IA-64 System Abstraction Layer Specification

January 2000

Order Number: 245359-001

Contents

Figures

Tables

Introduction 1

1.1 Objectives

This document describes the functionality of the IA-64 System Abstraction Layer (SAL).

This document specifies requirements to develop platform firmware for IA-64 systems. A companion document, *The Extensible Firmware Interface (EFI)*, 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 the IA-64 firmware.

This document is intended for firmware/BIOS (basic input/output device) designers, system designers and writers of diagnostic and low-level OS software. This document is a specification and does not specify implementation details.

The primary objectives of the IA-64 firmware layer are to:

- Enable boot of IA-64 OSes.
- Provide a uniform interface to the boot loaders of the OSes for all IA-64 platforms.
- Ensure that the firmware interfaces are sufficient to contain 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 hardware innovation and optimization of IA-64 platforms.
- Support OEM capability for platform differentiation.
- 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, multiprocessors (MPs), and broader and deeper I/O hierarchies (possibly greater than 100 I/O cards).
- Enable boot of shrink-wrapped versions of IA-32 operating systems (OSes). 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 be in the IA-64 Instruction Set Architecture (ISA).
- Enable the use of legacy PC peripherals, option ROMs and PCI cards with IA-32 Plug-and-Play expansion ROMs.

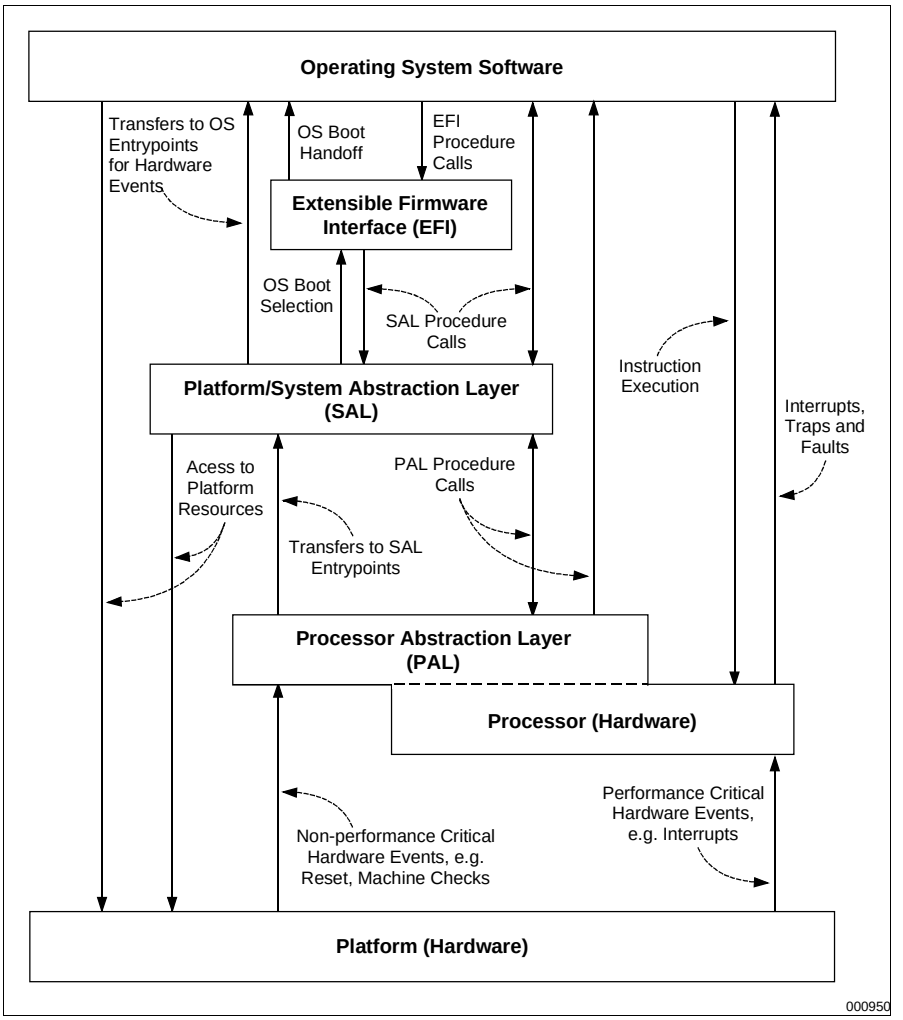
This document describes the platform dependent firmware interfaces needed to support these goals. However, this document is not intended to redocument the PC infrastructure specifications.

1.2 Firmware Model

As shown in Figure 1-1, IA-64 firmware consists of three major components, all of which are required:

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

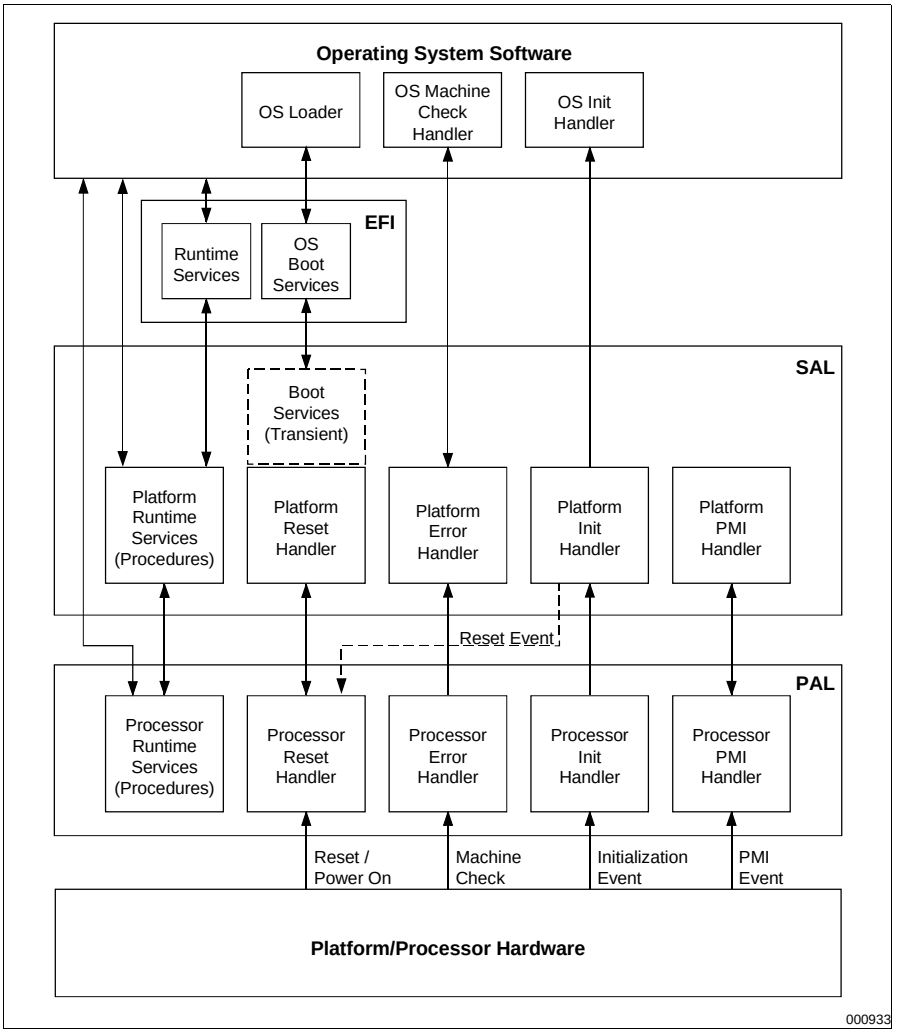
Figure 1-1. Firmware Model



PAL encapsulates the processor model specific hardware and is part of the IA-64 Instruction Set Architecture (ISA) extension. PAL is the firmware layer that abstracts the processor implementation-specific features and is independent of the number of processors. SAL is the platform specific firmware component that isolates OS 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 OS 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 IA-64 platform:

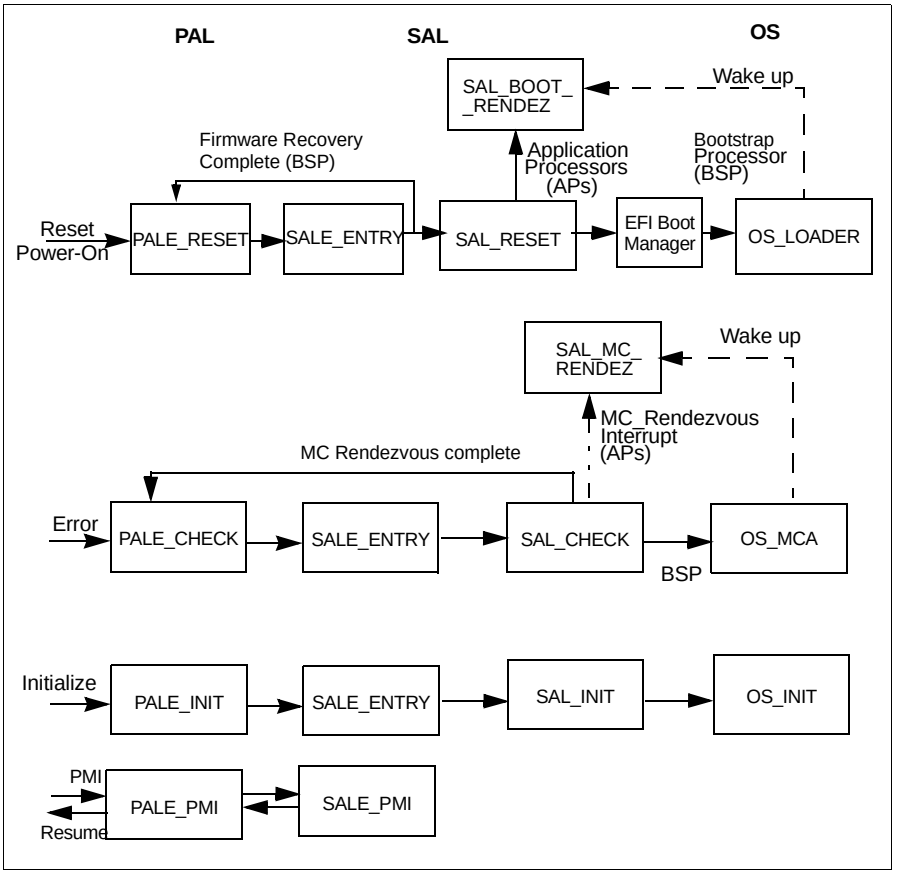
- Initialize, configure, and test the platform hardware. This includes the memory and I/O subsystems, the necessary boot devices and platform specific hardware.
- Select the bootstrap processor (BSP) in a MP platform and set the configurable processor features. The IA-64 processor provides its own PAL 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 to the rest of the system. For example, the 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 the EFI and the OS Loader in establishing the environment necessary for the OS to run in.
- Provide common data structures to the OS to convey initialization and configuration information.
- Provide the necessary services and common infrastructure to support MP configurations.
- Provide Runtime Service routines to encapsulate those functions of the platform necessary for the EFI and the OS 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.
- OS 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 a reset. This entrypoint within PAL calls SALE_ENTRYPOINT in SAL to test for firmware recovery indication. SALE_ENTRY, in turn, calls a procedure within SAL called SAL_RECOVERY_CHECK that performs the recovery if firmware recovery indication is present on the platform, else 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 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 SALE_ENTRY in SAL. SALE_ENTRY, in turn, branches to a procedure within SAL called SAL_CHECK to complete the error logging, correction, and reporting. PALE_CHECK is entered as a response to processor and/or platform errors.
3. PALE_INIT – saves the minimal processor state, initializes the processor, and branches to SALE_ENTRY in SAL. SALE_ENTRY, in turn, branches to a procedure within SAL called 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 SALE_PMI. 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 – PAL branches to this SAL entrypoint after a power-on reset, machine check or initialization event. The code at this entrypoint using the hand-off value in a General Register, jumps to different entrypoints within 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 – PAL branches to this SAL entrypoint for handling platform management events in an implementation dependent manner.

1.4.3 Operating System Entrypoints

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

- OS_LOADER – OS Loader. Entered from SAL_RESET on the BSP only, after the system has been initialized and the OS Loader image has been loaded by the EFI component from the boot device. Refer to the *EFI Specification* for details.
- OS_BOOT_RENDEZ – OS MP Rendezvous Handler. Entered from SAL when OS on the BSP wakes up the application processors (APs), to permit synchronization of APs in a MP environment.
- OS_MCA – OS 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 – OS Initialization Handler. Called from SAL_INIT to handle a valid initialization event.

1.5 Related Documents

The following documents contain additional material related to IA-64 processors:

- *Advanced Configuration and Power Interface Specification*, 1996 – 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
- *IA-64 Software Conventions and Runtime Architecture Guide* – HP/Intel
- *Intel® IA-64 Architecture Software Developer's Manual* – Intel
- *PCI BIOS Specification*, 1994 – PCI SIG
- *Plug and Play ISA Specification*, 1994 – Microsoft

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
	SAL Resources
0xFF00_0000	

The firmware address space is logically partitioned into two major functional blocks: the ROM area (shared by the SAL and PAL) and the SAL Resources area. The ROM area is placed in the address space such that its ending address is at 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 which the platform has implemented in the SAL Resources area. The hardware resources implemented can optionally include (but are not limited to) 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 and
- 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 (PAL_PROC) that can be invoked by SAL while performing a firmware recovery (refer to Volume 2 of the *Intel® IA-64 Architecture Software Developer's Manual* for details). The PAL_B block contains all the PAL procedures that can be invoked by SAL and the OS.

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

Following is a simplified example of the firmware address map that shows the *minimum* architectural components. Refer to Section 2.4.1 for description of the fields. This layout is not expected to be used with a flash ROM supporting the protected boot block feature. See Figure 2-2 for a different firmware organization supporting the protected boot block.

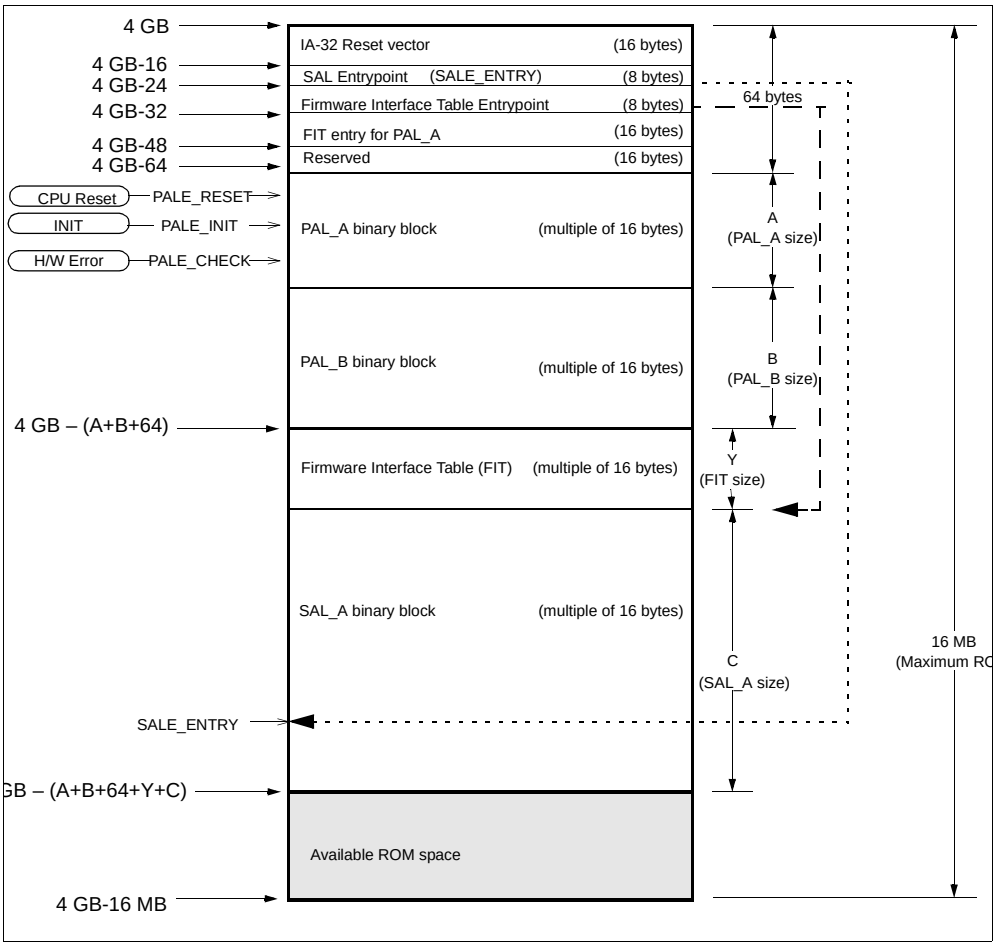
2.4 Firmware Organization using Protected Boot Block

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

Protected boot block refers to a block of the Flash ROM that is prevented from modifications by hardware. 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 was accidentally aborted. Firmware organization using 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



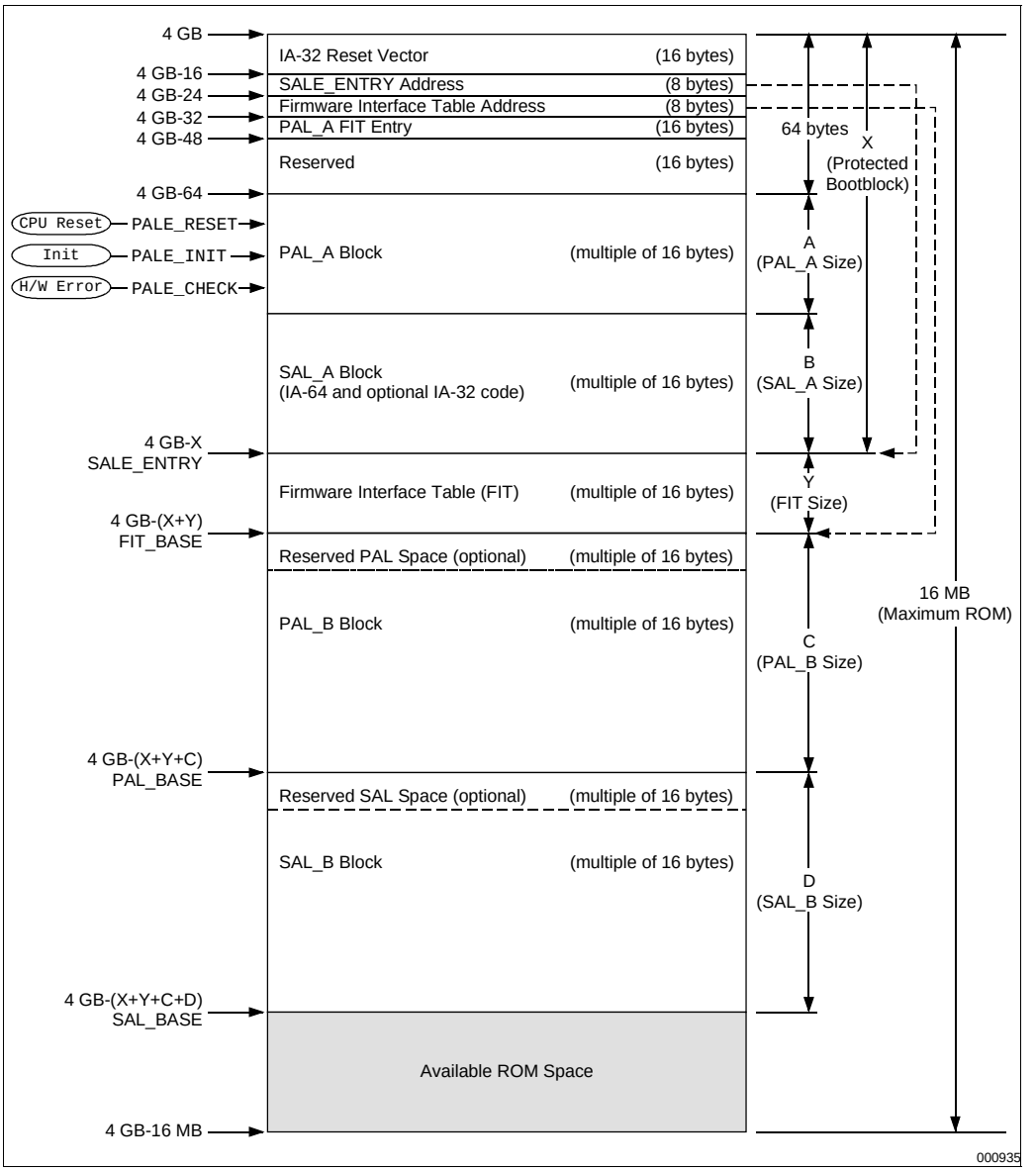
2.4.1 Firmware Components

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 below for a typical implementation.

The firmware address space contains the following regions and locations:

- The 16 bytes at (4GB – 16) contains the IA-32 Reset Code. This is typically an IA-32 far JMP instruction followed by the Date, the PC-AT* model signature, etc. This code is never executed but is present for PC-AT compatibility.

Figure 2-2. Firmware Address Map



- The 8 bytes at (4GB – 24) contain the address of the SALE_ENTRY entrypoint. Bit 63 of this address must be set to 1 to indicate the uncacheable memory attribute in physical addressing mode.
- The 8 bytes at (4GB – 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 (4GB – 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 (4GB – 64) are reserved for future use.
- The PAL_A code resides below the (4GB – 64) address. This area of variable size contains the hardware-triggered entrypoints PALE_RESET, PALE_INIT, and PALE_CHECK, as well as 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 INIT.

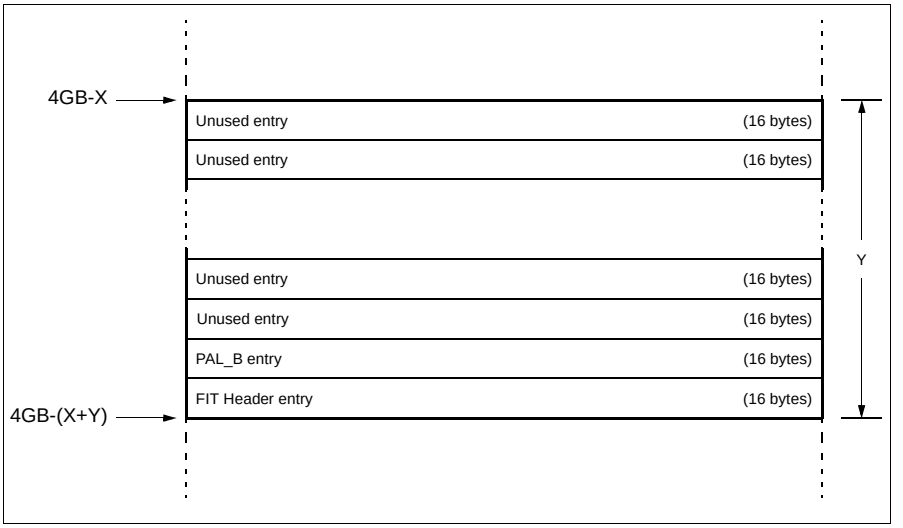
  The code in the PAL_A block contains enough capability to initialize the processor, invoke the SALE_ENTRY procedure for test of the recovery indication and continue with normal PAL execution in the PAL_B code area. The code in this area shall be identical for all IA-64 processors in the same family. This code shall be unaffected by processor stepping changes.
- 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 INIT, 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 INIT 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 comprises 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 which 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 IA-64 and IA-32 code components 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

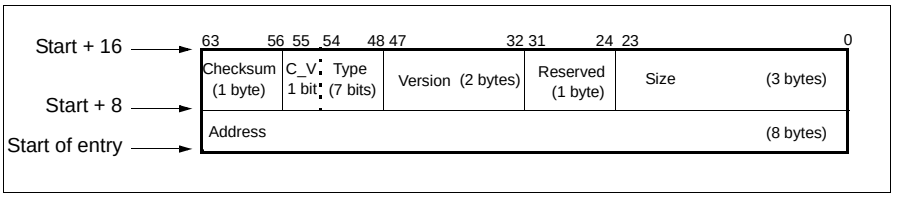
The Firmware Interface Table (FIT) contains starting addresses and sizes for the different firmware components that are outside the protected boot block. Because these code blocks may be compiled at different times and places, code in one block (such as PAL_A) cannot branch to code in another block (such as PAL_B) directly. The FIT allows code in one block to find entrypoints in another. The figure below 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 else 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 4GB – 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.

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 4GB.

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.

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

0x000F_FFFF	Shadowed IA-32 System BIOS
0x000F_0000	
0x000E_FFFF	Shadowed IA-32 Extended System BIOS/Option ROM/Memory area
0x000E_0000	

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

0x000D_FFFF 0x000C_0000	Shadowed IA-32 Option ROM BIOS or ISA Bus Expansion Memory
0x000B_FFFF	VGA Frame Buffer
0x000A_0000	
0x0009_FFFF	Memory
  0x0000_0500	
0x0000_04FF 0x0000_0400	IA-32 BIOS RAM Data Area
0x0000_03FF	IA-32 Interrupt Vector Area
0x0000_0000	

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.

IA-64 platforms may 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 OSes may rely on the existence of PC-AT compatible components. In order to execute such IA-32 code, all IA-64 platforms shall implement the I/O ports specified in Table 2-4. 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

Following are the SAL requirements from the chipset implementation:

- 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 alternatively, 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 INIT 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.

  IA-64 memory management features needed for IA-32 execution can be set up to prevent writes to the shadowed RAM areas. The IA-64 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 segments at E0000 to FFFFF to RAM.

2.8 Platform Support for Variant Architectural Features

Different platform implementations may vary with respect to each other in the features they implement and yet they could be 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 architectural variant 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 bus lock is implemented on a system, 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 OS 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® IA-64 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 OS, 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 OS by the EFI component. As an example of this architectural variant, 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® IA-64 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 a 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 MP 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 (IPIs). 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/algorithms will be very simple, to facilitate exchange of IPIs between processors (if needed), to determine the BSP node and the BSP processor in a 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-TLB (Translation Lookaside Buffer) coherence domain platform must provide a mechanism for detecting which TLB coherence domain the processor is located in.

2.10 Non-volatile Memory Requirements

IA-64 platform hardware must provide a minimum of 32KB of NVM to hold the Error log captured during machine check events. There may be additional NVM requirements to hold information on the OSes 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 feature is supported in SAL, the IA-64 platform must provide an implementation specific hardware mechanism to reflect the user selected *firmware recovery condition* to all the processors on the platform.
- IA-64 platforms must support simple hardware and/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.
- IA-64 platforms must provide mechanisms to determine the base frequency of the platform (clock input to the processor).
- IA-64 platform hardware must provide a mechanism for firmware to reset all components within the platform.
- IA-64 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”.
- IA-64 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 OS launch.

On Reset, all the processor(s) begin execution at PALE_RESET, a location within the PAL_A code area near 4 GB in ROM, in the IA-64 ISA. 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 entrypoint in SAL_A from code in the PAL_A and PAL_B blocks for Reset, Recovery, Machine check and INIT events. PAL code obtains SALE_ENTRY entrypoint from the 8-byte pointer at 4GB – 24. The state of the processor on entry into SALE_ENTRY is described in the *Intel® IA-64 Architecture Software Developer's Manual*. One of the general registers, indicates the event causing entry into SALE_ENTRY: Reset, Recovery check, Machine check or INIT. SALE_ENTRY uses this argument to jump to internal entrypoints 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	Disable interrupts and Machine Checks, then go into a spin loop
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. Disable interrupts and Machine Checks, then go into a spin loop
Functionally Restricted	Try to inform the user. Disable interrupts and Machine Checks, then go into a spin loop

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

- the platform indicates a recovery condition; or
- the PAL_A code reports an authentication failure on the PAL_B component in the firmware; or
- the PAL_A code reports checksum or other errors in the FIT or the PAL_B component; or
- 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 floppy disk or other OEM supported media. Note that 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® IA-64 Architecture Software Developer's Manual* for details). Further, if the SAL_A invokes the IA-32 BIOS, 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® IA-64 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 OS.

3.2 SAL_RESET

SAL_RESET is responsible for performing platform test and initialization, invoking EFI firmware which, in turn, loads the first level of OS Loader and jumps to it. 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.

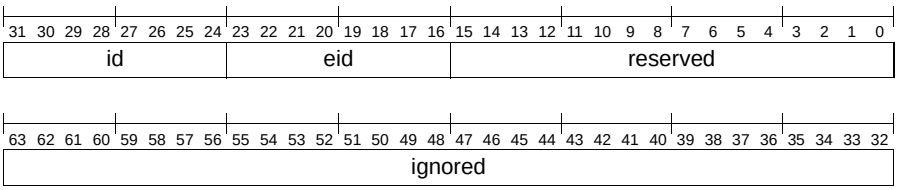
SAL_RESET functionality can be subdivided into the following phases:

- Initialization phase
- BSP identification phase
- Platform initialization phase
- OS 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® IA-64 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 a MP system. For uniprocessor (UP) systems, SAL must initialize this register prior to enabling interrupts. The OS 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 OS 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. The OS is free to change the virtual address component in the IOBASE register value but the value must be aligned on a 64 MB address boundary.

3.2.2 Bootstrap Processor Identification Phase in an MP 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 a MP configuration. The APs will set up processor-specific resources such as the Interrupt Vector Address (IVA) and enter 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 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 late self-test, they will return to the rendezvous state (EM_Rendezvous_2).

The BSP continues with platform initialization, loading the EFI firmware which, in turn, searches for bootable devices, loads the OS 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

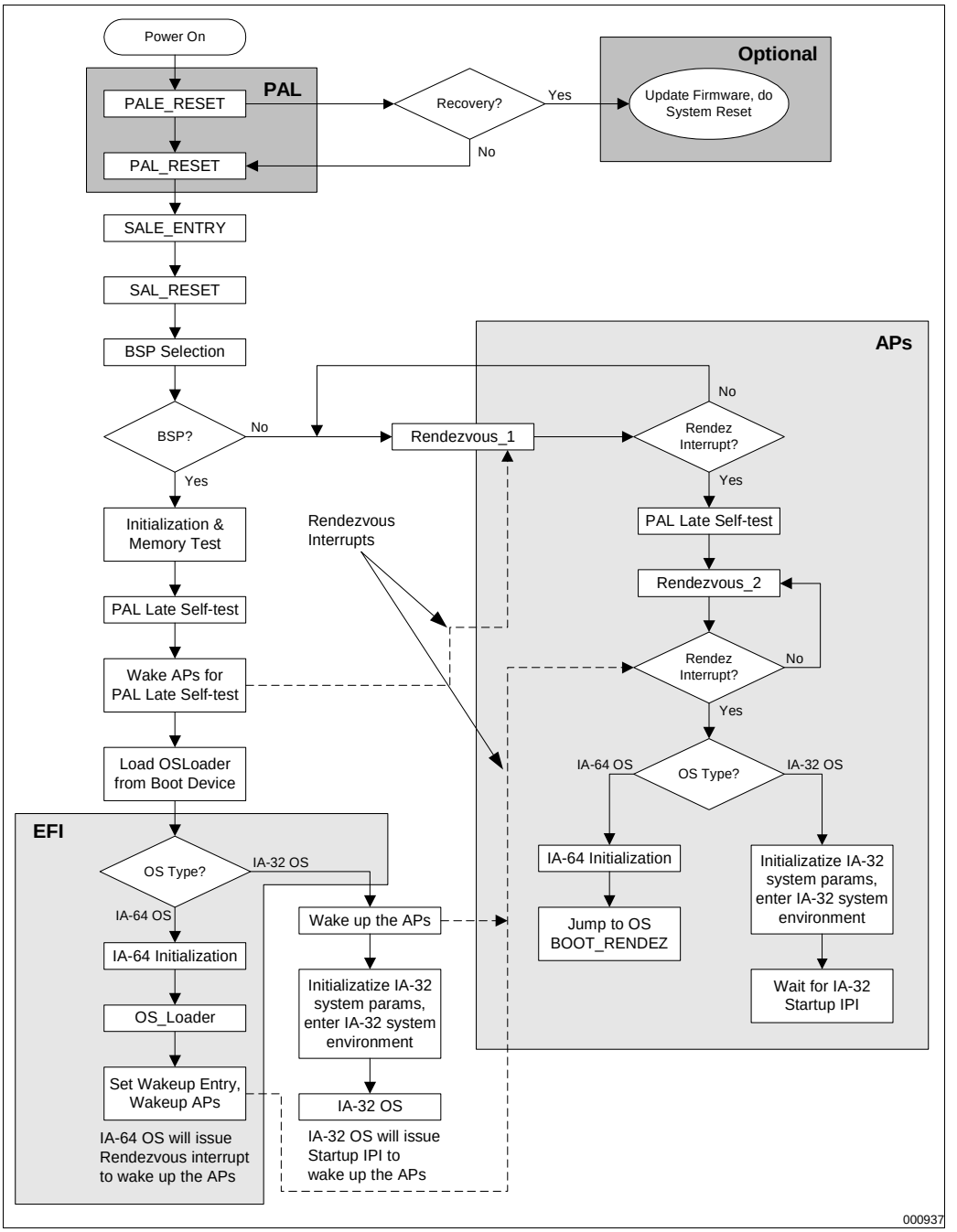
The rendezvous functionality is required only in MP environments and this functionality is utilized in two different situations:

- 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 OS 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 OS 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 semaphore to be used for the wake up. Refer to “SAL_MC_SET_PARAMS” on page 9-13 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 semaphore.

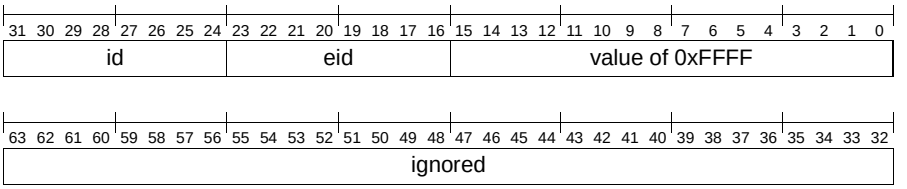
If external interrupt mechanism is chosen, APs will disable interrupts and poll the local SAPIC IRR register for the bit corresponding to the selected rendezvous interrupt to be set. The Task Priority Register (TPR) must be set such that a read of the IVR register will return the rendezvous interrupt 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 Multi-processor Configuration



If a memory semaphore mechanism is chosen, APs will disable the interrupts and poll the memory semaphore 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 semaphore to zero, execute the next phase of SAL code and, if necessary, return to the wait loop.

Figure 3-3. Memory Semaphore Format



SAL exports details of the wake-up mechanism to the OS through the SAL System Table (refer to Table 3-2) so that the OS kernel code on the BSP may wake up the APs when appropriate. While memory semaphore 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 OS. The OS shall not use the indicated external interrupt vector for its purposes until it takes over the IVA. The OS on the BSP will invoke the SAL_SET_VECTORS procedure to set the continuation point for the APs within the OS 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 ROM. 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 a 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 (not functionally restricted). This PAL procedure will not return to the SAL if the processor under test experiences a catastrophic failure. SAL must contain necessary logic to select a new BSP, if necessary. SAL shall shut down the system if there is not even a single healthy or a performance restricted processor on the system.

   After this point, the memory stack and RSE can be tested and enabled in the IA-64 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. RAM test is implementation-dependent. RAM test includes test 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 the IA-64 platforms, the autoscan function needs to be performed by the firmware SAL code in the IA-64 ISA.
7. Initialize the interrupt controllers with 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 IA-64 SAL code, 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 OS may cover the entire space using the same ITR. Refer to the *Intel® IA-64 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 IA-64 portion of the SAL code 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 ROM. 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 a MP environment.
12. Register the SAL_PMI entrypoint in RAM with PAL. This step must be accomplished on all the processors in a MP environment.
13. Call the PAL_MC_REGISTER_MEM procedure specifying where PAL code may deposit some minimal processor state information so that PAL code has sufficient resources 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_PROCESSOR_SET_FEATURES procedure. These steps must be accomplished on all the processors in a 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 IA-64 and suitable support needs to be provided by the IA-64 trap/fault handler code. These interactions are more fully described in Volume 2, Chapter 10 of the *Intel® IA-64 Architecture Software Developer's Manual*, and Chapter 7. As a side effect of supporting IA-32 Option ROMs, it is possible to have some of the SAL code implemented in IA-32 ISA.
18. Copy the EFI code into memory and transfer control to it. EFI firmware will search for bootable devices, load the OS 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 OS Boot Phase

This phase is executed only on the BSP. Refer to the *EFI Specification* for details of booting IA-64 OSes. If the selected OS is a Legacy IA-32 OS, SAL does the following:

1. SAL will construct a 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 OS. 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 OS. 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® IA-64 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 OSes 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® IA-64 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 OS 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 OSes and attempt to boot a different OS.

6. Some additional parameters are needed in a 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 OS 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 OS Loader Handoff State

The handoff to an IA-32 OS is compatible with the PC-AT industry standards. The handoff from firmware to the IA-64 OS 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 the IA-64 system registers at the time of handoff to the OS 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
- 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 OS-dependent MP rendezvous code. The OS code on the BSP registers this entrypoint by invoking SAL_SET_VECTORS, supplying the physical address of OS code that is 16-byte aligned. SAL exports details of the wake-up mechanism to the OS through the SAL System Table (refer to Table 3-11) so that the OS kernel code on the BSP may wake up the APs when appropriate. When SAL on the APs receives the wake-up, it will transition the APs to the registered OS_BOOT_RENDEZ entrypoint. Refer to Section 3.2.2.1, "Rendezvous Functionality" for additional details.

The state of the IA-64 system registers at the time of handoff to the OS_BOOT_RENDEZ is similar to that for the BSP.

3.2.7 SAL System Table

SAL uses the SAL System Table to export a variety of information to the OS Loader. The pointer to the SAL System Table is provided by EFI to the OS 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. OS software must step through each entry until it reaches the ENTRY_COUNT. The entries are sorted on entry type in ascending order. 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 IA-64 SAL specification supported by the SAL implementation in binary coded decimal (BCD) format. Byte 8 – Minor Byte 9 – Major
ENTRY_COUNT	10	2	The number of entries in the variable portion of the table. This field helps software in identifying the end of the table when stepping through the entries.
CHECKSUM	12	1	A modulo checksum of the entire table and the entries following this table. All bytes including the Checksum bytes must add up to zero.
RESERVED	13	7	Unused, must be zero.
SAL_A_VERSION	20	2	Version Number of the SAL_A firmware implementation in BCD format. Byte 20 – Minor Byte 21 – Major
SAL_B_VERSION	22	2	Version Number of the SAL_B firmware implementation in BCD format. Byte 22 – Minor Byte 23 – Major
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.

Table 3-2. SAL System Table Header (Continued)

Field	Offset (in bytes)	Length (in bytes)	Description
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 OS)
4	16	Purge Translation Cache (PTC) Coherence Descriptor
5	16	AP Wake-up Descriptor

a. All other types are reserved.
b. Not required for IA-64 OSes.

3.2.7.1 Entrypoint Descriptor Entry

The Entrypoint Descriptor entry provides the addresses in memory of PAL_PROC, SAL_PROC that may be used by the OS to invoke the procedures within the PAL and the SAL. When the OS 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 are used only while booting an IA-32 OS. IA-64 OSes obtain similar information from the EFI firmware component. The Memory Descriptor Table 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 64KB boundary as a minimum, for optimal TLB management. Note that memory usage values (byte 7 of the MDT entry) may change within a 64KB memory block and hence it is legal to have more than one MDT entry describing a 64KB memory region as long as the memory attribute (WB, WC, UC or UCE) does not change within that 64K 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 OS. 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 640K) as Boot Services Data in the Memory Usage field of the Memory Descriptor Table entries.

The EFI firmware component communicates the SAL's requirements for virtual address mappings to the OS. Once the OS 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 OS 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: 0: No 1: Yes
2	1	Encoded value of current Memory Attribute[b] setting in bits 0-2: 000: WB 100: UC 101: UCE 110: WC
3	1	Page Access Rights set up by SAL for the memory range[b]:
4	1	Memory Attributes[b] supported: Bit 0: WB Bit 1: UC Bit 2: UCE Bit 3: WC
5	1	Reserved (must be zero)

Table 3-5. Memory Descriptor Entry (Continued)

Offset (in bytes)	Length (in bytes)	Description[a] (unsigned integers)	
6	2	Memory Type (byte 6) 0 = Regular Memory	Memory Usage (byte 7) 0 = Unspecified[c] 1 = PAL Code 2 = Boot Services Code 3 = Boot Services Data 4 = Runtime Services Code 5 = Runtime Services Data 6 = IA-32 Option ROM 7 = IA-32 System ROM 8 = ACPI Reclaim Memory[d] 9 = ACPI NVS Memory 10 = SAL PMI Code 11 = SAL PMI Data 12 = Firmware Reserved Memory[e] 128-255 = Reserved for OEM
		1 = Memory mapped I/O	0 = Unspecified 1 = I2O Hidden space hole 2 = Video Memory 3-127 = Reserved 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® IA-64 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 OS after it reads the *Advanced Configuration and Power Interface Specification* tables.
e. This area is not visible in the IA-32 OS environment.

The SAL also provides the memory type and usage information to the EFI. Refer to the *EFI Specification* for details. The following table specifies the mapping between Memory Descriptor Table 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 1 = PAL Code 2 = Boot Services Code 3 = Boot Services Data 4 = Runtime Services Code 5 = Runtime Services Data 6 = IA-32 Option ROM 7 = IA-32 System ROM 8 = ACPI Reclaim Memory 9 = ACPI NVS Memory 10 = SAL PMI Code 11 = SAL PMI Data 12 = Firmware Reserved Memory 128-255 = Reserved for OEM	EfiConventionalMemory EfiPalCode EfiBootServicesCode EfiBootServicesData EfiRuntimeServicesCode EfiRuntimeServicesData EfiRuntimeServicesCode EfiRuntimeServicesCode EfiACPIReclaimMemory EfiACPIMemoryNVS EfiRuntimeServicesCode EfiRuntimeServicesData EfiRuntimeServicesData EfiRuntimeServicesCode
1 = Memory mapped I/O	<all values>	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 describes the features implemented on the platform. Refer to the *IA-64 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: Bit 0: 1 if Bus Lock is implemented Bit 1: 1 if the chipset supports redirection hint for interrupt messages originating from the platform (lowest priority interrupt) Bit 2: 1 if the chipset supports redirection hint for IPI messages originating from the processors Bits 3-7 = Reserved
2	14	Reserved

3.2.7.4 Translation Register Descriptor Entry

The Translation Register Descriptor entries describe the parameters used by the SAL during insertion of the TRs. These entries will be used by the OS to purge SAL's TRs after the OS 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: 0: Instruction Translation Register 1: Data Translation Register Other values: Reserved
2	1	Translation Register number
3	5	Reserved
8	8	Virtual address of the area covered by the Translation Register. Bits 61-63 of this field indicate the Region Register number.
16	8	Encoded value of the page size covered by the Translation Register. Refer to the *Intel® IA-64 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 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 OS.

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 describes the mechanism for waking up APs in an MP environment. Refer to Section 3.2.2.1, “Rendezvous Functionality” for details on OS 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: 0: External interrupt Other values: Reserved
2	6	Reserved (must be zero)
8	8	External Interrupt vector in the range of 0x10 to 0xFF

3.3 IA-64 OS Loader Requirements

The firmware will jump to the IA-64 OS Loader with the handoff state described in the *EFI Specification*. Included in this state information is a pointer to the SAL procedures the OS can invoke. These procedures are described in Chapter 9.

This section describes the requirements on the OS Loader while operating under the SAL execution environment.

3.3.1 Fault Handling

This section describes the guidelines to the OS Loader code as regards fault handling.

After the OS is completely loaded, it will take over the IVA, and replace the SAL environment with its own memory management. Until that time, the OS shall use SAL's virtual memory environment — IVA, Interrupt controller mode, TC mappings, etc., and it shall not change any of these resources.

The OS 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 OS 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 VHPT. If VHPT is not used, the Page Table Address (PTA) need not be initialized. SAL will turn off the PTA.ve bit to disable the processor walking the VHPT. VHPT is an optional feature of the IA-64 architecture. Avoiding VHPT usage also permits the IA-32 support code to operate out of ROM.
- Alternate Data TLB fault: This TLB fault occurs during data accesses if SAL does not implement the VHPT.
- 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 IA-64 ISA. 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 OS 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. SAL and OS Loader code must be compiled with speculation off, thereby avoiding the use of the above instructions. Turning off speculation should not have any impact on performance since most of SAL code relies on strong ordering.
- SAL shall not use advanced load (LD.a) or check load (LD.c) instructions, hence ALAT entries created by OS 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 *IA-64 Software Conventions and Runtime Architecture Guide*.

The OS 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 OS Loader code with speculation off.
- Unaligned fault: The OS Loader shall not make data references to misaligned data.
- General Exception fault: The OS 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 OS 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 OS Loader must not cause other traps or faults such as Debug, Single step, Taken branch, etc. Normally, the OS 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 OS 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 IA-64 OSes. The OS 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 OS Loader. Ideally, the OS Loader will set up the TRs for its memory mappings and not cause TLB faults. However, should the OS 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 OS 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

IA-64 virtual address is 85 bits wide and IA-64 physical address 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 IA-64 TLB mechanism as described in the *Intel® IA-64 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 OS Loader and jump to it. The OS Loader will load the rest of the OS using the firmware boot services procedures. While SAL can operate with identity mapping, there may be a need for the OS 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 OS Loader may need to load additional layers of software and fix up address relocations using virtual addressing. The OS Loader may need to load software at physical address 0.5 GB mapped to virtual address of 2.5 GB. When OS 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, OS Loader cannot use the TLB mapping set up by SAL for this case.

This problem can be solved by using different Region registers and Region ID values for SAL and OS. Differing Region ID values ensure that earlier TC/TR entries with a different Region ID value no longer cause TLB hits. SAL will use Region ID of 0x1000 for all its TLB mappings, if physical address space is less than or equal to 2^{61} bytes and OS Loader shall be restricted from using Region ID values of 0x1000 to 0x1003 until OS is ready to take over the memory management and the IVA. If this restriction is not followed by the OS Loader, a machine check abort might result when SAL attempts to insert a TC entry using the ITC.i or ITC.d instruction.

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. SAL will use only Region Register 0 and set up the same with a Region ID value of 0x1000, if physical address space is less than or equal to 2^{61} bytes. If physical memory is larger, it will load Region Registers 1 to 3 with Region ID values of 0x1001 to 0x1003 respectively.

The OS Loader will need to refer to the data structures common to SAL and OS in the process of loading the OS kernel. Similarly, the OS will need to pass parameters to SAL through pointers in Memory Stack Pointer (SP) and Global Data Pointer (GP) registers. SAL and OS 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.

While operating in the virtual mode, the OS 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. This restriction is not relevant after the OS takes over the memory management and the IVA.

Should the OS Loader set up any of the Region Registers for its use, it must

- 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 4KB.

3.3.3 Other Restrictions on the OS

The OS 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® IA-64 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 OS may lower the CMCI, MCA and BERR promotion strategy set by SAL by invoking the PAL_PROCESSOR_SET_FEATURES procedure, but this is not recommended.

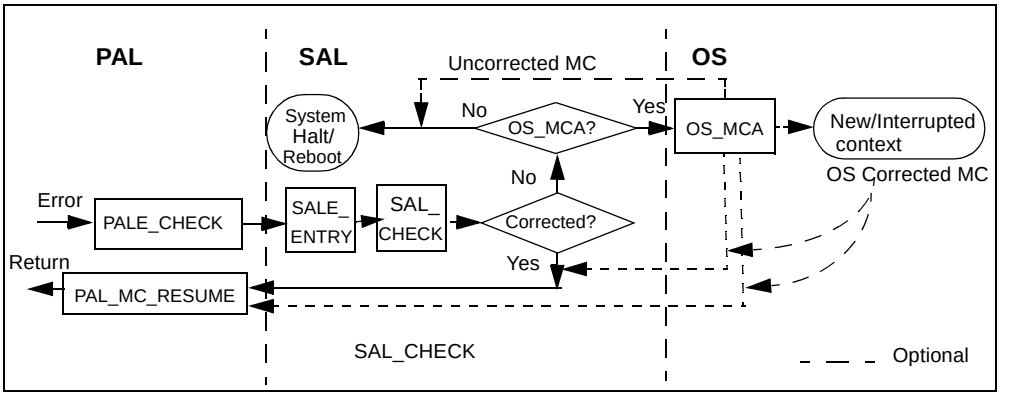
Machine Checks 4

Machine checks, including Machine Check Aborts (MCAs), Corrected Machine Check Interrupts (CMC Interrupts), and expected machine checks cause processor execution to vector to PALE_CHECK code in the IA-64 ISA. Please refer to Volume 2, Chapter 11 in the *Intel® IA-64 Architecture Software Developer's Manual* for details regarding PALE_CHECK processing. Also refer to the *IA-64 Error Handling Guide* for error handling from a system software perspective.

When PALE_CHECK has finished processing, it will pass control to SALE_ENTRY which, in turn, branches to the SAL_CHECK entrypoint in the IA-64 ISA. The entry conditions for SALE_ENTRY are described in the *Intel® IA-64 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 OS layer. The control flow differs between CMCs and uncorrected machine checks. For corrected machine checks, the OS CMC interrupt handler will be optionally 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:

- Log 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 log resources and re-enable future information collection.
- Halt the processor or platform as necessary.
- Handle 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 OS 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. This information is preserved in a platform-dependent location. Once the processor error logging 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. The OS does this by invoking SAL_CLEAR_STATE_INFO.

When multiple processors experience machine checks simultaneously, SAL selects a “monarch” machine check processor to accumulate all the error logs at the platform level and continue with the machine check processing.

SAL is responsible for reporting the state information to the OS via SAL_PROC get state information calls so that the OS can make the determination to:

- Fix the error and return,
- Create a new context and continue, or
- Reset the platform.

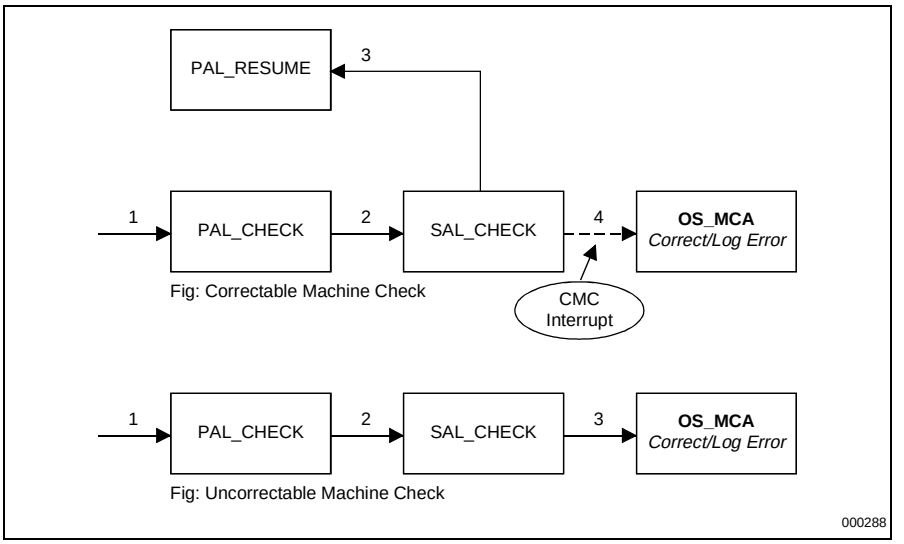
PALE_CHECK and SAL_CHECK shall not hide any architectural state from the OS_MCA layer and cannot make assumptions on whether OS_MCA would return to PAL or SAL. This permits the OS_MCA layer to run unencumbered. OS_MCA can save the processor and platform state and re-enable future machine checks as soon 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 OS 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). SAL computes and saves the checksum of this code area. Prior to entering 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. There may also 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 OS layer, SAL will call the PAL_MC_RESUME procedure to resume execution. See Section 4.6.1 for other options.

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

Figure 4-2. Machine Check Code Flow



4.2 Corrected Machine Checks

There are different categories of corrected machine checks pertaining to the IA-64 processor:

- Corrected internally by the processor hardware, e.g. single bit data ECC error on L1 cache.
- Corrected by PAL, e.g. double bit data ECC error on a clean L1 cache line, during an instruction fetch operation. To recover from this error, PAL layer may need to invalidate the L0 instruction cache and flush the L1 unified cache.

- Corrected by SAL. These are primarily platform errors that can be corrected by SAL without immediate involvement by the OS, e.g. BERR caused by a temperature/voltage sensor warning.

None of these categories will require rendezvousing of the other processors.

The SAL_CHECK processing steps for corrected machine checks are similar to the steps for the uncorrected machine checks. SAL will log the processor and platform error information and save the state of the processor and platform. SAL will perform any platform hardware-specific correction and then call PAL_MC_RESUME. PAL_MC_RESUME procedure provides an option for generating a Corrected Machine Check interrupt to the OS. The CMCV register specifies the CMC interrupt vector and its mask status.

For corrected machine checks, SAL does not call the OS_MCA layer immediately but the OS CMC interrupt handler will be invoked some time after returning to the interrupted process unless the CMC interrupt is masked by the CMCV register. The CMC interrupt handler of the OS must invoke SAL_GET_STATE_INFO procedure to obtain the processor information associated with the error(s). The SAL_GET_STATE_INFO procedure will accumulate all the processor error logs from PAL and provide the same to the OS CMC interrupt handler.

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 OS_MCA

When the OS 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 be specified so that SAL can ensure the integrity of the OS_MCA code by verifying its checksum. The OS 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 logging and state information and dumping such information for subsequent debug analysis. After registering the OS_MCA address, the OS can re-enable machine checks by clearing the PSR.mc bit to 0. The OS must call the SAL_GET_STATE_INFO_SIZE procedure to obtain the maximum size of machine check state information that SAL would return for processor and platform errors.

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 logging and state information. When it has finished this task it must call SAL_CLEAR_STATE_INFO to release these resources for future logging and state save. By calling SAL_CLEAR_STATE_INFO, the OS signifies the completion of its machine check handling. OS_MCA can then re-enable machine checks by clearing the PSR.mc bit to 0.

OS may perform any corrections on the OS controlled hardware resources. The OS 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 OS 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 OS 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 or by passing the same as an argument to the PAL_MC_RESUME procedure. Refer to the *Intel® IA-64 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." This is the recommended approach for errors corrected by the OS. The OS may set a new context in the Min-State save area and SAL will then invoke PAL_MC_RESUME to return to the interrupted or the new context. If the interrupted context was in the firmware address range and the OS decides to set a new context, the OS must take steps for resumption of the firmware code eventually, otherwise the system may become unstable.
- Correct the error and invoke PAL_MC_RESUME to return to the interrupted or a new context.
- Correct the error and jump to the interrupted context, or set a new context and jump to the new context. In this case, OS_MCA should re-enable future machine checks by setting PSR.mc bit to 0.
- In the event of an uncorrected error, return to SAL_CHECK with the uncorrected status value and an indication for SAL to halt or reboot the system.
- In the event of an uncorrected error, reboot the system.

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

4.4 Procedures used in Machine Check Handling

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

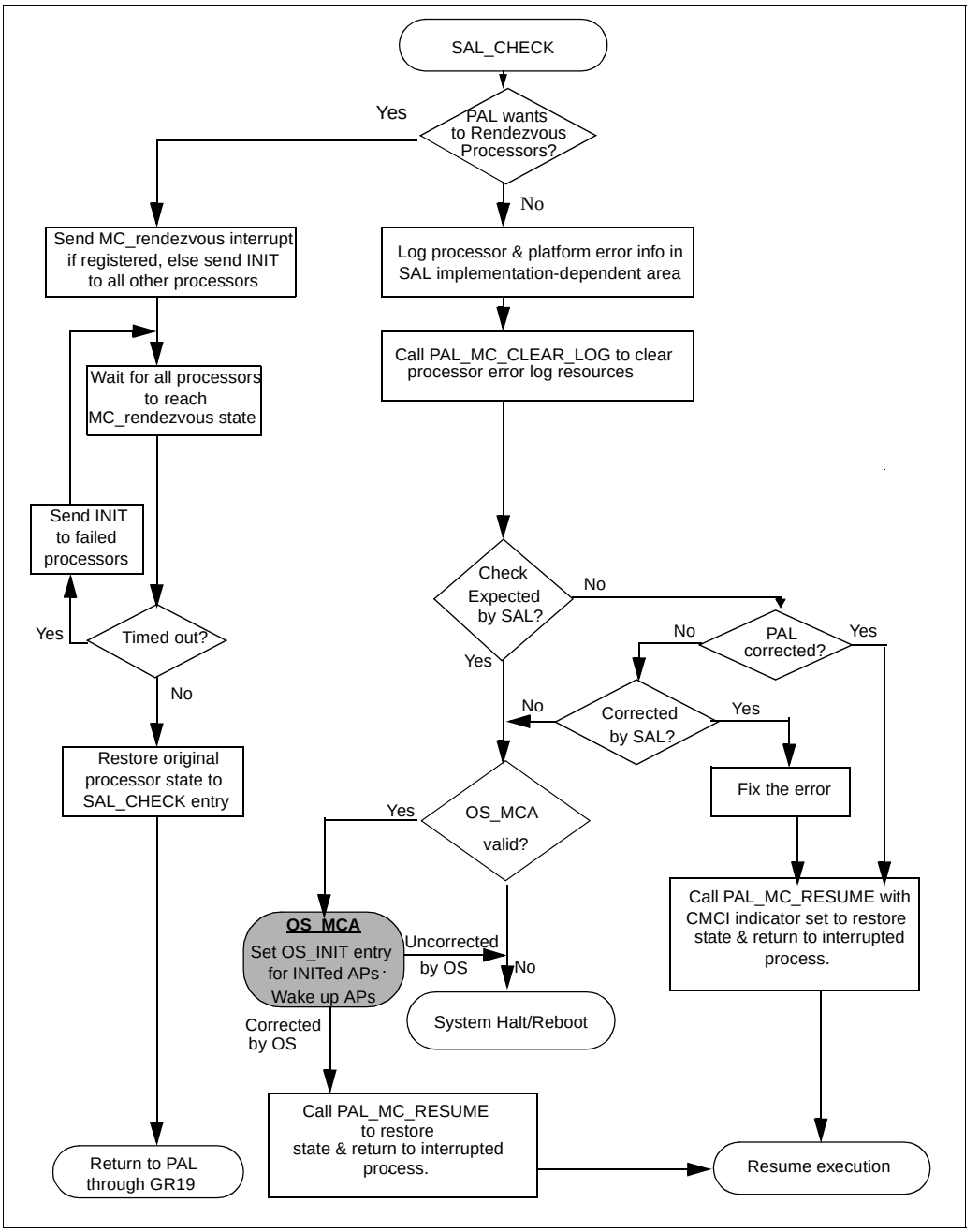
Following are typical PAL procedures that may be invoked by SAL_CHECK:

- 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_GET_FEATURES
- PAL_MC_REGISTER_MEM
- PAL_MC_EXPECTED

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



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

- PAL_MC_DRAIN

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

- SAL_MC_SET_PARAMS
- SAL_SET_VECTORS

OS_MCA may invoke any of the PAL and SAL procedures. Following are typical SAL procedures that may be invoked:

- SAL_MC_RENDEZ
- SAL_GET_STATE_INFO
- SAL_GET_STATE_INFO_SIZE
- SAL_CLEAR_STATE_INFO

4.5 Machine Checks in MP Configurations

There are certain machine check scenarios that require additional actions and considerations in MP configurations. For example, if a recoverable cache error occurs that requires the flushing of modified data to memory, the other processors unaffected by the machine check in the system (non-monarch processors) must be rendezvoused prior to the flush in order to maintain error containment in the system. Refer to Section 3.2.2.1, “Rendezvous Functionality” for details of how the rendezvous mechanism works.

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).

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 resume forwarding transactions. Also, one can imagine a platform cache situation similar to the one described above. Suffice it to say these conditions exist.

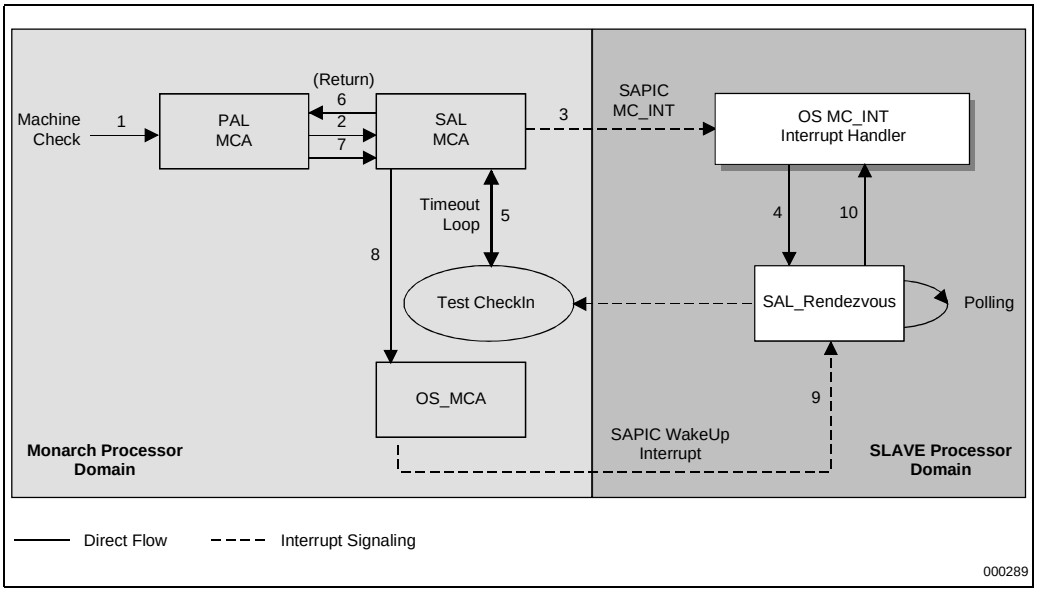
In order to facilitate these types of situations, the OS does the following:

- 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 OS will employ to wake up the non-monarch processors at the end of machine check processing.

On receipt of the MC_rendezvous interrupt, the OS on the non-monarch processors will:

- Disable further interrupts and PMI.
- 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.

Figure 4-4. Normal SAL Rendezvous Flow

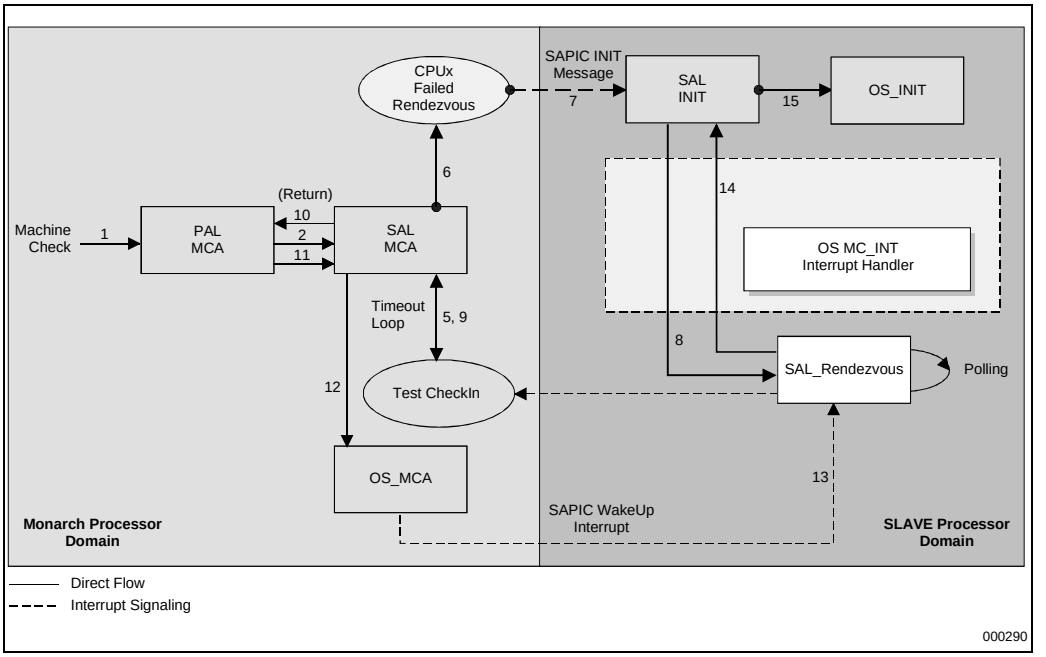


SAL on the monarch processor will wait a specified amount of time for the signalled processors to enter the SAL_MC_RENDEZ procedure. The wait time is specified as a parameter to the SAL_MC_SET_PARAMS procedure. Assuming all processors report in as expected, the PAL and SAL will perform the appropriate state save functions and proceed to the OS_MCA entrypoint to allow the OS to take the appropriate error recovery actions.

In situations where either the OS 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. 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. If 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® IA-64 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 processors rendezvoused using the MC_rendezvous interrupt message, the continuation point is merely a return from the SAL_MC_RENDEZ procedure. It is the responsibility of the OS to clear the IRR bits for the MC_rendezvous interrupt and the wake up interrupt, if any. The OS must re-enable future interrupts, PMI and machine checks.

Figure 4-5. Failed SAL Rendezvous Flow



If some non-monarch processors were rendezvoused using an INIT IPI message, their continuation point on wake up would be the OS_INIT procedure registered for the monarch by the SAL_SET_VECTORS (INIT) call. OS must register this entrypoint prior to the wake up, else SAL will reset the system. Refer to the Section 5.3, “OS_INIT Handoff State” for the parameters on entry to the OS_INIT procedure.

It should be noted that some 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 logs at the platform level. Once this is done, OS_MCA procedure will be invoked on all the processors that experienced the machine checks in a serial fashion. The OS_MCA layer may need to implement a similar “monarch” processor selection for the error recovery phase.

4.6 OS_MCA Handoff State

The OS_MCA interface defines the boundary between SAL_CHECK and the OS machine check handler, OS_MCA. 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_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® IA-64 Architecture Software Developer's Manual*) except as below:

GR1 = OS_MCA Global Pointer (GP) registered by OS (OS'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 PAL_CHECK and hence was 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 by PAL but was unsuccessful

GR12 = Return address to a location within the SAL_CHECK procedure
GRs13-31 = Refer to the *Intel® IA-64 Architecture Software Developer's Manual*
BR0 = Unspecified

Note: On entry into SAL_CHECK, the RSE has been set to enforced lazy mode configuration. he OS must not make cacheable accesses to the MinState area, else machine checks might occuras a result of a cache hit to an uncacheable page

4.6.1 Return from OS_MCA Procedure

The OS_MCA procedure may or may not return to SAL_CHECK in the case of uncorrected machine checks. If OS_MCA procedure does return to 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 OS 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 execute the processor-dependent INIT handler (PALE_INIT), in the IA-64 ISA. 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® IA-64 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 OS 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 OS to register separate entrypoints for the first processor (monarch) to enter the SAL_INIT layer and subsequent processors (non-monarchs).

INIT is also used during machine check handling in MP environments to transition the non-monarch processors to the rendezvous state. SAL code must recognize this condition using its internal variables and call SAL_MC_RENDEZ procedure. It must not invoke the OS INIT handler for this situation.

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.
- If the INIT was intended to transition APs to rendezvous state during a MP platform machine check, SAL_MC_RENDEZ procedure needs to be invoked. Refer to Section 4.5, "Machine Checks in MP Configurations" for details.
- If INIT is not due to a MP platform machine check rendezvous, check if OS_INIT handlers for the monarch and non-monarch processors are registered and that both of them are valid. When the OS_INIT procedures and their lengths were registered with SAL, SAL would have computed and saved the checksums of such code. On receipt of INIT, SAL verifies the checksum of the code at the OS_INIT procedure addresses before jumping to the same.

- 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 OSes to effect a crash dump on a hung system. On IA-64 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 OS 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.

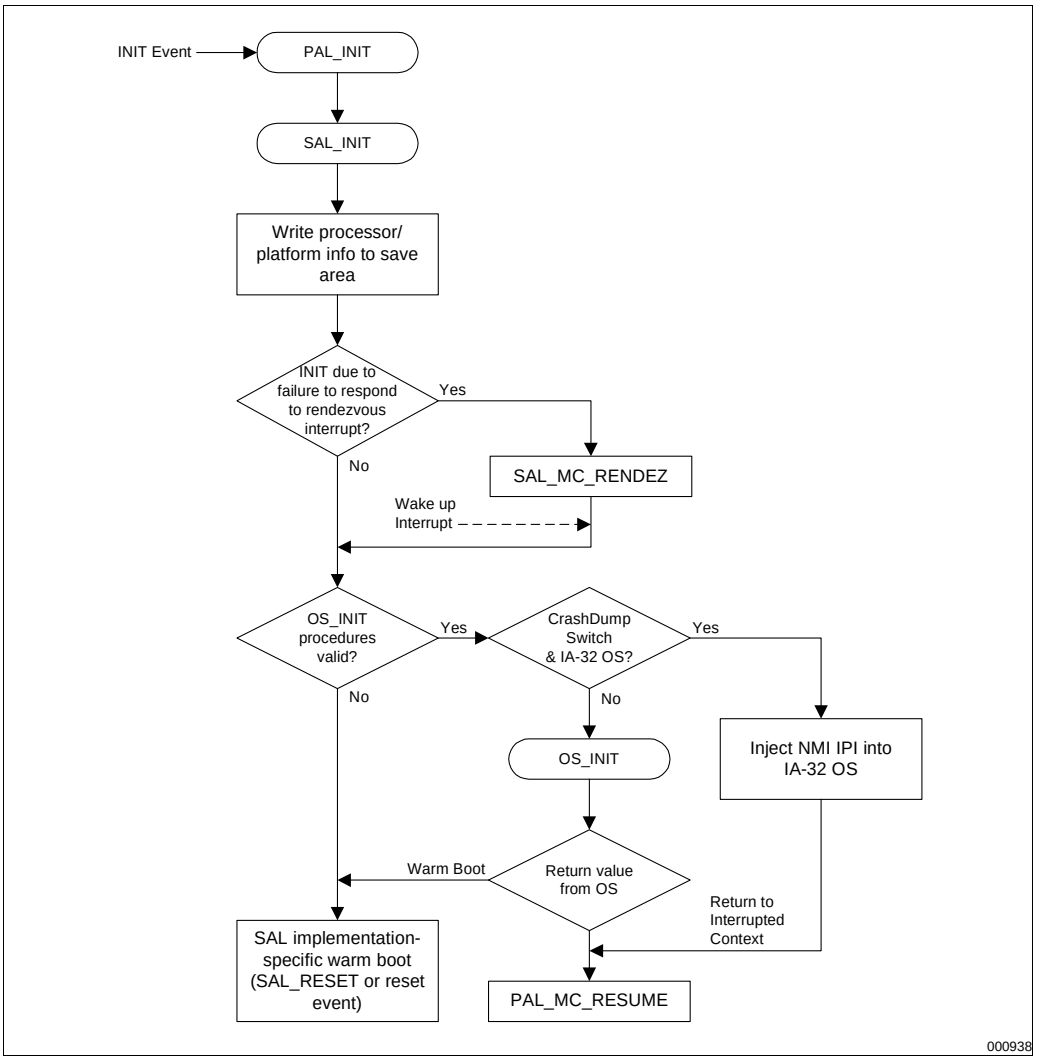
5.2 OS_INIT

OS_INIT is an entrypoint into the OS to deal with the initialization event. The exact definition of OS_INIT functionality is OS-dependent. SAL_SET_VECTORS is called by the OS 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 OS intends to make the monarch selection in the OS 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.

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. By calling SAL_CLEAR_STATE_INFO, the OS signifies the completion of its INIT processing. OS_INIT can then re-enable further INITs and machine checks by clearing the PSR.mc bit to 0.

The OS_INIT handler may return to SAL with an indication to effect a warm reset or a return to the interrupted context. OS_INIT may alternatively invoke PAL_MC_RESUME to return to the interrupted context. 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® IA-64 Architecture Software Developer's Manual* for the layout of this structure.

Figure 5-1. SAL_INIT Control Flow



5.3 OS_INIT Handoff State

The OS_INIT interface defines the boundary between SAL_INIT and the OS 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® IA-64 Architecture Software Developer's Manual*) except as below:

GR1 = Physical address of the OS_INIT Global Pointer (GP) registered by OS (OS'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 events other than CrashDump switch assertion
- 1 = Received wake up signal on this processor at the end of an OS_MCA corrected machine check
- 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® IA-64 Architecture Software Developer's Manual*
BR0 = Unspecified

Note: On entry into SAL_INIT, the RSE has been set to enforced lazy mode configuration. The OS must not make cacheable accesses to the MinState area, else machine checks might occur as a result of a cache hit to an uncacheable page.

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

If OS_INIT procedure returns to 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 OS must restore the processor state to the same as on entry to OS_INIT except as follows:

GR1 = 0 if SAL must warm boot the system
= 1 if SAL must return to interrupted context using PAL_MC_RESUME
GR2-9 = Unspecified
GR10 = GP (Physical address value) for SAL
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

BR0 = Unspecified
PSR = Same as on entry from SAL_INIT except that PSR.mc may be either 0 or 1

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 OS, 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.
- If a processor enters SAL_INIT as the result of a platform machine check rendezvous event, then the SAL_INIT must invoke the SAL_MC_RENDEZ procedure. Normally, the OS would have registered an interrupt using the SAL_MC_SET_PARAMS procedure to register the external interrupt vector to be used to interrupt the OS on the processors unaffected by the machine check. On receipt of such an interrupt, the OS would invoke the SAL_MC_RENDEZ procedure. If for some reason any of the processors do not respond to this interrupt, or if the OS fails to register such an interrupt vector, then the processor handling the machine check will INIT such processors and they will enter SAL_INIT for the platform machine check rendezvous event. Since all the processors reach SAL_MC_RENDEZ, the effect is the same (almost) as if the processor had responded to the interrupt. The difference is that processors entering SAL_MC_RENDEZ through SAL_INIT will be less likely to be recoverable.

  At the end of machine check processing, OS_MCA procedure on the monarch processor will wake up all the other processors using the wake up mechanism specified by the SAL_MC_SET_PARAMS procedure. The processors that received the INIT would jump to the registered OS_INIT procedure for the monarch processor. The OS_INIT procedure may analyze the reason why the processor needed the INIT (or reasons for not responding to the MC_rendezvous interrupt). If INIT occurred when PSR.ic bit was 1, there is no loss of machine state. OS_INIT can return to SAL specifying resumption of the interrupted context by invoking PAL_MC_RESUME.

Platform Management Interruptions 6

Platform Management Interruptions (PMIs) provide an OS-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® IA-64 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 OS 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 OS 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 below).

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 a 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® IA-64 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 is entered in the IA-64 ISA 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 a 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 7

7.1 IA-32 Support Model

This chapter describes the 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 IA-64 processors, refer to Volume 1, Chapter 6 and Volume 2, Chapter 10 of the *Intel® IA-64 Architecture Software Developer's Manual*.

IA-32 support code in SAL cannot be used after an OS (IA-32 or IA-64) has taken control of the translation resources. Most IA-64 OSes will provide their own IA-32 support code and not use the code in SAL. If the user boots an IA-32 OS, SAL would have invoked the PAL_ENTER_IA_32_ENV procedure which activates the PAL layer in support of IA-32 OSes 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® IA-64 Architecture Software Developer's Manual*.

During the platform initialization phase of the boot sequence, the IVA may point to a 32 KB IVT in ROM. 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

IA-64 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 OS. 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 IA-64 SAL layer 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 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 IA-64 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 IA-64 ISA. SAL will provide support the Extended Memory Move function (IA-32 INT 0x15, subfunction 0x87) for moving data to and from addresses above 1MB.
- 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 IA-64 code. Refer

to the *Intel® IA-64 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:

  - CLTS, HLT, INT 3, INTO, INVD, INVLPG, IRET, IRETD, MOV SS, POP SS, 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 OS kernel takes over. Thus, an IA-32 or IA-64 OS 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 OS 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 the IA-64 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 IA-64 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 IA-64 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 IA-64 software interruption handlers. 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® IA-64 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 CMPXCHG, LOCK, XADD, XCHG instructions. This intercept can be avoided by enabling the lock feature in the IA-64 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® IA-64 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, subfunction 0x87) in order that real mode code may move data to and from addresses over 1MB without requiring a transition to the IA-64 instruction set. 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:
  1. Get the Software interrupt number *nn* from ISR.vector.
  2. 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.
  3. Store the two byte FLAGS on IA-32 stack.
  4. Store the segment:offset address of the IA-32 instruction following the *INT nn* on IA-32 stack.
  5. Store the IA-32 *segment:offset* addresses in the appropriate IA-64 registers corresponding to IP, CS selector, CS

segment descriptor and transition to IA-32 code using *RFI* instruction.

6. 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 IA-64 ISA 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 OS 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

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

PSR Bit	Description	Entry	Exit	Class
mfh	Floating-point registers f16-f127 written	C	C	preserved
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
dfh	Disabled FP register f16 to f127	C	C	unchanged
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[c]
		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. 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 *IA-64 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; in0 -in95: input arguments (SAL index must be in0) loc0 – loc95: local variables out0 – out95: output arguments

The GP for the SAL code should be known to system software as SAL passes it as one of the boot parameters. The caller must initialize the GP and SP prior to calling a SAL procedure. 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 *IA-64 Software Conventions and Runtime Architecture Guide*. SAL shall not use the floating-point registers 32 to 127, thus eliminating the need for the OS 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 *IA-64 Software Conventions and Runtime Architecture Guide*.

8.1.7 Branch Registers

The conventions for these registers follows the *IA-64 Software Conventions and Runtime Architecture Guide*.

8.1.8 Application Special Registers

The application registers follow the *IA-64 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 OS 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 OS runtime execution environment must provide the proper mapping for the parameter buffers.

8.2 Software Interface Conventions for SAL Procedures

A generic IA-64 interface is provided between IA-64 OS and SAL. IA-64 OS 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 range. The 0x03XXXXXX range is reserved exclusively for Firmware vendors.

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:

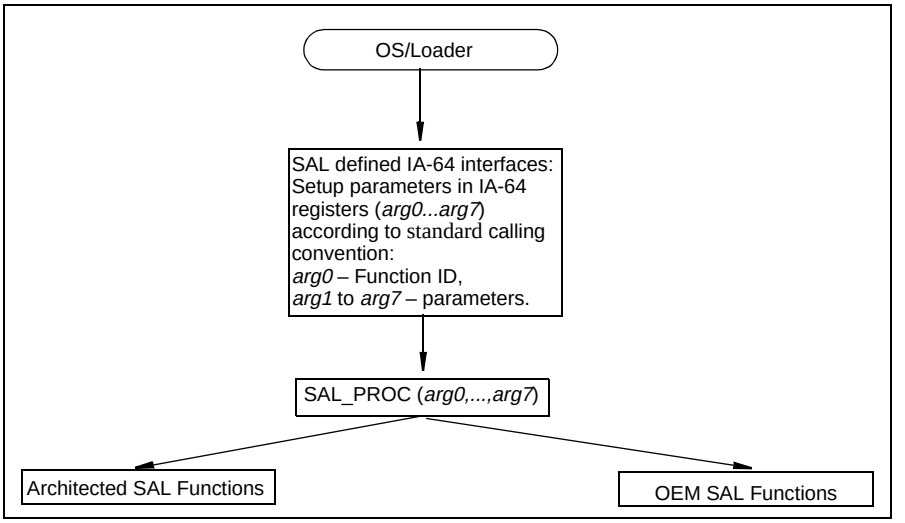
ret0 – return status: positive number indicates successful, negative number indicates failure.

ret1 to ret3 – other return parameters.

8.2.1 Control Flow of the SAL Interface

OS/Loader follows the standard calling convention to call both architected and OEM specific SAL functions. OS/Loader sets up the appropriate parameters in IA-64 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 OS/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. 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.

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. The following table 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
–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 OS when it is executing. These services provide a platform-independent interface for hardware components. runtime services contain procedures called by the OS to access platform hardware features on behalf of the OS. runtime services should take no more time to perform an action than it would take the OS 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:

- OS runtime execution environment. The normal OS execution environment is with translation on and interrupts enabled but OS may choose to call SAL runtime services in physical mode.
- OS Machine Check and INIT 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 IA-64 calling convention. SAL runtime services shall be implemented completely in the IA-64 ISA.
- SAL procedures are not re-entrant with respect to any runtime service (including itself).
- SAL procedures are not MP-safe except for the SAL_MC_RENDEZ, SAL_CACHE_FLUSH and SAL_CACHE_INIT procedures. The OS is required to enforce single threaded access to the other SAL procedures.
- Architected SAL runtime procedures are called either in virtual or physical mode under the OS 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 OS execution environment is with translation on and interrupts enabled but OS 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 OS in the Entrypoint Descriptor (refer to

Table 3-4 on page 3-12). 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 OS Loader will indicate the SAL's requirements for virtual address mappings. (Refer to the *EFI Specification* for details). In a MP configuration, the virtual addresses registered by the OS must be valid globally on all the processors in the system. The *EFI Specification* also provides the interfaces for the OS 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. These dependencies are described in Table 9-1 below. Prior to invoking the SAL procedures in virtual mode, the OS 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 OS (i.e. without changing the PSR.dt, PSR.rt and PSR.it bits). While invoking these SAL procedures, the OS must provide the appropriate translation resources required by PAL (i.e. ITR and DTC covering the PAL code area). However, if a particular PAL procedure needs to be invoked in physical mode, SAL will turn off translations and then invoke PAL.
3. The SAL_UPDATE_PAL procedure will invoke some PAL procedures in the firmware address space. The OS must register the virtual address of the firmware address space (ending at 4 GB). The OS must provide a contiguous virtual address mapping for the entire firmware address space. If SAL_UPDATE_PAL procedure is called in virtual mode, SAL will compute the virtual addresses of the relevant PAL procedures in the firmware address space and shall call the same in virtual mode.
4. The OS must register the virtual addresses of the Firmware Reserved Memory (refer to Table 3-5 on page 3-13). Such registration must be done prior to making SAL calls in virtual mode and the OS must provide a contiguous virtual address mapping for each of the data areas.

.

Table 9-1. SAL Procedures Invoking PAL Procedures

SAL Procedures	PAL Procedures
SAL_CACHE_FLUSH	PAL_CACHE_FLUSH
SAL_GET_STATE_INFO	PAL_MC_ERROR_INFO

9.1.2 Access to Resources not Supported by OS

In order to access resources for which the OS 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


;;                          //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 OS functions.

9.2 SAL Procedure Summary

.

Table 9-2. SAL Procedures

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

SAL_CACHE_FLUSH

Purpose: To flush the instruction or data caches.

Calling Conventions: Standard. Callable by the OS 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 to be flushed: 1 = instruction cache 2 = data cache 3 = instruction & data cache 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, at all levels of cache hierarchy, controlled by the platform and the processor. The *i_or_d* parameter specifies the instruction and/or data caches. Unified caches are flushed with both instruction and data caches. This procedure has the effect of invalidating all instruction cache lines, or causing a writeback and then invalidating all data cache lines.

This SAL procedure invokes the PAL procedure, PAL_CACHE_FLUSH. The PAL procedure may return to SAL without completing the flush operation should there be an intervening interrupt. This procedure will then re-invoke the PAL call. 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.

Platform Requirements: None

SAL_CACHE_INIT

Purpose: To initialize the instruction and data caches.

Calling Conventions: Standard. Callable by the OS 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

Description: Initializes the instruction and data caches controlled by the *platform only.* The OS 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 writeback.

Platform Requirements: None

SAL_CLEAR_STATE_INFO

Purpose: This procedure is used to invalidate the processor and platform information logged by SAL with respect to the machine state at the time of MCAs, INITs or CMCs.

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

Arguments:

Argument	Description
func_id	Function ID of SAL_CLEAR_STATE_INFO call within the list of SAL procedures.
type	The type of information being invalidated: 0 – MCA information 1 – INIT information 2 – CMC information Other values are reserved
sub-type	The type of machine check state information being cleared: 0 – Processor information 1 – Platform information Other values are reserved
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
–2	Invalid Argument
–3	Call completed with error
–4	Virtual address not registered

Description: This call will invalidate any processor or platform information logged by SAL for the specified event type. Once the log has been invalidated, any subsequent calls to SAL_GET_STATE_INFO will get a –5 return value (no information available). In a MP environment, processor log information pertains to the processor on which this call is executed and the platform log information pertains to the entire platform.

If an MCA has been logged and the OS fails to invalidate the log prior to another MCA then this may be considered fatal. For this reason the log will always be invalidated on booting. This means that the log information should be read as part of the OS_MCA handler.

This procedure enables the OS (and diagnostic software) to invalidate information obtained by SAL with respect to the machine state at the time of MCAs, INITs and CMCs. By calling this procedure, the OS signifies the completion of its machine check handling.

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 OS in physical or virtual mode.

Arguments:

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

Returns:

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

Status:

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

Description: This procedure is a runtime interface to determine the platform clock frequencies and to facilitate the OS 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.

The platform base clock frequency (*clock_freq* return parameter for *clock_type* of 0), in conjunction with the ratios returned by the PAL_FREQ_RATIOS, may be used to determine 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: IA-64 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 or CMCs.

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

Arguments:

Argument	Description
func_id	Function ID of SAL_GET_STATE_INFO call within the list of SAL procedures.
type	The type of information being requested: 0 – MCA information 1 – INIT information 2 – CMC information Other values are reserved
sub-type	The type of machine check state information being requested: 0 – Processor information 1 – Platform information Other values are reserved
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 OS (and diagnostic software) to gather information obtained by SAL with respect to the machine state at the time of MCAs, INITs and CMCs.

This call will return any information logged by SAL for the specified event *type* and *sub-type*. In response to the MCA or CMC event, the OS must call this procedure twice to obtain the processor and/or platform error information that triggered the machine check. The *processor information* returned is that of the processor that invoked this procedure. The *platform information* returns error information for memory and I/O devices.

The OS is expected to call this procedure to retrieve all data related to an event. The OS may retrieve the same information multiple times prior to clearing the log. The log is cleared by the OS calling SAL_CLEAR_STATE_INFO. Once the log has been cleared, any subsequent calls will get a –5 return value (no information available). The OS must be prepared to handle the –5 return value.

The maximum length of the buffer required to hold the requested log information is obtained by calling the SAL_GET_STATE_INFO_SIZE procedure. The OS is expected to allocate the memory

buffer according to the returned size and provide the same for the *memaddr* argument. SAL returns as many error logs as would fit into the memory buffer area provided by the *memaddr* argument.

The information returned in the *memaddr* argument will contain the error information logged for the processors, memory controller, and I/O devices (including host bridges) in the system. The exact format of the logs will be implementation dependent but the log for each type of device will follow an architected structure to allow the OS to parse the logs and extract the information. Refer to Appendix C ‘Error Log Structures’ for format of the error log information returned in the *memaddr* argument.

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

In a MP environment, processor log information pertains to the processor on which this call is executed and the platform log information pertains to the entire platform.

If an MCA has been logged and the OS fails to clear the log prior to another MCA then this may be considered fatal. Hence, the MCA log information should be read as part of the OS_MCA handler. On the other hand, if a CMC occurs prior to the OS 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* and *sub-type*, SAL shall discard the error logs for the latter occurrences.

SAL's error logs shall be cleared on a re-boot.

Platform Requirements: None

SAL_GET_STATE_INFO_SIZE

Purpose: This procedure is used to obtain the maximum size of the information logged by SAL with respect to the machine state at the time of MCAs, INITs or CMCs.

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

Arguments:

Argument	Description
func_id	Function ID of SAL_GET_STATE_INFO_SIZE call within the list of SAL procedures.
type	The type of information being requested: 0 – MCA information 1 – INIT information 2 – CMC information Other values are reserved
sub-type	The type of machine check state information being requested: 0 – Processor information 1 – Platform information Other values are reserved
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* and *sub-type*. The OS must make this call to determine the maximum size of data logged by SAL for each *type* and *sub-type* of log. The OS 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 OS 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 semaphore 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 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 rendezvous interrupt to be set.

If a memory semaphore mechanism is chosen, SAL spin loop routine will poll the memory semaphore 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 semaphore to zero and return. This procedure may be called in virtual or physical mode but when memory semaphore 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 semaphore.

The non-monarch processor will enter the spin loop routine and begin polling the wake up mechanism within 1 second after invocation of this call.

When this procedure returns, it is the responsibility of the OS 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 OS 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 OS to specify the interrupt number to be used by SAL to interrupt the OS 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 OS in virtual or physical mode.

Arguments:

Argument	Description
func_id	Function ID of SAL_MC_SET_PARAMS call within the list of SAL procedures
param_type	Unsigned 64-bit integer value for the parameter type of the machine check interface: 1 = rendezvous interrupt 2 = wake up Other values are reserved
i_or_m	Unsigned 64-bit integer value indicating whether interrupt vector or memory address is specified: 1 = interrupt vector 2 = memory address Other values are reserved
i_or_m_val	Unsigned 64-bit integer value specifying the interrupt vector or the memory address associated with the *i_or_m* parameter specified above.
time_out	Unsigned 64-bit integer value for rendezvous time out (in milliseconds). The minimum value is 1 second. Any value less than 1000 defaults to 1000.
Reserved	0
Reserved	0
Reserved	0

Returns:

Return Value	Description
status	Return status of SAL_MC_SET_PARAMS procedure
Reserved	0
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 is required for MP support. Section 3.2.2.1 provides details on how the rendezvous mechanism works in a 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 OS to register the interrupt number it wishes to use for this purpose. Typically, when the OS on the non-monarch processor receives the rendezvous interrupt, it will turn around and call SAL_MC_RENDEZ to go into a SAL spin loop routine. If the OS 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 OS.

The *param_type* parameter indicates whether the rendezvous interrupt or wake up mechanism is being specified. If *param_type* is 1, the *i_or_m* parameter is ignored.

The *i_or_m* parameter specifies whether an interrupt or memory semaphore is used. Interrupt is the only valid choice for the rendezvous function since the idea is to interrupt the non-monarch processor as quickly as possible. Either interrupt or memory may be used for the wake up mechanism and this is OS 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 semaphore must be aligned on an 8-byte boundary and coherent across the system fabric.

For the interrupt vector, a value of 0 indicates use of PMI as the interrupt mechanism. The PMI interrupt mechanism shall not be employed by IA-64 OSes as either the rendezvous or the wake-up interrupt. Only the PAL layer to support IA-32 OSes may use the PMI as the rendezvous interrupt since all the external interrupt vectors may be in use by the IA-32 OS. 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 OS.

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

Except for the 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 semaphore mechanism). This is because the OS is responsible for clearing the IRR bit associated with the wake up interrupt vector by reading the IVR and 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 with a minimum value of 1 second. This parameter is only applicable to the *param_type* of rendezvous interrupt. If the non-monarch processor does not invoke SAL_MC_RENDEZ 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 in local MCA and thus not be capable of servicing external interrupts or INIT.

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 OS 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: Bits 0..7 – Register address Bits 8..10 – Function number Bits 11..15 – Device number Bits 16..23 – Bus number Bits 24..31 – Segment number Bits 32..63 – Reserved (0) 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 OS 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: Bits 0..7 – Register address Bits 8..10 – Function number Bits 11..15 – Device number Bits 16..23 – Bus number Bits 24..31 – Segment number Bits 32..63 – Reserved (0) 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 OS in virtual or physical mode.

Arguments:

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

Returns:

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

Status:

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

Description: This procedure is used by the OS to register the new *physical* addresses of the PAL_PROC procedure in memory. If the OS 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 OS in virtual or physical mode.

Arguments:

Argument	Description
func_id	Function ID of SAL_SET_VECTORS call within the list of SAL procedures
vector_type	Type of event handler: 0 = Machine Check 1 = INIT 2 = BOOT_RENDEZ 3–64 = Reserved other values are implementation dependent
phys_addr_1	Physical address of the event handler.
gp_1	Global pointer (GP) of the event handler. This field must be a 16-byte aligned address.
length_1	Size of the event handler procedure in bytes
phys_addr_2	Physical address of the event handler.
gp_2	Global pointer (GP) of the event handler. This field must be a 16-byte aligned address.
length_2	Size of the event handler procedure in bytes

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 OS (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 OS or diagnostics in order to pass control. The OS registers the *physical* address where the specific handler resides. SAL uses these addresses to vector to on occurrence of the event.

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_1* arguments specify the entrypoint, gp-value and the length respectively of the OS_INIT procedure for the monarch and the *phys_addr_2*, *gp_2* and *length_2* arguments respectively specify the entrypoint, gp-value and the length of the OS_INIT procedure for the non-monarch processors. The entrypoints within the OS for the monarch and non-monarch processors could be the same if the OS intends to perform the monarch selection.

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 *gp_n* field has the physical address of the GP for the event handler to be called by SAL. The *length_n* argument contains the length in bytes of the OS procedure (or at least the first level OS_MCA, OS_INIT, OS_BOOT_RENDEZ procedure). If the *length_n* argument is non-zero, SAL computes and saves the checksum of the OS procedure. If this procedure were invoked in the virtual addressing mode, the OS must provide read access to the code area for calculating the checksum. Before invoking the registered OS procedure, SAL shall authenticate the OS 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 OS 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: 0= Do not store checksum of this component in its FIT entry 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) 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 OS must rendezvous all the other processors on the node whose firmware is being updated. At the end of the firmware update, the OS must invoke the PAL_MC_ERROR_INFO procedure to ascertain whether any machine checks occurred while they were masked and take corrective actions. The OS 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 OS 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 OS

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 OS had called this procedure in virtual mode). SAL will then return to the OS 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 OS. The OS 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 OS 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 invalidate its instruction cache by invoking the PAL_CACHE_INIT procedure as it would be non-coherent with respect to the data cache.
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 OS 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.

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).

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.

EFI

Extensible Firmware Interface. Firmware that provides a legacy free API interface to the OS.

EOI

End of Interrupt.

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.

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 Architecture Software Developer's Manual*.

IA-64

The new ISA with 64-bit instruction capabilities, new performance enhancing features, and support for the IA-32 instruction set.

IA-64 OS

An operating system which is written using the IA-64 code that can run IA-64 applications (IA-64, IA-32 code).

INTA

Interrupt Acknowledge.

IPI

Interprocessor Interrupts.

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 IA-64 OS for interrupting the IA-64 OS 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.

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.

MPS
Multiprocessor Specification.

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.

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

SAL_REV
The revision number of the IA-64 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 IA-64 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 64-bit IA-64 ISA extensions to the 32-bit Intel Architecture (IA-32). 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.

Swizzling

This term refers to mapping a 32-bit virtual linear address space into four virtual regions of the 64-bit virtual address space. Swizzling is defined as:

```
virtual_address{63} = 0
virtual_address{62:61} = 32-bit_virtual_address{31:30}
virtual_address{60:32} = 0
virtual_address{31:0} = 32-bit_virtual_address{31:0}
```

TLB

Translation Lookaside Buffer.

TSS

Task State Segment.

USB

Universal Serial Bus.

VHPT

Virtual Hash Page Table.

WBL

Write-back with Limited Speculation.

Error Log Structures B

B.1 Overview

The goals of the IA-64 Error Log structures is to keep it generic and flexible enough to be extensible and to abstract processor or platform implementation dependencies from the OS layers, at the same time providing as much error information as possible to the OS for error handling purposes.

B.2 Error Log Structure

The error log structure consists of two different components namely processor and platform. Both the processor and platform error log structures have a similar header, followed by the actual device specific (processor or platform) error log info. Since multiple errors are possible, the error log information will be structured in the form of a linked list of Error Log structures with each entry describing one error. The last Error Log in the linked list will contain a value of zero in the *Next_Log* field.

B.2.1 Header

The format of the header for both the platform and processor error log is as shown below:

Refer to the *Intel® IA-64 Architecture Software Developer's Manual* for explanation of fields not described in this document.

Offset	Field	Description
0	NEXTLOG	Offset of the next log from the beginning of this structure (0 if none present)
8	LOG_LEN	Length of this error log in bytes
12	LOG_TYPE	This is an unsigned integer to indicate the type of the log: 0 – Processor log 1 – Platform log
14	OEM_SUB_TYPE	Sub type of the log as defined by the OEM
16	TIME_STAMP	Timestamp recorded when MCA, INIT or CMC occurred

```
HEADER_INFORMATION:
{
          NEXT_LOG              8 bytes
          LOG_LEN               4 bytes
          LOG_TYPE              2 bytes
          OEM_SUB_TYPE          2 bytes
          TIME_STAMP            8 bytes (values in BCD format)
             Seconds               Byte 0
```

```
                Minutes                 Byte 1
                Hours                   Byte 2
                Reserved                Byte 3
                Day                     Byte 4
                Month                   Byte 5
                Year                    Byte 6
                Century                 Byte 7
}
```

The Device specific error log follows the header. For processor log, this field will contain an area that is architected for all IA-64 processors. For platform log, this field will contain information specific to the hardware implementation.

B.2.2 Processor Specific Error Log

Refer to the *Intel® IA-64 Architecture Software Developer's Manual* for explanation of fields.

```
PROCESSOR_SPECIFIC_ERROR_LOG STRUCTURE
{
  VALIDATION_BITS[1]                    8 bytes
     PROCESSOR_STATE_PARAMETER_VALID_BITBit 0
     CACHE_CHECK_VALID_BIT               Bit 1 to 6 (for cache errors 1 to
                                         6)
     TLB_CHECK_VALID_BIT                 Bit 7 to 12 (for TLB errors 1 to 6)
     BUS_CHECK_VALID_BIT                 Bit 13
     RESERVED                            Bits 14-31
     MINSTATE_VALID_BIT                  Bit 32
     BR_VALID_BIT                        Bit 33
     CR_VALID_BIT                        Bit 34
     AR_VALID_BIT                        Bit 35
     RR_VALID_BIT                        Bit 36
     FR_VALID_BIT                        Bit 37
     RESERVED                            Bit 38-63
  PROCESSOR_STATE_PARAMETER              8 bytes
  struct {                               96 bytes[2] (for cache errors 1 to 6)
     CACHE_CHECK_INFO                    8 bytes
     CACHE_TARGET_ADDR                   8 bytes
  } CACHE_ERROR_STRUCT[6]
TLB_CHECK_INFO                          48 bytes (for TLB errors 1 to 6)
struct {                                32 bytes
     BUS_CHECK_INFO                      8 bytes
     BUS_REQUESTOR                       8 bytes
     BUS_RESPONDER                       8 bytes
     BUS_TARGET                          8 bytes
  } BUS_ERROR_STRUCT
```

1. The amount of state saved by SAL is implementation dependent and SAL provides validation bits indicating the saved state information.
2. Contains a field indicating the level of cache. Refer to Cache Check Fields in the *Intel® IA-64 Architecture Software Developer's Manual.*

struct {	Processor Static Information
Minimal State Save Info Structure	Refer to the *Intel® IA-64 Architecture Software Developer's Manual*
BRs 0-7	64 bytes
CRs 0-127	1024 bytes[1,2]
ARs 0-127	1024 bytes[1,2]
RRs 0-7	64 bytes
FRs 0-127	2048 bytes
} PSI_STATIC_STRUCT	
}	

B.2.3 Platform Specific Error Log

```
PLATFORM_SPECIFIC_ERROR_LOG STRUCTURE
{
  implementation specific information for memory and I/O errors
}
```

1. The number of Control and Application registers on a processor is processor implementation dependent.
2. 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.