Platform-Level Error Handling Strategies for Intel Systems

White Paper

Ai Bee Lim
Eric D Heaton
Senior Platform Application Engineer
Embedded Communications Group
Communications Infrastructure Division
Intel Corporation

May 2011

325486

Executive Summary

This paper provides an overview of common error detection and notification capabilities found in Intel® Architecture (IA)-based systems and how they can be used to implement a number of platform-level error handling schemes.

The Intel® Embedded Design Center provides qualified developers with web-based access to technical resources. Access Intel Confidential design materials, step-by step guidance, application reference solutions, training, Intel's tool loaner program, and connect with an e-help desk and the embedded community. Design Fast. Design Smart. Get started today. www.intel.com/embedded/edc.

Contents

Introduction

Fault tolerant systems have always been in high demand. What was once the realm of high-end servers and High Performance Computing (HPC) machines are now the norm for all types of applications. Customers who are producing Embedded or Communications products expect high system up-times and demand ease of management/repair in the event of correctable and uncorrectable errors. Chip manufacturing technologies continue to use smaller and smaller transistors and employ lower and lower voltages as they chase higher performance via power per watt metrics. This comes with the unintended consequence of systems that are more susceptible to soft errors caused by manufacturing variation, Alpha particles, neutrons, and electro-magnetic forces.

Given a sufficiently large sample size or time scale, errors will occur and data will be corrupted somewhere in the system. If there isn't a robust set of detection mechanisms or alerts to handle a full range of system problems, the performance of the system may degrade to the point of full system shutdown or reset. Many systems "go down" unexpectedly and it is often puzzling when they do so without warning or explanation (e.g., Blue screen).

For systems that are expected run continuously, an error condition that can take it offline would be unacceptable. Modern systems employ various techniques to make the platform more reliable and serviceable. This helps keep downtime to a minimum. New platforms must have a way to identify errors and have a set of strategies for gracefully dealing with the aftermath.

Modern Intel® processors and chipsets provide two major error-handling paradigms to help accomplish this goal across all elements in the system:

1. Machine Check Architecture (MCA), for Core and Uncore modules.
2. Advanced Error Reporting (AER), for PCI Express* devices and Integrated IO modules.

While some errors are corrected automatically by the hardware (e.g., single-bit ECC error in memory), others require outside hardware or software intervention (e.g., graphics device gets poisoned data over PCI Express interface). MCA and AER in parallel give system architects the ability to classify errors and give fine-grained control over how to respond to such problems.

Both MCA and AER offer flexibility in how they respond to any given error condition:

- How to classify this event?
- Which system component should be notified?
- Are there special operating modes for error handling available in the CPU/chipset?
- Can I change the configuration during system operation?

These questions need to be answered early in the project timeline because some of the features of MCA and AER require specific hardware connections if they are to be available for use by software. If you do not explicitly plan and design for particular MCA or AER features during the hardware design phase, then your future software options may be limited. This can restrict your ability to meet certain RAS requirements. Given the rich feature-set and configuration options for both of these paradigms, do not leave these important decisions to chance when develop a comprehensive platform-level error handling strategy.

While we will leave the register-level details to the relevant Intel Product Datasheets, the Intel® Software Developers Manual, and the appropriate PCI Express Specifications, this document will describe some of the major elements of MCA and AER configuration so that a system architect, in conjunction with their test/diagnostics/BIOS engineers, will be able to design a system that efficiently and reasonably responds to error conditions and other run-time anomalies.

Note: If you were familiar with past generations of MCA and AER, we would encourage you to revisit them in the context of their implementation in the latest generation of CPUs and chipsets; both paradigms have been expanded to provide a much larger set of features to handle a wider variety of exceptions.

Document Structure

The Introduction section provides motivation for the discussion of system-level error handling options in IA-based platforms.

The Component-Level Error Reporting and Alerting section will provide an overview of the system-level error logging, reporting, and alerting features generally found in IA-based platforms.

The Platform-Level Error Handling Strategies on Intel® Platforms section will take these features a step further and present common error handling strategies utilizing the aforementioned system-level error logging, reporting, and alerting features.

Note: This paper does not describe the details of stand-alone RAS features, like Memory Mirroring or Rank Sparing. For these, please see the Datasheet, External Design Specification, or BIOS Writer’s Guide for the particular component of interest; this document only focuses on the platform-level response and the “flow” of error handling.

Component-Level Error Reporting and Alerting

An IA-based platform typically includes an Intel® processor, an Intel® chipset, and a set of peripheral components (e.g., Ethernet NICs). Before we can talk about designing a platform-level error propagation strategy, it is prudent to understand the general capabilities inherent in each component of the system.

While most of the error logging, reporting, and alerting features mentioned in this document are found on most recent Intel® processors and chipsets, not all of them are. Consult the Intel® Software Developers Manuals and the appropriate Intel® Datasheet for each component to make sure that the ones that you would like to make use of are implemented and available.

Note: This document uses the Intel® Xeon® C5500/C3500 series processors and associated chipsets in its examples, but the concepts apply to many other Intel® platform components.

Error Reporting Options for Intel® Processors

The Intel® P5 Architecture introduced a standard methodology of error detection and reporting for the Core known as Machine Check Architecture (MCA). Since that first generation of the technology, the MCA has been enhanced and improved as successive processors have integrated memory controllers and other peripherals into its domain. The latest generations of processors use MCA to log and report errors in both the Core (CPU, caches, etc.) and the Uncore (e.g., QPI bus interconnect, integrated memory controllers, Intel® Virtualization Technology logic, etc.).

While the MCA is described fully in the IA32/64 Software Developer's Manual Volume 3A, the short version is that the MCA includes a number of well-defined registers that allow a system designer to decide which error conditions should be logged in real time and how the system should be alerted when they occur. The alert options range from a simple, well-known (i.e., hardcoded) software interrupt vector (i.e., "INT18") to the assertion of a dedicated hardware pin called Catastrophic Error (i.e., the CATERR# or CAT_ERR_N signal).

Processors that implement integrated I/O devices, like the integrated PCI Express controllers in the Intel® Xeon® C5500/C3500 series processors, will have, in addition to MCA, logic for Advanced Error Reporting as defined by the PCI Express specification. When an error is logged thorough the AER system, the system can be alerted via a number of methods. These include by MSI, by a common software interrupt vector, or by the assertion of external hardware pins (i.e., the ERR[2:0] or SYS_ERR[2:0] signals). Many processors that support AER also support a separate global error unit that will collect all AER-generated error messages/indicators in the system into a compact set of registers at the top-level of the bus hierarchy. This unit helps to streamline the error handling process by collecting the error status of each and every PCI or PCI Express device in the system and presenting this information within a small set of registers. If you want to root-cause why a certain AER error occurred, you would start with these global error registers, then follow the error register hierarchy all the way down to the PCI and PCI Express-based configuration/status registers inside the problem device.

Note: As of this writing, the global error unit only aggregates errors in the AER domain and not those errors that are handled via the MCA.

Intel® processors all implement a version of MCA and some –those with integrated I/O devices – will implement AER features in parallel. When an error occurs, the system can be alerted via different types of interrupts or via a dedicated set of hardware pins. These options are all configurable with software. If you want to rely on hardware pins to communicate system-error situations to external devices (e.g., a BMC), then these connections must be accounted for during the board design phase.

Error Reporting Options for Intel® Chipsets

Intel® chipsets have typically contained a PCI hierarchy with a bus starting at zero and associated internal controllers (e.g., USB controller, SATA controller, PCI Express root ports, etc.) implemented with all the typical configuration registers per the relevant PCI specification. Most modern Intel® chipsets include one or more integrated devices (ex. PCI Express controllers) that support various aforementioned AER capabilities. Such devices might contain special register sets to detect/report error conditions defined only for its particular interface (e.g., PxSERR registers for SATA, or USB2.0_STS for USB, etc.), beyond any defined by the normal PCI specification such as PCISTS. Some Intel® chipsets, like certain processors described in the previous section, also support a global error unit to provide a central place for error logging and reporting.

When the chipset encounters an error, the system can be alerted via hardware or software methods. On the software side, Intel® chipsets contain one or more legacy 8259 and IOxAPIC interrupt controllers (which traditionally were the central interrupt collectors for the system) that can be configured to assert an SMI/NMI or any regular software-based interrupt. With the advent of PCI Express and modern operating systems, interrupts are no longer limited to the physical wires in the system as provided by the 8259 or IOxAPIC interrupt controllers and can be delivered directly to the CPU virtually via MSI messages on any of the Intel® QPI or PCI Express busses within the system. This new option has an advantage in that MSI's bypass the legacy interrupt controller logic in the chipset and directs internal error messages straight to the local APIC of a particular logical CPU for immediate action.

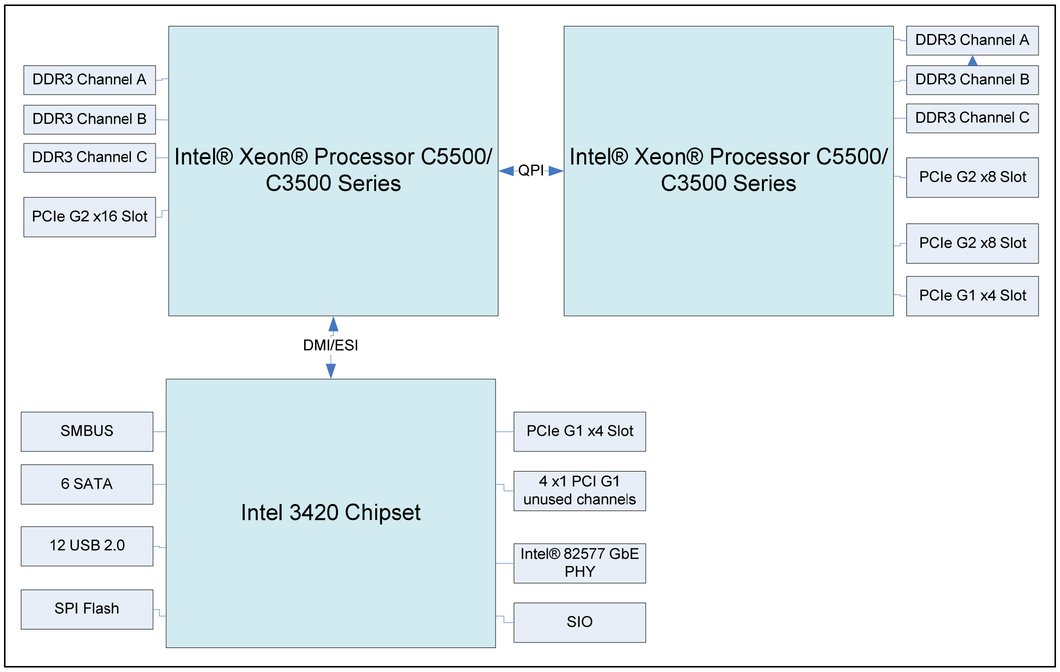
On the hardware side, there are many pins that can be used to signal that an error has occurred. Some of these pins are dedicated to specific error conditions (e.g., the PERR#, SERR#, SERIRQ, and PIRQ[H:A] signals), but there is usually a large set of GPIO pins in the chipset that can be configured to assert when other types of errors occur.

The devices integrated in the Intel® chipset will implement the normal PCI or PCI Express error logging registers inside an overall AER infrastructure. Once an error condition is encountered, the system can be alerted through software interrupts (MSIs usually being the best choice, if there are no legacy considerations) or hardware pins.

An example of this would be an Intel® Xeon® C5500/C3500 Processor and Intel® 3420 Chipset Platform.

To support our descriptions of the "potential" error-handling features that you may find in an IA-based platform, let's look at the specific features of a platform containing the Intel® Xeon® C5500/C3500 Processor and the Intel® 3420 chipset. Figure 1 shows how a typical two-processor system is connected to memory and various I/O components.

Figure 1 Block Diagram for an Intel® Xeon® C5500/C3500 Processor and Intel® 3420 Chipset-based Platform

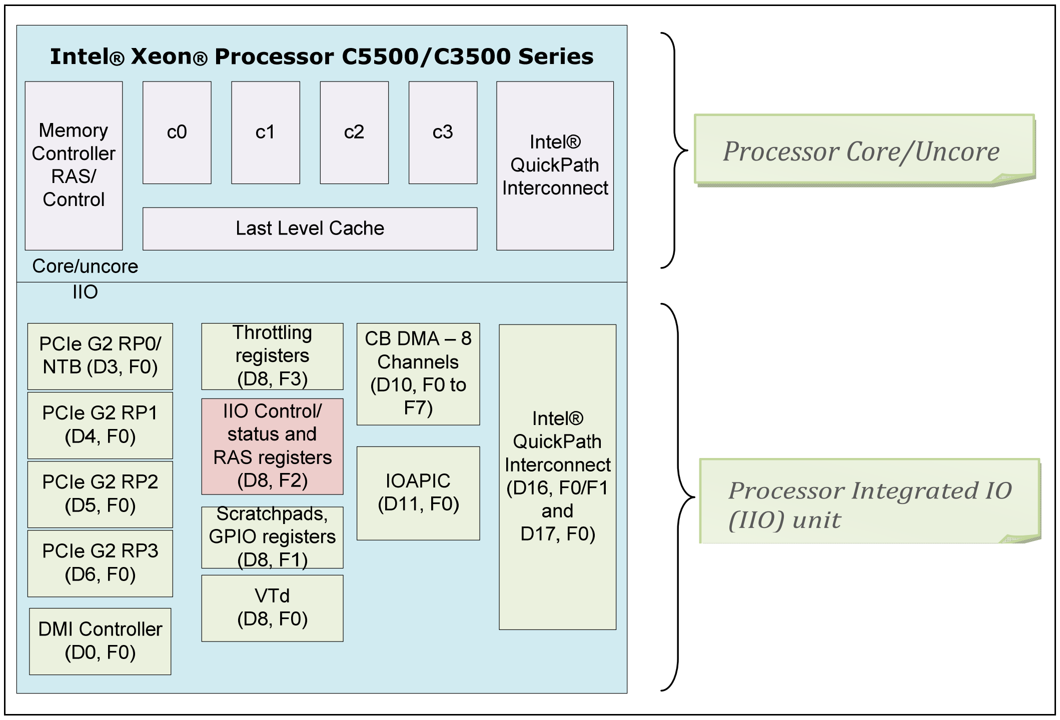


In this system, the two Intel® Xeon® C5500/C3500 Processors consist of up to four execution units, each supporting SMT (i.e., four physical cores, with up to eight logical cores with private caches), a large shared 8MB Last Level Cache (LLC), three integrated DDR3 Memory controllers, and an Integrated IO (IIO) unit.

The IIO unit includes:

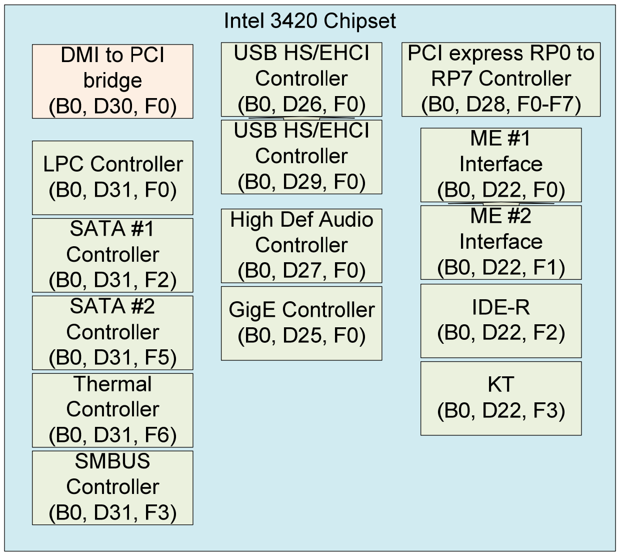
- One x16 PCI Express Gen2 external interface - split across four independent controllers, if desired
- DMI interface - similar to a x4 PCI Express Gen1 root complex
- Crystal Beach DMA engine - up to eight independent channels
- Intel® QPI interface between processors
- Intel® VT-d features such as virtual-to-physical address remapping and interrupt remapping
- Other miscellaneous functions such as a GPIO controller

Figure 2 Internal Functional Block for Intel® Xeon® C5500/C3500 Processor


Intel® Xeon® Processor C5500/C3500 Series
Memory Controller RAS/ Control
c0
c1
c2
c3
Intel® QuickPath Interconnect
Last Level Cache
Core/uncore
IIO
PCIe G2 RP0/ NTB (D3, F0)
PCIe G2 RP1 (D4, F0)
PCIe G2 RP2 (D5, F0)
PCIe G2 RP3 (D6, F0)
DMI Controller (D0, F0)
Throttling registers (D8, F3)
IIO Control/ status and RAS registers (D8, F2)
Scratchpads, GPIO registers (D8, F1)
VTd (D8, F0)
CB DMA – 8 Channels (D10, F0 to F7)
IOAPIC (D11, F0)
Intel® QuickPath Interconnect (D16, F0/F1 and D17, F0)
Processor Core/Uncore
Processor Integrated IO (IIO) unit

Besides the DMI connection to the processor, the Intel® 3420 Chipset provides a number of interfaces for common system peripherals like legacy PCI devices, USB devices, SATA hard drives, SPI flash devices, GigE devices, LPC devices, and PCI Express Gen1 devices (two x4s, with several configurations available).

Figure 3 Internal Functional Block for Intel® 3420 Chipset


Intel 3420 Chipset
DMI to PCI bridge (B0, D30, F0)
USB HS/EHCI Controller (B0, D26, F0)
PCI express RP0 to RP7 Controller (B0, D28, F0-F7)
LPC Controller (B0, D31, F0)
USB HS/EHCI Controller (B0, D29, F0)
ME #1 Interface (B0, D22, F0)
SATA #1 Controller (B0, D31, F2)
High Def Audio Controller (B0, D27, F0)
ME #2 Interface (B0, D22, F1)
SATA #2 Controller (B0, D31, F5)
GigE Controller (B0, D25, F0)
IDE-R (B0, D22, F2)
Thermal Controller (B0, D31, F6)
KT (B0, D22, F3)
SMBUS Controller (B0, D31, F3)

The MCA error domain maps to the Processor Core/Uncore functional blocks and the AER error domain maps to the Processor Integrated IO Unit and all devices in the chipset.

The MCA error domain in the Intel® Xeon® C5500/C3500 Processor acts as described in the Intel® Software Developer's Manual Vol. 3A for all the functional blocks mentioned above, noting that model-specific error codes of the IA32_MCi_STATUS MSR's can be found in the Nehalem BIOS Writer's Guide. The processor implements the usual OOB error signal, the CATERR# pin, which is pulsed for 16 BCLK's for all but the most catastrophic problems (in which case it goes low until reset). This pin should be connected to a GPIO on the Intel® 3420 chipset or to a BMC that is programmed to respond to MCE's. For more information, see the section titled *Platform-Level Error Handling Strategies on Intel® Platforms*.

The AER error domain runs across both the Intel® Xeon® C5500/C3500 processors and the Intel® 3420, but there two differences to note:

1. As a benefit for those writing system error handling code, the Intel® Xeon® C5500/C3500 processor provides an integrated central error control/collection point (Device 8 / Function 2). The main function of this device is to provide the central control of error detection and reporting for the entire Processor IIO unit. The Intel® 3420 chipset has no such global error collector. You can set up the system to route all system errors, even those of the chipset, to the processor and use its global error collector for error handling.
2. The processor has dedicated error pins that are specifically mapped to each of the three error severities defined in the AER spec. The processor documentation calls these pins SYS_ERR_STAT[2:0] or Error[2:0]. The chipset does not have these same dedicated error pins, but it can be

configured to assert GPIOs when a given error occurs. Either set of pins can be hooked up to a BMC for OOB error handling.

Please consult the documentation for both the Intel® Xeon® C5500/C3500 processor and the Intel® 3420 chipset for a full description of all of the error handling features mentioned in this section.

Platform-Level Error Handling Strategies on Intel® Platforms

The previous chapter discussed the error reporting and logging mechanisms that are typically found in modern IA processors and chipsets. IA-based platforms provide the flexibility to handle system problems through a variety of hardware or software mechanisms. This chapter will explore the most common and useful of these options, noting that you may mix-and-match any of these techniques to meet the specific requirements of your platform.

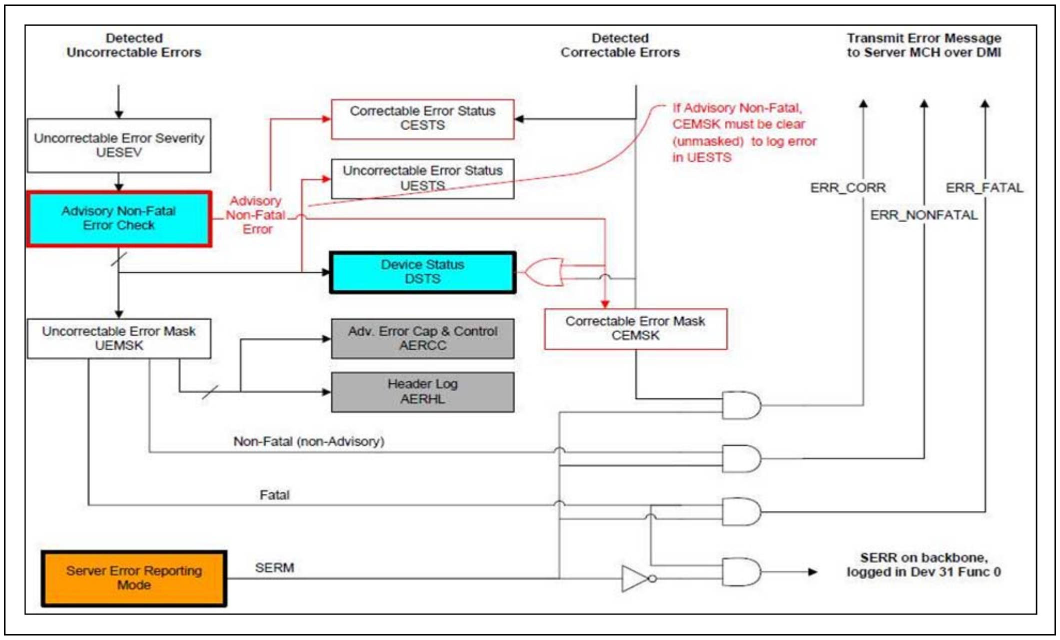
Once done with this paper, we encourage system architects to look at RAS, MCA, and AER features of the Intel components in your particular design and use this information to create a flow-chart that details how each error (or class or errors) should be handled, and "who" – this is, which component – in the system will be responsible for handling it.

Centralized IO Error Reporting

Errors occurring within the CPU itself (or to devices connected to the CPU directly, such as memory or peripherals) will be logged and reported via the MCA and AER mechanisms mentioned earlier. What do we do about the PCI/PCIe-type errors happening on devices inside of or attached to an ICH or PCH connected to the CPU?

As a general rule, it is best to have all IO errors routed to the CPU so that the system can decide to handle the problem there (via various types of error handlers available within the processor) or send it to an external board management controller (via special error pins on the processor). Intel® chipsets can be configured to operate in "Server Error Reporting Mode" (SERM). Any error in the chipset will be forwarded to the CPU via internal messages on the DMI interface.

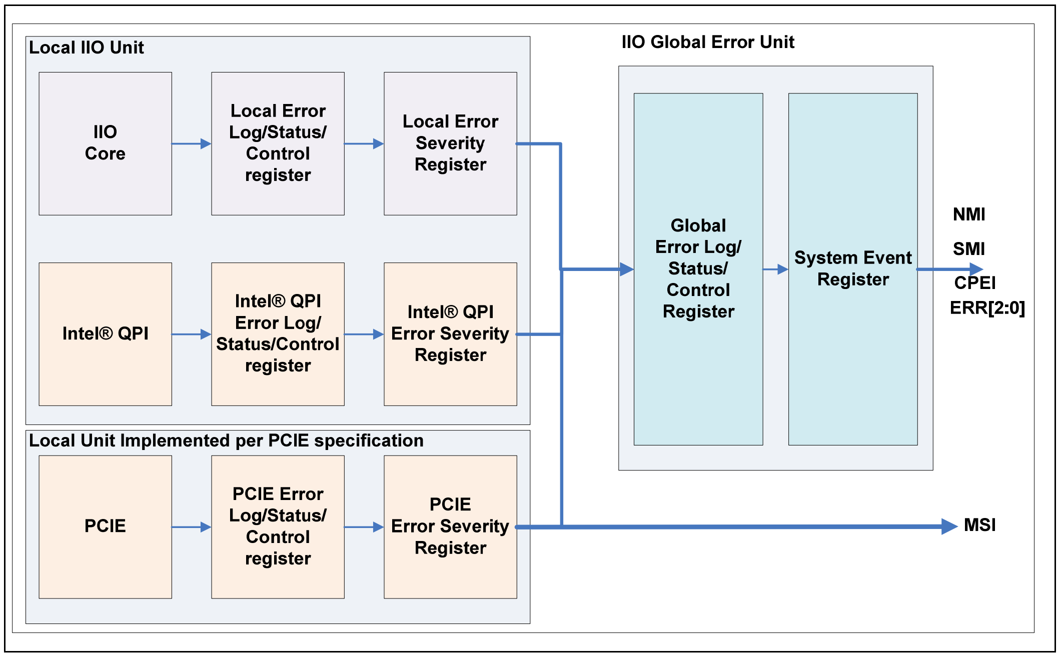
Figure 4 Generic diagram for SERM enabling logic


Detected Uncorrectable Errors
Detected Correctable Errors
Transmit Error Message to Server MCH over DMI
Uncorrectable Error Severity UESEV
Correctable Error Status CESTS
Uncorrectable Error Status UESTS
If Advisory Non-Fatal, CEMSK must be clear (unmasked) to log error in UESTS
Advisory Non-Fatal Error Check
Advisory Non-Fatal Error
Device Status DSTS
ERR_CORR
ERR_NONFATAL
ERR_FATAL
Uncorrectable Error Mask UEMSK
Adv. Error Cap & Control AERCC
Header Log AERHL
Correctable Error Mask CEMSK
Non-Fatal (non-Advisory)
Fatal
Server Error Reporting Mode
SERM
SERR on backbone, logged in Dev 31 Func 0

SERM can be enabled by setting a bit in the General Control and Status Register (GCS) found in all modern Intel® chipsets. In this mode, if the chipset detects any type of PCI/PCIe error, fatal, non-fatal, or correctable, it will propagate the internal error message to the CPU, which will treat it like any other IIO error. When any of these error messages arrive at the CPU (shown as ERR_CORR, ERR_NONFATAL, and ERR_FATAL in Figure 4), the information contained within the message will be populated in the local DMI interface registers and the Global Error Registers.

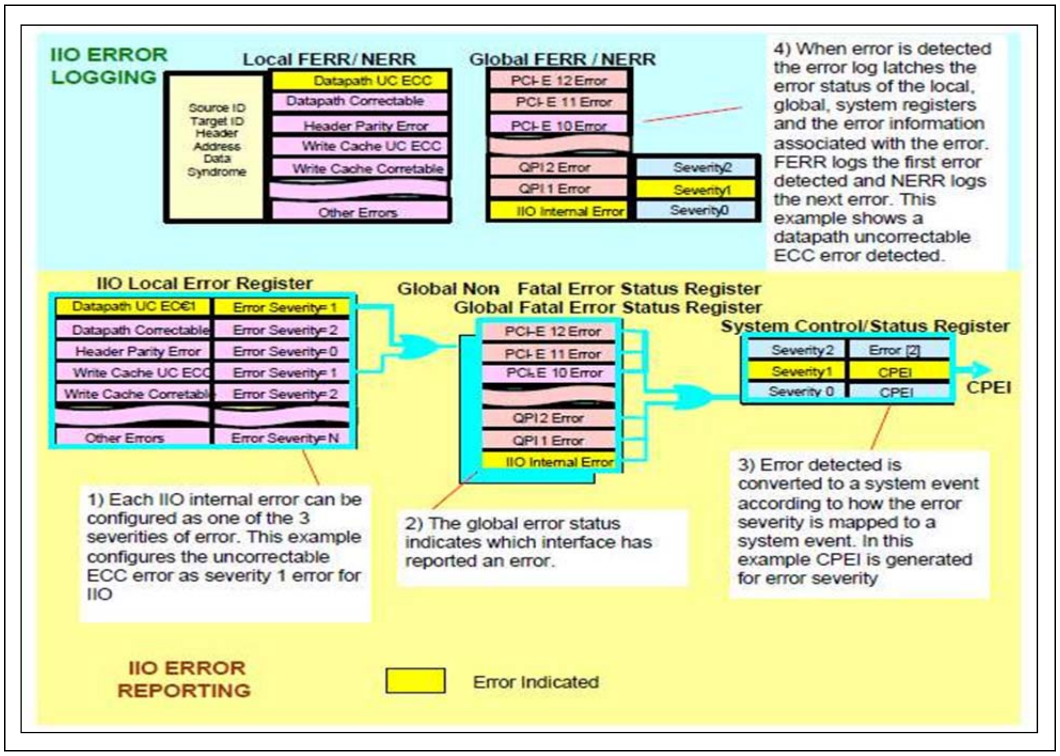
Note: Even if the chipset is set for SERM, a particular error might not show up at the CPU because it was masked or disabled somewhere else in the chipset. This document assumes that each PCI/PCIe device in the system is programmed correctly to record and generate error messages regardless of being in SERM or not. Setting the SERM bit is simply the last step in the process to say that system error messages should also be sent up to the CPU even as the details are recorded in the chipset.

Figure 5 Local Error Registers Feeding Into the Global Error Registers


Local IIO Unit
IIO Core
Local Error Log/Status/ Control register
Local Error Severity Register
Intel® QPI
Intel® QPI Error Log/ Status/Control register
Intel® QPI Error Severity Register
Local Unit Implemented per PCIE specification
PCIE
PCIE Error Log/Status/ Control register
PCIE Error Severity Register
IIO Global Error Unit
Global Error Log/ Status/ Control Register
System Event Register
NMI
SMI
CPEI
ERR[2:0]
MSI

There is a hierarchy of error reporting that starts at the device (i.e., the local error reporting PCI registers). Each device can be configured to send such error reports to the interrupt controller (usually the APIC), which generates an interrupt (IRQ or MSI – your choice). The best place for an overview of the interrupt hierarchy is Chapter 10 of the IA32 Software Developers Manual (SDM) Vol 3A – for APIC operation.

Figure 6   Error Propagation from the Local Unit to the Global Unit


IIO ERROR LOGGING
Local FERR/ NERR
Source ID
Target ID
Header
Address
Data
Syndrome
Datapath UC ECC
Datapath Correctable
Header Parity Error
Write Cache UC ECC
Write Cache Corretable
Other Errors
Global FERR / NERR
PCI-E 12 Error
PCI-E 11 Error
PCI-E 10 Error
QPI 2 Error
QPI 1 Error
IIO Internal Error
Severity2
Severity1
Severity0
4) When error is detected the error log latches the error status of the local, global, system registers and the error information associated with the error. FERR logs the first error detected and NERR logs the next error. This example shows a datapath uncorrectable ECC error detected.
IIO Local Error Register
Datapath UC ECC1
Datapath Correctable
Header Parity Error
Write Cache UC ECC
Write Cache Corretable
Other Errors
Error Severity= 1
Error Severity= 2
Error Severity= 0
Error Severity= 1
Error Severity= 2
Error Severity= N
Global Non Fatal Error Status Register
Global Fatal Error Status Register
PCI-E 12 Error
PCI-E 11 Error
PCI-E 10 Error
QPI 2 Error
QPI 1 Error
IIO Internal Error
System Control/Status Register
Severity2
Severity1
Severity 0
Error [2]
CPEI
CPEI
CPEI
1) Each IIO internal error can be configured as one of the 3 severities of error. This example configures the uncorrectable ECC error as severity 1 error for IIO
2) The global error status indicates which interface has reported an error.
3) Error detected is converted to a system event according to how the error severity is mapped to a system event. In this example CPEI is generated for error severity
IIO ERROR REPORTING
Error Indicated

With SERM enabled in the chipset, the Global Error Register set of the CPU is a hub for normal system errors. In the case of errors in the chipset (or devices attached to it), these registers will indicate that the problem is in “DMI port 0”. This set of registers will indicate which internal operating unit generated (or, at least, received) the error. For example, besides DMI port 0, the issue could be in one of the integrated PCIe ports, the VT-d logic, etc. This points the error handler to the PCI error registers of that specific unit for more information.

At start-up time, software will configure the Global Error register set (which shows up as its own B/F/D in PCI space of the processor) with information on how the system should react to various types of errors.

For example, when a certain correctable error occurs, the system may want to be alerted via a simple software interrupt or via the assertion of an external pin into a BMC implemented on the board.

While the details might change slightly depending on the processor family, this configuration involves a few registers in both the end device and in the root ports themselves. In the Intel® Xeon® C3500/C5500 Processor Family, these are the registers that control if/how errors will propagate from a device in the chipset, up to the processor, and on the chipset:

- GCS, bit 9 – to enable SERM
- NMI_SC, bit 2 – enable for NMI generation, if using NMI to signal errors
- PCICMD, bit 8 – to enable SERR generation by an individual PCIe device
- BCTRL, bit 1 – to forward error messages received by the device to the backbone
- DCTL, bits 0-2 – to enable reporting of error messages to the root control register
- RCTL, bits 0-2 – to enable reporting of errors in the root port for devices below it
- UEM, all bits – for masking various uncorrectable errors on the device
- UEV, all bits – for assigning a severity to uncorrectable errors for the device
- CEM, all bits – for masking various correctable errors on the device
- On the processor:
  - MISCCTRLSTS – bits 33 -35, to override system error enables for individual devices
  - DEVCON, bits 0-2 – to enable reporting of error messages upwards in the hierarchy
  - ROOTCON, bits 0-2 – to enable reporting of errors in the root port for devices below it
  - PCICMD, bit 8 – to enable SERR generation by an individual PCIe device
  - DEVCTRL, bits 0-2 – to enable reporting of errors received to the root control register
  - UNCERRMSK, all bits – for masking various uncorrectable errors on the device
  - UNCERRSEV, all bits – for assigning a severity to uncorrectable errors for the device
  - CORERRMSK, all bits – for masking various correctable errors on the device
  - RPERRCMD, bits 0-2 – to enable MSI generation for errors events, if desired
  - ERRPINCTL, bits 0-5 – to control the behavior or the ERR[2:0] pins (i.e., to assert them or not) when an error occurs anywhere in the system

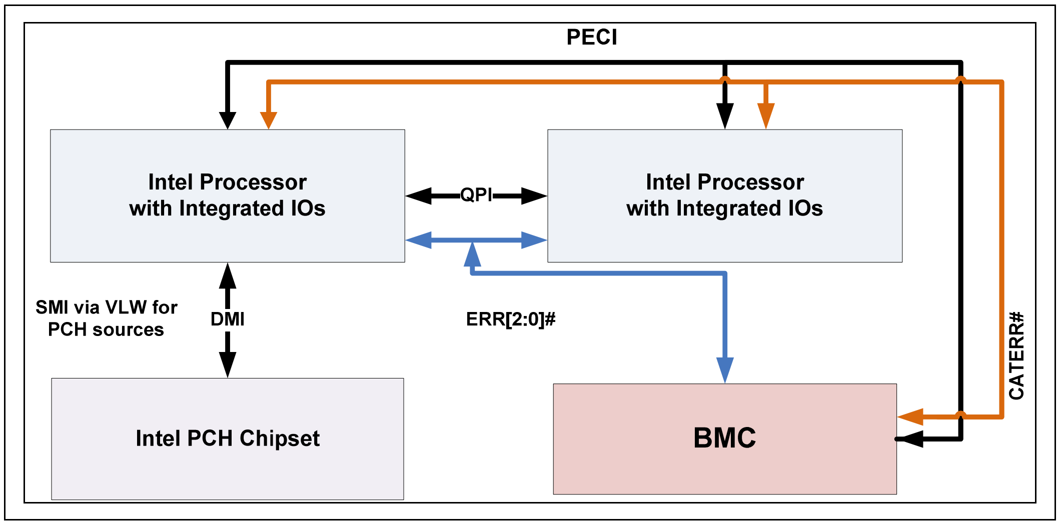
Software (e.g., a software-based error handler) can use the ERRPINDAT register to assert any of the ERR[2:0] pins at will.

Remember that AER (and all PCIe constructs) works as a hierarchy of devices. Associated events and a BIOS/diagnostics engineer need to make sure that all the mask and enable bits for each one of these registers are set correctly at each level of the system (e.g., at the end-device, at the root port, at the chipset control, etc.), for both external and internal devices.

Using a Board Management Controller

Catastrophic errors may be handled by an independent Board Management Controller (BMC) connected to the processor and/or chipset via a set of pins dedicated to error reporting. With such a setup, the platform is more likely to recover from a wider range of problems because the system does not rely on code running on the CPU.

Figure 7 All BMC Solution for Error Handling


PECI
Intel Processor with Integrated IOs
QPI
Intel Processor with Integrated IOs
SMI via VLW for PCH sources
DMI
ERR[2:0]#
CATERR#
Intel PCH Chipset
BMC

In previous generations of IA processors, the MCA used two different pins to indicate a problem. The first, MCERR, would indicate a "fatal" error occurred, and the second, IERR, would indicate that a "catastrophic" error occurred. The definition of which errors are "fatal" and which errors are "catastrophic" was generally defined for a family of processors. In the Intel® Xeon® C5500/C3500 series generation of processors, the CPU simply uses a single pin (the CATERR# pin) for either situation.

- If an "MCERR class" error occurs, CATERR# is asserted for 16 BCLKs.
- If an "IERR class" error occurs, CATERR# remains asserted until warm or cold reset.

The notion of fatal and catastrophic errors is still with us; a BMC can figure out which error occurred by counting the number of clocks CATERR# asserted and proceed accordingly with its analysis/handling of the error condition.

In the IIO, the processor will use three pins, called "ERROR_N[2:0]" or "ERR[2:0]" to indicate various severities of errors to an external agent.

- ERR[0]# = Hardware correctable error (no software action necessary)
- ERR[1]# = Non-fatal error (OS or firmware action required to contain and recover)
- ERR[2]# = Fatal error (system reset likely required to recover)

The ERRPINCTL register in the processor defines if and how the AER logic asserts these three pins if any type of IO error occurs on the platform. For example, this register can tell the AER logic to assert ERR[2]# to a BMC if a fatal error occurs, but not bother asserting ERR[1]# or ERR[0]# for non-fatal or corrected errors, respectively (because they will be handled via other, software-controlled mechanisms). The ERRPINDAT register can be used to assert any one of these pins at any time via software. For example, a software error handler may be called for every non-fatal error, but this code may determine that the BMC should handle a certain subset of these after doing some initial triage of the problem. In this case, the software could assert the ERR[1]# pin when it wants to signal the BMC.

When a BMC sees that a CATERR# or any of the three ERR[2:0]# pins are asserted, the BMC knows that there is a problem, but it is up to the system architect / BMC programmer to determine how exactly to respond. In some systems, the appropriate response to any type of error would be logging a MCE or IO error and then resetting the system. This approach does not distinguish among the many different instances of each type of error, but may work for some. More likely, the BMC will run a routine to query the state of the CPU and/or chipset and decide to take one of many actions based on what it finds.

How does the BMC communicate with these other components and read debug data and registers? The answer depends on the interfaces available to the BMC (and which interfaces are supported by the BMC itself). Each processor and chipset contain several different interfaces (PECI, SMBus, and PCIe) and each bus will allow an outside entity, like a BMC, different types of permissions/access to its registers. It is up to the system architect to decide which bus(es) or interface(s) will be used to connect the BMC to the rest of the platform components.

Commonly, though, PECI is the interface used by BMCs for this purpose, but note that different chips will provide different amounts of system access via this bus. For example, the Intel® Xeon® C5500/C3500 Processor mentioned in the previous section allows access to many MSRs related to error handling as well as to various portions of PCI configuration space, but previous generations typically limit access only to the latter resources. If your hardware-based error-handling paradigm requires access to certain information in the processor or chipset, be sure that the interface connecting the component allows the BMC access to it. If the PECI bus doesn't allow the BMC the view into the system that is required of the BMC software, use another one, like SMBus or PCIe, instead.

Interrupt-based Error Handling

Some systems may not implement a BMC at all, or may choose to handle certain classes of errors via software (most likely, those that are non-fatal). In these cases, a system architect or diagnostics engineer may use some of the various interrupt-based error reporting and handling schemes that are supported by IA® processors and chipsets.

Types of Interrupts

On IA-based platforms, various interrupts have a given set of behaviors, norms, and typical usage patterns. Beyond the general software interrupts enabled by legacy 8259 logic and various local APIC's and IOAPIC's (see Section 6 in Volume 3A of the Intel® 64 and IA-32 Architectures Software Developer's Manual for more information about these general interrupts) there are several types that are specifically designed to assist in the processing of serious system errors.

With MCA, interrupt 18 in the Interrupt Descriptor Table (IDT), is architecturally assigned to assert any time a Machine Check Error occurs. The vector at this IDT location could point to a software handler that performs one or more of the following actions:

- Determine the scope of the problem and try to recover the system to a normal operating state, by reloading or resetting only the software/hardware that is affected,
- Collect as much information as possible, given the current state of the system, and store this for later analysis (i.e., by BIOS or other system management software),
- Set up the system so that another entity continues the error handling process "up the chain". In this case, the MCE interrupt handler does triage and determines that another component in the system (e.g. a BMC) would be better to return the system to a normal state.

In a similar fashion, interrupt 2 in the IDT will assert whenever a non-maskable interrupt (NMI) occurs. The NMI vector would perform exactly the same set of actions as described for an MCE (i.e., interrupt 18).

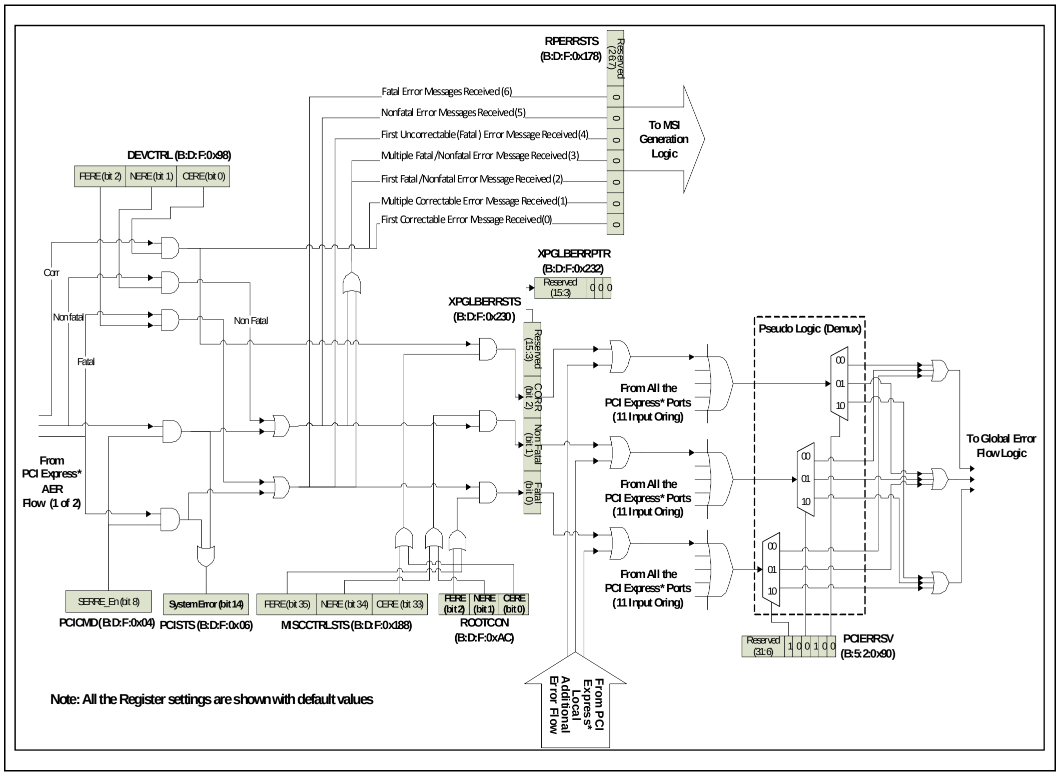
The Corrected Machine Check Interrupt (CMCI) is a relatively new type of interrupt that is part of the MCA and is used to alert system software that a certain number (user-configurable) of correctable errors have occurred. It is signaled like normal software interrupt and its descriptor will point to a particular vector in the IDT.

With AER, Message Signaled Interrupts (MSIs) are the most common type of interrupt. They will be applied to most of the IO error conditions that a system normally encounters. The AER logic of a processor (if it contains IIO) or a chipset generally allows any PCIe device to send an MSI regardless of the action (e.g., one of the ERR[n]# pins are asserted). If all the enable or mask bits for an error are programmed so they fully propagate through the

AER hierarchy (per the list in Section 3.1), then a MSI is asserted (and a software handler called), these additional registers need to be set:

- MSICTRL, bit 0 – to enable MSI messages for the device
- MSIAR, all bits – to specify how the MSI should be routed in the system
- MSRDR, all bits – to specify the interrupt vector for a particular device
- MSIMSK, bits 0-1 – to mask particular MSI messages for a device

Figure 8 Example of PCI Express AER Flow, focused on MSI support


RPERRSTS (B:D:F:0x178)
Reserved (26:7)
Fatal Error Messages Received (6)
Nonfatal Error Messages Received (5)
First Uncorrectable (Fatal) Error Message Received (4)
Multiple Fatal /Nonfatal Error Message Received (3)
First Fatal /Nonfatal Error Message Received (2)
Multiple Correctable Error Message Received (1)
First Correctable Error Message Received (0)
0 0 0 0 0 0 0
To MSI Generation Logic
DEVCTRL (B:D:F:0x98)
FERE (bit 2) NERE (bit 1) CERE (bit 0)
Corr
Non fatal
Fatal
Non Fatal
XPGLBERRPTR (B:D:F:0x232)
Reserved (15:3) 0 0 0
XPGLBERRSTS (B:D:F:0x230)
Reserved (15:3)
CORR (bit 2)
Non Fatal (bit 1)
Fatal (bit 0)
From PCI Express* AER Flow (1 of 2)
From All the PCI Express* Ports (11 Input Oring)
Pseudo Logic (Demux)
00 01 10
To Global Error Flow Logic
SERRE_En (bit 8)
PCICMD (B:D:F:0x04)
System Error (bit 14)
PCISTS (B:D:F:0x06)
FERE (bit 35) NERE (bit 34) CERE (bit 33)
MISCCTRLSTS (B:D:F:0x188)
FERE (bit 2) NERE (bit 1) CERE (bit 0)
ROOTCON (B:D:F:0xAC)
Reserved (31:6) 1 0 0 1 0 0
PCIERRSV (B:5:2:0x90)
From PCI Express* Local Additional Error Flow
Note: All the Register settings are shown with default values

Note: This document does not discuss legacy PCI interrupts; it is assumed that they are covered sufficiently by other sources.

Routing PIRQ Inputs to Specific IRQs

To support legacy PCI devices, a BIOS or an operating system will route PCI interrupts (PIRQs) to interrupt requests (IRQs) inside the chipset. PIRQs can be asserted by physical devices connected to these pins on the board or by some of the PCI devices integrated inside the chipset itself. For the latter case, the chipset allows software to choose which PIRQ is used by each integrated PCI device. A PCI function internal to the PCH and capable of driving an interrupt can be configured to drive any one of the INTA#, INTB#, INTC#, INTD# pins (originally, these were physical pins on an Intel® chipset, but have since been internalized). These can be mapped to any one of the PIRQs in the system. This mapping is done on a function-by-function basis via the Device X Interrupt Pin Registers, located at offsets 0x3000-0x316F in the memory-mapped space of Chipset Configuration Registers (i.e. the registers accessed using the Root Complex Base Address – RCBA – register of the PCI-to-LPC bridge).

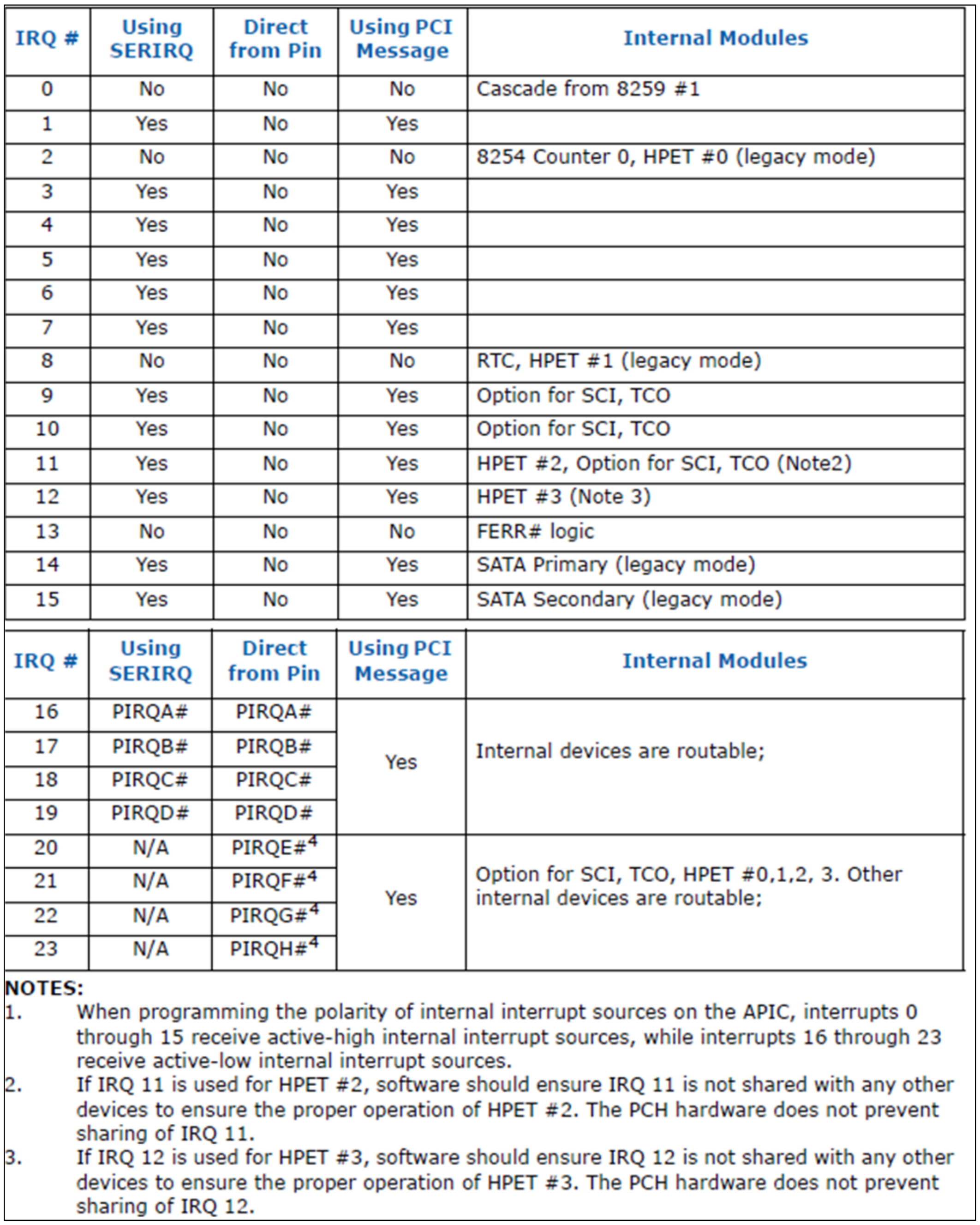
Note: Internal sources of PIRQs, including SCI and TCO interrupts, cause the associated external PIRQ pin to be asserted.

Once the INTx to PIRQ mapping is done for internal PCI devices that require it, the next step is to map PIRQs to IRQs on either the legacy 8250 interrupt controller or the IOxAPIC.

In 8259 mode, software can map the PIRQs to any one of the following IRQs on the 8259 PIC: 3–7, 9–12, 14 or 15, all other IRQs are reserved for other purposes. The assignment is done through the PIRQx Route Control registers, located at 0x60–0x63 and 0x68–0x6B in Device 31: Function 0. If desired, one or more PIRQ lines can be routed to the same IRQ input on the 8259.

In APIC mode, the PIRQs are connected to the internal IOxAPIC in the following fashion (and is not configurable): PIRQA# is connected to IRQ16, PIRQB# to IRQ17, PIRQC# to IRQ18, PIRQD# to IRQ19, PIRQE# is connected to IRQ20, PIRQF# to IRQ21, PIRQG# to IRQ22, and PIRQH# to IRQ23.

Table 1 IRQ routing Table for Intel 3420 Chipset in APIC mode



IRQ #	Using SERIRQ	Direct from Pin	Using PCI Message	Internal Modules
0	No	No	No	Cascade from 8259 #1
1	Yes	No	Yes	
2	No	No	No	8254 Counter 0, HPET #0 (legacy mode)
3	Yes	No	Yes	
4	Yes	No	Yes	
5	Yes	No	Yes	
6	Yes	No	Yes	
7	Yes	No	Yes	
8	No	No	No	RTC, HPET #1 (legacy mode)
9	Yes	No	Yes	Option for SCI, TCO
10	Yes	No	Yes	Option for SCI, TCO
11	Yes	No	Yes	HPET #2, Option for SCI, TCO (Note2)
12	Yes	No	Yes	HPET #3 (Note 3)
13	No	No	No	FERR# logic
14	Yes	No	Yes	SATA Primary (legacy mode)
15	Yes	No	Yes	SATA Secondary (legacy mode)

IRQ #	Using SERIRQ	Direct from Pin	Using PCI Message	Internal Modules
16	PIRQA#	PIRQA#	Yes	Internal devices are routable;
17	PIRQB#	PIRQB#		
18	PIRQC#	PIRQC#		
19	PIRQD#	PIRQD#		
20	N/A	PIRQE#[4]	Yes	Option for SCI, TCO, HPET #0,1,2, 3. Other internal devices are routable;
21	N/A	PIRQF#[4]		
22	N/A	PIRQG#[4]		
23	N/A	PIRQH#[4]		

NOTES:

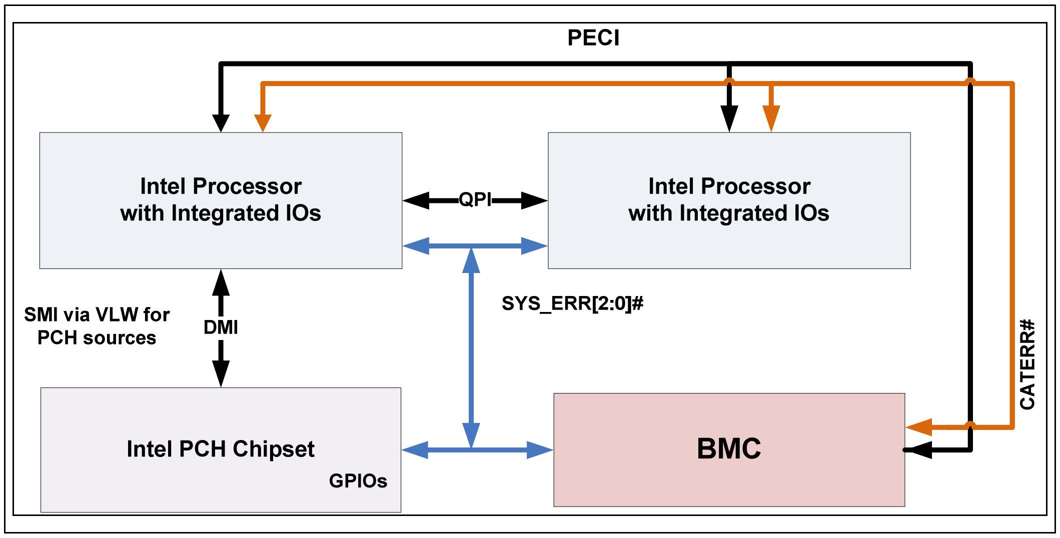
1. When programming the polarity of internal interrupt sources on the APIC, interrupts 0 through 15 receive active-high internal interrupt sources, while interrupts 16 through 23 receive active-low internal interrupt sources.
2. If IRQ 11 is used for HPET #2, software should ensure IRQ 11 is not shared with any other devices to ensure the proper operation of HPET #2. The PCH hardware does not prevent sharing of IRQ 11.
3. If IRQ 12 is used for HPET #3, software should ensure IRQ 12 is not shared with any other devices to ensure the proper operation of HPET #3. The PCH hardware does not prevent sharing of IRQ 12.

Routing SYS_ERR to NMI/SMI

The Intel® Xeon® C5500/C3500 series processors contain an IIO unit that supports a global error reporting and alerting hierarchy (AER) that can assert hardware pins when different types of errors occur (i.e. SYS_ERR[2:0]#). The most robust error handling solutions will connect these pins to one or more components in the platform. If an error is so severe that it makes the processor unreliable, it may still be handled by the system in a known and graceful manner.

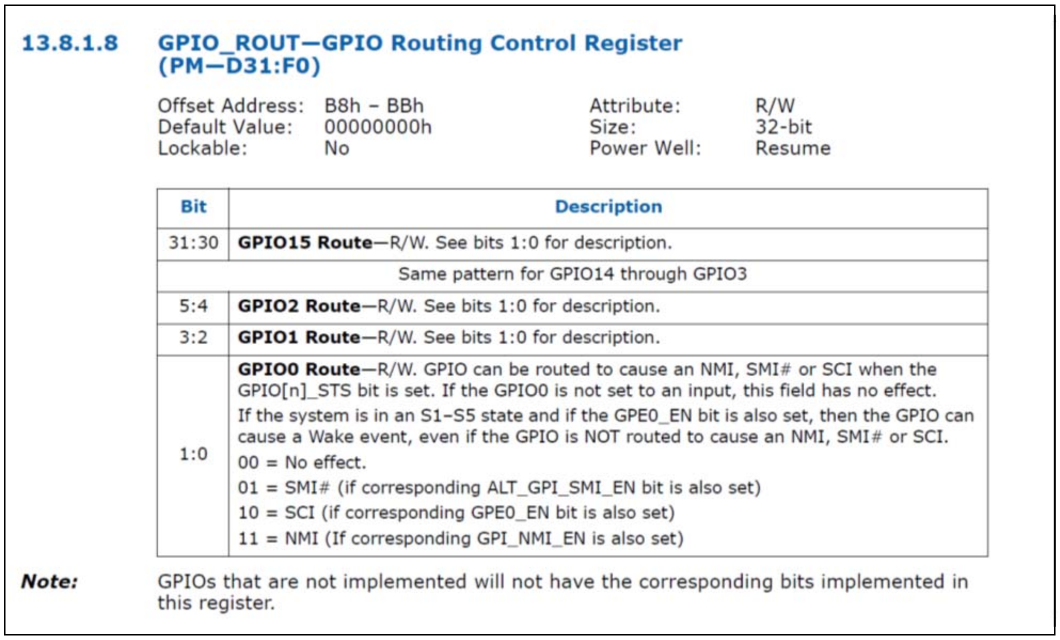
Figure 9 shows an example of routing of SYS_ERR[2:0]# pins to both the chipset and to an independent BMC. The BMC could be programmed to respond to the assertion of these pins. The BMC would start by reading the global error registers in the processor and then work its way through the error reporting hierarchy to figure out what went wrong. Alternatively, the SYS_ERR pins could be routed to a set of GPIOs in the chipset that have special logic that will cause either an NMI or SMI to occur. From there, the NMI or SMI interrupt handlers running on the processor would try to figure out what's going on.

Figure 9 Processor SYS_ERR pins connected to both the PCH and a BMC


PECI
Intel Processor with Integrated IOs
QPI
Intel Processor with Integrated IOs
SMI via VLW for PCH sources
DMI
SYS_ERR[2:0]#
CATERR#
Intel PCH Chipset
GPIOs
BMC

On the Intel® 3420 chipset, there are 16 GPIOs which can be programmed to assert a NMI or SMI as their input goes high. By default, the behavior is for them to do nothing, but this can be changed via the GPIO_ROUT register.

Figure 9 GPIO Routing Control Register Descriptions



13.8.1.8 GPIO_ROUT—GPIO Routing Control Register (PM—D31:F0)

Offset Address:	B8h – BBh	Attribute:	R/W
Default Value:	00000000h	Size:	32-bit
Lockable:	No	Power Well:	Resume

Bit	Description
31:30	**GPIO15 Route**—R/W. See bits 1:0 for description.
	Same pattern for GPIO14 through GPIO3
5:4	**GPIO2 Route**—R/W. See bits 1:0 for description.
3:2	**GPIO1 Route**—R/W. See bits 1:0 for description.
1:0	**GPIO0 Route**—R/W. GPIO can be routed to cause an NMI, SMI# or SCI when the GPIO[n]_STS bit is set. If the GPIO0 is not set to an input, this field has no effect. If the system is in an S1–S5 state and if the GPE0_EN bit is also set, then the GPIO can cause a Wake event, even if the GPIO is NOT routed to cause an NMI, SMI# or SCI. 00 = No effect. 01 = SMI# (if corresponding ALT_GPI_SMI_EN bit is also set) 10 = SCI (if corresponding GPE0_EN bit is also set) 11 = NMI (If corresponding GPI_NMI_EN is also set)

Note: GPIOs that are not implemented will not have the corresponding bits implemented in this register.

Most Intel® chipsets have a similar feature, but consult the specific datasheet for the chipset that you are using in your project to ensure that its GPIO signals support routing to an NMI or SMI. You will generally find the GPIO routing register in the Power Management set of registers in the LPC interface.

Summary

Comprehensive system-level error handling has never been higher, customers require high levels of fault tolerance, ease of management, and robust security. Given the fast pace of change in the architecture of systems (via the integration of previously-separate devices), it may be unclear to new system architects and engineers what features to look for in modern processors and chipsets to help them implement these requirements.

On Intel-based platforms, the two over-arching error handling paradigms are the Machine Check Architecture – which generally handles those problems of the processor, cache, and memory subsystems – and Advanced Error Reporting, which handles everything else connected to the PCI Express hierarchy (both on the processor and the chipset and on the devices connected to these system components). At this point, these systems are orthogonal to each other and a comprehensive error-handling strategy must take both into account.

MCA and AER both provide methods to handle errors through hardware or software methods. For the most robust system, review the design ideas that are outlined in this paper, with an eye towards situations that would compromise system security or would otherwise diminish the ability of the system to fulfill its central duties:

- Overview of MCA and AER Error Domains
- Centralized I/O Error Reporting
- Using a Board Management Controller
- Routing PIRQ Inputs to Specific IRQs
- Routing SYS_ERR to NMI/SMI

Discuss these options with both your hardware and software design teams as early as possible during the design phase. Some of the discussed methods require particular hardware hook-ups to fully implement.

Authors

Ai Bee Lim is a Senior Platform Application Engineer with the Embedded Communications Group at Intel Corporation.

Eric Heaton is a Senior Platform Application Engineer with the Embedded Communications Group at Intel Corporation.

§