causes the processor to branch to the processor-dependent INIT handler (PALE_INIT) in the Itanium architecture. The 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.*

## 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).

The warm boot mechanism is SAL implementation-dependent and can be done either by calling the SAL_RESET entrypoint 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 on 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.

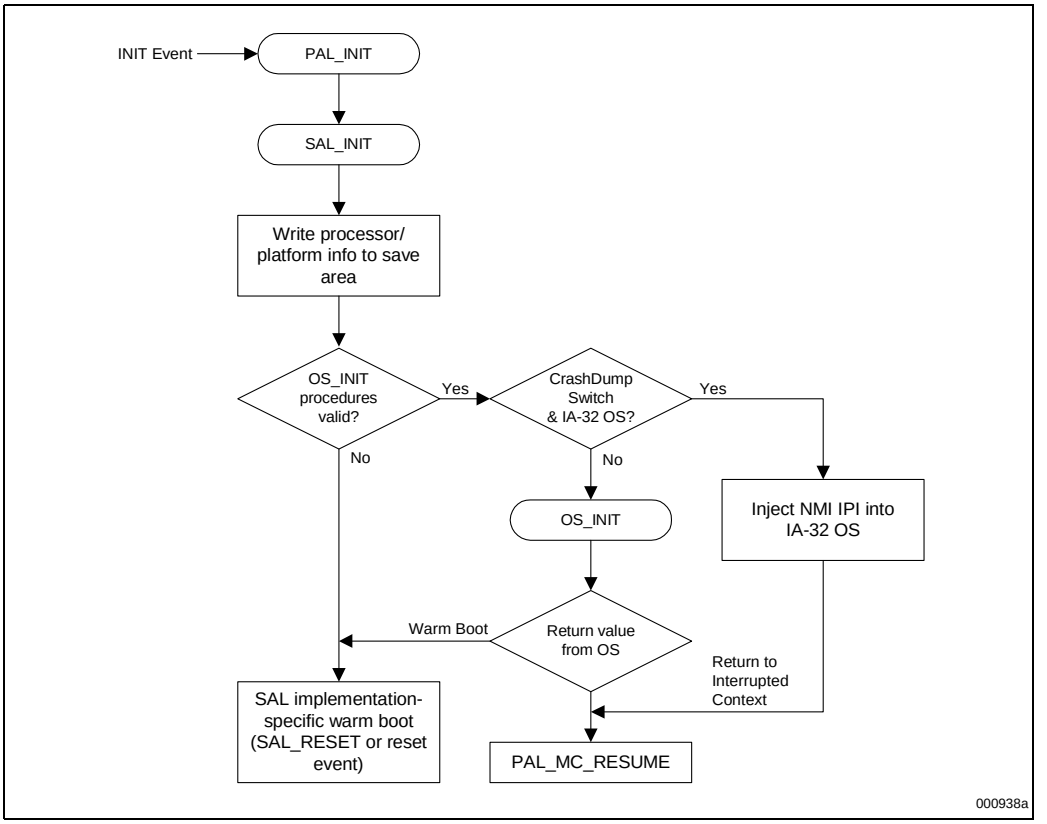
- Check if the OS_INIT handlers for the monarch and non-monarch processors are registered and that both of them are valid. When the OS_INIT procedures were registered with the SAL, the operating system may optionally supply the modulo checksum for the code areas (all bytes of the code area including 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.
- 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.

On some PC-AT platforms, the platform provides a switch that can generate an NMI signal and this is used by IA-32 operating systems to effect a crash dump on a hung system. On Itanium-based systems, a similar function will be performed by an INIT switch as the NMI signal is masked by the PSR.i bit of the processor. If SAL_INIT gains control due to the platform's INIT switch while an IA-32 operating system is executing, the SAL_INIT layer shall send an SAPIC IPI message to the same processor with the interrupt type of NMI and then return to the interrupted context using the PAL_MC_RESUME procedure.

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 Handoff 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 handoff 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 MinState 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.

## 5.4 Return from OS_INIT Procedure

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.<br>–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.<br>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 on the system and then select a BSP for further system initialization. If rebooting is required while running an IA-32 operating system, SAL will use the currently selected BSP for performing the rendezvous of the other processors.

## 5.5 MP INIT Support

There are a few situations when processors enter SAL_INIT in MP configurations which deserve specific mention.

- If a processor enters SAL_INIT and there are no registered OS_INIT handlers for the monarch and non-monarch processors or their checksums are incorrect, then the processor will reset the system (warm boot). In the MP environment, the processor performing the reset shall reset the system, not just itself.

# Platform Management Interruptions 6

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* 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.

Unlike the System Management Interrupt (SMI) on IA32 systems, 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 as regards 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.

## 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 processor 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 B0. 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 on 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 save the interruption resources and set the PSR.*ic* bit to 1 as early as possible.

# IA-32 Support (Optional) 7

## 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 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-based operating system) has taken control of the translation resources. Most Itanium-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 Volume 4, Chapter 8 of the *Intel® Itanium™ Architecture Software Developer's Manual*.

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-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-based platforms may itself be coded in IA-32 ISA (i.e. 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 PC-AT compatibility.

- PC-AT 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.
- PC-AT 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.

- PC-AT 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 PC-AT 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 PC-AT 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 processor 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, etc. 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-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-based 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 processor 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 1MB without requiring a transition to the Itanium processor 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 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 processor 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.

# Calling Conventions 8

## 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 |

## Table 8-2. State Requirements for PSR (Cont'd)

| PSR Bit | Description | Entry | Exit | Class |
|---|---|---|---|---|
| ic | Interruption state collection enable | C | C | Preserved[a] |
| | | 0 | 0 | Unchanged |
| i | Interrupt unmask | C | C | Preserved[b] |
| 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[c] |
| dfh | Disabled FP register f16 to f127 | C | C | Unchanged[c] |
| 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 | C | Preserved[d] |
| | | 1 | 1 | 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 |

a. If this bit is 0 on entry, the value of this bit shall be 0 on exit and it must be classified as unchanged.
b. SAL procedures shall not enable interrupts if interrupts are disabled on entry.
c. If this bit is 1 on entry, a Disabled FP-register vector fault may occur.
d. 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[a] |
| 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] |
| 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[b] |
| TC | Translation Cache | Scratch |
| IBR/DBR | Break Point Registers | Preserved |
| PMC | Performance Monitor Control Registers | Preserved |
| PMD | Performance Monitor Data Registers | Unchanged[c] |

a. SAL procedures may not update these registers, but the arrival of asynchronous interrupts may cause them to change.
b. If an implementation provides a means to read TRs through a PAL procedure call, this should be preserved.
c. 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. |
| Bank 0 Registers (GR16 – GR23) | Preserved. |
| Bank 0 Registers (GR 24 – GR31) | Scratch. |
| GR32 – GR127 | Stacked registers:<br>in0 – in95: input arguments (SAL index must be in0)<br>loc0 – loc95: local variables<br>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. 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*.

### 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. 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-based operating system and SAL. An Itanium-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. Currently 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:

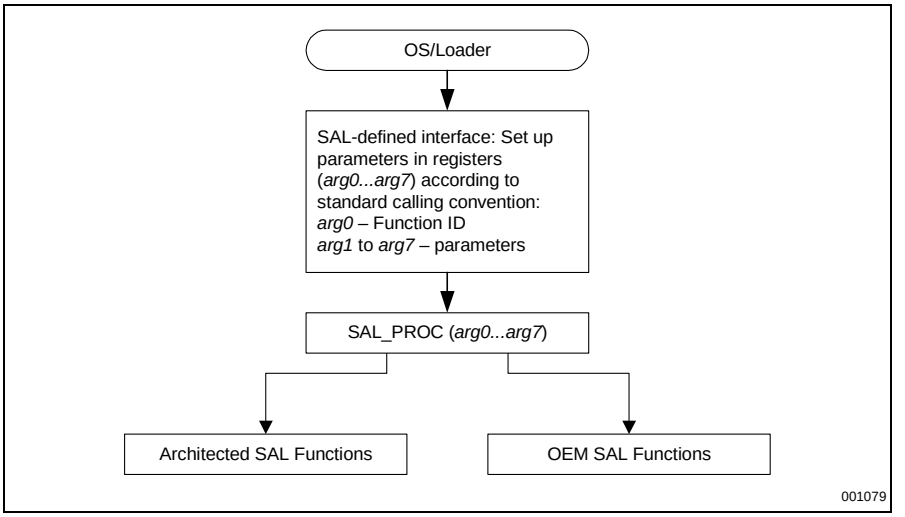
ret0 – return status: positive number indicates successful, negative number indicates failure.

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 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 or firmware error. |
| –4 | Virtual address not registered. |
| –5 | No information available. |
| –9 | Scratch buffer required. |

# SAL Procedures 9

## 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 processors. The SAL runtime services shall be implemented completely in the Itanium processor system environment.
- 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, e.g., 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 9.3. 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, i.e. 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. OEM specific SAL Runtime procedures may not support both virtual and physical modes of operation.
- 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, i.e. 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 3-4). 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 *EFI 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 *EFI 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, e.g. 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 (i.e. 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 (i.e. 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.

## 9.1.2 Access to Resources not Supported by the Operating System

In order to access resources for which the operating system does not provide the mapping, SAL runtime services will access the platform resources in physical addressing mode. This will be done by disabling the interrupts and turning the data translation off before accessing the platform resources. SAL will restore the state of the data translation and interrupt enable bits in the PSR after accessing the device. The following is a suggested code sequence:

```
mov    r2=psr.l            //Save current PSR, low 32 bits
rsm    (1<<14) | (1<<17)   //Mask Interrupt (PSR bit 14) and
                           //disable data translation (PSR bit 17)
```

```
;;                          //End of instruction group
srlz.d                      //Serialize
;;                          //End of instruction group

ld/st.......                //Perform load/store to platform specific
                            //device using physical address

mf.a                        //Ensure platform acceptance

;;                          //End of instruction group
mov    psr.l=r2             //Restore original PSR, low 32 bits
;;                          //End of instruction group
srlz.d                      //Serialize
;;                          //End of instruction group
```

The code sequence (from *rsm* to the second *srlz.d*) must exist in a single page of memory and the translation for this code sequence must exist. The code sequence must not cause any NaT consumption faults. All the memory accesses in this code sequence must be naturally aligned to avoid unaligned data reference faults. If disabling of interrupt and data translation are done separately, interrupts need to be disabled first and then the data translation. The code sequence may not work if the data translation is disabled first followed by interrupt disabling. The restoring of the processor state must be done in the *reverse* order. In general, interrupt and data translation should be disabled to access the devices in physical mode and then interrupt and data translation must be re-enabled as soon as possible.

The duration of interrupt and data translation disabled state should be kept at a minimum to preclude impacting normal operating system functions.

## 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 | Re-entrancy Requirement |
|---|---|---|---|
| SAL_SET_VECTORS | 0x01000000 | Register software code locations with SAL. | None |
| SAL_GET_STATE_INFO | 0x01000001 | Return Machine State information obtained by SAL. | Yes |
| SAL_GET_STATE_INFO_SIZE | 0x01000002 | Obtain size of Machine State information. | Yes |
| SAL_CLEAR_STATE_INFO | 0x01000003 | Clear Machine State information. | Yes |
| SAL_MC_RENDEZ | 0x01000004 | Cause the processor to go into a spin loop within SAL. | MP-safe |
| SAL_MC_SET_PARAMS | 0x01000005 | Register the machine check interface layer with SAL. | None |
| SAL_REGISTER_PHYSICAL_ADDR | 0x01000006 | Register the physical addresses of locations needed by SAL. | None |
| SAL_CACHE_FLUSH | 0x01000008 | Flush the instruction or data caches. | MP-safe |
| SAL_CACHE_INIT | 0x01000009 | Initialize the instruction and data caches. | MP-safe |
| SAL_PCI_CONFIG_READ | 0x01000010 | Read from the PCI configuration space. | Yes |
| SAL_PCI_CONFIG_WRITE | 0x01000011 | Write to the PCI configuration space. | Yes |
| SAL_FREQ_BASE | 0x01000012 | Return the base frequency of the platform. | MP-safe |
| SAL_UPDATE_PAL | 0x01000020 | Update the contents of firmware blocks. | None |

## 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:<br>1 = Flush instruction cache<br>2 = Flush data cache<br>3 = Flush instruction & data cache<br>4 = Make local instruction caches coherent with the data caches<br>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. If the platform caches are coherent with the memory hierarchy, the SAL implementation is not required to perform flushes of such caches. If platform caches are node specific, this SAL procedure must be invoked on each node.

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, i.e. 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 |
| –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.

If platform caches are node specific, this SAL procedure must be invoked on each node.

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 or Corrected Platform 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:<br>0 – MCA event information<br>1 – INIT event information<br>2 – Processor CMC event information<br>3 – Corrected Platform event information<br>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). In an MP environment, processor record information pertains to the processor on which this call is executed and the platform record information pertains to the entire platform. 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:<br>0 = Platform base clock frequency (clock input to the processor)<br>1 = Input frequency to the Interval Timer on the platform (optional)<br>2 = Input frequency to the Real time clock on the platform (optional)<br>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 front side 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 front side 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-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 or Corrected Platform 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:<br>0 – MCA event information<br>1 – INIT event information<br>2 – Processor CMC event information<br>3 – Corrected Platform Event information<br>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 |

**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 operating system may retrieve the same information multiple times prior to clearing the record. 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. In the case of multiple pending error records of the same type, the operating system has to get and clear the current record before it can get the next one.

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.

In an MP environment, processor record information pertains to the processor on which this call is executed and the platform record information pertains to the platform. 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 Appendix B, "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 shall discard the error logs for the latter occurrences.

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 NVM storage for backing up the MCA error records. The SAL is not required to log CMC or CPE error records to the NVM 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 NVM 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 or CMCs.

**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:<br>0 – MCA event information<br>1 – INIT event information<br>2 – Processor CMC event information<br>3 – Corrected Platform Event information<br>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. This procedure will disable interrupts and set an implementation dependent check-in flag within the SAL data area to indicate to the monarch processor that the non-monarch processor has reached the SAL layer. 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 wake up signal from the monarch processor. The wake up 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 wake up 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.

While waiting for the wake-up from the monarch processor, the SAL on the non-monarch processors shall mask further machine checks and escalate future MCA and BERR events to BINIT using the PAL_PROC_SET_FEATURES procedure. This step is important from error containment perspective. 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:<br>1 = rendezvous interrupt<br>2 = wake up<br>3 = Corrected Platform Error Vector<br>Other values are reserved |
| i_or_m | Unsigned 64-bit integer value indicating whether interrupt vector or memory address is specified:<br>1 = interrupt vector<br>2 = memory address<br>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-based operating systems as either the rendezvous or the wake-up interrupt. Only the PAL layer to support IA-32 operating systems may use the PMI as the rendezvous interrupt since all the external interrupt vectors may be in use by the IA-32 operating system. The SAPIC IPI message signalling the MC_rendezvous interrupt of PMI type shall specify a value of 13 in the vector field of the IPI message. The PMI interrupt mechanism shall not be employed as the wake-up interrupt by any operating system.

The PMI interrupt mechanism needs to be supported only on platforms that support IA-32 operating systems and SAL may return an error status on other platforms.

The *mca_opt* argument specifies the options that the SAL_MCA 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. This is the recommended setting for error containment. 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. Good programming practices dictate that indexed accesses to the configuration space be serialized in order to be MP-safe.

**Arguments:**

| Argument | Description |
|---|---|
| func_id | Function ID of SAL_PCI_CONFIG_READ within the list of SAL procedures |
| address | PCI configuration address:<br>Bits 0..7 – Register address<br>Bits 8..10 – Function number<br>Bits 11..15 – Device number<br>Bits 16..23 – Bus number<br>Bits 24..31 – Segment number<br>Bits 32..63 – Reserved (0)<br>Must be naturally aligned with respect to the size of the read. |
| size | PCI config size (1, 2 or 4 bytes) |
| 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. Good programming practices dictate that indexed accesses to the configuration space be serialized in order to be MP-safe.

**Arguments:**

| Argument | Description |
|---|---|
| func_id | Function ID of SAL_PCI_CONFIG_WRITE within the list of SAL procedures |
| address | PCI configuration address:<br>Bits 0..7 – Register address<br>Bits 8..10 – Function number<br>Bits 11..15 – Device number<br>Bits 16..23 – Bus number<br>Bits 24..31 – Segment number<br>Bits 32..63 – Reserved (0)<br>Must be naturally aligned with respect to the size of the write. |
| size | PCI config size (1, 2 or 4 bytes) |
| value | Value to write to PCI config space |
| 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

**Arguments:** 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<br>0 = PAL_PROC<br>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:** This procedure is used by the operating system to register the new *physical* addresses of the PAL_PROC procedure in memory. If the operating system were to copy PAL procedures to a different memory location (using the PAL_COPY_PAL procedure), it must register the new PAL_PROC entrypoint address with the SAL. The SAL layer will then be in a position to invoke the PAL procedures in physical mode.

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:<br>0 = Machine Check<br>1 = INIT<br>2 = BOOT_RENDEZ<br>3–64 = Reserved<br>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, i.e. 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.<br>1 = Checksum information provided by the operating system in bits 40-47. |
| 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 the length argument is non-zero, the SAL saves the operating system provided checksum for the procedure. Before invoking the registered operating system procedure, SAL shall authenticate the operating system code by verifying its checksum.

**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 (e.g. 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:<br>0= Do not store checksum of this component in its FIT entry<br>1=Calculate & 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 (e.g. 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 2-7)<br>1 = PAL_B; 0x0F = 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 –7 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 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 on 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 |
| –13 | Insufficient space in the storage device |

In some firmware architectures (e.g. 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, i.e. 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.

# Glossary A

**ACPI**

Advanced Configuration and Power Interface Specification.

**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. Warm Boot, on the other hand, refers to a hardware/software event that sets the circuitry of any or all of the processor(s) on the system to an initial state. Warm Boot may be triggered by the INIT event. Both Cold and Warm Boot events occur at cycle boundaries and do not corrupt any pending cycles. Destructive memory tests are not performed during 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.

**FSB**

Processor Frontside Bus.

**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-based Operating System**

An operating system which is written in the Itanium instruction set that can run Itanium-based applications (code containing Itanium instructions and/or IA-32 instructions).

**INTA**

Interrupt Acknowledge.

**IPI**

Inter processor interrupt signaling using the local SAPIC within the processor.

**IPL**

Initial Program Load.

**ISA**

Instruction Set Architecture.

**IVT**

Interrupt Vector Table.

**MBR**

Master Boot Record.

**MC_rendezvous Interrupt**

An external interrupt vector provided to SAL by the Itanium-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.

**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.

**NVM**

Non-volatile 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 or from multiple processors.

**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. The major version is incremented when the SAL API changes. The minor version is incremented when underlying functionality changes but the API remains the same. SAL implementations pertaining to a particular SAL revision specification shall be compatible with each other at the published SAL external interfaces.

**SAPIC**

Streamlined Advanced Programmable Interrupt Controller. The code name for the high performance interrupt architecture for the Itanium processor. 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.

**Wakeup 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-based operating system with the SAL.

**WBL**

Write-back with Limited Speculation.

# Error Record Structures B

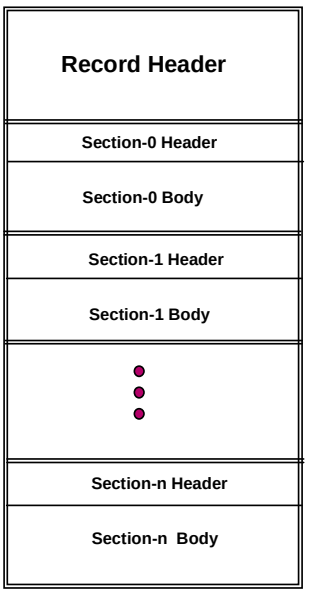
## B.1 Overview

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.

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 etc.), 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: The ERR_SEVERITY information reflects the error severity based on the PROC_STATE_PARAMETER field in the processor section. The SAL may increase the severity if a platform component has experienced severe errors. The operating system is free to analyze the section error information, decide if it can correct or continue without the device represented by the section and ignore the fatal severity.

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

| Offset | Length | Field | Description |
|---|---|---|---|
| 0 | 8 bytes | RECORD_ID | Unique monotonically increasing ID for MCA, INIT, CMC and CPE event Records. |
| 8 | 2 bytes | REVISION | 2-byte Major and Minor revision number of the Record in BCD format:<br>Byte0 – Minor (02)<br>Byte1 – Major (00) |
| 10 | 1 byte | ERR_SEVERITY | This encoded field indicates error severity. See glossary section for details on the definition:<br>0 – Recoverable<br>1 – Fatal<br>2 – Corrected<br>Others – Reserved |
| 11 | 1byte | VALIDATION_BITS | Bit 0 = If 1, the OEM_PLATFORM_ID field below contains valid information.<br>Bits 1-7 – 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 in BCD format:<br>Byte 0 – Seconds<br>Byte 1 – Minutes<br>Byte 2 – Hours<br>Byte 3 – Reserved<br>Byte 4 – Day<br>Byte 5 – Month<br>Byte 6 – Year<br>Byte 7 – Century |
| 24 | 16 bytes | OEM_PLATFORM_ID | A unique identifier of the OEM platform. |

## 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 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 etc.).

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:<br>Byte0 – Minor (02)<br>Byte1 – Major (00) |
| 18 | 1 byte | ERROR_RECOVERY_INFO | Bit 7 = If 1, remaining bits contain information about the error.<br>Bit 6-3 = Reserved, must be 0.<br>Bit 2 = Reset. If set, the component must be re-initialized or re-enabled by the operating system prior to use.<br>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.<br>Bit 0 = If set, the error has been corrected. |
| 19 | 1 byte | RESERVED | Reserved. |
| 20 | 4 bytes | SECTION_LEN | Length of this error device section in bytes, including the header. |

**Table B-1. GUID Format**

The GUID structure is as follows:

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

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-3) in the error status field within the section.

The ERROR_SEVERITY_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_SEVERITY_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.

## B.2.3 Processor Device Error Info

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
      PSI_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
      RESERVED                             Bits 24-63
   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
   } 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.File 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 {                                   Processor Static Information
   VALID_FIELD_BITS[2]                     8 bytes
         MINSTATE_VALID_BIT                  Bit 0
         BR_VALID_BIT                        Bit 1
         CR_VALID_BIT                        Bit 2
```

1. The amount of information reported by SAL is implementation dependent. The validity of each field is indicated by either a validation bit or an encoded number field. Data areas corresponding to *invalid* fields will be padded. For CACHE, TLB, BUS, REG, MS fields, the encoded NUM field indicates the number of MOD_ERROR_INFO_STRUCTs for each category, ranging from 0-15. *For these five categories only,* if the encoded NUM field is zero, then the data area corresponding to that category will be absent.
2. Data areas corresponding to Invalid fields will be padded.

```
            AR_VALID_BIT                        Bit 3
            RR_VALID_BIT                        Bit 4
            FR_VALID_BIT                        Bit 5
            RESERVED                            Bit 6-63
        Minimal State Save Info Structure[1]    1024 bytes
        BRs 0-7                                 64 bytes
        CRs 0-127                               1024 bytes[2,3]
        ARs 0-127                               1024 bytes[2,3]
        RRs 0-7                                 64 bytes
        FRs 0-127                               2048 bytes
} PSI_STATIC_STRUCT
}

The MOD_ERROR_INFO_STRUCT structure is defined as below:
struct{                                       48 bytes[4](Mod)
        VALID_FIELD_BITS                      8 bytes
            CHECK_INFO_VALID_BIT                Bit 0
            REQUESTOR_IDENTIFIER_VALID_BIT      Bit 1
            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[5]_ERROR_INFO_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. The size of the MinState Structure is about 1Kbytes. For more details on the size and contents of the structure, please refer to the *Intel® Itanium™ Architecture Software Developer's Manual.*
2. The number of Control and Application registers on a processor is processor implementation dependent.
3. 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.
4. The size of this structure will always be 48 bytes, with invalid fields being padded with null values.
5. The MOD structure is common across CACHE, TLB, BUS, REGISTER_FILE and Microarchitectural structure error records.

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:<br>Bit 0 – MEM_ERROR_STATUS_VALID_BIT<br>Bit 1 – MEM_PHYSICAL_ADDR_VALID_BIT<br>Bit 2– MEM_ADDR_MASK_BIT<br>Bit 3 – MEM_NODE_VALID_BIT<br>Bit 4 – MEM_CARD_VALID_BIT<br>Bit 5 – MEM_MODULE_VALID_BIT<br>Bit 6 – MEM_BANK_VALID_BIT<br>Bit 7 – MEM_DEVICE_VALID_BIT<br>Bit 8 – MEM_ROW_VALID_BIT<br>Bit 9 – MEM_COLUMN_VALID_BIT<br>Bit 10 – MEM_BIT_POSITION_VALID_BIT<br>Bit 11 – MEM_PLATFORM_REQUESTOR_ID_VALID_BIT<br>Bit 12 – MEM_PLATFORM_RESPONDER_ID_VALID_BIT<br>Bit 13 – MEM_PLATFORM_TARGET_VALID_BIT<br>Bit 14 – MEM_PLATFORM_BUS_SPECIFIC_DATA_VALID_BIT<br>Bit 15 – MEM_PLATFORM_OEM_ID_VALID_BIT<br>Bit 16 – MEM_PLATFORM_OEM_DATA_STRUCT_VALID_BIT<br>Bit 17-63 – RESERVED |
| 8 | 8 bytes | MEM_ERROR_STATUS | Memory device error status fields (see Table B-3). |
| 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. |
| 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-2 for the format of this structure. |

### Table B-2. 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. |
| 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, etc.). 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:<br>Bit 0 – PCI_BUS_ERROR_STATUS_VALID_BIT<br>Bit 1 – PCI_BUS_ERROR_TYPE_VALID_BIT<br>Bit 2 – PCI_BUS_ID_VALID_BIT<br>Bit 3 – PCI_BUS_ADDRESS_VALID_BIT<br>Bit 4 – PCI_BUS_DATA_VALID_BIT<br>Bit 5 – PCI_BUS_CMD_VALID_BIT<br>Bit 6 – PCI_BUS_REQUESTOR_ID_VALID_BIT<br>Bit 7 – PCI_BUS_RESPONDER_ID_VALID_BIT<br>Bit 8 – PCI_BUS_TARGET_ID_VALID_BIT<br>Bit 9 – PCI_BUS_OEM_ID_VALID_BIT<br>Bit 10 – PCI_BUS_OEM_DATA_STRUCT_VALID_BIT<br>Bit 11..63– RESERVED |
| 8 | 8 bytes | PCI_BUS_ERROR_STATUS | PCI Bus error status fields (see Table B-3). |
| 16 | 2 bytes | PCI_BUS_ERROR_TYPE | PCI Bus error types<br>Byte0:<br>0 – Unknown or OEM System Specific Error<br>1 – Data Parity Error<br>2 – System Error<br>3 – Master Abort<br>4 – Bus Time Out or No Device Present (No DEVSEL#)<br>5 – Master Data Parity Error<br>6 – Address Parity Error<br>7 – Command Parity Error<br>Others – Reserved<br>Byte1:<br>RESERVED |
| 18 | 2 bytes | PCI_BUS_ID | Designated PCI Bus identifier encountering error.<br>Bits 0..7 – Bus Number<br>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. |
| 48 | 8 bytes | PCI_BUS_REQUESTOR_ID | PCI Bus Requestor ID at the time of the event. |
| 56 | 8 bytes | PCI_BUS_RESPONDER_ID | PCI Bus Responder ID at the time of the event. |
| 64 | 8 bytes | PCI_BUS_TARGET_ID[a] | 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-2 for the format of this structure. |

a. 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:<br>Bit 0 – PCI_COMP_ERROR_STATUS_VALID_BIT<br>Bit 1 – PCI_COMP_INFO_VALID_BIT<br>Bit 2 – PCI_COMP_MEM_NUM_VALID_BIT<br>Bit 3 – PCI_COMP_IO_NUM_VALID_BIT<br>Bit 4 – PCI_COMP_REGS_DATA_PAIR_VALID_BIT<br>Bit 5 – PCI_COMP_OEM_DATA_STRUCT_VALID_BIT<br>Bit 6..63– RESERVED |
| 8 | 8 bytes | PCI_COMP_ERROR_STATUS | PCI Component error status fields (see Table B-3). |
| 16 | 16 bytes | PCI_COMP_INFO | PCI Component Information to identify the device:<br>Bytes 0-1 – Vendor ID<br>Bytes 2-3 – Device ID<br>Bytes 4-6 – Class Code<br>Byte 7 – Function Number<br>Byte 8 – Device Number<br>Byte 9 – Bus Number<br>Byte 10 – Segment Number<br>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 N bytes | PCI_COMP_REGS_DATA_PAIR | An array of PCI Component address/data register pair values. |
| 40+2x8xN | N bytes | PCI_COMP_OEM_DATA_STRUCT | OEM specific data of variable length. See Table B-2 for the format of this structure. |

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.

### 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_BIT_BITS | Validation Bits to indicate the validity of each of the subsequent fields:<br>Bit 0 – SEL_RECORD_ID_VALID_BIT<br>Bit 1 – SEL_RECORD_TYPE_VALID_BIT<br>Bit 3 – SEL_GENERATOR_ID_VALID_BIT<br>Bit 3 – SEL_EVM_REV_VALID_BIT<br>Bit 4 – SEL_SENSOR_TYPE_VALID_BIT<br>Bit 5 – SEL_SENSOR_NUM_VALID_BIT<br>Bit 6 – SEL_EVENT_DIR_TYPE_VALID_BIT<br>Bit 7 – SEL_EVENT_DATA1_VALID_BIT<br>Bit 8 – SEL_EVENT_DATA2_VALID_BIT<br>Bit 9 – SEL_EVENT_DATA3_VALID_BIT<br>Bit 10-63– RESERVED |
| 8 | 2 bytes | SEL_RECORD_ID | Record ID used for SEL record access. |
| 10 | 1 bytes | SEL_RECORD_TYPE | Indicates the record type:<br>0x02 – System Event Record<br>0xC0-0xDF – OEM time stamped, bytes 8-16 OEM defined<br>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<br>Byte1:<br>Bit 7:1 – 7-Bit system software ID.<br>Bit 0 – set to one (1) when using system software.<br>Byte 2:<br>Bit 7:2 – Reserved. Write as 0, ignore when read.<br>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:<br>Bit 7 – 0 for assertion; 1 for de-assertion.<br>Event Type:<br>Type of trigger for the event, e.g. critical threshold going high, state asserted, etc. 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.<br>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 *IPMI 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:<br>Bit 0 – SMBIOS_EVENT_TYPE_VALID_BIT<br>Bit 1 – SMBIOS_LENGTH_VALID_BIT<br>Bit 3 – SMBIOS_TIME_STAMP_VALID_BIT<br>Bit 3 – SMBIOS_DATA_VALID_BIT<br>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-2 for the format of this structure. |

Refer to the *SMBIOS Specification (http://download.intel.com/ial/wfm/smbios.pdf)* 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:<br>Bit 0 – PLATFORM_ERROR_STATUS_VALID_BIT<br>Bit 1 – PLATFORM_REQUESTOR_ID_VALID_BIT<br>Bit 2 – PLATFORM_RESPONDER_ID_VALID_BIT<br>Bit 3 – PLATFORM_TARGET_VALID_BIT<br>Bit 4 – PLATFORM_SPECIFIC_DATA_VALID_BIT<br>Bit 5 – PLATFORM_OEM_ID_VALID_BIT<br>Bit 6 – PLATFORM_OEM_DATA_STRUCT_VALID_BIT<br>Bit 7 – PLATFORM_OEM_DEVICE_PATH_VALID_BIT<br>Bit 8..63 – RESERVED |
| 8 | 8 bytes | PLATFORM_ERROR_STATUS | Platform generic error status fields (see Table B-3). |
| 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. |

| Offset | Length | Field | Description |
|---|---|---|---|
| 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-2 for the format of this structure. |
| 64+N bytes | X bytes | PLATFORM_OEM_DEVICE_PATH | OEM specific Vendor Device Path. Please refer to the *EFI Specification* for the format of this field. |

## 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-3. Error Status Fields**

| Bit Position | Description |
|---|---|
| Bit 0-Bit7 | Reserved. |
| Bit8 - Bit 15 | Encoded value for the Error_Type[a] (see Table B-4). |
| 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. |

a. 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-4.

**Table B-4. 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. |

## Table B-4. Error Types (Cont'd)

| | 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. |
| All Others | Reserved. |