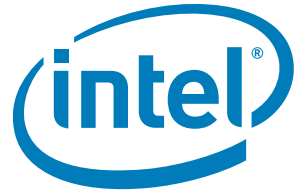

Intel® 64 Architecture x2APIC Specification

Reference Number: 318148-003
June 2008

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an ordering number and are referenced in this document, or other Intel literature, may be obtained from:

Intel Corporation
P.O. Box 5937
Denver, CO 80217-9808

or call 1-800-548-4725
or visit Intel's website at http://www.intel.com

CONTENTS

This page intentionally left blank

TABLES

This page intentionally left blank

FIGURES

This page intentionally left blank

CHAPTER 1
INTRODUCTION

1.1 INTRODUCTION

The xAPIC architecture provided a key mechanism for interrupt delivery in many generations of Intel processors and platforms across different market segments. This document describes the x2APIC architecture which is extended from the xAPIC architecture (the latter was first implemented on Intel® Pentium® 4 Processors, and extended the APIC architecture implemented on Pentium and P6 processors). Extensions to the xAPIC architecture are intended primarily to increase processor addressability. The x2APIC architecture provides backward compatibility to the xAPIC architecture and forward extendability for future Intel platform innovations. Specifically, x2APIC

- Retains all key elements of compatibility to the xAPIC architecture:
  - delivery modes,
  - interrupt and processor priorities,
  - interrupt sources,
  - interrupt destination types;
- Provides extensions to scale processor addressability for both the logical and physical destination modes;
- Adds new features to enhance performance of interrupt delivery;
- Reduces complexity of logical destination mode interrupt delivery on link based architectures.

1.2 IMPACTED PLATFORM COMPONENTS

x2APIC is architected to extend from the xAPIC architecture while minimizing the impact on platform components. Specifically, support for the x2APIC architecture can be implemented in the local APIC unit. All existing PCI/MSI capable devices and IOxAPIC unit should work with the x2APIC extensions defined in this document. The x2APIC architecture also provides flexibility to cope with the underlying fabrics that connect the PCI devices, IOxAPICs and Local APIC units.

The extensions provided in this specification translate into modifications to:

- the local APIC unit,
- the underlying fabrics connecting Message Signaled Interrupts (MSI) capable PCI devices to local xAPICs,
- the underlying fabrics connecting the IOxAPICs to the local APIC units.

However no modifications are required to PCI or PCI-e devices that support direct interrupt delivery to the processors via Message Signaled Interrupts. Similarly no modifications are required to the IOxAPIC. The routing of interrupts from these devices in x2APIC mode leverages the interrupt remapping architecture specified in the Intel Virtualization Technology for Directed I/O, Rev 1.1 specification.

Modifications to ACPI interfaces to support x2APIC are described in Appendix A, "ACPI Extensions for x2APIC Support".

1.3 GLOSSARY

This document uses the terms listed in the following table.

Table 1-1. Description of terminology

Term	Description
APIC	The set of advanced programmable interrupt controller features which may be implemented in a stand-alone controller, part of a system chipset, or in a microprocessor.
local APIC	The processor component that implements the APIC functionalities. The underlying APIC registers their functionalities are documented in Chapter 8 of "Intel® 64 and IA-32 Architectures Software Developer's Manual", Vol. 3B. Historically, this may refer narrowly to early generations of processor component in the Pentium and P6 processors. In this document, we also use this term generically across multiple generations of processor components.
I/O APIC	The system chipset component that implements APIC functionalities to communicate with a local APIC.
xAPIC	The extension of the APIC architecture that includes messaged APIC interface over the system bus and expanding processor physical addressability from 4 bits to 8 bits.
local xAPIC	The processor component that implements the associated xAPIC functionalities. This is supported by Intel® Pentium® 4 processors, Pentium® M processors, Intel® Core™ 2 Duo processors, and Intel® Xeon® processors based on Intel® NetBurst microarchitecture and Intel® Core™ microarchitecture.
x2APIC	The extension of xAPIC architecture to support 32 bit addressability of processors and associated enhancements.
local x2APIC	The processor component that implements the associated x2APIC functionalities.
xAPIC mode	The operating mode of a local xAPIC unit when it is enabled, or that of a local x2APIC unit when it is enabled but not in extended mode.
x2APIC mode	The operating mode of a local x2APIC unit when it is enabled and in extended mode.

Table 1-1. Description of terminology

Term	Description
APIC ID	A unique ID that can identify individual agent in a platform (or clustered configuration). The maximum bit-width supported is 8 bit, versus 32 bits in x2APIC.
local xAPIC ID	The value configured in the local APIC ID register in xAPIC mode. This is an 8-bit value for xAPIC, and x2APIC in xAPIC mode. Because this is used to specify a target destination in physical delivery mode, it is also referred to as physical xAPIC ID. The processor initializes local xAPIC ID.
physical xAPIC ID	See **"local xAPIC ID"**.
logical xAPIC ID	The APIC ID value that specifies a target processor to receive interrupt delivered in logical destination mode in a local xAPIC. See documentation on logical destination register (LDR) in Section 2.4.2. This is an 8-bit value. Logical xAPIC ID is not initialized by hardware.
initial APIC ID	The value reported by CPUID.01H:EBX[31:24]. Initial APIC ID is initialized by hardware.
x2APIC ID	The 32-bit value in the local APIC ID register defined by the x2APIC architecture. The value is initialized by hardware and can be accessed via RDMSR in x2APIC mode. It is also reported by CPUID.0BH:EDX. Application can query CPUID.0BH:EDX in user mode without RDMSR.
logical x2APIC ID	The APIC ID value that specifies a target processor to receive interrupt delivered in logical destination mode in a local x2APIC. This is a 32-bit value initialized by hardware.
RsvdZ	Reads of reserved bits return zero

1.4 REFERENCES

- Intel® 64 and IA-32 Architectures Software Developer's Manual (in five volumes)
  http://developer.intel.com/products/processor/manuals/index.htm
- Intel Virtualization Technology for Directed I/O, Rev 1.1 specification
  http://download.intel.com/technology/computing/vptech/Intel(r)_VT_for_Direct_IO.pdf
- Detecting Multi-Core Processor Topology in an IA-32 Platform
  http://www3.intel.com/cd/ids/developer/asmo-na/eng/recent/275339.htm

This page intentionally left blank

CHAPTER 2
LOCAL x2APIC ARCHITECTURE

2.1 x2APIC ENHANCEMENTS

The key enhancements provided by the x2APIC architecture over xAPIC are the following:

- Support for two modes of operation to provide backward compatibility and extensibility for future platform innovations:
  - In xAPIC compatibility mode, APIC registers are accessed through memory mapped interface to a 4K-Byte page, identical to the xAPIC architecture.
  - In x2APIC mode, APIC registers are accessed through Model Specific Register (MSR) interfaces. In this mode, the x2APIC architecture provides significantly increased processor addressability and some enhancements on interrupt delivery.
- Increased range of processor addressability in x2APIC mode:
  - Physical xAPIC ID field increases from 8 bits to 32 bits, allowing for interrupt processor addressability up to 4G-1 processors in physical destination mode. A processor implementation of x2APIC architecture can support fewer than 32-bits in a software transparent fashion.
  - Logical xAPIC ID field increases from 8 bits to 32 bits. The 32-bit logical x2APIC ID is partitioned into two sub-fields: a 16-bit cluster ID and a 16-bit logical ID within the cluster. Consequently, ((2^20) -16) processors can be addressed in logical destination mode. Processor implementations can support fewer than 16 bits in the cluster ID sub-field and logical ID sub-field in a software agnostic fashion.
- More efficient MSR interface to access APIC registers.
  - To enhance inter-processor and self directed interrupt delivery as well as the ability to virtualize the local APIC, the APIC register set can be accessed only through MSR based interfaces in the x2APIC mode. The Memory Mapped IO (MMIO) interface used by xAPIC is not supported in the x2APIC mode.
- The semantics for accessing APIC registers have been revised to simplify the programming of frequently-used APIC registers by system software. Specifically the software semantics for using the Interrupt Command Register (ICR) and End Of Interrupt (EOI) registers have been modified to allow for more efficient delivery and dispatching of interrupts.

The x2APIC extensions are made available to system software by enabling the local x2APIC unit in the "x2APIC" mode. The rest of this chapter provides details for detecting, enabling and programming features of x2APIC.

2.2 DETECTING AND ENABLING x2APIC

A processor's support to operate its local APIC in the x2APIC mode can be detected by querying the extended feature flag information reported by CPUID. When CPUID is executed with EAX = 1, the returned value in ECX[Bit 21] indicates processor's support for the x2APIC mode. If CPUID.(EAX=01H):ECX[Bit 21] is set, then the local APIC in the processor supports the x2APIC capability and can be placed into the x2APIC mode. This bit is set only when the x2APIC hardware is present.

- System software can place the local APIC in the x2APIC mode by setting the x2APIC mode enable bit (bit 10) in the IA32_APIC_BASE MSR at MSR address 01BH. The layout for the IA32_APIC_BASE MSR is shown in Figure 2-1.

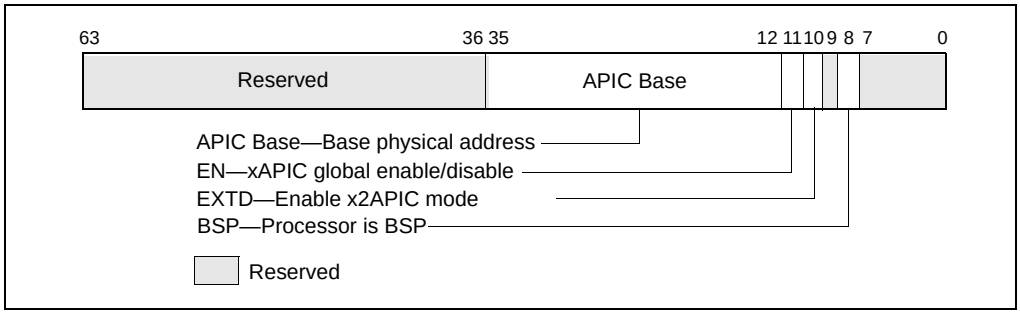

Figure 2-1. IA32_APIC_BASE MSR Supporting x2APIC

Table 2-1, "x2APIC operating mode configurations" describe the possible combinations of the enable bit (EN - bit 11) and the extended mode bit (EXTD - bit 10) in the IA32_APIC_BASE MSR.

Table 2-1. x2APIC Operating Mode Configurations

xAPIC global enable (IA32_APIC_BASE[11])	x2APIC enable (IA32_APIC_BASE[10])	Description
0	0	local APIC is disabled
0	1	Invalid
1	0	local APIC is enabled in xAPIC mode
1	1	local APIC is enabled in x2APIC mode

Once the local APIC has been switched to x2APIC mode (EN = 1, EXTD = 1), switching back to xAPIC mode would require system software to disable the local APIC unit. Specifically, attempting to write a value to the IA32_APIC_BASE MSR that has (EN= 1, EXTD = 0) when the local APIC is enabled and in x2APIC mode will raise a GP exception. Once bit 10 in IA32_APIC_BASE MSR is set, the only way to leave x2APIC mode using IA32_APIC_BASE would require a WRMSR to set both bit 11 and

bit 10 to zero. Section 2.7, “x2APIC STATE TRANSITIONS” provides a detailed state diagram for the state transitions allowed for the local APIC.

2.3 x2APIC MODE REGISTER INTERFACE

In xAPIC mode, the software model for accessing the APIC registers is through a memory mapped interface. Specifically, the APIC registers are mapped to a 4K-Byte region in the processor's memory address space, the physical address base of the 4K-Byte region is specified in the IA32_APIC_BASE MSR (Default value of FEE0_0000H).

In x2APIC mode, a block of MSR address range is reserved for accessing APIC registers through the processor’s MSR address space. This section provides details of this MSR based interface.

2.3.1 Instructions to Access APIC Registers

In x2APIC mode, system software uses RDMSR and WRMSR to access the APIC registers. The MSR addresses for accessing the x2APIC registers are architecturally defined and specified in Section 2.3.2, “APIC Register Address Space”. Executing the RDMSR instruction with APIC register address specified in ECX returns the content of bits 0 through 31 of the APIC registers in EAX. Bits 32 through 63 are returned in register EDX - these bits are reserved if the APIC register being read is a 32-bit register. Similarly executing the WRMSR instruction with the APIC register address in ECX, writes bits 0 to 31 of register EAX to bits 0 to 31 of the specified APIC register. If the register is a 64-bit register then bits 0 to 31 of register EDX are written to bits 32 to 63 of the APIC register. The Interrupt Command Register is the only APIC register that is implemented as a 64-bit MSR. The semantics of handling reserved bits are defined in Section 2.3.3, “Reserved Bit Checking”.

2.3.2 APIC Register Address Space

The MSR address range between 0000_0800H through 0000_0BFFH is architecturally reserved and dedicated for accessing APIC registers in x2APIC mode. Figure 2-2 provides the detailed list of the APIC registers in xAPIC mode and x2APIC mode. The MSR address offset specified in the table is relative to the base MSR address of 800H. The MMIO offset specified in the table is relative to the default base address of FEE00000H.

There is a one-to-one mapping between the legacy xAPIC register MMIO offset and the MSR address offset with the following exceptions:

- The Interrupt Command Register (ICR): The two 32-bit ICR registers in xAPIC mode are merged into a single 64-bit MSR in x2APIC mode.
- The Destination Format Register (DFR) is not supported in x2APIC mode.

- The SELF IPI register is available only if x2APIC mode is enabled.

The MSR address space is compressed to allow for future growth. Every 32 bit register on a 128- bit boundary in the legacy MMIO space is mapped to a single MSR in the local x2APIC MSR address space. The upper 32-bits of all x2APIC MSRs (except for the ICR) are reserved.

Table 2-2. Local APIC Register Address Map Supported by x2APIC

MMIO Offset (xAPIC mode)	MSR Offset (x2APIC mode)	Register Name	R/W Semantics	Comments
0000H-0010H	000H-001H	Reserved		
0020H	002H	Local APIC ID Register	Read only	See Section 2.7.1 for initial values.
0030H	003H	Local APIC Version Register	Read only.	Same version between extended and legacy modes. Bit 24 is available only to an x2APIC unit (in xAPIC mode and x2APIC modes, See Section 2.5.1).
0040H-0070H	004H-007H	Reserved		
0080H	008H	Task Priority Register (TPR)	Read/Write.	Bits 7:0 are RW. Bits 31:8 are Reserved.
0090H	009H	Reserved		
00A0H	00AH	Processor Priority Register (PPR)	Read only.	
00B0H	00BH	EOI Register	Write only.	0 is the only valid value to write. GP fault on non-zero write
00C0H	00CH	Reserved		
00D0H	00DH	Logical Destination Register	Read only.	Read/Write in xAPIC mode)
00E0H	00EH	Reserved[1]		GP fault on Read Write in x2APIC mode.
00F0H	00FH	Spurious Interrupt Vector Register	Read/Write.	Bits 0-8, 12 Read/Write; other bits reserved.
0100H	010H	In-Service Register (ISR); bits 0:31	Read Only.	
0110H	011H	ISR bits 32:63	Read Only.	

Table 2-2. Local APIC Register Address Map Supported by x2APIC (Contd.)

MMIO Offset (xAPIC mode)	MSR Offset (x2APIC mode)	Register Name	R/W Semantics	Comments
0120H	012H	ISR bits 64:95	Read Only.	
0130H	013H	ISR bits 96:127	Read Only.	
0140H	014H	ISR bits 128:159	Read Only.	
0150H	015H	ISR bits 160:191	Read Only.	
0160H	016H	ISR bits 192:223	Read Only.	
0170H	017H	ISR bits 224:255	Read Only.	
0180H	018H	Trigger Mode Register (TMR); bits 0:31	Read Only.	
0190H	019H	TMR bits 32:63	Read Only.	
01A0H	01AH	TMR bits 64:95	Read Only.	
01B0H	01BH	TMR bits 96:127	Read Only.	
01C0H	01CH	TMR bits 128:159	Read Only.	
01D0H	01DH	TMR bits 160:191	Read Only.	
01E0H	01EH	TMR bits 192:223	Read Only.	
01F0H	01FH	TMR bits 224:255	Read Only.	
0200H	020H	Interrupt Request Register (IRR); bits 0:31	Read Only.	
0210H	021H	IRR bits32:63	Read Only.	
0220H	022H	IRR bits 64:95	Read Only.	
0230H	023H	IRR bits 96:127	Read Only.	
0240H	024H	IRR bits 128:159	Read Only.	
0250H	025H	IRR bits 160:191	Read Only.	
0260H	026H	IRR bits 192:223	Read Only.	
0270H	027H	IRR bits 224:255	Read Only.	
0280H	028H	Error Status Register	Read/Write.	GP fault on non-zero writes
0290H-02E0H	029H-02EH	Reserved		
02F0H	02FH	Reserved		
0300H-0310H[2]	030H[3]	Interrupt Command Register (ICR); bits 0-63	Read/Write.	

Table 2-2. Local APIC Register Address Map Supported by x2APIC (Contd.)

MMIO Offset (xAPIC mode)	MSR Offset (x2APIC mode)	Register Name	R/W Semantics	Comments
0320H	032H	LVT Timer Register	Read/Write.	
0330H	033H	LVT Thermal Sensor Register	Read/Write.	
0340H	034H	LVT Performance Monitoring Register	Read/Write.	
0350H	035H	LVT LINT0 Register	Read/Write.	
0360H	036H	LVT LINT1 Register	Read/Write.	
0370H	037H	LVT Error Register	Read/Write.	
0380H	038H	Initial Count Register (for Timer)	Read/Write.	
0390H	039H	Current Count Register (for Timer)	Read Only.	
03A0H-03D0H	03AH-03DH	Reserved		
03E0H	03EH	Divide Configuration Register (for Timer)	Read/Write.	
Not supported	03FH	SELF IPI[4]	Write only	Only in x2APIC mode
	040H-3FFH	Reserved		

NOTES:

1. Destination format register (DFR) is supported in xAPIC mode at MMIO offset 00E0H.
2. APIC register at MMIO offset 0310H is accessible in xAPIC mode only
3. MSR 831H (offset 31H) is reserved; read/write operations will result in a GP fault.
4. SELF IPI register is supported only if x2APIC mode is enabled.

2.3.3 Reserved Bit Checking

Section 2.3.2 and Table 2-2 specifies the reserved bit definitions for the APIC registers in x2APIC mode. Non-zero writes (by WRMSR instruction) to reserved bits to these registers will raise a general protection fault exception while reads return zeros (RsvdZ semantics).

2.3.4 Error Handling

RDMSR and WRMSR operations to reserved addresses in the x2APIC mode will raise a GP fault. (Note: In xAPIC mode, an APIC error is indicated in the Error Status Register on an illegal register access.) Additionally reserved bit violations cause GP faults as detailed in Section 2.3.3. Beyond illegal register access and reserved bit violations, other APIC errors are logged in Error Status Register. The details on Error Status Register are in Section 2.3.5.4.

2.3.5 MSR Access Semantics

To allow for efficient access to the APIC registers in x2APIC mode, the serializing semantics of WRMSR are relaxed when writing to the APIC registers. Thus, system software should not use “WRMSR to APIC registers in x2APIC mode” as a serializing instruction. Read and write accesses to the APIC registers will occur in program order.

Additional semantics for the WRMSR instruction expected by system software for specific registers (EOI, TPR, SELF IPI) are described in Section 2.3.5.3, Section 2.3.5.2, and Section 2.4.5.

The RDMSR instruction is not serializing and this behavior is unchanged when reading APIC registers in x2APIC mode. System software accessing the APIC registers using the RDMSR instruction should not expect a serializing behavior. (Note: The MMIO-based xAPIC interface is mapped by system software as an un-cached region. Consequently, read/writes to the xAPIC-MMIO interface have serializing semantics in the xAPIC mode.)

There are some simplifications to the means used by system software for accessing the Interrupt Control Register via the register interface in the x2APIC mode. These changes are described in Section 2.3.5.1.

2.3.5.1 Interrupt Command Register Semantics

A processor generates an inter-processor interrupt (IPI) by writing to the Interrupt Command Register (ICR) in the local xAPIC unit. In xAPIC mode, ICR contains a delivery status bit (bit 12) that indicates the status of the delivery of this interrupt. The field has software read-only semantics. A value of 0 implies that there is currently no activity while a value of 1 implies that the transmission is pending. The delivery status bit gets cleared when the interrupt has been transmitted. With the legacy xAPIC interface, system software would poll the delivery status bit until it is clear prior to sending an IPI. Similarly if the semantics of the send operation required that the interrupt be sent from the local xAPIC unit, then system software would busy-wait for the delivery status bit to be cleared.

In the x2APIC mode, the semantics of programming Interrupt Command Register to dispatch an interrupt is simplified. A single MSR write to the 64-bit ICR (see Figure 2-5) is required for dispatching an interrupt.

Other semantics change related to reading/writing the ICR in x2APIC mode vs. xAPIC mode are:

- Completion of the WRMSR instruction to the ICR does not guarantee that the interrupt to be dispatched has been received by the targeted processors. If the system software usage requires this guarantee, then the system software should explicitly confirm the delivery of the interrupt to the specified targets using an alternate software mechanisms. For example, one possible mechanism would be having the interrupt service routine associated with the target interrupt delivery to update a memory location, thereby allowing the dispatching software to verify the memory location has been updated.
- A destination ID value of FFFF_FFFFH is used for broadcast of interrupts in both logical destination and physical destination modes.
- The Delivery Status bit of the ICR has been removed. Software need not poll on the Delivery Status bit before writing the ICR.
- ICR reads are still allowed to aid debugging. However software should not assume the value returned by reading the ICR is the last written value.

2.3.5.2 Task Priority Register Semantics

In x2APIC mode, the layout of the Task Priority Register has the same layout as in the xAPIC mode.

The semantics for reading and writing to the TPR register via the MSR interface are identical to those used for TPR access via the CR8 register. Specifically, the write to the TPR register ensures that the result of any re-prioritization action due to the change in processor priority is reflected to the processor prior to the next instruction following the TPR write. Any deliverable interrupts resulting from the TPR write would be taken at the instruction boundary following the TPR write.

2.3.5.3 End Of Interrupt Register Semantics

In xAPIC mode, the EOI register is written by an interrupt service routine to indicate that the current interrupt service has completed. System software performs a write to the EOI register to signal an EOI.

In the x2APIC mode, the write of a zero value to EOI register is enforced. Writes of a non-zero value to the EOI register in x2APIC mode will raise a GP fault. System software continues to have to perform the EOI write to indicate interrupt service completion. But in x2APIC mode, the EOI write is with a value of zero.

2.3.5.4 Error Status Register Semantics

The Error Status register (ESR) records all errors detected by the local APIC. In xAPIC mode, software can read/write to the ESR. In the x2APIC mode, the write of a zero value is enforced. Software writes zero's to the ESR to clear the error status.

Writes of a non-zero value to the Error Status Register in x2APIC mode will raise a GP fault.

The layout of ESR is shown in Figure 2-2. In x2APIC mode, a RDMSR or WRMSR to an illegal register address raises a GP fault. In xAPIC mode, the equivalent MMIO accesses would have generated an APIC error. So in the x2APIC mode, the Illegal Register Address field in the Error Status register will not have any errors logged.

Write to the ICR (in xAPIC and x2APIC modes) or to SELF IPI register (x2APIC mode only) with an illegal vector (vector <= 0FH) will set the "Send Illegal Vector" bit. On receiving an IPI with an illegal vector (vector <= 0FH), the "Receive Illegal Vector" bit will be set. On receiving an interrupt with illegal vector in the range 0H – 0FH, the interrupt will not be delivered to the processor nor will an IRR bit be set in that range. Only the ESR “Receive Illegal Vector” bit will be set.

If the ICR is programmed with lowest priority delivery mode then the "Re-directible IPI" bit will be set in x2APIC modes (same as legacy xAPIC behavior) and the interrupt will not be processed.

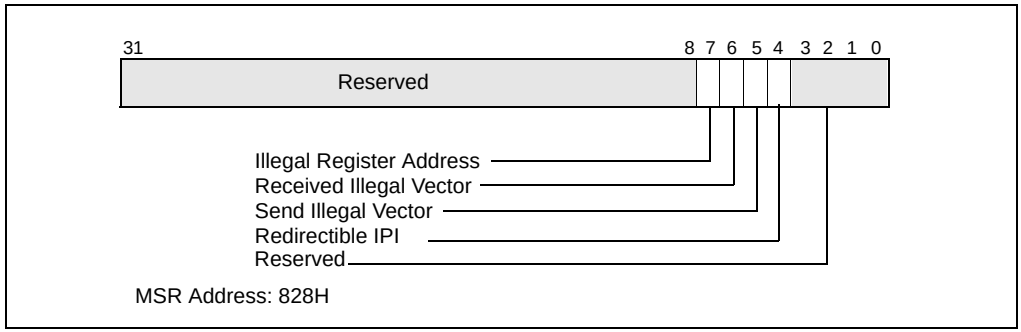

Figure 2-2. Error Status Register (ESR)

2.3.6 x2APIC Register Availability

The local APIC registers can be accessed via the MSR interface only when the local x2APIC has been switched to the x2APIC mode as described in Section 2.2. Accessing any APIC register in the MSR address range 0800H through 0BFFH via RDMSR or WRMSR when the local APIC is not in x2APIC mode will cause the instructions to raise a GP fault. In x2APIC mode, the memory mapped interface is not available and any access to the MMIO interface will behave similar to that of a legacy xAPIC in globally disabled state. Table 2-3 provides the interactions between the legacy & extended modes and the legacy and register interfaces.

Table 2-3. MSR/MMIO Interface of a Local x2APIC in Different Modes of Operation

	MMIO Interface	MSR Interface
xAPIC mode	Available	GP Fault
x2APIC mode	Behavior identical to xAPIC in globally disabled state	Available

2.3.7 VM-exit Controls for MSRs and x2APIC Registers

The VMX architecture allows a VMM to specify lists of MSRs to be loaded or stored on VMX transitions using the VMX-transition MSR areas (see VM-exit MSR-store address field, VM-exit MSR-load address filed, and VM-entry MSR-load address field in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*).

The X2APIC MSRs cannot to be loaded and stored on VMX transitions. A VMX transition fails if the VMM has specified that the transition should access any MSRs in the address range from 0000_0800H to 0000_08FFH (the range used for accessing the X2APIC registers). Specifically, processing of an 128-bit entry in any of the VMX-transition MSR areas fails if bits 31:0 of that entry (represented as ENTRY_LOW_DW) satisfies the expression: "ENTRY_LOW_DW & FFFFF800H = 00000800H". Such a failure causes an associated VM entry to fail (by reloading host state) and causes an associated VM exit to lead to VMX abort.

2.4 EXTENDED PROCESSOR ADDRESSABILITY

This section provides details on extensions to the physical xAPIC ID and the logical xAPIC ID to support extended processor addressability.

The x2APIC architecture also provides two destination modes - physical destination mode and logical destination mode. Each logical processor in the system has a unique physical xAPIC ID which is used for targeting interrupts to that processor in physical destination mode. The local APIC ID register provides the physical destination mode 8-bit or 32-bit ID for the processor, depending on xAPIC mode or x2APIC mode. Section 2.4.1 describes the 32-bit x2APIC ID in x2APIC mode.

Each logical processor in the system also can have a unique logical xAPIC ID which is used for targeting interrupts to that processor in logical destination mode. The Logical Destination Register specified in Section 2.4.2. It contains the logical x2APIC ID for the processor in x2APIC mode.

2.4.1 Local APIC ID Register

In x2APIC mode, the local APIC ID register is increased to 32 bits wide. This enables 2^32 -1 processors to be addressable in physical destination mode. This 32-bit value is referred to as "x2APIC ID". A processor implementation may choose to support less

than 32 bits in its hardware. System software should be agnostic to the actual number of bits that are implemented. All non-implemented bits will return zeros on reads by software.

The APIC ID value of FFFF_FFFFH and the highest value corresponding to the implemented bit-width of the local APIC ID register in the system are reserved and cannot be assigned to any logical processor.

In x2APIC mode, the local APIC ID register is a read-only register to system software and will be initialized by hardware. It is accessed via the RDMSR instruction reading the MSR at address 0802H. Figure 2-3 provides the layout of the Local x2APIC ID register.

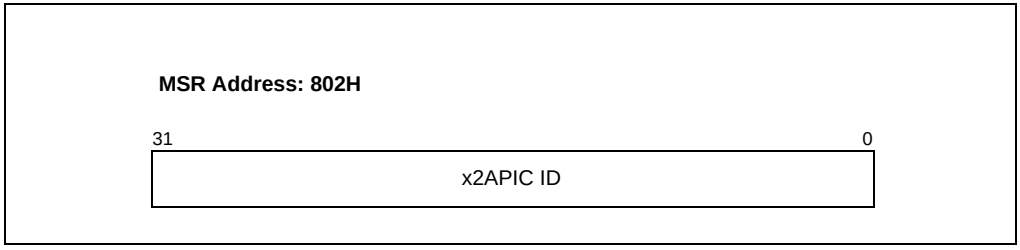

Figure 2-3. Local APIC ID Register in x2APIC Mode

Each logical processor in the system (including clusters with a communication fabric) must be configured with an unique x2APIC ID to avoid collisions of x2APIC IDs. On DP and high-end MP processors targeted to specific market segments and depending on the system configuration, it is possible that logical processors in different and "unconnected" clusters power up initialized with overlapping x2APIC IDs. In these configurations, a model-specific means may be provided in those product segments to enable BIOS and/or platform firmware to re-configure the x2APIC IDs in some clusters to provide for unique and non-overlapping system wide IDs before configuring the disconnected components into a single system.

2.4.2 Logical Destination Register

In x2APIC mode, the Logical Destination Register (LDR) is increased to 32 bits wide. It is a read-only register to system software. This 32-bit value is referred to as “logical x2APIC ID”. System software accesses this register via the RDMSR instruction reading the MSR at address 80DH. Figure 2-4 provides the layout of the Logical Destination Register in x2APIC mode.

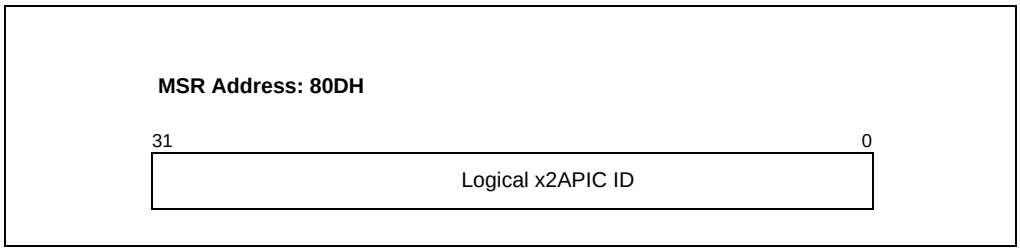

Figure 2-4. Logical Destination Register in x2APIC Mode

In the xAPIC mode, the Destination Format Register (DFR) through MMIO interface determines the choice of a flat logical mode or a clustered logical mode. Flat logical mode is not supported in the x2APIC mode. Hence the Destination Format Register (DFR) is eliminated in x2APIC mode.

The 32-bit logical x2APIC ID field of LDR is partitioned into two sub-fields:

- Cluster ID (LDR[31:16]): is the address of the destination cluster
- Logical ID (LDR[15:0]): defines a logical ID of the individual local x2APIC within the cluster specified by LDR[31:16].

This layout enables 2^16-1 clusters each with up to 16 unique logical IDs - effectively providing an addressability of ((2^20) - 16) processors in logical destination mode.

It is likely that processor implementations may choose to support less than 16 bits of the cluster ID or less than 16-bits of the Logical ID in the Logical Destination Register. However system software should be agnostic to the number of bits implemented in the cluster ID and logical ID sub-fields. The x2APIC hardware initialization will ensure that the appropriately initialized logical x2APIC IDs are available to system software and reads of non-implemented bits return zero. This is a read-only register that software must read to determine the logical x2APIC ID of the processor. Specifically, software can apply a 16-bit mask to the lowest 16 bits of the logical x2APIC ID to identify the logical address of a processor within a cluster without needing to know the number of implemented bits in cluster ID and Logical ID sub-fields. Similarly, software can create a message destination address for cluster model, by bit-Oring the Logical X2APIC ID (31:0) of processors that have matching Cluster ID(31:16).

To enable cluster ID assignment in a fashion that matches the system topology characteristics and to enable efficient routing of logical mode lowest priority device interrupts in link based platform interconnects, the LDR are initialized by hardware based on the value of x2APIC ID upon x2APIC state transitions. Details of this initialization are provided in Section 2.4.4.

2.4.3 Interrupt Command Register

In x2APIC mode, the layout of the Interrupt Command Register is shown in Figure 2-5. The lower 32 bits of ICR in x2APIC mode is identical to the lower half of the ICR in xAPIC mode, except the Delivery Status bit is removed since it is not needed in X2APIC mode. The destination ID field is expanded to 32 bits in x2APIC mode.

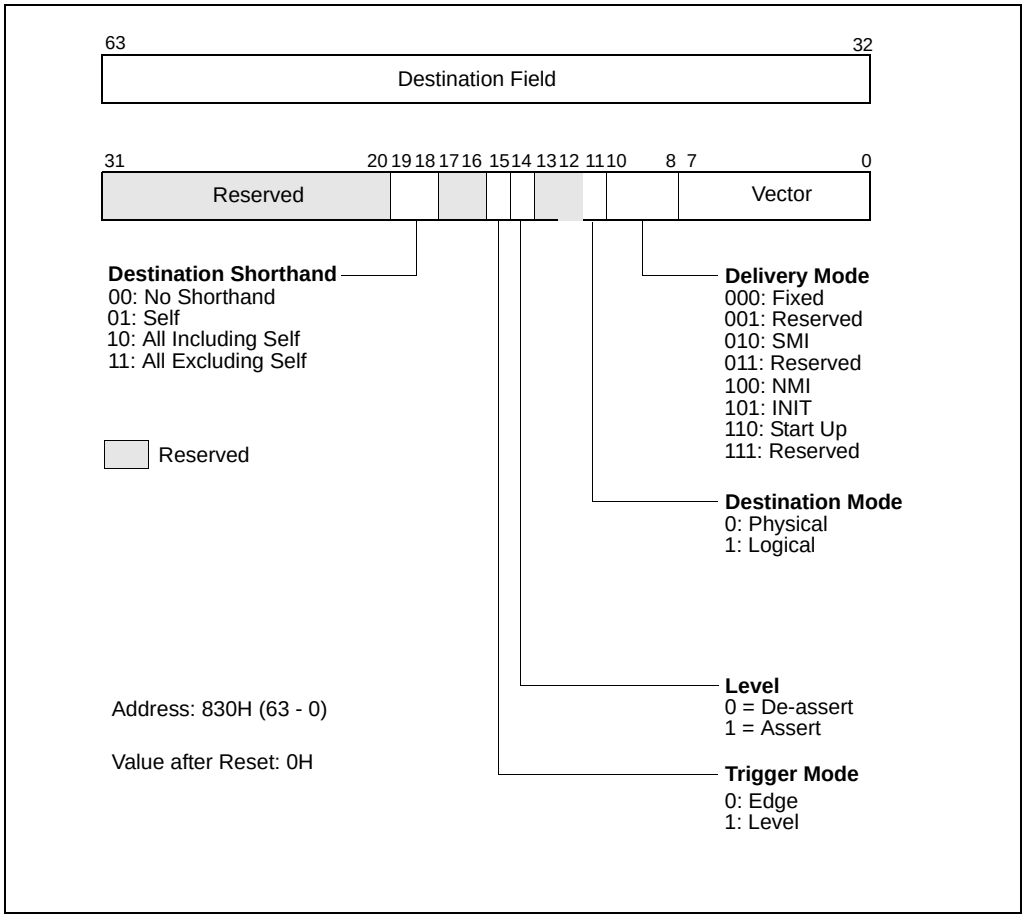

Figure 2-5. Interrupt Command Register (ICR) in x2APIC Mode

A single MSR write to the Interrupt Command Register is required for dispatching an interrupt in x2APIC mode. With the removal of the Delivery Status bit, system software no longer has a reason to read the ICR. It remains readable only to aid in debugging.

A destination ID value of FFFF_FFFFH is used for broadcast of interrupts in both logical destination and physical destination modes.

2.4.4 Deriving Logical x2APIC ID from the Local x2APIC ID

In x2APIC mode, the 32-bit logical x2APIC ID, which can be read from LDR, is derived from the 32-bit local x2APIC ID. Specifically, the 16-bit logical ID sub-field is derived by shifting 1 by the lowest 4 bits of the x2APIC ID, i.e. Logical ID = 1 << x2APIC ID[3:0]. The rest of the bits of the x2APIC ID then form the cluster ID portion of the logical x2APIC ID:

Logical x2APIC ID = [(x2APIC ID[31:4] << 16) | (1 << x2APIC ID[3:0])]

The use of lowest 4 bits in x2APIC ID implies that at least 16 APIC IDs are reserved for logical processors within a socket in multi-socket configurations. If more than 16 APIC IDS are reserved for logical processors in a socket/package then multiple cluster IDs can exist within the package.

The LDR initialization occurs whenever the x2APIC mode is enabled. This is described in Section 2.7.

2.4.5 SELF IPI register

SELF IPIs are used extensively by some system software. The xAPIC architecture provided a mechanism for sending an IPI to the current local APIC using the "self-IPI" short-hand in the interrupt command register (see Figure 2-5). The x2APIC architecture introduces a new register interface. This new register is dedicated to the purpose of sending self-IPIs with the intent of enabling a highly optimized path for sending self-IPIs.

Figure 2-6 provides the layout of the SELF IPI register. System software only specifies the vector associated with the interrupt to be sent. The semantics of sending a self-IPI via the SELF IPI register are identical to sending a self targeted edge triggered fixed interrupt with the specified vector. Specifically the semantics are identical to the following settings for an inter-processor interrupt sent via the ICR - Destination Shorthand (ICR[19:18] = 01 (Self)), Trigger Mode (ICR[15] = 0 (Edge)), Delivery Mode (ICR[10:8] = 000 (Fixed)), Vector (ICR[7:0] = Vector).

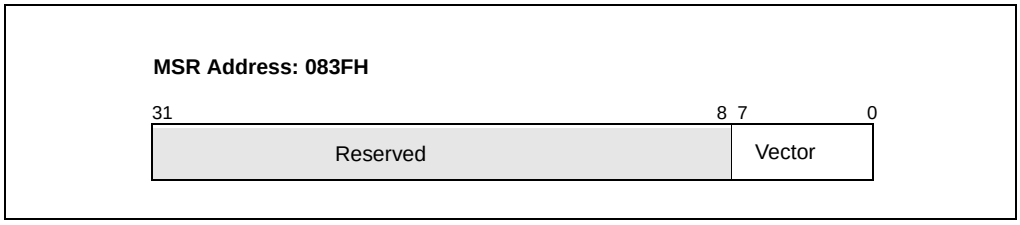

Figure 2-6. SELF IPI register

The SELF IPI register is a write-only register. A RDMSR instruction with address of the SELF IPI register will raise a GP fault.

The handling and prioritization of a self-IPI sent via the SELF IPI register is architecturally identical to that for an IPI sent via the ICR from a legacy xAPIC unit. Specifically the state of the interrupt would be tracked via the Interrupt Request Register (IRR) and In Service Register (ISR) and Trigger Mode Register (TMR) as if it were received from the system bus. Also sending the IPI via the Self Interrupt Register ensures that interrupt is delivered to the processor core. Specifically completion of the WRMSR instruction to the SELF IPI register implies that the interrupt has been logged into the IRR. As expected for edge triggered interrupts, depending on the processor priority and readiness to accept interrupts, it is possible that interrupts sent via the SELF IPI register or via the ICR with identical vectors can be combined.

2.5 x2APIC ENHANCEMENTS TO LEGACY xAPIC ARCHITECTURE

The x2APIC architecture also provides enhanced features for a local x2APIC unit operating in xAPIC mode. This section describes x2APIC enhancements that are common to xAPIC mode and x2APIC mode.

2.5.1 Directed EOI

To support level triggered interrupts, the legacy xAPIC architecture broadcasts EOI messages for level triggered interrupts over the system interconnect to all the IOxAPICs in the system indicating that the interrupt has been serviced. Broadcasting the EOIs can lead to system inefficiencies on a link-based system interconnect. Also, in systems with multiple IOxAPICs, where different IOxAPICs have been programmed with the same vector but different processor destinations, the broadcasting of the EOI message can lead to duplicate interrupts being delivered to the local xAPIC for the same event on an IO device.

Directed EOI capability is intended to enable system software to perform directed EOIs to specific IOxAPICs in the system. System software desiring to perform a directed EOI would do the following:

- inhibit the broadcast of EOI message by setting bit 12 of the Spurious Interrupt Vector Register, and
- following the EOI to the local x2APIC unit for a level triggered interrupt, perform a directed EOI to the IOxAPIC generating the interrupt by writing to its EOI register.

Supporting directed EOI capability would require system software to retain a mapping associating level triggered interrupts with IOxAPICs in the system.

Bit 12 of the Spurious Interrupt Vector Register (SVR) in the local x2APIC unit controls the generation of the EOI broadcast if the Directed EOI capability is

supported. This bit is reserved to 0 if the processor doesn't support Directed EOI. If SVR[bit 12] is set, a broadcast EOI is not generated on an EOI cycle even if the associated TMR bit is indicating the current interrupt is a level triggered interrupt. Layout of the Spurious Interrupt Vector Register is shown in Figure 2-7.

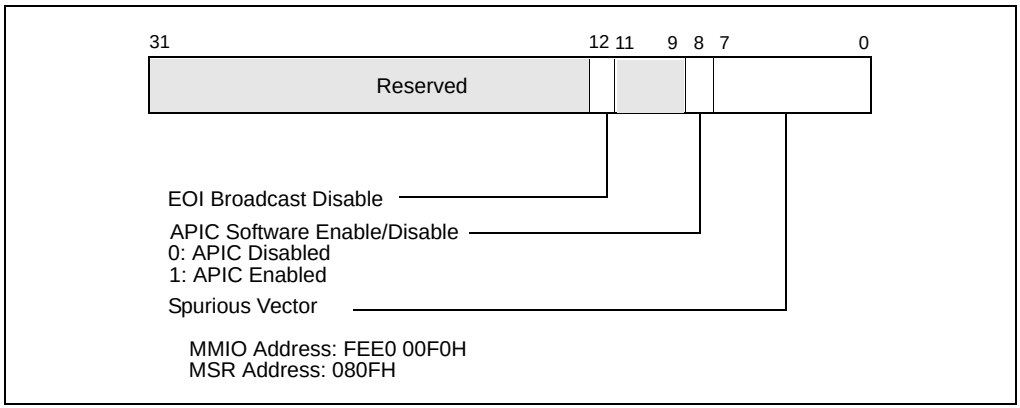

Figure 2-7. Spurious Interrupt Vector Register (SVR) of x2APIC

The default value for SVR[bit 12] is clear, indicating that an EOI broadcast will be performed.

The support for Directed EOI capability can be detected by means of bit 24 in the Local APIC Version Register. This feature is supported in both the xAPIC mode and x2APIC modes of a local x2APIC unit. Layout of the Local APIC Version register is as shown in Figure 2-8. The Directed EOI feature is supported if bit 24 is set to 1.

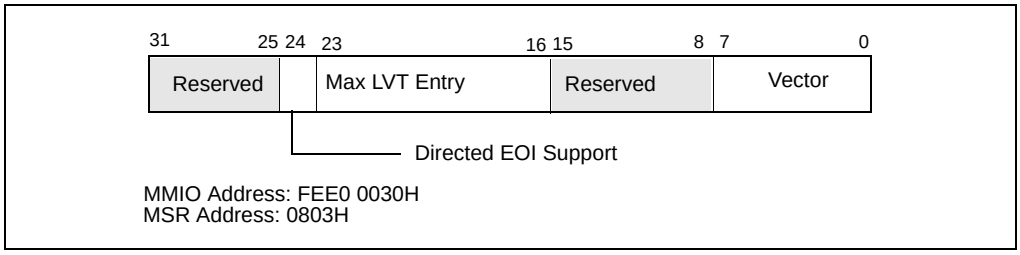

Figure 2-8. Local APIC Version Register of x2APIC

2.6 INTERACTION WITH PROCESSOR CORE OPERATING MODES

Similar to the xAPIC architecture, the APIC registers defined in the x2APIC architecture are accessible in the following operating modes of the processor: Protected

Mode, Virtual-8086 Mode, Real Mode, and IA-32e mode (both 64-bit and compatibility sub-modes).

2.7 x2APIC STATE TRANSITIONS

This section provides a detailed description of the x2APIC states of a local x2APIC unit, transitions between these states as well as interactions of these states with INIT and RESET.

2.7.1 x2APIC States

The valid states for a local x2APIC unit is listed in Table 2-1:

- APIC disabled: IA32_APIC_BASE[EN]=0 and IA32_APIC_BASE[EXTD]=0
- xAPIC mode: IA32_APIC_BASE[EN]=1 and IA32_APIC_BASE[EXTD]=0
- x2APIC mode: IA32_APIC_BASE[EN]=1 and IA32_APIC_BASE[EXTD]=1
- Invalid: IA32_APIC_BASE[EN]=0 and IA32_APIC_BASE[EXTD]=1

The state corresponding to EXTD=1 and EN=0 is not valid and it is not possible to get into this state. Values written to the IA32_APIC_BASE_MSR that attempt a transition from a valid state to this invalid state will cause a GP fault. Figure 2-9 shows the comprehensive state transition diagram for a local x2APIC unit.

On coming out of RESET, the local x2APIC unit is enabled and is in the xAPIC mode: IA32_APIC_BASE[EN]=1 and IA32_APIC_BASE[EXTD]=0. The APIC registers are initialized as:

- The local APIC ID is initialized by hardware with a 32 bit ID (x2APIC ID). The lowest 8 bits of the x2APIC ID is the legacy local xAPIC ID, and is stored in the upper 8 bits of the APIC register for access in xAPIC mode.
- The following APIC registers are reset to all zeros for those fields that are defined in the xAPIC mode:
  - IRR, ISR, TMR, ICR, LDR, TPR, Divide Configuration Register (See Chapter 8 of “Intel® 64 and IA-32 Architectures Software Developer’s Manual“, Vol. 3B for details of individual APIC registers),
  - Timer initial count and timer current count registers,
- The LVT registers are reset to 0s except for the mask bits; these are set to 1s.
- The local APIC version register is not affected.
- The Spurious Interrupt Vector Register is initialized to 000000FFH.
- The DFR (available only in xAPIC mode) is reset to all 1s.
- SELF IPI register is reset to zero.

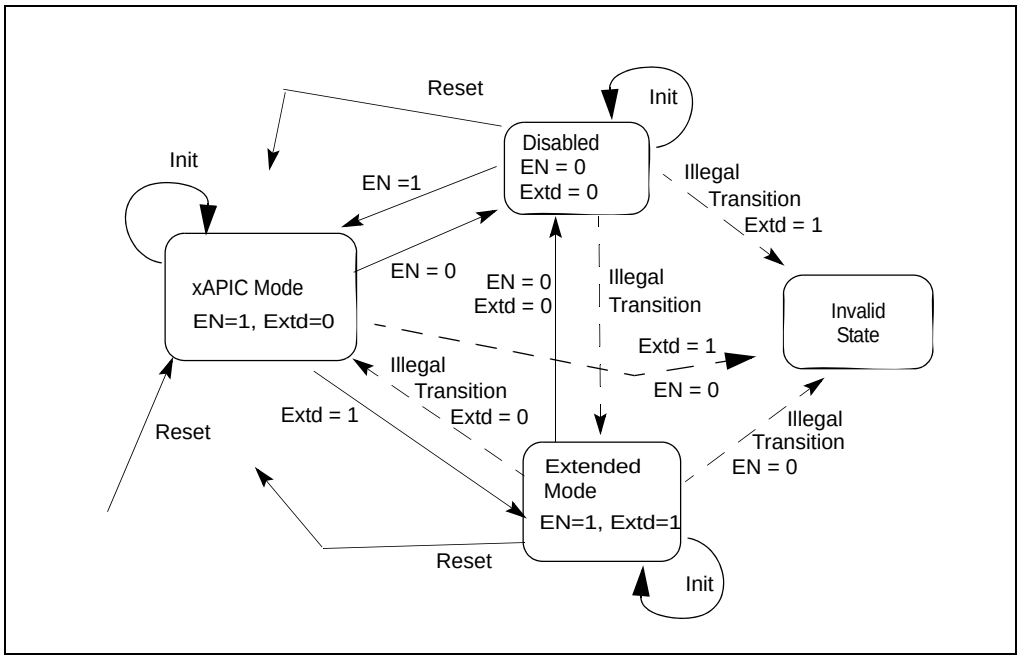

Figure 2-9. Local x2APIC State Transitions with IA32_APIC_BASE, INIT, and RESET

2.7.1.1 x2APIC After RESET

The valid transitions from the xAPIC mode state are:

- to the x2APIC mode by setting EXT to 1 (resulting EN=1, EXTD= 1). The physical x2APIC ID (see Figure 2-3) is preserved across this transition and the logical x2APIC ID (see Figure 2-4) is initialized by hardware during this transition as documented in Section 2.4.4. The state of the extended fields in other APIC registers, which was not initialized at RESET, is not architecturally defined across this transition and system software should explicitly initialize those programmable APIC registers.
- to the disabled state by setting EN to 0 (resulting EN=0, EXTD= 0).

The result of an INIT in the xAPIC state places the x2APIC in the state with EN= 1, EXTD= 0. The state of the local APIC ID register is preserved (the 8-bit xAPIC ID is in the upper 8 bits of the APIC ID register). All the other APIC registers are initialized as a result of INIT.

A RESET in this state places the x2APIC in the state with EN= 1, EXTD= 0. The state of the local APIC ID register is initialized as described in Section 2.7.1. All the other APIC registers are initialized described in Section 2.7.1.

2.7.1.2 x2APIC Transitions From x2APIC Mode

From the x2APIC mode, the only valid x2APIC transition using IA32_APIC_BASE is to the state where the x2APIC is disabled by setting EN to 0 and EXTD to 0. The x2APIC ID (32 bits) and the legacy local xAPIC ID (8 bits) are preserved across this transition. A transition from the x2APIC mode to xAPIC mode is not valid and the corresponding WRMSR to the IA32_APIC_BASE MSR will raise a GP fault.

A RESET in this state places the x2APIC in xAPIC mode. All APIC registers (including the local APIC ID register) are initialized as described in Section 2.7.1.

An INIT in this state keeps the x2APIC in the x2APIC mode. The state of the local APIC ID register is preserved (all 32 bits). However, all the other APIC registers are initialized as a result of the INIT transition.

2.7.1.3 x2APIC Transitions From Disabled Mode

From the disabled state, the only valid x2APIC transition using IA32_APIC_BASE is to the xAPIC mode (EN= 1, EXTD = 0). Thus the only means to transition from x2APIC mode to xAPIC mode is a two-step process:

- first transition from x2APIC mode to local APIC disabled mode (EN= 0, EXTD = 0),
- followed by another transition from disabled mode to xAPIC mode (EN= 1, EXTD= 0).

Consequently, all the APIC register states in the x2APIC, except for the x2APIC ID (32 bits), are not preserved across mode transitions.

A RESET in the disabled state places the x2APIC in the xAPIC mode. All APIC registers (including the local APIC ID register) are initialized as described in Section 2.7.1.

An INIT in the disabled state keeps the x2APIC in the disabled state.

2.7.1.4 State Changes From xAPIC Mode to x2APIC Mode

After APIC register states have been initialized by software in xAPIC mode, a transition from xAPIC mode to x2APIC mode does not affect most of the APIC register states, except the following:

- The Logical Destination Register is not preserved.
- Any APIC ID value written to the memory-mapped local APIC ID register is not preserved.
- The high half of the Interrupt Command Register is not preserved.

2.8 CPUID EXTENSIONS AND TOPOLOGY ENUMERATION

For Intel 64 and IA-32 processors that support x2APIC, the CPUID instruction provides additional mechanism for identifying processor topology information.

Specifically, a value of 1 reported by CPUID.01H:ECX[21] indicates that the processor supports x2APIC and the extended topology enumeration leaf (CPUID.0BH).

The extended topology enumeration leaf can be accessed by executing CPUID with EAX = 0BH. Software can detect the availability of the extended topology enumeration leaf (0BH) by performing two steps:

- Check maximum input value for basic CPUID information by executing CPUID with EAX= 0. If CPUID.0H:EAX is greater than or equal or 11 (0BH), then proceed to next step
- Check CPUID.EAX=0BH, ECX=0H:EBX is non-zero.

If both of the above conditions are true, extended topology enumeration leaf is available. The presence of CPUID leaf 0BH in a processor does not guarantee support for x2APIC. If CPUID.EAX=0BH, ECX=0H:EBX returns zero and maximum input value for basic CPUID information is greater than 0BH, then CPUID.0BH leaf is not supported on that processor.

The extended topology enumeration leaf is intended to assist software with enumerating processor topology on systems that requires 32-bit x2APIC IDs to address individual logical processors. For example, a system with greater than 256 logical processors or greater than 64 processor cores will require the OS to use 32-bit x2APIC IDs.

The basic concept of processor topology enumeration using 8-bit initial APIC ID (CPUID.01H:EBX[31:24]) on legacy systems is similar to using 32-bit x2APIC ID reported by CPUID leaf 0BH: apply appropriate bit masks on unique IDs to sort out levels of topology in a system.

Legacy processor enumeration algorithm is based on examining the initial APIC IDs and additional information from CPUID leaves 01H and 04H to infer system-wide processor topology. The relevant information in CPUID leaves 01H and 04H do not directly map to individual levels of the topology, but merely relate to the sharing characteristics below different levels.

The extended topology enumeration leaf of CPUID provides topology information and data that simplify the algorithm to sort out the processor topology within a physical package from a 32-bit x2APIC ID. Each level of the processor topology is enumerated by specifying a “level number“ in ECX as input when executing CPUID.EAX=0BH. This enumeration by level number allows the CPUID.0BH leaf to support more sophisticated topology than the limitation of legacy topology definitions (SMT, core, package).

The bit fields reported by CPUID.EAX=0BH include the x2APIC ID of the current logical processor (in EDX), an encoded value of hierarchy referred to as “level type“ (in ECX[15:8]), the number of enabled logical processors at each queried level type (below its immediate parent level type), and a bit-vector length field to simplify the parsing of 32-bit x2APIC ID into hierarchical components. The detailed bit field definitions for CPUID.0BH leaf are shown in Table 2-4.

Table 2-4. CPUID Leaf 0BH Information

Initial EAX Value	Information Provided about the Processor	
	CPUID leaves > 3 < 80000000 are visible only when IA32_MISC_ENABLES.BOOT_NT4[bit 22] = 0 (default).	
	Extended Topology Enumeration Leaf	
0BH		**NOTE:** Most fields of leaf 0BH output depends on the initial value in ECX. EDX output do not vary with initial value in ECX. ECX[7:0] output always reflect initial value in ECX. All other output value for an invalid initial value in ECX are 0.
	EAX	Bits 4-0: Number of bits to shift x2APIC ID right to get unique topology ID of next level type*. All logical processors with same next level ID share current level Bits 31-5: Reserved
	EBX	Bits 15-00: Number of enabled logical processors at this level type. The number reflects configuration as shipped by Intel** Bits 31-16: Reserved
	ECX	Bits 07-00: Level number. Same value as input Bits 15-08: Level Type*** Bits 31-16: Reserved.
	EDX	Bits 31-0: x2APIC ID of the current logical processor
		NOTES: * Software should use this field (EAX[4:0]) to enumerate processor topology of the system. ** Software must not use EBX[15:0] to enumerate processor topology of the system. This value in this field (EBX[15:0]) is only intended for display/diagnostic purposes. The actual number of logical processors available to BIOS/OS/Applications may be different from the value of EBX[15:0], depending on software and platform hardware configurations. ***The value of Level Type field is not related to level numbers in any way, higher level type values do not mean higher levels. Level Type field has the following encoding: 0 = Invalid 1 = SMT 2 = Core 3-255 = Reserved

The lowest level number is zero. Level number = 0 is reserved to specify SMT-related topology information (see Hyper-Threading Technology in Section 7.8 of “Intel® 64 and IA-32 Architectures Software Developer’s Manual“, Vol. 3A). If SMT is not present in a processor implementation but CPUID leaf 0BH is supported, CPUID.EAX=0BH, ECX=0 will return EAX = 0, EBX = 1 and level type = 1. Number of logical processors at the core level is reported at level type = 2.

CPUID.0BH leaf can report “level type“ and “level number“ in any order. Each level type defines a specific topology configuration within the physical package. Thus there is no level type corresponding to “package“ for CPUID.0BH leaf. The Level Type encodings indicate the topology level and need not correspond to any Level number except level number 0 is reserved for level type SMT.

The legacy processor topology enumeration fields in CPUID.01H and CPUID.04H will continue to report correct topology up to the maximum values supported by the fields and 8-bit initial APIC ID. For future processors with topology that exceeds the limits of CPUID.01H:EBX[23:16],CPUID.01H:EBX[31:24], CPUID.EAX=04H, ECX=0H:EAX[31:26], these legacy fields will report the respective modulo maximum values.

If CPUID.0BH returns EBX=0 when input ECX=0 then assume that CPUID.0BH leaf data for extended processor topology enumeration is not supported on this processor. Use CPUID.01H and CPUID.04H leaves for topology information.

2.8.1 Consistency of APIC IDs and CPUID

The consistency of physical x2APIC ID in MSR 802H in x2APIC mode and the 32-bit value returned in CPUID.0BH:EDX is facilitated by processor hardware.

CPUID.0BH:EDX will report the full 32 bit ID, in xAPIC and x2APIC mode. This allows BIOS to determine if a system has processors with IDs exceeding the 8-bit initial APIC ID limit (CPUID.01H:EBX[31:24]). Initial APIC ID (CPUID.01H:EBX[31:24]) is always equal to CPUID.0BH:EDX[7:0].

If the values of CPUID.0BH:EDX reported by all logical processors in a system are less than 255, BIOS can transfer control to OS in xAPIC mode.

If the values of CPUID.0BH:EDX reported by some logical processors in a system are greater or equal than 255, BIOS must support two options to hand off to OS:

- If BIOS enables logical processors with x2APIC IDs greater than 255, then it should enable X2APIC in Boot Strap Processor (BSP) and all Application Processors (AP) before passing control to the OS. Application requiring processor topology information must use OS provided services based on x2APIC IDs or CPUID.0BH leaf.
- If a BIOS transfers control to OS in xAPIC mode, then the BIOS must ensure that only logical processors with CPUID.0BH.EDX value less than 255 are enabled. BIOS initialization on all logical processors with CPUID.0B.EDX values greater than or equal to 255 must (a) disable APIC and execute CLI in each logical processor, and (b) leave these logical processor in the lowest power state so that these processors do not respond to INIT IPI during OS boot. The BSP and all the

enabled logical processor operate in xAPIC mode after BIOS passed control to OS. Application requiring processor topology information can use OS provided legacy services based on 8-bit initial APIC IDs or legacy topology information from CPUID.01H and CPUID 04H leaves.

2.9 SYSTEM TRANSITIONS

This section describes implications for the x2APIC across system state transitions - specifically initialization and booting.

The default will be for the BIOS to pass the control to the OS with the local x2APICs in xAPIC mode if all x2APIC IDs reported by CPUID.0BH:EDX are less than 255, and in x2APIC mode if there are any logical processor reporting its x2APIC ID at 255 or greater.

2.10 LEGACY xAPIC CLARIFICATIONS

The x2APIC architecture eliminates/deprecates some of the features provided by the legacy xAPIC and some of the legacy xAPIC features that were not used by prevailing commercial system software. This section provides a list of the features/capabilities that are not supported in the x2APIC architecture.

- Re-directible/Lowest Priority inter-processor interrupts are not supported in the x2APIC architecture.

This page intentionally left blank

APPENDIX A
ACPI EXTENSIONS FOR X2APIC SUPPORT

A.1 ACPI SPECIFICATION CHANGES TO SUPPORT THE X2APIC ARCHITECTURE

The APIC configuration interfaces described in the Advanced Configuration and Power Interface (ACPI) Specification must be augmented to enable operating system support for platforms employing x2APIC architecture-based components. This appendix describes the required changes to sections of the ACPI 3.0b specification that have been approved for incorporation in the next release of the ACPI specification (ACPI 4.0) to be published on the ACPI web site at: http://www.acpi.info

The scope of ACPI interfaces that are covered in this appendix include:

- ACPI's system description tables: The system tables relevant to x2APIC are:
  - Multiple APIC description table (MADT)
  - System Resource Affinity Table (SRAT)
  - ACPI namespace support for x2APIC

This appendix will be removed from this specification when ACPI 4.0 is published.

A.2 MULTIPLE APIC DESCRIPTION TABLE AND X2APIC

The ACPI interrupt model describes all interrupts for the entire system in a uniform interrupt model implementation. Supported interrupt models include the PC-AT-compatible dual 8259 interrupt controller, the Intel Advanced Programmable Interrupt Controller (APIC), and Intel Streamlined Advanced Programmable Interrupt Controller (SAPIC). The APIC interrupt model applies to several APIC architectures, including local APIC, I/O APIC, xAPIC, and x2APIC. The choice of the interrupt model(s) to support is up to the platform designer. The interrupt model cannot be dynamically changed by the system firmware; OS power management (OSPM) will choose which model to use and install support for that model at the time of installation. If a platform supports more than one models, an OS will install support for one model or the other; it will not mix models. Multi-boot capability is a feature in many modern operating systems. This means that a system may have multiple operating systems or multiple instances of an OS installed at any one time. Platform designers must allow for this.

This section describes the format of the ACPI Multiple APIC Description Table (MADT), which provides OSPM with information necessary for operation on systems with APIC (including multiple APIC functionality), or SAPIC implementations.

ACPI represents all interrupts as "flat" values known as global system interrupts. Therefore to support APICs or SAPICs on an ACPI-enabled system, each used APIC or SAPIC interrupt input must be mapped to the global system interrupt value used by ACPI. See Section 5.2.12. “Global System Interrupts" of the ACPI 3.0 specification for a description of Global System Interrupts.

Table A-1 lists the basic layout the MADT. All addresses in the MADT are processor-relative physical addresses.

Table A-1. Multiple APIC Description Table Format

Field	Byte Length	Byte Offset	Description
Header			
Signature	4	0	Contains the signature for the multiple APIC description table: “APIC”
Length	4	4	Length, in bytes, of the entire MADT
Revision	1	8	2
Checksum	1	9	Entire table must sum to zero
OEMID	6	10	OEM ID
OEM Table ID	8	16	For the MADT, Table ID is the manufacturer model ID
OEM Revision	4	24	OEM revision of MADT for supplied OEM Table ID
Creator ID	4	28	Vendor ID of utility that created the table. For tables containing Definition Blocks, this is the ID for the ASL Compiler
Creator Revision	4	32	Revision of utility that created the table. For tables containing Definition Blocks, this is the revision for the ASL Compiler
Local APIC Address	4	36	The 32-bit physical address at which each processor can access its local APIC
Flags	4	40	Multiple APIC flags. See Table 5-18 of ACPI 3.0 specification for a description of this field
APIC Structure[n]	4	44	A list of APIC structures for this implementation. This list will contain all of the I/O APIC, I/O SAPIC, Local APIC, Local SAPIC, Interrupt Source Override, Non-maskable Interrupt Source, Local APIC NMI Source, Local APIC Address Override, and Platform Interrupt Sources structures needed to support this platform. These structures are described in the following sections.

Immediately after the Flags field (byte offset 40) in the MADT at byte offset 44 is the starting address for a list of APIC structures that declare the APIC features of the machine. The first byte of each APIC structure declares the type of that structure and the second byte declares the length of that structure. ACPI structure type 0 through 8 are already defined in ACPI 3.0 specification. Support for x2APIC defines two new APIC structure types. ACPI structure types are listed in Table A-2.

Table A-2. MADT APIC Structure Type Definition

Value	Description
0	Processor Local APIC
1	I/O APIC
2	Interrupt Source Override
3	Non-maskable Interrupt (NMI) Source
4	Local APIC NMI Structure
5	Local APIC Address Override Structure
6	I/O SAPIC
7	Local SAPIC
8	Platform Interrupt Sources
9	Processor x2APIC
10	x2APIC NMI Structure
11-127	Reserved. OSPM skips structures of the reserved type.
128-255	Reserved for OEM use

The layout for ACPI structure corresponding to type 0-8 are listed in ACPI 3.0 specification. The layout for ACPI structure corresponding to type 9 and 10 are described next.

A.2.1 x2APIC Structure

The Processor X2APIC structure (type 9) is very similar to the processor local APIC structure (type 0). When using the X2APIC interrupt model, logical processors with APIC ID values of 255 and greater in the system are required to have a Processor X2APIC record and an ACPI Device object. OSPM does not expect the information provided in this table to be updated if the processor information changes during the lifespan of an OS boot. While in the sleeping state, logical processors are not allowed to be added, removed, nor can their X2APIC ID or x2APIC Flags change. When a logical processor is not present, the Processor X2APIC information is either not reported or flagged as disabled.

All logical processors with APIC ID values of 255 and greater will have their APIC reported through Processor X2APIC structure (type-9 entry type) only. All logical

processors with APIC ID less than 255 will have their APIC reported through Processor Local APIC (type-0 entry type) only. The format of x2APIC structure is listed in Table A-3.

Table A-3. Processor x2APIC Structure Format

Field	Byte Length	Byte Offset	Description
Type	1	0	09H
Length	1	1	Length, in bytes, of the x2APIC structure (16 bytes)
Reserved	2	2	Must be zero
x2APIC ID	4	4	Processor x2APIC ID
Flags	4	8	x2APIC flags. See Table A-4 for a description of this field
ACPI Processor UID	4	12	OSPM associates the X2APIC Structure with a processor object declared in the namespace using the Device statement, when the _UID child object of the processor device evaluates to a numeric value, by matching the numeric value with this field.

Table A-4. x2APIC Structure Flag Field Definition

x2APIC flag field	Bit Length	Bit Offset	Description
Enabled	1	0	If zero, this processor is unusable, and the operating system support will not attempt to use it.
Reserved	31	1	Must be zero

A.2.2 x2APIC NMI Structure

Local APIC or x2APIC NMI structures (type 4 and type 10) describe the interrupt input (LINTn) that NMI is connected to for each of the logical processors in the system where such a connection exists. Each NMI connection to a processor requires a separate NMI structure. This information is needed by OSPM to enable the appropriate APIC entry.

NMI connection to a logical processor with x2APIC ID 255 and greater requires an X2APIC NMI structure (type-10 entry type). NMI connection to a logical processor with x2APIC ID less than 255 require a Local APIC NMI structure (type-4 entry type). For example, if the platform has 8 logical processors with x2APIC ID 0-3 and 256-259 and NMI is connected LINT1 for processor 3, 2, 256 and 257 then two Local APIC NMI entries and two X2APIC NMI entries would be needed in the MADT.

Local NMI structure (type-4 entry type) is to be used to specify global LINTx for all processors if all logical processors have x2APIC ID less than 255. If there are any logical processors with x2APIC ID 255 or greater then an X2APIC NMI structure (type-10 entry type) must be used to specify global LINTx for all logical processors. The format of x2APIC NMI structure is listed in Table A-5.

Table A-5. x2APIC NMI Structure Format

Field	Byte Length	Byte Offset	Description
Type	1	0	0AH
Length	1	1	Length, in bytes, of the x2APIC NMI structure (12 bytes)
Flags	2	2	MPS INTI flags. See Table A-6 for a description of this field
ACPI Processor UID	4	4	UID corresponding to the ID listed in the processor Device object. A value of 0xFFFFFFFF signifies that this applies to all processors in the machine.
x2APIC LINT#	1	8	X2APIC interrupt input LINTn to which NMI is connected
Reserved	3	9	Reserved

Table A-6. MPS INTI Flag Field Definition

MPS INTI field	Bit Length	Bit Offset	Description	
Polarity	2	0	Polarity of the APIC I/O input signals:	
			00B	Conforms to the specifications of the bus (For example, EISA is active-low for level-triggered interrupts)
			01B	Active high
			10B	Reserved
			11B	Active low
Trigger Mode	2	2	Trigger mode of the APIC I/O input signals:	
			00B	Conforms to the specifications of the bus (For example, ISA is edge-triggered)
			01B	Edge-triggered
			10B	Reserved
			11B	Level-triggered
Reserved	12	4	1	Must be zero

A.3 SYSTEM RESOURCE AFFINITY TABLE (SRAT)

This optional table provides information that allows OSPM to associate processors and memory ranges, including ranges of memory provided by hot-added memory devices, with system localities / proximity domains. On NUMA platforms, SRAT information enables OSPM to optimally configure the operating system during a point in OS initialization when evaluation of objects in the ACPI Namespace is not yet possible. OSPM evaluates the SRAT only during OS initialization.

Table A-7 lists the basic layout the SRAT. All addresses are processor-relative physical addresses.

Table A-7. System Resource Affinity Table Format

Field	Byte Length	Byte Offset	Description
Header			
Signature	4	0	Contains the signature for the system resource affinity table: "SRAT"
Length	4	4	Length, in bytes, of the entire SRAT. The length implies the number of Entry fields at the end of the table.
Revision	1	8	2
Checksum	1	9	Entire table must sum to zero
OEMID	6	10	OEM ID
OEM Table ID	8	16	For the SRAT, Table ID is the manufacturer model ID
OEM Revision	4	24	OEM revision of SRAT for supplied OEM Table ID
Creator ID	4	28	Vendor ID of utility that created the table.
Creator Revision	4	32	Revision of utility that created the table.
Reserved	4	36	Reserved to be 1 for backward compatibility
Reserved	8	40	Reserved
Static Resource Allocation Structure[n]	4	48	A list of static resource allocation structures for the platform. This list can contain processor Local APIC/SAPIC Affinity structure, memory affinity structure and x2APIC affinity structure.

Starting at byte offset 48 of the SRAT is a list of static resource allocation structures such as processor local APIC affinity and memory affinity structures. The first byte of each static resource allocation structure declares the type of that structure and the second byte declares the length of that structure. SRAT structure type 0 (processor local APIC affinity structure) and 1 (memory affinity structure) are already defined in ACPI 3.0 specification. Support for x2APIC defines a new SRAT entry type, 2, for processor x2APIC affinity structure.

The Processor X2APIC Affinity structure provides the association between the X2APIC ID of a logical processor and the proximity domain to which the logical processor belongs. The Processor X2APIC affinity structure must be used corresponding to a logical processor which is reported using Processor X2APIC structure (MADT entry type 9). The Processor Local APIC/SAPIC Affinity structure must be used corresponding to a processor which is reported using Processor Local APIC Structure (MADT entry type 0). The format of x2APIC Affinity structure is listed in Table A-8.

Table A-8. Processor x2APIC Affinity Structure Format

Field	Byte Length	Byte Offset	Description
Type	1	0	02H
Length	1	1	Length, in bytes, of the x2APIC Affinity structure (16 bytes)
Reserved	2	2	Must be zero
Proximity Domain[31:0]	4	4	Proximity Domain to which the logical processor belongs
x2APIC ID	4	8	Processor x2APIC ID
Flags	4	12	x2APIC Affinity Structure flags. See Table A-9 for a description of this field

Table A-9. x2APIC Affinity Structure Flag Field Definition

x2APIC Affinity flag field	Bit Length	Bit Offset	Description
Enabled	1	0	If clear, the OSPM ignores the contents of the Processor x2APIC Affinity Structure. This allows system firmware to populate the SRAT with a static number of structures but only enable them as necessary.
Reserved	31	1	Must be zero

A.4 ACPI NAMESPACE AND X2APIC SUPPORT

ACPI interface provides a hierarchical namespace to refer to system objects (CPU, system links, devices, etc.), section 5.3 of the ACPI 3.0 specification provides an overview of ACPI namespace.

Each logical processor in the system must be declared in the ACPI namespace in either the _SB or _PR scope but not both. Declaration of a logical processor in the _PR scope is required for platforms desiring compatibility with ACPI 1.0-based

OSPM implementations. Logical processors are declared either via the ASL Processor statement or the ASL Device statement. A Processor definition declares a processor object that provides processor configuration information and points to the processor register block (P_BLK). A Device definition for a processor object is declared using the ACPI0007 hardware identifier (HID). In this case, processor configuration information is provided exclusively by objects in the processor device's object list.

When the platform uses the APIC interrupt model, OSPM associates logical processors declared in the namespace with entries in the MADT. Prior to ACPI 3.0, this was accomplished using the processor object's Processor ID and the ACPI Processor ID fields in MADT entries. UID fields have been added to MADT entries in ACPI 3.0. By expanding processor declaration using Device definitions, UID object values under a processor device can now be used to associate processor devices with entries in the MADT. This removes the previous 256 processor declaration limit. The hand-off to OSPM will have processor IDs in the range of 0 to 254 for xAPIC/x2APIC and 0 to 255 for SAPIC declared as either Processor() or Device() objects, but not both. Processor IDs outside these ranges must be declared as Device() objects.

Processor-specific objects may be included in the processor object's optional object list or declared within the processor device's scope. These objects serve multiple purposes including providing alternative definitions for the registers described by the processor register block (P_BLK) and processor performance state control. Other ACPI-defined device-related objects are also allowed in the processor object's object list or under the processor device's scope (for example, the unique identifier object _UID).

With device-like characteristics attributed to processors, it is implied that a processor device driver will be loaded by OSPM to, at a minimum, process device notifications. OSPM will enumerate processors in the system using the ACPI Namespace, processor-specific native identification instructions, and optionally the _HID method.

OSPM will ignore definitions of ACPI-defined objects in an object list of a processor object declared under the _PR namespace.

For more information on the declaration of the processor object, see section 17.5.93, "Processor (Declare Processor)" of the ACPI 3.0 specification. Processor-specific objects are described in the following sections.

INDEX

S

T

X

This page intentionally left blank