## Contents

1 Introduction ................................................................................................................. 1  
1.1 Purpose ............................................................................................................. 1  
1.2 Target Audience ............................................................................................. 1  
1.3 Related Documents ........................................................................................... 1  
1.4 Terminology ..................................................................................................... 1  

2 Machine Check Architecture ........................................................................................... 7  
2.1 Overview ........................................................................................................... 7  
2.2 Itanium® Processor Family Firmware Model ............................................................ 7  
2.3 Machine Check Error Handling Model ...................................................................... 8  
2.4 MCA Scope ....................................................................................................... 10  
2.5 Error Severity ................................................................................................... 11  
2.5.1 Hardware-Corrected Errors ...................................................................... 11  
2.5.2 Firmware-Corrected Errors ...................................................................... 12  
2.5.3 OS Recoverable Errors ............................................................................ 12  
2.5.4 Fatal Errors ........................................................................................... 12  
2.6 Software Handling ............................................................................................. 13  
2.6.1 PAL Responsibilities ............................................................................... 13  
2.6.2 SAL Responsibilities ............................................................................... 13  
2.6.3 Operating System Responsibilities ............................................................ 14  
2.7 Multiple Errors .................................................................................................. 16  
2.7.1 SAL Issues Related to Nested Errors ......................................................... 17  
2.8 Expected MCA Usage Model ................................................................................ 18  

3 Processor Error Handling ............................................................................................. 19  
3.1 Processor Errors ................................................................................................ 19  
3.1.1 Processor Cache Check ........................................................................... 19  
3.1.2 Processor TLB Check............................................................................... 20  
3.1.3 System Bus Check.................................................................................. 20  
3.1.4 Processor Register File Check................................................................... 20  
3.1.5 Processor Microarchitecture Check............................................................ 21  
3.2 Processor Error Correlation ................................................................................. 21  
3.3 Processor CMC Signaling .................................................................................... 21  
3.3.1 CMC Masking ......................................................................................... 22  
3.3.2 Error Severity Escalation ......................................................................... 22  

4 Platform Error Handling............................................................................................... 23  
4.1 Platform Errors ................................................................................................ 23  
4.1.1 Memory Errors ....................................................................................... 23  
4.1.2 I/O Errors ............................................................................................. 23  
4.1.3 OEM-Specific Errors ............................................................................ 24  
4.2 Platform Error Correlation................................................................................. 24  
4.3 Platform-Corrected Error Signaling .................................................................... 24  
4.3.1 Scope of Platform Errors ...................................................................... 24  
4.3.2 Handling Corrected Platform Errors........................................................... 24  
4.4 Platform MCA Signaling .................................................................................... 25  
4.4.1 Global Signal Routing ........................................................................... 26  
4.4.2 Error Escalation ..................................................................................... 27  

**Figures**

2-1 Itanium® Processor Family Firmware Machine Check Handling Model...................... 8  
2-2 Machine Check Error Handling Flow ......................................................................... 10
2-3 Error Types and Severity .................................................................11
2-4 Multiple MCA Events.................................................................17

Tables
3-1 Processor Machine Check Event Masking ........................................22
3-2 Machine Check Event Escalation ..................................................22
## Revision History

<table>
<thead>
<tr>
<th>Version</th>
<th>Description</th>
<th>Date</th>
</tr>
</thead>
</table>
| -004    | Updates for Intel® QuickPath Interconnect-based platforms.  
          | Corrected definitions in the Terminology section.  
          | Added reference to the DIG64 Corrected Platform Error Polling Interface Specification.  
          | Clarified global error signaling and error containment requirements.  
          | Removed Chapter 5: Error Records because of redundancy with the Intel® Itanium® Processor Family System Abstraction Layer Specification. | February 2010 |
| -003    | Updated links to related documents.  
          | Changes to reflect updated trademarks.  
          | Provided differentiation for machine checks vs. MCA references.  
          | Updated PAL, EFI, PMI, Data Poisoning and MCA definitions.  
          | Updated diagram for the Error Handling Flow.  
          | Removed references to IA-32 Operating Environment.  
          | Removed Chapter 6: OS Error Handling.  
          | Corrected Local and Global MCA references. | September 2003 |
| -002    | Added definitions for Domain, CMCI, and Corrected Error.  
          | Multiple clarifications to Chapter 2, Machine Check Architecture. Added sections on SAL/OS responsibilities and SAL issues with nested errors.  
          | Clarifications to Chapter 3, Processor Error Handling.  
          | Clarifications to Chapter 4, Added detail and new sections on the scope and handling of platform-corrected errors.  
          | Added detail to Error Record definition in Chapter 5, including new sections on required/optional error records.  
          | Added clarifications to Section 6.2, Identifying the Errant and Affected Threads.  
          | Added additional detail to Appendix A, Error Injection. | August 2001 |
| -001    | Initial release of the document. | January 2001 |
1 Introduction

1.1 Purpose

This Intel® Itanium® Processor Family Error Handling Guide describes error handling on Itanium®-based systems. It provides guidelines for firmware and operating systems to take advantage of the Itanium Advanced Machine Check Architecture. This document references the Processor Abstraction Layer (PAL) and System Abstraction Layer (SAL) specifications and shows how firmware, platform design, and the operating system cooperate to address corrected errors and machine check aborts.

1.2 Target Audience

This document is intended for Itanium architecture SAL developers, platform designers, and OS developers. Implementation-specific details of processors are not discussed in this document. The target audience is expected to be familiar with the PAL and SAL specifications.

1.3 Related Documents

This document refers to the following publications:


1.4 Terminology

ACPI
Advanced Configuration and Power Interface Specification.

AP
Application Processor. Any processor not selected to be the bootstrap processor.

API
Application Programming Interface.
Introduction

**BERR#**
Bus Error Signal. A platform may assert BERR# to bring all processors to a continuable MCA. BERR# may also be driven by the processor on fatal errors to bring other processors on the bus into MCA. Unlike BINIT#, BERR# does not cause the processor to dump internal state before entering MCA. On BERR# MCA, all error logging and system state (including memory dump) is available for logging. This signal is maskable by the processor’s psr.mc bit.

**BINIT#**
Bus Initialization Signal. An unmaskable system-wide signal driven by the processor or platform on bus-based systems to indicate a fatal machine check abort that requires the processor to dump most processor state to maintain error containment. After BINIT#, the processors come into MCA and min-state save area is available and some PAL error information is available for logging, but other system state is not available for logging. (For Intel QuickPath Interconnect-based systems, refer to ERROR[1]_N.)

**BIOS**
Basic Input/Output System. A collection of routines that includes Power On Self-test (POST), system configuration and a software layer between the OS and hardware.

**BSP**
Bootstrap Processor. The processor selected to be primarily responsible for system initialization.

**CMC**
Corrected Machine Check from processor corrected errors.

**CMCI**
Corrected Machine Check Interrupt. An interrupt generated by processor hardware following a processor corrected or PAL-corrected error.

**CPE**
Corrected Platform Errors originate from platform-detected errors.

**CPEI**
Corrected Platform Error Interrupt. An interrupt which may be generated by the platform following a platform-corrected error.

**CPEV**
Corrected Platform Error interrupt vector.

**Data Poisoning**
Refers to deferring handling of non-correctable errors to consumption of the error by retaining multi-bit errors in cache or memory.

**Domain**
Dividing the system resources into separate OS partitions with processor nodes, I/O nodes and memory. A domain is isolated from the hardware errors occurring on other domains, except for errors on common system hardware such as power supplies, interconnects, etc.

**ECC**
Error Correcting Code. This term refers to a code that is used to detect and correct data corruption for the storage media or data transmission on a bus.
**Introduction**

**Error Categories:**

**Corrected Error**
These errors may have been corrected by the processor (corrected machine check), platform hardware (corrected platform error), firmware, or the OS.

**Recoverable Error**
An error in which hardware was unable to correct but was able to contain the error and reesteer execution to a machine check abort flow for firmware and OS handling. This type of error requires firmware/OS analysis and corrective action to either correct the error or terminate the affected applications to maintain system availability.

**Fatal Error**
Fatal errors occur when a non-correctable error occurs and the processor was unable to precisely capture the interrupted context to allow resumption of the interrupted context or system reset was needed to contain the error.

**ERROR[0]_N**
On QuickPath Interconnect-based Itanium processors, ERROR[0]_N is typically programmed to signal corrected processor errors or loss of lockstep.

**ERROR[1]_N**
A QuickPath Interconnect signal that can be asserted to maintain error containment by signaling a global fatal error. This error clears processor state and results in the platform bringing all processors to MCA. ERROR[1]_N is analogous to BINIT# on bus-based systems.

**GUID**
A Globally Unique Identifier is a number of sufficient length to guarantee uniqueness when assigned to a member of a group. GUIDs on Itanium architecture are used to identify structures, procedures, and data in firmware.

**Hard Fail Bus Response**
A processor system bus response used to indicate a transaction failure to the requesting agent of the transaction.

**IA-32 Architecture**
The 32-bit Intel architecture as described in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual*.

**Intel® 64 Architecture**
The 64-bit Intel architecture as described in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual*.

**Itanium® Architecture**
Intel’s explicitly parallel 64-bit architecture with architected firmware and support for IA-32 applications as described in the *Intel® Itanium® Architecture Software Developer’s Manual*.

**Itanium® -based Operating System**
An operating system for Itanium-based platforms that can run Itanium-based applications and may support IA-32 applications.

**IPI**
Inter-processor interrupt signaling using the local SAPIC within the processor.
MC Rendezvous Interrupt
An interrupt used to signal the OS to enter a rendezvous spin-loop in firmware to quiesce the system during an MCA.

MCA
Machine Check Abort. MCAs are classified into two categories: Local and Global.

Local MCA
A local MCA is limited to the processor, core, or logical processor that encountered the error and does not involve other processors in the system. At any time, multiple processors, cores, or logical processors in the system may experience local MCAs, which would be handled independently. Local MCAs may result due to uncorrectable processor errors, data poisoning error consumption, or Hard Fail bus responses.

Global MCA
During a global MCA, all the processors in the same system domain will be notified of an MCA event. On bus-based Itanium-based systems, global MCAs are signaled via the BERR# and BINIT# bus signals. On Intel QuickPath Interconnect-based platforms, global MCAs are signaled via ERROR[1]_N or viral signaling.

Min-State Save Area
Area registered by SAL with PAL for saving minimal processor state during machine check and INIT processing. See the Intel® Itanium® Architecture Software Developer’s Manual for details.

Monarch Processor
The processor elected by some implementations of SAL or the OS to coordinate the platform error handling when multiple simultaneous machine check events occur in a multiprocessor system.

MP
Multiprocessor.

Node
A large cache-coherent Non Uniform Memory Access system may consist of multiple nodes - groups of tightly-interconnected processors, memory and in some cases I/O devices. Latencies within a node are lower than inter-node latencies.

NVM
Non-volatile memory.

OS
Operating System.

PAL
The Processor Abstraction Layer of firmware, which provides runtime services and abstracts implementation-dependent processor features.

PMI
Platform Management Interrupts (PMI) provide an operating system-independent interrupt mechanism to support OEM and vendor-specific hardware events. See the Intel® Itanium® Architecture Software Developer’s Manual for details.
Introduction

**SAL**
The System Abstraction Layer of firmware, which provides runtime services and abstracts implementation-dependent system features.

**SAPIC**
Streamlined Advanced Programmable Interrupt Controller is the high performance interrupt mechanism used on Itanium® architecture-based systems. The **Local SAPIC** resides within the processor and accepts interrupts sent on the system bus. The **IOxAPIC** resides in the I/O subsystem and provides the mechanism by which I/O devices inject interrupts into the system.

**TLB**
Translation Lookaside Buffer.

**UEFI**
The Unified Extensible Firmware Interface is the interface between the OS and platform firmware. The interface consists of platform-related information, plus boot and runtime service calls that are available to the OS and its loader. This provides a standard environment for booting an OS and running pre-boot applications.

**Wakeup Interrupt**
Interrupt sent by the OS to wake up the APs from the SAL_MC_RENDEZVOUS spin loop. This interrupt vector is registered by the OS with SAL.
2 Machine Check Architecture

2.1 Overview

The Itanium Advanced Machine Check Architecture provides error handling features for high reliability, availability, and serviceability. Error containment is the highest priority, followed by error correction without program interruption, and error logging.

While error detection, correction, and containment are mostly accomplished through hardware, software plays a role in correction, recovery, and error logging. These capabilities require coordination and cooperation between the processors, platform, firmware and operating system.

This chapter provides an overview of Itanium® architecture firmware and describes the machine check error handling model. A discussion on the scope and classification of errors will be provided. Finally, topics that deviate from the normal error handling model are covered.

2.2 Itanium® Processor Family Firmware Model

The Itanium architecture defines three firmware layers: the Processor Abstraction Layer (PAL), the System Abstraction Layer (SAL), and the Unified Extensible Firmware Interface (UEFI) layer.

- PAL encapsulates processor functions that are likely to change between processor implementations so that SAL firmware and operating system software can maintain a consistent interface to the processor. These include non-performance critical functions such as processor initialization, configuration, and error handling.

- SAL is the platform-specific firmware component provided by OEMs and firmware vendors. SAL provides runtime services to the OS and provides a consistent implementation-independent interface to the operating system.

- UEFI is the firmware layer that provides a legacy-free API interface to the OS loader for boot and runtime services. SAL and UEFI isolate the OS and other higher level software from implementation differences in the platform.

PAL, SAL, and the OS work together to handle machine check aborts, processor-corrected errors, and platform-corrected errors. Figure 2-1 provides an overview of how the firmware and OS interact for machine check handling.
2.3 Machine Check Error Handling Model

The Itanium® architecture error handling model consists of multiple software components, which cooperate with hardware to handle different error conditions. PAL, SAL, and the OS have error handling components, which are integrated through well-defined interfaces.

System errors may be handled by any of the following components:

1. Processor Hardware
2. Platform Hardware
3. PAL
4. SAL
5. Operating System

*Hardware Error Handling:* When the processor or platform hardware corrects an error, a notification of the corrected event may be signaled through a CMCI for processor-corrected errors or a CPEI for platform-corrected errors. The OS has the choice to disable this automatic interrupt notification and periodically poll the firmware to collect corrected error information. See the [DIG64 Corrected Platform Error Polling Interface Specification](#). The OS can also choose to mask interrupts using external interrupt vector...
Machine Check Architecture

numbers for Corrected Machine Check events (for more information about masking interrupts using CMCV.m, refer to the Intel® Itanium® Architecture Software Developer’s Manual, Volume 2).

**Firmware Error Handling:** When the processor or platform hardware detects an error that is not hardware correctable or requires firmware assistance to cleanse, a machine check abort (MCA) is triggered. The MCA event will pass control to the firmware. The PAL and SAL firmware will correct any errors that they are capable of handling. Errors that are corrected by firmware are logged, and the control is returned back to the interrupted context. Firmware-corrected errors require no OS intervention for error handling. If an error is not correctable by firmware, control is then passed to the OS.

**Operating System Error Handling:** When an error is not corrected by the hardware or firmware layers, the control is transferred to the OS. If the OS can correct the error, it can return to the interrupted context. If the OS cannot correct the error, it can terminate the interrupted context and switch to a new context or return to SAL with a request to reset the system.

**Figure 2-2** illustrates the high level view of the machine check error handling flow. For more details, refer to the Itanium® Processor Family System Abstraction Layer (SAL) Specification.
2.4 MCA Scope

MCAs may be classified as local or global, but this classification is not explicitly reported in either the PAL or SAL architectural state hand-offs or error logs. Although error scope is not explicitly provided to software, it is described here to illustrate error handling scenarios in an MP system.

**Local MCA:** The scope of a local MCA is limited to the processor, core, or logical processor that encountered the error. A local MCA will affect other processors in the system. At any time, more than one processor in the system may experience a local MCA and handle it without notifying the other processors in the system. In certain cases, the firmware may rendezvous other processors in the system for coordinating the error handling. Local MCAs may be raised due to internal processor errors, multi-bit errors, or Hard-Fail bus responses.
MCA with Rendezvous: PAL may determine that an MCA requires coordination between all processors in the system and request a rendezvous. In addition to PAL-requested rendezvous, SAL may choose to rendezvous processors for error handling or the OS may request rendezvous for all processors by setting the `rz_always` flag when invoking SAL_MC_SET_PARAMS.

In rendezvous, the monarch sends a interrupt to all processors not in MCA to bring them into spin-loop. The monarch returns control to PAL. After PAL rendezvous error handling has completed, PAL hands back to SALE_ENTRY a second time for error handling.

See the *Intel Itanium Processor Family System Abstraction Layer Specification* for additional information about rendezvous.

Global MCA: A global MCA occurs due to system-wide broadcast of an error condition, such as BINIT#, ERROR[1]_N, or viral signaling. In a global MCA, all the processors in the domain will enter MCA and start processing the error event. The system firmware and OS layers will each coordinate the handling of the error in the MP environment.

Note that on bus-based platforms, BERR# may be routed either globally or to a subset of the OS domain. Processor-asserted BERR# will always request rendezvous to bring any processors not wired to BERR# into MCA. A platform may also assert BERR# to bring processors into platform-initiated MCA.

2.5 Error Severity

Errors are classified into five different categories based on the severity and the scope of errors as shown in Figure 2-3.

**Figure 2-3. Error Types and Severity**

<table>
<thead>
<tr>
<th>Error Handling</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Reset</td>
<td>Non-Recoverable / “Fatal”</td>
</tr>
<tr>
<td>Multi-bit Error in Kernel</td>
<td></td>
</tr>
<tr>
<td>OS Recoverable: System Available</td>
<td>Recoverable</td>
</tr>
<tr>
<td>Multi-bit Error in Application</td>
<td></td>
</tr>
<tr>
<td>OS Corrected: Execution Continues</td>
<td>Corrected</td>
</tr>
<tr>
<td>Translation Register Error</td>
<td></td>
</tr>
<tr>
<td>Firmware Corrected: Execution Continues</td>
<td></td>
</tr>
<tr>
<td>1-bit Error in Write-Through Cache</td>
<td></td>
</tr>
<tr>
<td>Hardware Corrected: Execution Continues</td>
<td></td>
</tr>
<tr>
<td>Most 1-bit Errors</td>
<td></td>
</tr>
</tbody>
</table>

2.5.1 Hardware-Corrected Errors

Errors of this type are either corrected by the processor or platform hardware and have no impact on the currently executing process. The OS may be notified of this event (CMCI/CPEI) to facilitate error logging. An OS can configure the system to disable the notification of the corrected error events, in which case it shall poll for these events...
through the SAL_GET_STATE_INFO procedure. There may be some loss of corrected error information due to overflow if the OS polling frequency of SAL_GET_STATE_INFO is not adequate. Refer to the DIG64 Corrected Platform Error Polling Interface Specification for more information.

Figure 2-4 represents a machine check error flow in which the errors were corrected without involving the OS.

### 2.5.2 Firmware-Corrected Errors

This type of error is not corrected by the processor or platform hardware and is corrected by firmware. On detecting such a non-hardware correctable error, the processor signals a local MCA, forcing control transfer to the firmware. Processor-detected errors of this type are corrected by PAL, whereas platform-detected errors of this type are corrected by SAL. The firmware handlers correct the error and resume the execution of the interrupted context. When the error is corrected by the firmware layers, the corrected event may be signaled to the OS as a CMC or CPE interrupt, otherwise the OS needs to poll for this information.

An example of this type of error is an error that occurs in a write-through cache data structure containing unmodified data. The firmware invalidates the affected lines in the structure and returns execution to the interrupted process. Intel Cache Safe Technology is another example of a firmware-corrected error.

### 2.5.3 OS Recoverable Errors

OS recoverable errors can be due to a local MCA or global MCA as described below.

OS recoverable errors of the local MCA type cannot be corrected by either the hardware or firmware and requires OS handling. Successful recover depends upon the state of the system and OS and the OS capabilities. When an error is not recoverable, the system should be rebooted to return the system to an available state.

OS recoverable errors have the following characteristics:

1. The error is contained.
2. Critical system state is intact.
3. The process or the system is continuable.

An example of an OS recoverable error with a local MCA is an application register parity error. If the OS can identify the offending process from the error information logged, it can recover by terminating the processes that needed to consume this data and allow the system to continue executing.

### 2.5.4 Fatal Errors

This type of error cannot be corrected by the processor, platform, firmware, or the OS, and the system must be rebooted. Fatal MCAs may be either local or global.

Fatal local MCAs result in a processor handing off to SAL with a fatal condition. SAL logs the fatal error information to NVRAM. If SAL determines that the system has sufficient functionality to allow the OS to report the error, SAL may hand off to OS_MCA. If the system does not have sufficient functionality to support OS_MCA execution, SAL resets the system and the fatal error is reported on reboot from NVRAM.
Fatal global MCAs are delivered using BINIT#, ERROR[1]_N, or viral signaling. Please refer to Section 4.4, “Platform MCA Signaling.” On detecting a global error event condition, all the processors enter their MCA handlers.

On bus-based systems, BINIT# (Bus Initialization signal) invalidates the state of outstanding memory and bus transactions. After a BINIT# signal, the first bus transaction must be the fetch to the MCA handler. The BINIT# error condition requires all the processors to handle the BINIT# reset. The BINIT# signal is not maskable by the processor’s psr.mc bit.

On Intel QuickPath Interconnect-based platforms, ERROR[1]_N may be asserted for hardware fatal conditions that prevent branching directly into an MCA flow. On ERROR[1]_N assertion, the platform responds by asserting an error reset, which results in processor warm-logic reset and brings the processors into MCA. At this point, SAL can log error information and should initiate a cold reboot.

On Intel QuickPath Interconnect-based platforms, viral signaling may be used for hardware fatal conditions. Each Intel QPI packet contains a viral bit. When a protocol agent detects a fatal error or receives a viral packet, the agent turns viral and sets the viral bit on all outbound packets until the agent is reset or viral status has been cleared. The caching agent will enter MCA to perform error logging and reset the system.

2.6 Software Handling

The Itanium® architecture requires PAL, SAL, and the OS to share the responsibility for machine check handling. The responsibilities of each of these components are described below.

2.6.1 PAL Responsibilities

Machine check abort events initially go to PAL for handling. PAL has the following responsibilities when receiving a machine check:

- Save the processor state in min-state save area of memory registered by SAL.
- If necessary, contain the error by requesting a rendezvous for all processors in the domain. Refer to section 13.3.1.1 in Intel® Itanium® Architecture Software Developer’s Manual.
- Attempt to correct the error.
- Hand off control to SAL for further handling and logging.
- Return the processor error information when the SAL requests it using the PAL_MC_ERROR_INFO procedure.
- Return to the interrupted context by restoring the state of the processor when PAL_MC_RESUME is called or when PAL corrects the error.

2.6.2 SAL Responsibilities

The responsibilities of the SAL may be categorized into initialization and runtime responsibilities.

2.6.2.1 SAL Responsibilities During System Initialization

The SAL has the following responsibilities during system initialization to set up the processor and the platform for appropriate MCA signalling:
Optionally invoke the PAL_PROC_SET_FEATURES procedure to specify CMC, MCA, and BERR# promotion. Refer to Section 3.3.2, “Error Severity Escalation” for more details. CMC promotion forces hardware corrected CMCs through the PAL and SAL layers.

Program the platform to enable ECC generation on the data delivered by the platform to the processor and to enable error correction and detection logic on the data received by the platform.

Request virtual address registration for the platform hardware that the SAL needs to access during machine check processing, (e.g. chipset registers, NVM, etc.).

The SAL may log MCA events that occur during the boot prior to the registration of the OS_MCA layer. However, it must preserve any unconsumed MCA records from the previous boot session as these may not have been recorded by the OS in persistent storage.

In general, the SAL should take maximum advantage of the error correction and detection capabilities provided by the platform hardware.

### 2.6.2.2 SAL Responsibilities During Machine Check Aborts

The SAL has the following responsibilities during machine checks:

- Attempt to rendezvous other processors in the system if requested to by PAL.
- Process MCA handling after hand-off from PAL.
- Initiate processor rendezvous, if the error situation warrants one.
- Elect a monarch processor if multiple processors enter the SAL MCA handler at the same time to coordinate error handling.
- For corrected, recoverable, and fatal errors, retrieve the processor error record information from PAL_MC_ERROR_INFO for logging.
- For corrected, recoverable, and fatal errors, issue a PAL_MC_CLEAR_LOG request to clear the log and to enable further error logging.
- Retrieve platform state for the MCA.
- Attempt to correct platform errors. If the error is not corrected and the OS has specified the “always rendezvous” flag through SAL_MC_SET_PARAMS, SAL will rendezvous the processors.
- Log the uncorrected error information to NVM and hand off control to the OS_MCA handler if the platform has sufficient functionality to support OS_MCA error reporting. If the OS_MCA handler is absent or corrupted, then the SAL will either reset the system or take OEM-specific actions.
- On return from OS_MCA, if the error was corrected, return to the interrupted context through the PAL_MC_RESUME procedure. If the error was not corrected, the SAL may reset the system.

### 2.6.3 Operating System Responsibilities

The OS depends on SAL to interact with PAL to get information about machine check errors for further handling. The responsibilities of OS machine check handler may be categorized into initialization and runtime responsibilities.
2.6.3.1 Operating System Responsibilities During OS Initialization

To minimize the boot time, the following steps are required to be handled by the Bootstrap processor:

- Register spin loop/Rendezvous and Wakeup Request Interrupt Vectors.
- Specify the options during MCA processing (rendezvous always for MP systems, MCA to BINIT# escalation, etc.) by invoking the SAL_MC_SET_PARAMS procedure.
- Register an OS_MCA handler entry point by invoking the SAL_SET_VECTORS procedure.
- Initialize the CMCV register to enable CMC interrupt on the processor and install a handler for the interrupt. This is not required if the OS chooses to poll for corrected processor errors.
- Initialize the Corrected Platform Error Interrupt vectors (CPEI) in the I/O SAPIC. The details of the interrupt line on which CPEI is signaled is provided by the SAL in the Platform Interrupt Source Structure within the ACPI tables. The OS must also register the CPEV with the SAL using the SAL_MC_SET_PARAMS procedure. These steps are not required if the OS chooses to poll for corrected platform errors.
- Invoke the SAL_GET_STATE_INFO procedure on the bootstrap processor with argument type MCA (and optionally CMC and CPE) to retrieve any unconsumed error records from the previous boot. Then, invoke the SAL_CLEAR_STATE_INFO procedure to mark such records as consumed, thereby freeing up the NVM storage for future logging of uncorrected errors.
- If an OS supports a mechanism to have an OEM call-back function for MCAs, it should set up this call-back function if required by the platform. The interface for such a function is OS specific.

The OS should then enable maskable interrupts on all the processors in the system. Machine checks would already have been unmasked when the OS gains control from EFI.

2.6.3.2 Operating System Responsibilities During a CMC at Runtime

On receipt of the CMCI, the OS should invoke the SAL_GET_STATE_INFO and the SAL_CLEAR_STATE_INFO procedures with the CMC argument type on the processor on which the CMCI was signaled. This permits the SAL to mark the error record as consumed and free up memory resources for future SAL use. The SAL_GET_STATE_INFO call must be invoked repeatedly until the SAL returns a "No Information Available" status to the OS. If polling is employed, these procedures should be called periodically.

2.6.3.3 Operating System Responsibilities During a CPE at Runtime

On receipt of the CPEI interrupt, invoke the SAL_GET_STATE_INFO and the SAL_CLEAR_STATE_INFO procedures with the CPE argument type on the processor on which the CPEI was signaled. This permits SAL to mark the error record as consumed and free up memory resources for future SAL use. The SAL_GET_STATE_INFO call must be invoked repeatedly until the SAL returns a "No Information Available" status to the OS. If polling is employed, these procedures should be called periodically. Refer to Section 4.3.1, “Scope of Platform Errors” for additional information.

2.6.3.4 Operating System Responsibilities During an MCA at Runtime

The OS has the following responsibilities during machine checks:
• Elect a monarch processor, if multiple processors enter OS_MCA at the same time to coordinate error handling.

• Retrieve error records from SAL by making repeated calls to SAL_GET_STATE_INFO.

• Recover from the error if possible. If the OS cannot continue, it will return to the SAL and request a reboot.

• Call back the registered OEM machine check handler, if any.

• If the OS cannot continue further, return to the SAL, requesting a reboot. The SAL implementation will log the current MCA event to NVM and provide the MCA event information during the next reboot so that the OS may record it subsequently in persistent storage.

• If the error has already been corrected (as indicated by the ERR_SEVERITY field in the Record Header), or if the OS can continue, it should mark the MCA error record as consumed by invoking the SAL_CLEAR_STATE_INFO procedure.

• At the end of machine check processing, wake up all slave processors from the SAL rendezvous by sending the wake-up interrupt. If any processors go through INIT during the rendezvous, set up further processing steps as part of the OS_INIT procedure.

• On all the processors that entered the OS_MCA handler, the OS_MCA handler can specify a return to the interrupted context or branch to a new context by modifying the processor state that the OS provides to the SAL MCA handler.

2.6.3.5 Operating System Responsibilities During a Rendezvous Interrupt at Runtime

The OS has the following responsibilities during rendezvous interrupts:

• Identify the processors to be awakened at the end of OS_MCA processing, using an implementation-specific structure. This variable may be used by the OS to identify the processors that need to be woken up at the end of OS_MCA processing. The OS implementations may also use this variable to distinguish between a rendezvous interrupt and an INIT.

• Invoke the SAL_MC_RENDEZ procedure.

• On return from SAL_MC_RENDEZ at the end of MCA event processing, clear the interrupt pending bits within the processor for the rendezvous and wake-up interrupt vectors.

• Resume the interrupted context.

2.7 Multiple Errors

All machine check errors detected within a window before the processor masks machine check detection hardware (PSR.mc) are lumped together as a single MCA condition, locally visible on that processor only. This is shown in Figure 2-4.

Multiple MCA events within a detection window on a particular processor may be reported as:

1. A single error, if the same error is detected multiple times in the same structure (cache, TLB or system bus structural units).

2. A single error with an overflow indicator, if multiple unique errors are detected in the same structure it may be the result of shared logging and may be aggregated in a single overflow message.
3. Multiple unique errors in different structures.

**Figure 2-4. Multiple MCA Events**

![Multiple MCA Events](image)

*Nested MCA*: A nested MCA is an MCA that occurs after the MCA detection window is closed. All further MCAs occurring after the detection window are held pending and may be unrecoverable. The Machine Check Architecture allows for multiple nested MCAs to occur on each processor, but only one MCA may be handled at a time. Note that errors detected and corrected by hardware trigger the optional CMCI or CPEI event and are not considered to be MCAs or nested MCAs.

*Multiprocessor System*: Error handling may depend on the number of processors in a system. In an MP environment, because of the possibility of a global MCA error or simultaneous local MCAs on multiple processors, firmware and OS_MCA handlers must perform synchronization during error handling. The SAL firmware may perform a rendezvous of the processors based on the error encountered or may be configured by the OS to always rendezvous using the SAL_MC_SET_PARAMS procedure. Likewise, the OS may perform its own rendezvous of the processors based on the error encountered if not already done by the firmware.

### 2.7.1 SAL Issues Related to Nested Errors

*Reentrancy*: The *Itanium® Processor Family System Abstraction Layer Specification* requires that the SAL procedures invoked during an MCA be made reentrant. This permits a SAL invocation for CMC or CPE to be interrupted by SAL procedure calls to retrieve MCA information. Reentrancy becomes important as operating systems develop strategies to service simultaneous MCAs in the error handling flow to allow building and retrieval of error records on the same processor. For the environments where the OS does not attempt recovery from MCA, the minimum SAL implementation requirements are that:

- The MCA error information is provided to the OS_MCA layer.
- The MCA error record is logged to the NVM.

To simplify SAL implementation, it is strongly recommended that SAL process all MCAs by handing off to the OS as soon as possible to prevent timeouts from occurring. The SAL may maintain a variable in the SAL data area that indicates whether SAL, on one of the processors, is already handling an MCA. If so, MCA processing on other processors will wait within the SAL MCA handler until the current MCA is processed. This situation may arise when local MCAs occur on multiple processors. The SAL code must also implement methods of detecting which processors have already reached the SAL MCA handler and avoid steps to rendezvous such processors (using MC_rendezvous interrupt or INIT).
While SAL firmware is handling the MCA event, further machine checks are masked by the PSR.mc bit in the processor status register (PSR). This does not, however, mask a subsequent BINIT# or ERROR[1]_N event. SAL firmware must be designed to allow for a BINIT# or ERROR[1]_N occurring in the middle of a machine check event processing and must not wait on any semaphores that may have been held by the earlier MCA event. At a minimum, the SAL must log the BINIT# error to NVM and provide the error information pertaining to the BINIT# to the OS_MCA layer.

Indexed I/O Accesses: Even if the re-entrancy issue is solved by careful SAL implementation, there are atomicity issues due to legacy hardware that requires multiple accesses. A number of I/O devices are accessed using indexed accesses (e.g. PCI configuration space). To access a register within the PCI configuration space, an index needs to be written followed by a read or write to the data port (e.g. I/O ports CF8, CFC). It is possible for an MCA to occur while the SAL is in the middle of the SAL_PCI_CONFIG_READ/WRITE procedure. The index may have been written but the data may not have been read/written. If the SAL MCA handler needs to access the PCI configuration space, it must first save the index, access the areas of interest and then restore the index.

The combination of restricting processing to one MCA at a time, rendezvousing all the processors on an MCA, and the save/restore of the index will permit the SAL to work around indexed I/O issues. The SAL must also ensure that other layers within SAL, such as SAL_PMI, do not access the PCI configuration space. This is solvable by maintaining a SAL data variable reflecting current usage (0=Unused; 1=OS; 2=PMI; 3=MCA) and establishing rules for pre-emption. The CMPXCHG instruction may be used for atomic updates to the variable.

The PCI configuration space is a resource abstracted by SAL and it is possible to solve the indexed I/O problem for this resource. There are no solutions if multiple software layers access indexed I/O resources without any coordination. Hence, the SAL code handling the MCA event shall not use other indexed I/O resources such as the Real-Time Clock NVM if this resource is also used by the firmware during runtime calls by the OS.

Memory Attribute Aliasing: The SAL code and data areas will be accessed both during machine check processing and during normal OS execution. The normal execution mode during the MCA path is uncacheable (UC) using the firmware image (around 4GB), with address translation disabled. The normal execution mode while the OS retrieves the CMC and CPE error records is cacheable and writeback (WB) using the copy of SAL in memory, with address translation enabled. Both these contexts would need to access common data variables. The SAL implementation must be careful to avoid accessing the same location with differing memory attributes (UC vs. WB), otherwise additional MCAs may surface due to memory attribute aliasing.

2.8 Expected MCA Usage Model

In addition to the error handling model described in Section 2.3, "Machine Check Error Handling Model", the Machine Check Architecture provides a Machine Check Expected configuration option for platform/software testing purpose. This machine check expected option is enabled or disabled through the PAL_MC_EXPECTED procedure. When this option is set, the PAL machine check handler will deviate from its normal handling and will not attempt to perform error handling other than log generation (hardware error recovery action is not affected), but hands off control to SAL directly. The Machine Check Architecture does not restrict the usage of the Machine Check Expected option, but it is intended to be used for software diagnostics only.
3 Processor Error Handling

On detecting an internal error, processor hardware may correct the error or steer execution to a machine check abort. Processor-corrected errors are signaled directly to the OS using CMCI by default. If the error is not hardware-correctable but is containable and software intervention is required, the processor branches to PALE_CHECK to begin the error handling flow. If required for containment, BINIT# or ERROR[1]_N is asserted to contain the error and reset the platform.

3.1 Processor Errors


Processor Machine Check Errors and Deconfigured Processor Machine Check Errors are reported using five different structures. At any point in time, a processor may encounter an MCA or CMC event due to errors reported in one or more of the following structures:

1. Processor Cache
2. Processor TLB
3. System Bus
4. Processor Register File
5. Processor Microarchitectural

Refer to the *Intel® Itanium® Architecture Software Developer’s Manual* for detailed processor error reporting information regarding PAL_MC_ERROR_INFO. An overview of the types of processor machine check errors is presented in the following sections.

3.1.1 Processor Cache Check

Itanium processors may have several levels of processor cache. An implementation may organize a level of cache as separate instruction and data caches or as a unified cache. To make the error information independent of the processor’s cache implementation, a processor will report error information in an architecturally-defined manner for recovery and logging.

PAL_MC_ERROR_INFO may return the following information when a cache error occurs:

1. Instruction or data/unified cache failure identification.
2. Data or tag failure identification.
3. Type of operation that caused the failure.
4. The cache way and level of failed location.
5. Index of the failed cache line.
6. Physical address that generated the machine check.
3.1.2 Processor TLB Check

Itanium architecture implementations may have several levels of processor cache translation look-aside buffers (TLBs). An implementation may choose to have separate instruction and data TLBs or a unified TLB.

PAL_MC_ERROR_INFO may return the following information when a TLB error occurs:

1. Translation register (TR) or the translation cache (TC) error identification.
2. Indication of whether the error occurred in an instruction or data/unified TLB structure.
3. Type of operation that caused the TLB MCA.
4. Level of the TLB where the error was encountered.
5. Slot number of the TR that experienced the MCA.
6. Physical address that generated the machine check.

3.1.3 System Bus Check

Itanium® architecture implementations will report a bus machine check for system bus transaction errors or Intel QuickPath Interconnect errors.

PAL_MC_ERROR_INFO may return the following information when a bus error occurs:

1. Size of the transaction that caused the machine check.
2. Indication of whether this machine check was due to an internal processor error or due to an external bus notification.
3. The type of bus transaction that generated the machine check.
4. Identification of the requester and responder of the bus transaction that generated the machine check.

3.1.4 Processor Register File Check

Itanium processor implementations have large register files, which may be protected to detect errors. Errors encountered on protected register files will be returned in the register file check.

PAL_MC_ERROR_INFO may return the following information when a register error occurs:

1. Register File ID and register number for the failure.
2. Operation that generated the register file error.

1. For further details on the TLB error info, please refer to Intel® Itanium® Architecture Software Developer’s Manual.
2. For further details on the bus error info, please refer to Intel® Itanium® Architecture Software Developer’s Manual.
3. For further details on the register file error info, please refer to Intel® Itanium® Architecture Software Developer’s Manual.
3.1.5 Processor Microarchitecture Check

Itanium processor implementations have many internal arrays and structures that may not be architecturally defined yet may still be designed to detect errors. Any errors detected in architecturally undefined structures are reported using the microarchitecture check. These error conditions may not be recoverable by OS software but may be logged for serviceability.

PAL_MC_ERROR_INFO may return the following information when a microarchitecture error occurs:

1. Structure ID, array ID, way and level where the error occurred.
2. Operation that triggered the error.

3.2 Processor Error Correlation

SAL must iteratively call PAL_MC_ERROR_INFO to retrieve all of the information associated with a machine check event. SAL calls PAL_MC_ERROR_INFO to get the error severity within the Processor State Parameter (PSP) and error map information through Processor Error Map. Subsequent calls are made to obtain detailed error information. The PSP and the processor error map values returned by the PAL have a global summary of the error, which enable SAL to identify and make subsequent PAL calls to get detailed error information for each structure.

Note: As defined in the SAL specification, SAL returns processor error information for the processor on which the SAL_GET_STATE_INFO call is made. To get the error information for all processors, multiple SAL calls must be made on each processor.

3.3 Processor CMC Signaling

Corrected machine check signaling may be due to either hardware-corrected errors or PAL-corrected errors.

A machine check error corrected either by processor hardware or PAL translates into a CMC condition with eventual notification to the OS. The notification of the CMC condition to the OS is only necessary for recording the error information. The OS can use this information for system management. For the processor hardware or firmware to deliver the CMC interrupt to the OS, the CMC interrupt must be enabled on each of the processors with CMC vector initialized in the processor’s CMCV register.

On a processor-corrected error, a CMC event can be transferred to the OS by two different methods. The OS may either initialize the processor to generate an interrupt (CMCI) for automatic signaling of a CMC, or the OS can periodically poll the CMC condition through SAL_GET_STATE_INFO. The OS can choose any low priority interrupt vector for this purpose by programming the processor’s CMCV register.

If the OS chooses to use polling for corrected processor error events, it must periodically call SAL_GET_STATE_INFO with argument type of CMC on each of the processors in the system to check for the validity of CMC events and error records. The polling frequency is implementation-specific, and should be frequent enough to prevent loss of current data while providing space for additional logging.

1. For further details on the Microarchitectural check info., please refer to Intel® Itanium® Architecture Software Developer’s Manual.
3.3.1 CMC Masking

System software may disable corrected machine check interrupts by modifying the CMCV register. These capabilities are highlighted in Table 3-1.

Table 3-1. Processor Machine Check Event Masking

<table>
<thead>
<tr>
<th>Processor Register</th>
<th>Field</th>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMC Interrupt Vector Register</td>
<td>CMCV.m</td>
<td>CMCI</td>
<td>Mask or deliver the CMC interrupt.</td>
</tr>
</tbody>
</table>

3.3.2 Error Severity Escalation

To simplify error handling or to give certain errors a different priority level, system software may escalate errors to a higher severity level. PAL firmware may permit SAL or the OS to escalate errors.

Although these features are architecturally defined, a particular Itanium® architecture implementation may not support all of these capabilities. The PAL_BUS_GET_FEATURES and PAL_PROC_GET_FEATURES procedures can determine the processor capabilities.

Table 3-2 shows the different events that can be elevated in severity.

Table 3-2. Machine Check Event Escalation

<table>
<thead>
<tr>
<th>Processor Detected Machine Check Event</th>
<th>Available Escalation Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Corrected Machine Check Events</td>
<td>May be promoted to an MCA condition. When this option is chosen, the processor signals a MCA on all CMC conditions. A promoted MCA may be further promoted to BINIT# or ERROR[1]_N. On bus-based systems, a promoted MCA may be further promoted to BERR#.</td>
</tr>
<tr>
<td>All MCA Events</td>
<td>May be promoted to BINIT# or ERROR[1]_N. On bus-based systems, MCA may also be promoted to BERR#.</td>
</tr>
<tr>
<td>Only BERR# Event</td>
<td>On bus-based systems, BERR# may be promoted to BINIT# error condition. When this option is chosen for the detecting processor, the processor treats all BERR# errors as BINIT# error condition.</td>
</tr>
</tbody>
</table>
4 Platform Error Handling

Detecting and reporting platform errors are platform specific. Platform hardware may record error information and return it to the firmware and OS layers, which may have platform-specific error handling capabilities.

4.1 Platform Errors

MCA-enabled platforms may report several different error types, depending upon the platform design. The platform errors can be classified into three categories:

1. Memory errors
2. I/O errors
3. OEM-specific errors

Each of the error types will be associated with a unique GUID for identification. OEMs can define their own error type and associated GUID. When these platform errors, the processors will indicate external bus errors in their processor error records to the system software (see Section 3.1.3, "System Bus Check" on page 20). The platform firmware is responsible for further queries to the platform hardware to identify the source of the error and build an appropriate platform error record.

Since SAL is the platform-specific component of the firmware, it should be able to retrieve any error information from the chipset and possibly correct some errors with a hand-off to the OS. The OS MCA handler in conjunction with the SAL can effectively handle platform-detected errors.

4.1.1 Memory Errors

Errors detected on the external memory subsystem, such as local main memory single-bit or multi-bit errors, will be reported using the Platform Memory Device Error Section. The errors detected on any platform-level cache will also fall into this category.

4.1.2 I/O Errors

*Intel Itanium Processor Family System Abstraction Layer Specification* defines the following I/O error sections:

- Platform PCI Bus Errors
- Platform PCI Component Errors
- Platform System Event Log Device Errors
- Platform SMBIOS Device Errors
- PCI-Express* 1.1 Errors
- PCIe-Express* 1.1/2.0 Errors
4.1.3 OEM-Specific Errors

The SAL Specification defines a Platform-Specific Error Section to allow platforms to report OEM hardware errors not captured by the architected error sections.

4.2 Platform Error Correlation

The OS must iteratively call SAL_GET_STATE_INFO to retrieve all of the information associated with a machine check event. The processor error record provides the severity of error, which is coupled with the platform error sections returned by SAL. The processor-logged bus error information has an external bus error flag, which is set for errors detected and reported by the platform.

4.3 Platform-Corrected Error Signaling

Corrected platform error events are signaled (if OS polling is not used) using two different control paths:

1. Hardware-corrected platform errors
2. Firmware-corrected platform errors

For hardware-corrected platform error, the event notification is sent to the OS with the corrected platform interrupt vector.

If the SAL firmware corrects an error, a corrected platform error event will be signaled to the OS if signaling is enabled. In this case, the SAL will send an IPI to a processor with the corrected platform error vector number.

4.3.1 Scope of Platform Errors

The scope of platform errors depends upon the platform and firmware implementations. Depending upon the platform topology, a single physical platform may consist of multiple processor nodes. A processor node in this context is defined to be a section of the platform that contains a set of inter-connected processors with its own error event generation and notification ability.

When SAL_GET_STATE_INFO is called for MCA or corrected errors for the platform, SAL returns the error record for the processor node associated with the processor on which the call is made. If a SAL implementation is capable of accessing error information for the entire multinode system from one processor, it is permitted to aggregate all the platform error sections within one error record.

Returning error information on a processor node basis helps to efficiently manage platform resources for error event notification and error record building when the system has a large number of processors and platform resources.

4.3.2 Handling Corrected Platform Errors

The OS may employ either the CPEI interrupt mechanism or polling to retrieve corrected platform errors. When the OS uses the polling option for the platform corrected error event, it must call SAL_GET_STATE_INFO with argument type of CPE on each of the processor nodes to collate the error information for the entire platform.

Note that it is unnecessary to make this call on each processor within a node. If polling is employed, the frequency may be based on past history.
The most common source of corrected platform errors are 1-bit, soft or transient memory errors. This is usually corrected by the 1-bit error correction or double bit error detection feature of the memory controller. The platform hardware would provide the corrected data to the processor but the memory location may still contain erroneous data. Hardware scrubbing will eventually correct the error but this may not be soon enough. It is possible to experience several 1-bit errors from the same location. There may be an abundance of errors if the device driver handling the 1-bit error is located on the faulty location(s) or has the need to access the faulty location(s).

The OS can maintain statistics on the corrected platform errors corrective action since the CPEI interrupt is signaled to the OS first. The only method by which the SAL can perform any pre-processing and thresholding is to define the interrupt signal line for the CPEI to be a PMI interrupt type in the ACPI tables. This is not a desirable option from a performance perspective as the PMI interrupts are handled at the highest priority level. Further, the resources to maintain statistics may be limited within the SAL data areas.

The OS has several strategies for handling excessive CPEI interrupts:

- Turn off the CPEI interrupt by turning on the “mask” bit within the redirection table entry (RTE) of the I/O SAPIC. This will have the effect of turning off interrupts from all the platform sources that are connected to the RTE. Subsequent to the masking, the OS can use the polling mechanism by invoking the SAL_GET_STATE_INFO procedure. This SAL procedure will clear the error registers within the chipset and enable future logging of errors. The OS can maintain statistics on the location of error, frequency, etc. and vary the polling frequency depending on past history.

- In the event of a HF condition, the code/data from the faulty page should be remapped to a different page and thereafter avoid use of the faulty page. This step is essential so that the 1-bit error does not degrade to a multi-bit error over time. This may be the only way to prevent hard failure memory errors. This solution does not apply if OS invokes the firmware in physical addressing mode and the errant location is part of the firmware image in memory.

- Correct the faulty location permanently by rewriting. The error record associated with the CPEI event would specify the physical address and granularity (address mask) of the faulty memory location(s). The OS should mask the CPEI interrupt for the 1-bit error, read the faulty memory location(s), write it back and execute the flush cache (fc) instruction to ensure that the location(s) in memory is rewritten. The OS should then invoke SAL procedures to retrieve and clear the error record to ensure that the logging registers within the chipset are freed up for future error logging. The OS may then access the location in uncacheable mode to check if the correction was successful. If errors persist, it indicates a stuck-at-fault error.

The SAL may optionally maintain statistics of CPEI interrupts when the OS invokes the SAL_GET_STATE_INFO procedure. For the corrected errors not caused by memory transactions, such as internal buses within the chipset, the SAL may program the chipset registers to avoid corrected error signalling. It may also take platform-specific actions such as system management alerts, enabling alternate paths, etc., if such actions are transparent to the OS execution.

4.4 Platform MCA Signaling

Depending on the severity of an error, a platform has the choice of signaling an error to the processor either in-band to the bus, or out of band.

Options for platform signaling of MCAs on bus-based platforms are:
• **BERR# Pin Assertion:** Since the processors do not reset their internal state, platform errors that are signaled this way may be recoverable. BERR# errors are global events when a platform design connects all the processors’ BERR# pins together in a system.

• **BINIT# Pin Assertion:** Since the processors reset part of their internal state, platform errors signaled this way are not recoverable. BINIT# signaling causes a system-wide MCA event (refer to Section 4.4.1, “Global Signal Routing” on page 26 for further details).

• **Forcing a multi-bit Data Error:** Platforms may only use this method on a transaction that requires data return to the processors. On receiving a multi-bit error indicator, for cacheable data, the processor poisons and stores the data in the internal caches. A multi-bit error response is local to the receiving processor.

• **Hard Fail Response Error:** The Hard Fail response is supported by the system bus protocol. On receiving such an error, the processor treats it in the same manner as a multi-bit data error. A Hard Fail error response is local to the receiving processor.

Options for platform signaling of MCAs on Intel QuickPath-based platforms are:

• **ERROR[1]_N Assertion:** ERROR[1]_N errors are not recoverable, but core and uncore error information may be logged.

• **Data Poisoning bit:** If the data poisoning bit is set on a received packet, a recoverable cache_check MCA will occur.

• **RspStatus = Failed:** A packet with RspStatus = Failed will be reported as a recoverable MCA with bus_check.bsi = 3.

• **Viral MCA.** Identified by checking bus_check.bsi = 1 and uncore CSR U_CSR_SESP.vs = 1.

Each Intel QPI packet contains a viral bit. When a protocol agent detects a fatal error or receives a viral packet, the agent turns viral and sets the viral bit on all outbound packets until the agent is reset or viral status has been cleared. The caching agent will enter MCA to perform error logging and reset the system.

### 4.4.1 Global Signal Routing

#### 4.4.1.1 Bus-based Platforms

At the node and system levels, error containment relies on PAL, SAL, and the OS with the use of the BERR# and BINIT# pins to achieve error containment. This section suggests a usage model of the BERR# and BINIT# pins on bus-based Itanium-based platforms.

Broadcast of BERR# across processor nodes (see Section 4.3.1, “Scope of Platform Errors” on page 24 for the definition of a processor node) is an implementation choice based on the needed platform topology and functionality. Irrespective of the nature of the signal routing, the impact of these platform signal routing choices shall be abstracted from the OS. An example of such abstraction is when a platform does not route the BERR# signal across processor nodes, SAL must perform a SAL rendezvous of the processors on the neighboring processor nodes.

**Use of BERR#:** The BERR# pin can be used on an Itanium architecture-based platforms as a way for the platform to signal a recoverable platform MCA. The SAL MCA Handler can specify the error recoverability for platform-asserted BERR# by reporting the appropriate error severity in the error record as being fatal or recoverable. If any platform components lost essential state information due to the assertion of the BERR# pin, then SAL must report the error as fatal.
Bus-based Itanium processors drive the BERR# pin for errors that are not recoverable but are contained within the processor. The processor drives this pin to notify other processors that there is an unrecoverable error and to get the other processors to stop their currently executing programs, in order to reduce the chance of one of these processors noticing the same error (this would cause the processor to assert the BINIT#). This increases the chance of allowing the firmware and OS to get a good error log stored before having to reboot the system. For very large multi-node systems, platforms may not want to tie the BERR# pins together across processor nodes, but can achieve a similar global event notification by using the rendezvous mechanism to coordinate error handling.

Use of BINIT#: The BINIT# pin is used for system level error containment. The processor only asserts this pin for fatal errors that may cause loss of error containment. These pins shall be tied together in a multi-node system. BINIT# assertion is a system-wide MCA event.

To be consistent with the Machine Check Architecture, it is recommended that the platform hardware generate a BERR#, multi-bit errors, data poisoning, or Hard Failure bus responses for recoverable errors and a BINIT# for fatal errors. Since BERR# assertion allows the MCA handlers to make forward progress, some platforms may choose to report containable fatal errors through this means rather than asserting BINIT#.

4.4.1.2 Intel QPI-based Platforms

Use of ERROR[0]_N: Processors or platform components may assert ERROR[0]_N to signal corrected errors to system management.

Use of ERROR[1]_N: ERROR[1]_N is routed system-wide to guarantee error containment. Processors and platform components assert ERROR[1]_N when an agent is unable to contain an error. Upon observing assertion of ERROR[1]_N, platforms are required to assert Reset. Upon reset, PAL hands off through SALE_ENTRY to SAL_CHECK with a fatal MCA. SAL saves core, uncore, and IOH error information in an NVRAM error record and triggers cold reset. After reboot, the OS will consume the MCA error records and may also consume any pending CMC or CPE error information.

4.4.2 Error Escalation

In addition to the processor error masking capabilities, the platform hardware may also provide an implementation-specific way of masking the platform to external interrupts for error severity escalation.

Platform errors that are visible to the processor can be escalated on a per processor basis by setting the processor configuration bits as shown in Table 3-2.
The following pseudocode is provided to show the steps an OS would take to try and recover from a processor cache error. This pseudocode is not meant to be an exhaustive definition of what an OS needs to do for all machine check handling, but rather a starting point for OS designers.

// Definitions - These are provided to attempt to make the pseudo code easier to read and are not meant to be real definitions that can be used.

/* Processor State Parameter is located in PSP=r18 at hand off from SAL to the OS_MCA handler. Define some PSP bit fields. */

#define TLB_Error ProcessorStatParameter[60]

#define Record_ID_Offset 0
#define Err_Severity_Offset 10
#define Recoverable 0
#define Fatal 1
#define Corrected 2
#define Record_Length_Offset 12
#define Record_Header_Length 40

#define GUID_Offset 0
#define Section_Length_Offset 20
#define Processor_GUID E429FAF1-3CB7-11D4-BCA70080C73C8881
#define Section_Header_Length 24
/* SAL Processor Error Record Definitions */

#define Validation_Bit_Structure
    Proc_Error_Map_Valid       = bit 0
    Cache_Check_Valid          = bits [7:4]
    TLB_Check_Valid            = bits [11:8]
    Bus_Check_Valid            = bits [15:12]
    Reg_File_Check_Valid       = bits [19:16]
    MS_Check_Valid             = bits [23:20]

#define Error_Validation_Bit_Length  = 8
#define Check_Info_Valid_Bit       = bit 0
#define Target_Address_Valid_Bit   = bit 3
#define Precise_IP_Valid_Bit       = bit 4

#define Check_Info_Offset          = 0
#define Target_Address_Offset       = 24
#define Precise_IP_Offset           = 32

/* Cache Check Info Bit definitions */

#define PrecisePrivLevel       = bits [57:56]
#define PrecisePrivLevel_Valid = bits 58

/*----------------------------------------------------------------------------*/
/* OS Machine Check Initialization */
/*----------------------------------------------------------------------------*/

OS_MCA_Initialization;
{
    /* this code is executed once by OS during boot, on the Bootstrap processor. 
    Register OS_MCA Interrupt parameters by calling SAL_MC_SET_PARAMS */

    Install OS_Rendez_Interrupt_Handler
    Install OS_Rendez_WakeUp_Interrupt_Handler /* ISR clean up wrapper */
    Register_Rendez Interrupt_Type&Vector;
Register_WakeUpInterrupt_Type&Vector;

Specify the options during MCA processing (rendezvous always, MCA to BINIT escalation);

Register_CorrectedPlatformErrorInterrupt_Vector with SAL;

Program the I/O SAPIC for CPEI interrupt;

Initialize_CMC_Vector_Masking;

/* Register OS_MCA EntryPoint parameters by calling SAL_SET_VECTORS */

Register_OS_MCA_EntryPoint;

Register_OS_INIT_EntryPoint;
}

/*===========================END======================================*/

/*=========================BEGIN======================================*/

/* OS Machine Check Rendez Interrupt Handler */
/*====================================================================*/

OS_Rendez_Interrupt_Handler()
{
    /* go to spin loop */
    Mask_All_Interrupts;
    Set_Flag to indicate Rendezvous occurrence (with value based on CPUID);
    Call SAL_MC_RENDEZ();

    /* clean-up after wakeup from exit */
    Clear_Flag to indicate Rendezvous occurrence;
    Enable_All_Interrupts;

    /* return from interruption */
    return;
}

/*===========================END======================================*/

/*=========================BEGIN======================================*/

/* OS Corrected Error Interrupt Handler (processor and platform) */

/*=========================END======================================*/
/*----------------------------------------------*/
OS_Corrected_Error_Interrupt_Handler( )
{
  /* handler for corrected machine check intr.*/
  /* get error log */
  if(ProcessorCorrectedError)
    Sal_Get_State_Info(processor);
  else
    Sal_Get_State_Info(platform)

  /* Save log of MCA */
  Save_Error_Log( );

  /* now we can clear the errors */
  if(ProcessorCorrectedError)
    Call Sal_Clear_State_Info(processor);
  else
    Call Sal_Clear_State_Info(platform);

  /* return from interruption */
  return;
}
/*----------------------------------------------*/

/*-----------------------------------------------*/
/* OS Core Machine Check Handler */
/*====================================================================*/
OS_MCA_Handler( )
{
  /* handler for uncorrected machine check event */
  Save_Processor_State();

  if(ErrorType!=Processor TLB)
SwitchToVirtualMode();
else
    StayInPhysicalMode();

/* Assuming that the OS can call SAL in physical mode to get info */
SAL_GET_STATE_INFO(MCA);

/* check for error */
if(ErrorType==processor)
{
    ErrorCorrectedStatus=OsProcessorMca();
}
If(ErrorType==Platform)
{
    Elect a Monarch processor;
    ErrorCorrectedStatus|=OsPlatformMca();
}

Note: If the error is not corrected, OS may want to reboot the machine and can do it by returning to SAL with a failure return result.

If(ErrorCorrectedStatus==failure)
    branch=ReturnToSAL_CHECK

Note: Errors are corrected, so try to wake up processors which are in Rendezvous. OS data structures should indicate which processors have rendezvoused.

/* completed error handling */
if(ErrorCorrectedStatus==success && InRendezvous()==true)
    WakeUpApplicationProcessorsFromRendezvous();

Note: If saving of the error record is to disk or the OS event log, then this is core OS functionality.

/* as a last thing */
Save_Error_Log();

Note: This is a very important step, as this clears the error record and also indicates the end of machine check handling by the OS. SAL uses this to clear any state information it may have related to which processors are in the MCA and any state of earlier rendezvous.
Call Sal_Clear_State_Info(MCA);

ReturnToSAL::
    /* return from interruption on all the processors that entered OS_MCA*/
    SwitchToPhysicalMode();
    Restore_Processor_State();

    /* return to SAL CHECK, SAL would do a reset if OS fails to correct*/
    return(ErrorCorrectedStatus)
}

/*===========================END======================================*/

/*=========================BEGIN======================================*/
/* Os Platform Machine Check Handler */
/*====================================================================*/
OsPlatformMca()
{
    ErrorCorrectedStatus=True;

    /* check if the error is corrected by PAL or SAL */
    If(ErrorRecord.Severity==not corrected)
        /* call sub-routine to try and correct the Platform MCA */
        ErrorCorrectedStatus=Correctable_Platform_MCA(platform_error_type);

    return(ErrorCorrectedStatus);
}

/*===========================END======================================*/
/ * check if the error is corrected by Firmware */

If(ErrorRecord.Severity==not corrected)
    ErrorCorrectedStatus=TryProcessorErrorCorrection( );

Return(ErrorCorrectedStatus);

} /*----------------------------------------END----------------------------------------*/

/*----------------------------------------BEGIN----------------------------------------*/

/* Try Individual Processor Error Correction */

/*====================================================================*/

Note: Now the OS has the data logs. Start parsing the log retrieved from SAL. The sub-routine Read_OS_Error_Log will read data from the error log copied from SAL. An offset is passed to identify the data being read and the base pointer is assumed to be known by the Read_OS_Error_Log sub-routine just to simplify the pseudo-code.

TryProcessorErrorCorrection( )
{
    /* extract appropriate fields from the record header */
    Record_ID = Read_OS_Error_Log(Record_ID_Offset);
    Severity = Read_OS_Error_Log(Err_Severity_Offset);

    Note: It is unlikely that the OS can write to persistent storage in physical mode. If it is possible, the OS should do so. If it is not, the SAL firmware should still have a copy of the error log stored to NVM that will be persistent across resets.

    if (Severity == Fatal)
        SystemReset() or return(failure);

    if (Severity == Corrected)
        return(ErrorCorrectedStatus=True);

    Note: These errors may be recoverable by the OS depending on the OS capability and the information logged by the processor. Call the sub-routine, OS_MCA_Recovery_Code and on return set up a min-state save area to return to a context of choice. The PAL_MC_RESUME call made by SAL allows the OS to return to the interrupted context, which includes enabling of all pending MCAs.
if (Severity == Recoverable)
{
    ErrorCorrectedStatus=OS_MCA_Recovery();
    Set_Up_A_Min_State_For_OS_MCA_Recovery(my_minstate);
}
return(ErrorCorrectedStatus);

} /* End of TryProcessorErrorCorrection Handler */
/*---------------------------------------------------------------*/

/*---------------------------------------------------------------*/
/* OS_MCA Recovery Code */
/*---------------------------------------------------------------*/

Note: At this point the OS is running with address translations enabled. This is needed otherwise the OS would not be able to access all of its data structures needed to analyze if the error is recoverable or not.

OS_MCA_Recovery( )
{
    /* Set up by default that the errors are not corrected */
    CorrectedErrorStatus = CorrectedCacheErr = CorrectedTlbErr =
    CorrectedBusErr = CorrectedRegFileErr = CorrectedUarchErr = 0;

    /* Start parsing the error log */
    RecordLength = Read_OS_Error_Log(Record_Length_Offset);
    Section_Header_Offset = OS_Error_Log_Pointer + Record_Header_Length;

    /* Find the processor error log data */
    Processor_Error_Log_Found = 0;

    /* traverse the error record structure to find processor section */
    while (Processor_Error_Log_Found == 0)
    {
        SectionGUID = Read_OS_Error_Log(Section_Header_Offset + GUID_Offset);
SectionLength = Read_OS_Error_Log(Section_Header_Offset + Section-Length_Offset);

if (SectionGUID == Processor_GUID)
    Processor_Error_Log_Found = 1;

Section_Body_Pointer = Section_Header_Offset + Section_Header_Length;
Section_Header_Offset = Section_Header_Offset + SectionLength;

if (Section_Header_Offset >= RecordLength)
    InternalError(); /* Expecting a processor log */

Note: Start parsing the processor error log. Section_Body_Pointer was set up to point to the first offset of the processor error log in the while loop above. Check the valid bits to see which part of the structure has valid info. The Read_OS_Error_Log sub-routine is assumed to know the initial pointer and just an offset is passed. This was done to allow the pseudo-code to be more readable.

Proc_Valid_Bits = Read_OS_Error_Log(Section_Body_Pointer);
Section_Body_Pointer = Section_Body_Pointer + Validation_Bit_Length;

/* Read the Processor Error Map if the valid bit is set. */
if (Proc_Valid_Bits[Proc_Error_Map_Valid] == 1)
    Proc_Error_Map = Read_OS_Error_Log(Section_Body_Pointer);

/* Extract how many errors are valid in the error log and determine which type */
Cache_Check_Errs = Proc_Valid_Bits[Cache_Check_Valid];
TLB_Check_Errs = Proc_Valid_Bits[TLB_Check_Valid];
Bus_Check_Errs = Proc_Valid_Bits[Bus_Check_Valid];
Reg_File_Errs = Proc_Valid_Bits[Reg_File_Check_Valid];
Uarch_Errs = Proc_Valid_Bits[MS_Check_Valid];

/* These sub-routines will return an indication of if the error can be corrected by killing the affected processes. */
if (Cache_Check_Errs != 0)
{
    /* If uncorrected cache errors are reported, the OS should attempt not to */
/ * use the cache. This would require the OS to not caching any of the */
/* code or data used during recovery. */
/* */
/* Check to see if one or multiple cache errors occurred */
if (Cache_Check_Errs == 1)
    CorrectedCacheErr = Handle_Single_Cache_Error(Section_Body_Pointer);
else
    CorrectedCacheErr = Handle_Multiple_Cache_Errors(Section_Body_Pointer);
}

if (TLB_Check_Errs != 0)
{
    /* Check to see if one or multiple TLB errors occurred */
    if (TLB_Check_Errs == 1)
        CorrectedTlbErr = Handle_Single_TLB_Error(Section_Body_Pointer);
    else
        CorrectedTlbErr = Handle_Multiple_TLB_Errors(Section_Body_Pointer);
}

if (Bus_Check_Errs != 0)
{
    /* Check to see if one or multiple Bus errors occurred */
    if (Bus_Check_Errs == 1)
        CorrectedBusErr = Handle_Single_Bus_Error(Section_Body_Pointer);
    else
        CorrectedBusErr = Handle_Multiple_Bus_Errors(Section_Body_Pointer);
}

if (Reg_File_Errs != 0)
{
    /* Check to see if one or multiple Register file errors occurred */
    if (Reg_File_Errs == 1)
        CorrectedRegFileErr = Handle_Single_Reg_File_Error(Section_Body_Pointer);
    else
        CorrectedRegFileErr = Handle_Multiple_Reg_File_Errors(Section_Body_Pointer);
Pseudocode – OS_MCA

if (Uarch_Errs != 0)
{
/* Check to see if one or multiple uarch file errors occurred */
if (Uarch_Errs == 1)
    CorrectedUarch_Err = Handle_Single_Uarch_Error(Section_Body_Pointer);
else
    CorrectedUarch_Err = Handle_Multiple_Uarch_Errors(Section_Body_Pointer);
}

CorrectedErrorStatus = CorrectedCacheErr | CorrectedTlbErr | CorrectedBusErr |
                    CorrectedRegFileErr | CorrectedUarch_Err;

return(CorrectedErrorStatus);
} /* end OS_MCA_Recovery_Code */

/*-------------------------------END======================================*/