IA-32 Intel® Architecture Software Developer's Manual

Documentation Changes

November 2003

Notice: The IA-32 Intel® Architecture may contain design defects or errors known as errata that may cause the product to deviate from published specifications. Current characterized errata are documented in this specification update.

Document Number: 252046-006

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL® PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT. Intel products are not intended for use in medical, life saving, or life sustaining applications.

Intel may make changes to specifications and product descriptions at any time, without notice.

Designers must not rely on the absence or characteristics of any features or instructions marked “reserved” or “undefined.” Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them.

The IA-32 Intel® Architecture may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

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

I^2C is a two-wire communications bus/protocol developed by Philips. SMBus is a subset of the I^2C bus/protocol and was developed by Intel. Implementations of the I^2C bus/protocol may require licenses from various entities, including Philips Electronics N.V. and North American Philips Corporation.

Intel, Pentium, Celeron, Intel SpeedStep, Intel Xeon and the Intel logo, and the Intel logo are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

*Other names and brands may be claimed as the property of others.

Copyright © 2002-2003, Intel Corporation

Contents

Revision History 4

Preface.. 5

Summary Table of Changes.. 6

Documentation Changes 8

Revision History

Version	Description	Date
-001	Initial Release	November 2002
-002	Added 1-10 Documentation Changes. Removed old Documentation Changes items that already have been incorporated in the published Software Developer's manual	December 2002
-003	Added 9 -17 Documentation Changes Removed Documenation Change #6 - References to bits Gen and Len Deleted Removed Documenation Change #4 - VIF Information Added to CLI Discussion	February 2003
-004	Removed Documentation changes 1-17 Added Documentation changes 1-24	June 2003
-005	Removed Documentation Changes 1-24 Added Documentation Changes 1-15	September 2003
-006	Added Documentation Changes 16- 34	November 2003

Preface

This document is an update to the specifications contained in the Affected Documents/Related Documents table below. This document is a compilation of documentation changes. It is intended for hardware system manufacturers and software developers of applications, operating systems, or tools.

Affected Documents/Related Documents

Document Title	Document Number
IA-32 Intel® Architecture Software Developer's Manual: Volume 1, Basic Architecture	245470-011
IA-32 Intel® Architecture Software Developer's Manual: Volume 2, Instruction Set Reference	245471-011
IA-32 Intel® Architecture Software Developer's Manual: Volume 3, System Programming Guide	245472-011

Nomenclature

Documentation Changes include errors or omissions from the current published specifications. These changes will be incorporated in the next release of the Software Development Maunal.

Summary Table of Changes

The following table indicates documentation changes which apply to the IA-32 Intel Architecture. This table uses the following notations:

Codes Used in Summary Table

Change bar to left of table row indicates this erratum is either new or modified from the previous version of the document.

Summary Table of Documentation Changes

Number	DOCUMENTATION CHANGES
1.	IA32_THERM_CONTROL has Been Changed to IA32_CLOCK_MODULATION
2.	INTER-PRIVILEGE" was not Spelled Corretly in Pseudocode Entry
3.	Confusing Text Artifact Removed
4.	IA32_MISC_CTL has Been Removed From the List of Architectural MSRs
5.	Typo Corrected in Figure 8-24
6.	Typo Corrected in Figure 8-23
7.	Corrupted Text Corrected
8.	Corrected an Error in PACKSSDW Illustration
9.	SSM Corrected to SMM
10.	Exiting From SMM Text Updated
11.	L1 Data Cache Context Mode Description has Been Udpated
12.	#DE Should be #DB in Description of EFLAGS.RF
13.	There Have Been Revisions to the Table That States Priority Among Simultaneous Exceptions and Interrupts
14.	Corrections to Page-Directory-Pointer-Table Entry Desciption
15.	Behavior Notes on the Accessed (A) Flag and Dirty (D) Flag
16.	Interrupt 11 Discussion Concerning EXT Flag Functioning Has Been Updated
17.	Improved Information on Interpreting Machine-Check Error Codes
18.	More information on the Functioning of Debug BPs after POP SS/MOV SS Has Been Provided
19.	More Information on the LBR Stack Has Been Provided
20.	Limited Availability of Two MSRs Has Been Documented
21.	The Section On Microcode Update Facilities Has Been Refreshed
22.	A Mechanism for Determining Sync/Async SMIs Has Been Documented

Summary Table of Documentation Changes

Number	DOCUMENTATION CHANGES
23.	Omitted Debug Data Has Been Restored
24.	CLTS Exception Information Improved
25.	The MOVSS Description Have Been Updated
26.	An Instruction Listing (PULLHUW) Has Been Deleted
27.	Some Data Entry Errors in Table B-20 Have Been Corrected
28.	Figure 8-22 Has Been Corrected
29.	The Description of Minimum Thermal Monitor Activation Time Has Been Updated
30.	Corrected Description of Exception- or Interrupt-Handler Procedures
31.	CMPSD and CMPSS Exception Information Updated
32.	PUNPCKHB*/PUNPCKLB* Exception Information Improved
33.	MOVHPD, MOVLPD, UNPCKHPS, UNPCKLPS Exception Information Improved.
34.	PEXTRW - PINSRW Exception Information Improved

Documentation Changes

1. IA32_THERM_CONTROL has been Changed to IA32_CLOCK_MODULATION

The name of the MSR IA32_THERM_CONTROL has been changed to IA32_CLOCK_MODULATION. This was done to avoid confusion about the MSR's function.

The following corrected table segment is from Appendix B, Table B-3, the *IA-32 Intel Architecture Software Developer's Manual, Volume 3*. See the reproduced text below

Register Address		Register Name	Bit Description
Hex	Dec		
19AH	410	IA32_CLOCK_MODULATION	**Clock Modulation.** (R/W) Enables and disables on-demand clock modulation and allows the selection of the on-demand clock modulation duty cycle. (See Section 13.15.3., *Software Controlled Clock Modulation*. NOTE: IA32_CLOCK_MODULATION MSR was originally named IA32_THERM_CONTROL MSR.

2. INTER-PRIVILEGE" Was Not Spelled Correctly in Pseudocode Entry

The term inter-privilege was incorrectly spelled in pseudocode provided as part of the "INT n/INTO/INT 3—Call to Interrupt Procedure" section, Chapter 3, *IA-32 Intel Architecture Software Developer's Manual, Volume 2.*

The corrected text segment is reproduced below.

```
...INTER-PRIVILEGE-LEVEL-INTERRUPT
    (* PE=1, interrupt or trap gate, non-conforming code segment, DPL<CPL *)
    (* Check segment selector and descriptor for stack of new privilege level in current TSS *)
    IF current TSS is 32-bit TSS
        THEN
            TSSstackAddress ← (new code segment DPL ∗ 8) + 4.....
```

3. Confusing text Artifact Removed

There were some materials in the OPCODE table that should have been deleted. This error has been corrected. The corrected table segment (reproduced below) is in Appendix A, Table A-3, *IA-32 Intel Architecture Software Developer's Manual, Volume 2*. See address 0x0f0b

.

	8	9	A	B	C	D	E	F
0	INVD	WBINVD		**UD2**				

4. IA32_MISC_CTL Has Been Removed from the List of Architectural MSRs

The MSR IA32_MISC_CTL has been removed from the list of architectural MSRs . Note that this MSR is still listed in other locations.

The impacted segment (reproduced below) is from Appendix B, Table B-5, *IA-32 Intel Architecture Software Developer's Manual, Volume 3*. The change bars show where the table row was deleted.

79H	121	IA32_BIOS_UPDT_TRIG	BIOS_UPDT_TRIG	P6 Family Processors
8BH	139	IA32_BIOS_SIGN_ID	BIOS_SIGN/BBL_CR_D3	P6 Family Processors
FEH	254	IA32_MTRRCAP	MTRRcap	P6 Family Processors
174H	372	IA32_SYSENTER_CS	SYSENTER_CS_MSR	P6 Family Processors
175H	373	IA32_SYSENTER_ESP	SYSENTER_ESP_MSR	P6 Family Processors

5. Typographical Error Corrected in Figure 8-24

ExINT should be ExtINT in Figure 8-24, located in the “Message Data Register Format” section, Chapter 8, *IA-32 Intel Architecture Software Developer’s Manual, Volume 3*. The corrected figure is reproduced below.

Figure 8-24. Layout of the MSI Message Data Register

6. Typographical Error Corrected in Figure 8-23

0FEEH was incorrectly represented as 0FEEEH in Figure 8-23, located the “Message Address Register Format” section, Chapter 8, *IA-32 Intel Architecture Software Developer’s Manual, Volume 3*.

The corrected figure is reproduced below.

Figure 8-23. Layout of the MSI Message Address Register

7. Corrupted Text Corrected

There was some corrupted text in the "State of the Logical Processors" section, Chapter 7, *IA-32 Intel Architecture Software Developer's Manual, Volume 3*.

The correction is shown in the segment below. See the changebar.

7.6.1.1 State of the Logical Processors

The following features are considered part of the architectural state of a logical processor with HT Technology. The features can be subdivided into three groups:

- Duplicated for each logical processor
- Shared by logical processors in a physical processor
- Shared or duplicated, depending on the implementation

The following features are duplicated for each logical processor:

- General purpose registers (EAX, EBX, ECX, EDX, ESI, EDI, ESP, and EBP)

....

8. Corrected an Error in PACKSSDW Illustration

Operation of the PACKSSDW instruction was incorrectly illustrated in Figure 3-6, the "PACKSSWB/PACKSSDW—Pack with Signed Saturation" section, Chapter 3, *IA-32 Intel Architecture Software Developer's Manual, Volume 2*.

The corrected figure is reproduced below.

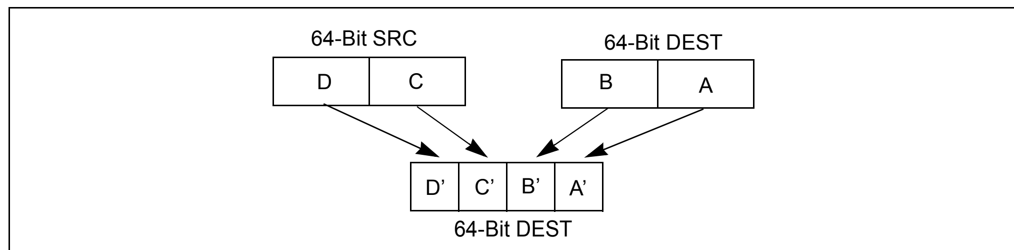

Figure 3.6. Operation of the PACKSSDW Instruction Using 64-bit Operands

9. SSM Corrected to SMM

In several places, SSM was still being used as an acronym for 'system management mode.' The correct usage is SMM. Corrections were made in the "Modes of Operation" section, Chapter 3, *IA-32 Intel Architecture Software Developer's Manual, Volume 1*. The updated paragraph is reproduced below.

....

System management mode (SMM). This mode provides a transparent mechanism for implementing platform-specific functions such as power management and system security. The processor enters SMM when the external SMM interrupt pin (SMI#) is activated or an SMI is received from the advanced programmable interrupt controller (APIC). In SMM, the processor switches to a separate address space while saving the basic context of the currently running program or task. SMM-specific code may then be executed transparently. Upon returning from SMM, the processor is placed back into its state prior to the system management interrupt. SMM was introduced with the Intel386™ SL and Intel486™ SL processors and became a standard IA-32 feature with the Pentium processor family.

....

This change was also made in the "RSM—Resume from System Management Mode" section, Chapter 3, *IA-32 Intel Architecture Software Developer's Manual, Volume 2*. The corrected segments are reproduced below.

....

Returns program control from system management mode (SMM) to the application program or operating-system procedure that was interrupted when the processor received an SMM interrupt. The processor's state is restored from the dump created upon entering SMM. If the processor detects invalid state information during state restoration, it enters the shutdown state....

```
...
ReturnFromSMM;
ProcessorState ← Restore(SMMDump);
...
```

10. Exiting from SMM Text Updated

A paragraph in the "Exiting from SMM" section, Chapter 13, *IA-32 Intel Architecture Software Developer's Manual, Volume 3* has been updated. The information previously provided was not complete. The corrected text segment is reproduced below. See the change bar for location.

- (For the Pentium and Intel486 processors only.) If the address stored in the SMBASE register when an RSM instruction is executed is not aligned on a 32-KByte boundary. This restriction does not apply to the P6 family processors.

In the shutdown state, Intel processors stop executing instructions until a RESET#, INIT# or NMI# is asserted. Processors do recognize the FLUSH# signal in the shutdown state. While Pentium family processors recognize the SMI# signal in shutdown state, P6 family and Intel486 processors do not. Intel does not support using SMI# to recover from shutdown states for any processor family; the response of processors in this circumstance is not well defined. On Pentium 4 and later processors, shutdown will inhibit INTR and A20M but will not change any of the other inhibits. On these processors, NMIs will be inhibited if no action is taken in the SMM handler to uninhibit them (see Section 13.7.).

If the processor is in the HALT state when the SMI is received, the processor handles the return from SMM slightly differently (see Section 13.10., "Auto HALT Restart"). Also, the SMBASE address can be changed on a return from SMM (see Section 13.11., "SMBASE Relocation").

11. L1 Data Cache Context Mode Description Has Been Udpated

In Appendix B, Table B-1, *IA-32 Intel Architecture Software Developer's Manual, Volume 3*; the "L1 Data Cache Context Mode (RW)" table cell has been updated. Information about adaptive mode was clarified.

The updated table segment is reproduced below.

Register Address		Register Name Fields and Flags	Shared/ Unique[1]	Bit Description
Hex	Dec			
		24		**L1 Data Cache Context Mode (R/W).** When set to 1, this bit places the L1 Data Cache into shared mode. When set to 0 (the default), this bit places the L1 Data Cache into adaptive mode. When the L1 Data Cache is running in adaptive mode and the CR3s are identical, data in L1 is shared across logical processors. Otherwise, data in L1 is not shared and cache use is competitive. NOTE: If the Context ID feature flag, ECX[10], is not set to 1 after executing CPUID with EAX = 1; the ability to switch modes is not supported and the BIOS must not alter the contents of IA32_MISC_ENABLE[24].

12. #DE Should Be #DB in Description of EFLAGS.RF

In the "System Flags and Fields in the EFLAGS Register" section, Chapter 2, *IA-32 Intel Architecture Software Developer's Manual, Volume 3*; there was a sentence that began "When set, this flag temporarily disables debug exceptions (#DE)". Debug exceptions are noted as #DB, not #DE. This error has been corrected.

The corrected entry is reproduced below.

RF **Resume (bit 16).** Controls the processor's response to instruction-breakpoint conditions. When set, this flag temporarily disables debug exceptions (#DB) from being generated for instruction breakpoints; although, other exception conditions can cause an exception to be generated. When clear, instruction breakpoints will generate debug exceptions.

The primary function of the RF flag is to allow the restarting of an instruction following a debug exception that was caused by an instruction breakpoint condition. Here, debugger software must set this flag in the EFLAGS image on the stack just prior to returning to the interrupted program with the IRETD instruction, to prevent the instruction breakpoint from causing another debug exception. The processor then automatically clears this flag after the instruction returned to has been successfully executed, enabling instruction breakpoint faults again.

See Section 15.3.1.1., *Instruction-Breakpoint Exception Condition*, for more information on the use of this flag.

13. There Have Been Revisions to the Table That States Priority among Simultaneous Exceptions and Interrupts

We have made a number of updates to Table 5-2, located in the “Priority Among Simultaneous Exceptions and Interrupts” section, Chapter 5, *IA-32 Intel Architecture Software Developer’s Manual, Volume 3*.

The updated cells are reproduced below.

Priority	Descriptions (continued)...
5	External Interrupts - NMI Interrupts - Maskable Hardware Interrupts
6	Code Breakpoint Fault
7	Faults from Fetching Next Instruction - Code-Segment Limit Violation - Code Page Fault
8	Faults from Decoding the Next Instruction - Instruction length > 15 bytes - Invalid Opcode - Coprocessor Not Available
9 (Lowest)	Faults on Executing an Instruction - Overflow - Bound error - Invalid TSS - Segment Not Present - Stack fault - General Protection - Data Page Fault - Alignment Check - x87 FPU Floating-point exception - SIMD floating-point exception

14. Corrections to Page-Directory-Pointer-Table Entry Desciption

In Figure 3-20 and 3-21, located in the “Page-Directory and Page-Table Entries With Extended Addressing Enabled” section, Chapter 3, *IA-32 Intel Architecture Software Developer’s Manual, Volume 3*; Bit 0 of both representations of the Page-Directory-Pointer-Table Entry now indicate P (showing the the location of the ‘present flag’ bit).

The corrected tables are reproduced below.

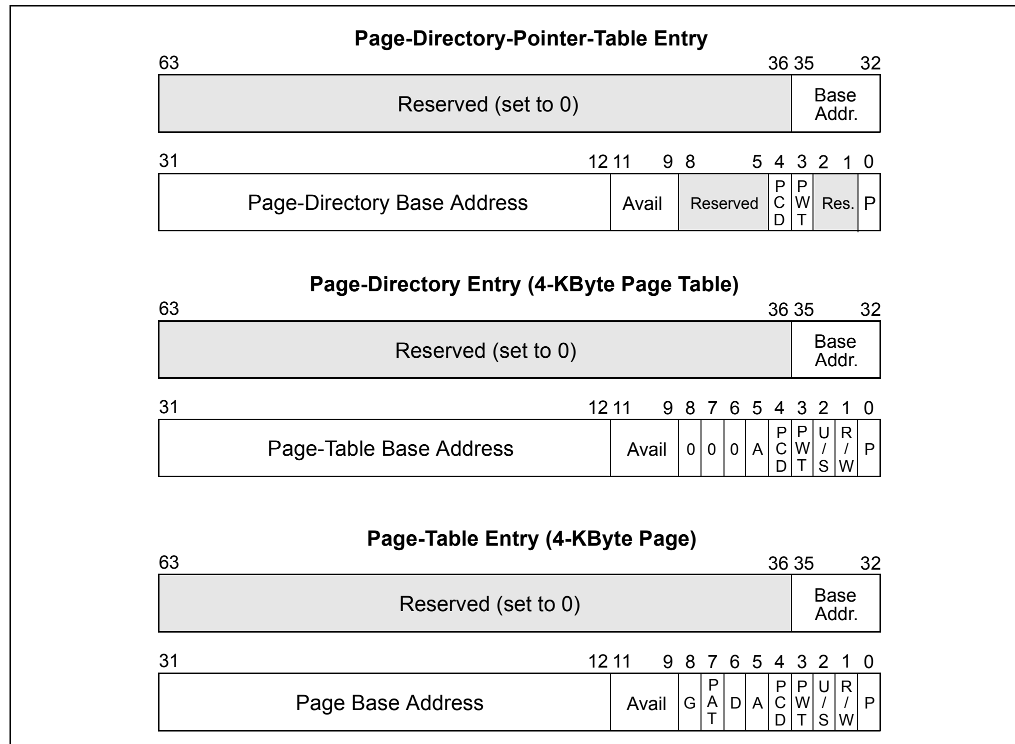

Figure 3-20. Format of Page-Directory-Pointer-Table, Page-Directory, and Page-Table Entries for 4-KByte Pages with PAE Enabled

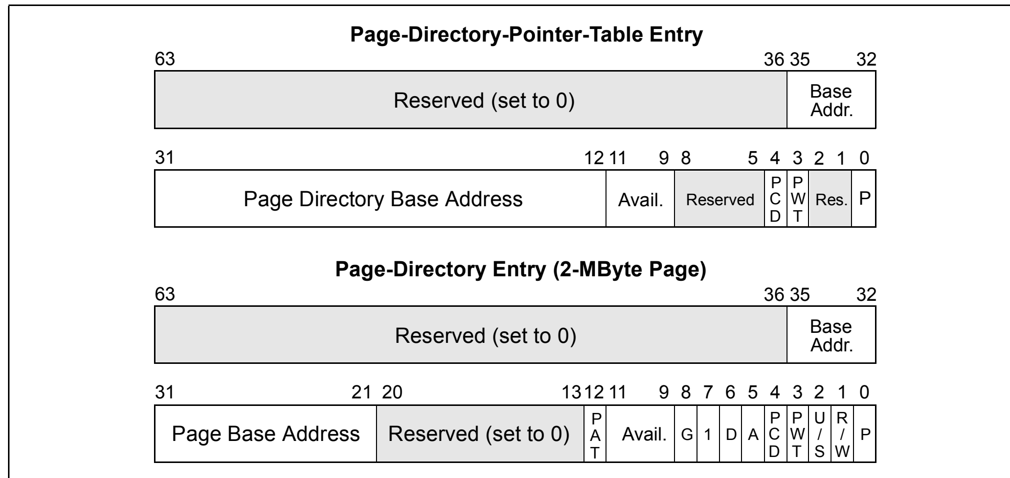

Figure 3-21. Format of Page-Directory-Pointer-Table and Page-Directory Entries for 2-MByte Pages with PAE Enabled

In addition, the paragraph discussing the present flag has been updated. this text is also located in the “Page-Directory and Page-Table Entries With Extended Addressing Enabled” section, Chapter 3, *IA-32 Intel Architecture Software Developer’s Manual, Volume 3*.

The applicable text segment is reproduced below. Note the change bar.

For all table entries (except for page-directory entries that point to 2-MByte pages), the bits in the page base address are interpreted as the 24 most-significant bits of a 36-bit physical address, which forces page tables and pages to be aligned on 4-KByte boundaries. When a page-directory entry points to a 2-MByte page, the base address is interpreted as the 15 most-significant bits of a 36-bit physical address, which forces pages to be aligned on 2-MByte boundaries.

The present flag (bit 0) in the page-directory-pointer-table entries can be set to 0 or 1. If the present flag is clear, the remaining bits in the page-directory-pointer-table entry are available to the operating system. If the present flag is set, the fields of the page-directory-pointer-table entry are defined in Figures for 4KB pages and Figures for 2MB pages.

The page size (PS) flag (bit 7) in a page-directory entry determines if the entry points to a page table or a 2-MByte page. When this flag is clear, the entry points to a page table; when the flag is set, the entry points to a 2-MByte page. This flag allows 4-KByte and 2-MByte pages to be mixed within one set of paging tables.

15. Behavior Notes on the Accessed (A) Flag and Dirty (D) Flag

Notes have been added to two sub-paragraphs of the “Page-Directory and Page-Table Entries” section, Chapter 3, *IA-32 Intel Architecture Software Developer’s Manual, Volume 3*. The notes clarify a limitation on the processor’s Self-Modifying Code detection logic in the Accessed (A) flag and Dirty (D) flag context.

The applicable sections are reproduced below. See the change bars.

Accessed (A) flag, bit 5

Indicates whether a page or page table has been accessed (read from or written to) when set. Memory management software typically clears this flag when a page or page table is initially loaded into physical memory. The processor then sets this flag the first time a page or page table is accessed.

This flag is a “sticky” flag, meaning that once set, the processor does not implicitly clear it. Only software can clear this flag. The accessed and dirty flags are provided for use by memory management software to manage the transfer of pages and page tables into and out of physical memory.

NOTE: The accesses used by the processor to set this bit may or may not be exposed to the processor’s Self-Modifying Code detection logic. If the processor is executing code from the same memory area that is being used for page table structures, the setting of the bit may or may not result in an immediate change to the executing code stream.

Dirty (D) flag, bit 6

Indicates whether a page has been written to when set. (This flag is not used in page-directory entries that point to page tables.) Memory management software typically clears this flag when a page is initially loaded into physical memory. The processor then sets this flag the first time a page is accessed for a write operation.

This flag is “sticky,” meaning that once set, the processor does not implicitly clear it. Only software can clear this flag. The dirty and accessed flags are provided for use by memory management software to manage the transfer of pages and page tables into and out of physical memory.

NOTE: The accesses used by the processor to set this bit may or may not be exposed to the processor’s Self-Modifying Code detection logic. If the processor is executing code from the same memory area that is being used for page table structures, the setting of the bit may or may not result in an immediate change to the executing code stream.

16. Interrupt 11 Discussion Concerning EXT Flag Functioning Has Been Updated

The Volume 3, Chapter 5, Interrupt 11: Error Code section has been updated. This section now provides a more complete description of the EXT flag. The impacted text is reproduced below.

Exception Error Code

An error code containing the segment selector index for the segment descriptor that caused the violation is pushed onto the stack of the exception handler. If the EXT flag is set, it indicates that the exception resulted from either:

- an external event (NMI or INTR) that caused an interrupt, which subsequently referenced a not-present segment.
- a benign exception that subsequently referenced a not-present segment. A contributory exception or page fault that subsequently referenced a not-present segment would cause a double fault (#DF) to be generated instead of #NP.

17. Improved Information on Interpreting Machine-Check Error Codes

In Volume 3, Appendix E has been re-written to incorporate new IA32_MCi_STATUS data. Encoding of the model-specific and other information fields is different for the 06H and 0FH processor families. Changes are documented in the following sections.

E.1. DECODING FAMILY 06H SPECIFIC MACHINE ERROR CODES

Machine error code reporting by processor family 06H is based on values read from IA32_MC*i*_STATUS (Figure E-1).

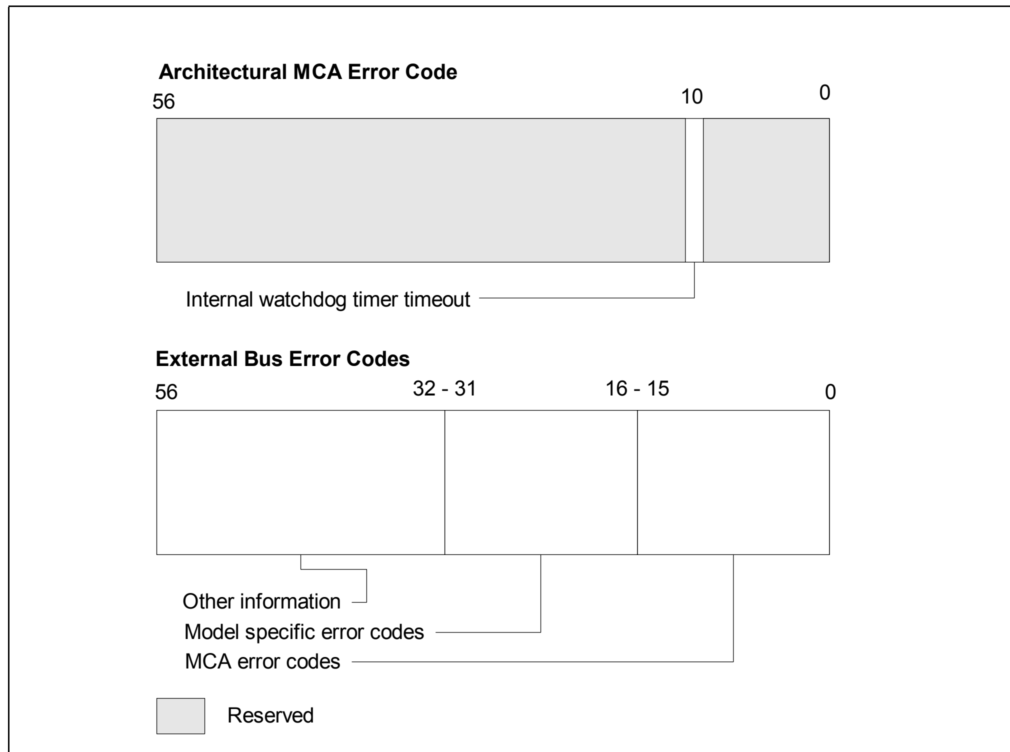

Figure E-1. IA32_MC*i*_STATUS Encoding for Family 06H

Table E-1 shows how to interpret internal watchdog timer timeout machine-check errors reported in IA32_MC*i*_STATUS for processor family 06H.

Table E-1. Family 06H Encoding of Internal Watchdog Timer Errors Reported in IA32_MC*i*_STATUS

	Bit No.	Bit Function	Bit Description
Architectural MCA error code	0-15	0000010000000000	Internal watchdog timer timeout. Note that a watch-dog timer time-out only occurs if the BINIT driver is enabled.
Model-specific error codes	16-31	Reserved	Reserved
Other information	32-56	Reserved	Reserved

Table E-2 shows how to interpret errors that occur on the external bus.

Table E-2. Family 06H Encoding 32_MC*i*_STATUS for External Bus Errors

Type	Bit No.	Bit Function	Bit Description
MCA error codes	0-1	Reserved	Reserved.
	2-3	For external bus errors: special cycle or I/O For internal timeout: Reserved	For external bus errors: • Bit 2 is set to 1 if the access was a special cycle. • Bit 3 is set to 1 if the access was a special cycle OR a I/O cycle. For internal timeout: Reserved
	4-7	For external bus errors: Read/Write For internal timeout: Reserved	For external bus errors, 00WR: W = 1 for writes R = 1 for reads For internal timeout: Reserved
	8-9	Reserved	Reserved
	10-11	10 01	External bus errors Internal watchdog timer timeout
	12-15	Reserved	Reserved
Model specific errors	16-18	Reserved	Reserved

Table E-2. Family 06H Encoding 32_MC*i*_STATUS for External Bus Errors (Continued)

Type	Bit No.	Bit Function	Bit Description
Model specific errors	19-24	Bus queue request type	000000 for BQ_DCU_READ_TYPE error 000010 for BQ_IFU_DEMAND_TYPE error 000011 for BQ_IFU_DEMAND_NC_TYPE error 000100 for BQ_DCU_RFO_TYPE error 000101 for BQ_DCU_RFO_LOCK_TYPE error 000110 for BQ_DCU_ITOM_TYPE error 001000 for BQ_DCU_WB_TYPE error 001010 for BQ_DCU_WCEVICT_TYPE error 001011 for BQ_DCU_WCLINE_TYPE error 001100 for BQ_DCU_BTM_TYPE error 001101 for BQ_DCU_INTACK_TYPE error 001110 for BQ_DCU_INVALL2_TYPE error 001111 for BQ_DCU_FLUSHL2_TYPE error 010000 for BQ_DCU_PART_RD_TYPE error 010010 for BQ_DCU_PART_WR_TYPE error 010100 for BQ_DCU_SPEC_CYC_TYPE error 011000 for BQ_DCU_IO_RD_TYPE error 011001 for BQ_DCU_IO_WR_TYPE error 011100 for BQ_DCU_LOCK_RD_TYPE error 011110 for BQ_DCU_SPLOCK_RD_TYPE error 011101 for BQ_DCU_LOCK_WR_TYPE error 000010 for BQ_IFU_DEMAND_TYPE error 000011 for BQ_IFU_DEMAND_NC_TYPE error 000100 for BQ_DCU_RFO_TYPE error 000101 for BQ_DCU_RFO_LOCK_TYPE error 000110 for BQ_DCU_ITOM_TYPE error 001000 for BQ_DCU_WB_TYPE error 001010 for BQ_DCU_WCEVICT_TYPE error 001011 for BQ_DCU_WCLINE_TYPE error 001100 for BQ_DCU_BTM_TYPE error 001101 for BQ_DCU_INTACK_TYPE error 001110 for BQ_DCU_INVALL2_TYPE error 001111 for BQ_DCU_FLUSHL2_TYPE error 010000 for BQ_DCU_PART_RD_TYPE error 010010 for BQ_DCU_PART_WR_TYPE error 010100 for BQ_DCU_SPEC_CYC_TYPE error 011000 for BQ_DCU_IO_RD_TYPE error 011001 for BQ_DCU_IO_WR_TYPE error 011100 for BQ_DCU_LOCK_RD_TYPE error 011110 for BQ_DCU_SPLOCK_RD_TYPE error 011101 for BQ_DCU_LOCK_WR_TYPE error
Model specific errors	27-25	Bus queue error type	000 for BQ_ERR_HARD_TYPE error 001 for BQ_ERR_DOUBLE_TYPE error 010 for BQ_ERR_AERR2_TYPE error 100 for BQ_ERR_SINGLE_TYPE error 101 for BQ_ERR_AERR1_TYPE error

Table E-2. Family 06H Encoding 32_MC*i*_STATUS for External Bus Errors (Continued)

Type	Bit No.	Bit Function	Bit Description
Model specific errors	28	FRC error	1 if FRC error active
	29	BERR	1 if BERR is driven
	30	Internal BINIT	1 if BINIT driven for this processor
	31	Reserved	Reserved
Other information	32-34	Reserved	Reserved
	35	External BINIT	1 if BINIT is received from external bus.
	36	RESPONSE PARITY ERROR	This bit is asserted in IA32_MC*i*_STATUS if this component has received a parity error on the RS[2:0]# pins for a response transaction. The RS signals are checked by the RSP# external pin.
	37	BUS BINIT	This bit is asserted in IA32_MC*i*_STATUS if this component has received a hard error response on a split transaction (one access that has needed to be split across the 64-bit external bus interface into two accesses).
	38	TIMEOUT BINIT	This bit is asserted in IA32_MC*i*_STATUS if this component has experienced a ROB time-out, which indicates that no micro-instruction has been retired for a predetermined period of time. A ROB time-out occurs when the 15-bit ROB time-out counter carries a 1 out of its high order bit. The timer is cleared when a micro-instruction retires, an exception is detected by the core processor, RESET is asserted, or when a ROB BINIT occurs. The ROB time-out counter is prescaled by the 8-bit PIC timer which is a divide by 128 of the bus clock (the bus clock is 1:2, 1:3, 1:4 of the core clock). When a carry out of the 8-bit PIC timer occurs, the ROB counter counts up by one. While this bit is asserted, it cannot be overwritten by another error.
	39-41	Reserved	Reserved
	42	HARD ERROR	This bit is asserted in IA32_MC*i*_STATUS if this component has initiated a bus transactions which has received a hard error response. While this bit is asserted, it cannot be overwritten.
	43	IERR	This bit is asserted in IA32_MC*i*_STATUS if this component has experienced a failure that causes the IERR pin to be asserted. While this bit is asserted, it cannot be overwritten.

Table E-2. Family 06H Encoding 32_MC*i*_STATUS for External Bus Errors (Continued)

Type	Bit No.	Bit Function	Bit Description
Other information	44	AERR	This bit is asserted in IA32_MC*i*_STATUS if this component has initiated 2 failing bus transactions which have failed due to Address Parity Errors (AERR asserted). While this bit is asserted, it cannot be overwritten.
	45	UECC	The Uncorrectable ECC error bit is asserted in IA32_MC*i*_STATUS for uncorrected ECC errors. While this bit is asserted, the ECC syndrome field will not be overwritten.
	46	CECC	The correctable ECC error bit is asserted in IA32_MC*i*_STATUS for corrected ECC errors.
	47-54	ECC syndrome	The ECC syndrome field in IA32_MC*i*_STATUS contains the 8-bit ECC syndrome only if the error was a correctable/uncorrectable ECC error and there wasn't a previous valid ECC error syndrome logged in IA32_MC*i*_STATUS. A previous valid ECC error in IA32_MC*i*_STATUS is indicated by IA32_MC*i*_STATUS.bit45 (uncorrectable error occurred) being asserted. After processing an ECC error, machine-check handling software should clear IA32_MC*i*_STATUS.bit45 so that future ECC error syndromes can be logged.
	55-56	Reserved	Reserved.

E.2. DECODING FAMILY 0FH SPECIFIC MACHINE ERROR CODES

Machine error code reporting by processor family 0FH is also based on values read from IA32_MC*i*_STATUS (Figure E-2).

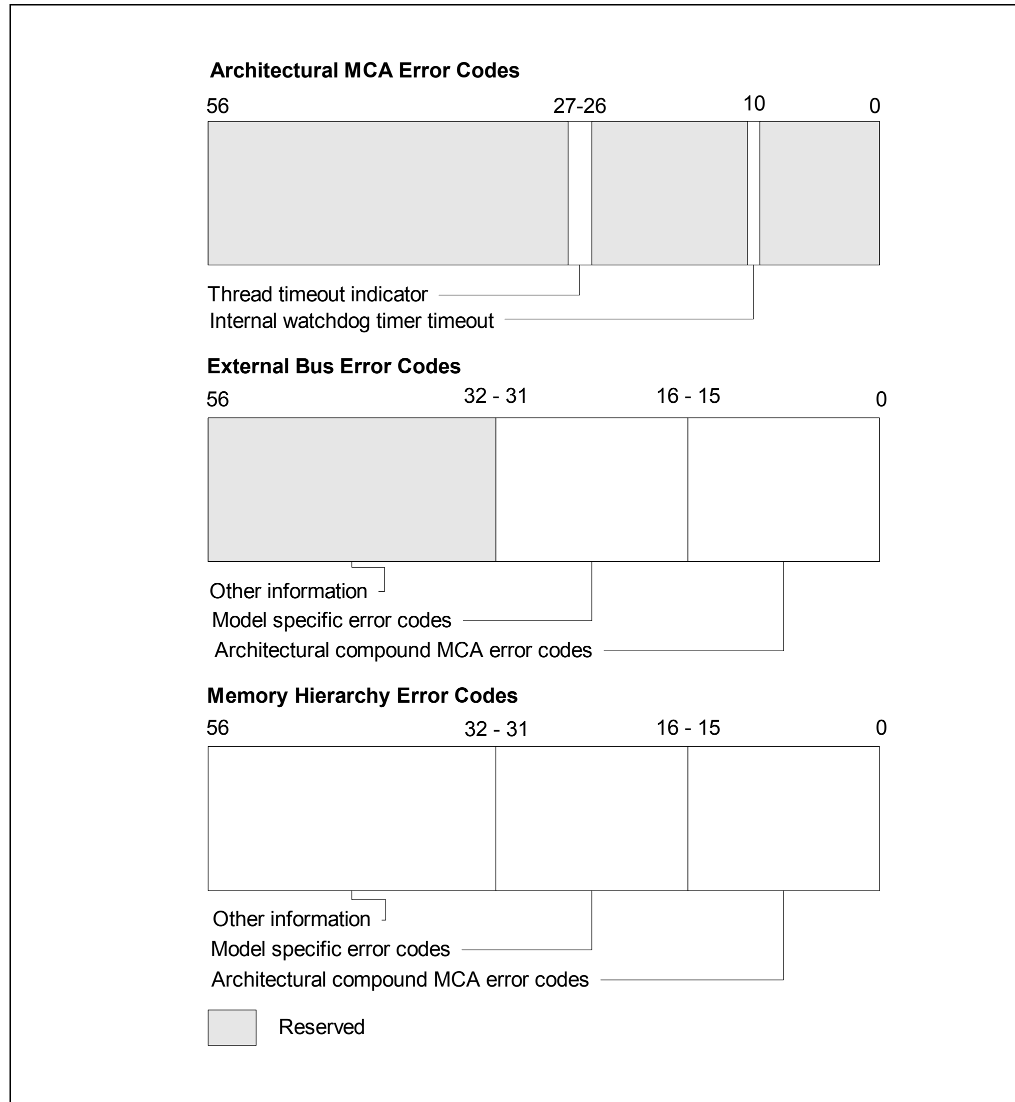

Figure E-2. IA32_MC*i*_STATUS Encoding for Family 0FH

Table E-3 provides information on how to interpret processor family 0FH error code fields for internal watchdog timer timeout machine-checks.

Table E-3. Family 0FH Encoding of IA32_MC*i*_STATUS for Internal Watchdog Timer Errors

	Bit No.	Bit Function	Bit Description
Architectural MCA error code	0-15	0000010000000000	Internal watchdog timer timeout. Note that a watch-dog timer time-out only occurs if the BINIT driver is enabled.
Model-specific error code	16-25	Reserved	Reserved
	26-27	Thread timeout indicator (TT)	Contains the indication of the thread which timed out: 01 - Thread 0 timed out 10 - Thread 1 timed out 11 - Both threads timed out
	28-31	Reserved	Reserved
Other information	32-56	Reserved	Reserved

Table E-4 provides the information to interpret errors that occur on the external bus. Note that processor family 0FH uses the compound MCA code format for external bus errors. Refer to *Chapter 14, Machine-Check Architecture* for more information.

Table E-4. Family 0FH Encoding of IA32_MC*i*_STATUS for External Bus Errors

	Bit No.	Bit Function	Bit Description
Architectural compound MCA error codes	0-1	Memory hierarchy level (LL)	Refer to Table 14-5 for detailed decoding of the memory hierarchy level (LL) sub-field.
	2-3	Memory and I/O (II)	Refer to Table 14-7 for a detailed decoding of the memory or IO (II) sub-field.
	4-7	Request (RRRR)	Refer to Table 14-6 for a detailed decoding of the request (RRRR) sub-field.
	8	Timeout (T)	Refer to Table 14-7 for a detailed decoding of the Timeout (T) Sub-Field.
	9-10	Participation (PP)	Refer to Table 14-7 for a detailed decoding of the participation (PP) sub-field.
	11-15	00001	Bus and interconnect errors
Model-specific error codes	16	FSB address parity	Address parity error detected: 1 = Address parity error detected 0 = No address parity error
	17	Response hard fail	Hardware failure detected on response
	18	Response parity	Parity error detected on response
	19	PIC and FSB data parity	Data Parity detected on either PIC or FSB access
	20	Processor Signature = 00000F04H: Invalid PIC request All other processors: Reserved	Processor Signature = 00000F04H. Indicates error due to an invalid PIC request (access was made to PIC space with WB memory): 1 = Invalid PIC request error 0 = No Invalid PIC request error Reserved
	21	Pad state machine	The state machine that tracks P and N data-strobe relative timing has become unsynchronized or a glitch has been detected.
	22	Pad strobe glitch	Data strobe glitch
	23	Pad address glitch	Address strobe glitch
	24-31	Reserved	Reserved
Other Information	32-56	Reserved	Reserved

Table E-5 provides information on how to interpret errors that occur within the memory hierarchy.

Table E-5. Family 0FH Encoding of IA32_MC*i*_STATUS for Memory Hierarchy Errors

	Bit No.	Bit Function	Bit Description
Architectural compound MCA error code	0-1	Memory Hierarchy Level (LL)	Refer to Table 14-5 for a detailed decoding of the memory hierarchy level (LL) sub-field.
	2-3	Transaction Type (TT)	Refer to Table 14-5 for a detailed decoding of the transaction type (TT) sub-field.
	4-7	Request (RRRR)	Refer to Table 14-6 for a detailed decoding of the request type (RRRR) sub-field.
	8-15	00000001	Memory hierarchy error format
Model specific error codes	16-17	Tag Error Code	Contains the tag error code for this machine check error: 00 = No error detected 01 = Parity error on tag miss with a clean line 10 = Parity error/multiple tag match on tag hit 11 = Parity error/multiple tag match on tag miss
	18-19	Data Error Code	Contains the data error code for this machine check error: 00 = No error detected 01 = Single bit error 10 = Double bit error on a clean line 11 = Double bit error on a modified line
	20	L3 Error	This bit is set if the machine check error originated in the L3 (it can be ignored for invalid PIC request errors): 1 = L3 error 0 = L2 error
	21	Invalid PIC Request	Indicates error due to invalid PIC request (access was made to PIC space with WB memory): 1 = Invalid PIC request error 0 = No invalid PIC request error
	22-31	Reserved	Reserved
Other Information	32-39	8-bit Error Count	Holds a count of the number of errors since reset. The counter begins at 0 for the first error and saturates at a count of 254.
	40-56	Reserved	Reserved

18. More information on the Functioning of Debug BPs after POP SS/MOV SS Has Been Provided

In Volume 3, Section 15.3.1.1; more information has been provided on the functioning of code instruction breakpoints immediately after POP SS/MOV SS instructions. This data is reprinted below (in context). Footnotes have been added to the POP and MOV sections in Volume 2 of the IA-32 Intel Architecture Software Developer's Manual which contain the same information for POP SS/MOV SS (the footnotes are not reproduced here).

\---

15.3.1.1. INSTRUCTION-BREAKPOINT EXCEPTION CONDITION

The processor reports an instruction breakpoint when it attempts to execute an instruction at an address specified in a breakpoint-address register (DB0 through DR3) that has been set up to detect instruction execution (R/W flag is set to 0). Upon reporting the instruction breakpoint, the processor generates a fault-class, debug exception (#DB) before it executes the target instruction for the breakpoint.

Instruction breakpoints are the highest priority debug exceptions. They are serviced before any other exceptions detected during the decoding or execution of an instruction. Note, however, that if a code instruction breakpoint is placed on an instruction located immediately after a POP SS/MOV SS instruction, it may not be triggered. In most situations, POP SS/MOV SS will inhibit such interrupts (see "MOV-Move" and "POP-Pop a Value from the Stack" in the *IA-32 Intel Architecture Software Developer's Manual, Volume 2*).

19. More Information on the LBR Stack Has Been Provided

The following information has been added to Volume 3 of the IA-32 Intel Architecture Software Developer's Manual, Section 15.5. This information describes the LBR stack and MSR_LASTBRANCH_TOS

\---

- Last Branch Record (LBR) Stack — The LBR stack is a circular stack that consists of four MSRs (MSR_LASTBRANCH_0 through MSR_LASTBRANCH_3) for the Pentium 4 and Intel Xeon processor family [CPUID family 0FH, models 0H-02H]. The LBR stack consists of 16 MSR pairs (MSR_LASTBRANCH_0_FROM_LIP through MSR_LASTBRANCH_15_FROM_LIP and MSR_LASTBRANCH_0_TO_LIP through MSR_LASTBRANCH_15_TO_LIP) for the Pentium 4 and Intel Xeon processor family [CPUID family 0FH, model 03H].
- Last Branch Record Top-of-Stack (TOS) Pointer — The TOS Pointer MSR contains a 2-bit pointer (0-3) to the MSR in the LBR stack that contains the most recent branch, interrupt, or exception recorded for the Pentium 4 and Intel Xeon processor family [CPUID family 0FH, models 0H-02H]. This pointer becomes a 4-bit pointer (0-15) for the Pentium 4 and Intel Xeon processor family [CPUID family 0FH, model 03H].

See also: Table 15-2, Figure 15-3, and Figure 15-4 below.

Table 15-2. LBR MSR Stack Structure for the Pentium 4 and Intel Xeon Processor Family

LBR MSRs for Family 0FH, Models 0H-02H; MSRs at locations 1DBH-1DEH.	**Decimal Value of TOS Pointer in MSR_LASTBRANCH_TOS (bits 0-1)**
MSR_LASTBRANCH_0 MSR_LASTBRANCH_1 MSR_LASTBRANCH_2 MSR_LASTBRANCH_3	0 1 2 3
LBR MSRs for Family 0FH, Models; MSRs at locations 680H-68FH.	**Decimal Value of TOS Pointer in MSR_LASTBRANCH_TOS (bits 0-3)**
MSR_LASTBRANCH_0_FROM_LIP MSR_LASTBRANCH_1_FROM_LIP MSR_LASTBRANCH_2_FROM_LIP MSR_LASTBRANCH_3_FROM_LIP MSR_LASTBRANCH_4_FROM_LIP MSR_LASTBRANCH_5_FROM_LIP MSR_LASTBRANCH_6_FROM_LIP MSR_LASTBRANCH_7_FROM_LIP MSR_LASTBRANCH_8_FROM_LIP MSR_LASTBRANCH_9_FROM_LIP MSR_LASTBRANCH_10_FROM_LIP MSR_LASTBRANCH_11_FROM_LIP MSR_LASTBRANCH_12_FROM_LIP MSR_LASTBRANCH_13_FROM_LIP MSR_LASTBRANCH_14_FROM_LIP MSR_LASTBRANCH_15_FROM_LIP	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
LBR MSRs for Family 0FH, Model 03H; MSRs at locations 6C0H-6CFH.	
MSR_LASTBRANCH_0_TO_LIP MSR_LASTBRANCH_1_TO_LIP MSR_LASTBRANCH_2_TO_LIP MSR_LASTBRANCH_3_TO_LIP MSR_LASTBRANCH_4_TO_LIP MSR_LASTBRANCH_5_TO_LIP MSR_LASTBRANCH_6_TO_LIP MSR_LASTBRANCH_7_TO_LIP MSR_LASTBRANCH_8_TO_LIP MSR_LASTBRANCH_9_TO_LIP MSR_LASTBRANCH_10_TO_LIP MSR_LASTBRANCH_11_TO_LIP MSR_LASTBRANCH_12_TO_LIP MSR_LASTBRANCH_13_TO_LIP MSR_LASTBRANCH_14_TO_LIP MSR_LASTBRANCH_15_TO_LIP	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

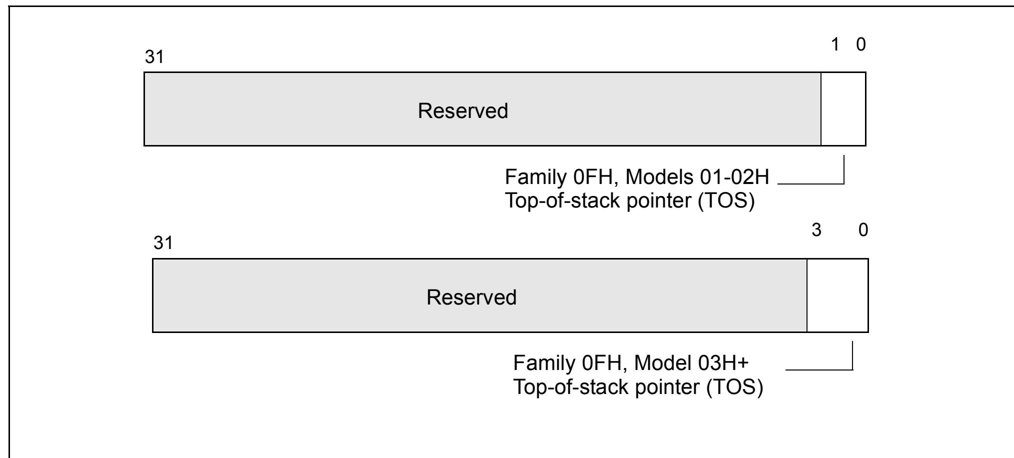

Figure 15-3. MSR_LASTBRANCH_TOS MSR Layout for the Pentium 4 and Intel Xeon Processor Family

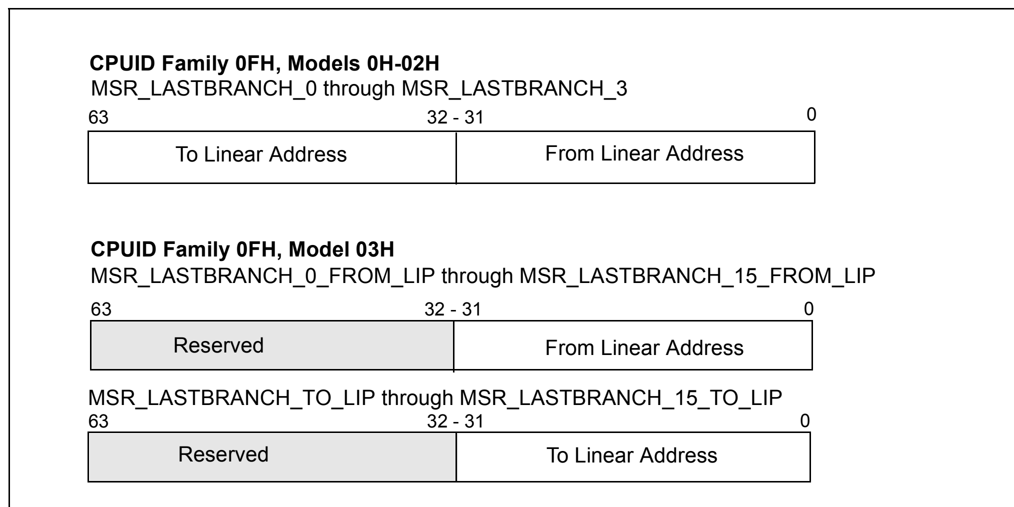

Figure 15-4. LBR MSR Branch Record Layout for the Pentium 4 and Intel Xeon Processor Family

Volume 3, Appendix B, Table B-1 has also been updated to reflect new LBR stack information. Impacted cells are reproduced below.

--

Table B-1. MSRs in the Pentium 4 and Intel Xeon Processors

Register Address		Register Name Fields and Flags	Model Avail-ability	Shared/ Unique[1]	Bit Description
Hex	Dec				
1DAH	474	MSR_LASTBRANCH _TOS	0, 1, 2, 3	Unique	**Last Branch Record Stack TOS.** (R) Contains an index (0-3 or 0-15) that points to the top of the last branch record stack (that is, that points the index of the MSR containing the most recent branch record.
1DBH	475	MSR_LASTBRANCH _0	0, 1, 2	Unique	**Last Branch Record 0.** (R/W) One of four last branch record registers on the last branch record stack. It contains pointers to the source and destination instruction for one of the last four branches, exceptions, or interrupts that the processor took. NOTE: MSR_LASTBRANCH_0 through MSR_LASTBRANCH_3 at 1DBH-1DEH are available only on family 0FH, models 0H-02H. They have been replaced by the MSRs at 680H-68FH and 6C0H-6CFH..
1DCH	476	MSR_LASTBRANCH _1	0, 1, 2	Unique	**Last Branch Record 1.** See description of the MSR_LASTBRANCH_0 MSR at 1DBH.
1DDH	477	MSR_LASTBRANCH _2	0, 1, 2	Unique	**Last Branch Record 2.** See description of the MSR_LASTBRANCH_0 MSR at 1DBH.
1DEH	478	MSR_LASTBRANCH _3	0, 1, 2	Unique	**Last Branch Record 3.** See description of the MSR_LASTBRANCH_0 MSR at 1DBH.
680H	1664	MSR_LASTBRANCH _0_FROM_LIP	3	Unique	**Last Branch Record 0.** (R/W) One of 16 pairs of last branch record registers on the last branch record stack (680H-68FH). This part of the stack contains pointers to the **source instruction** for one of the last 16 branches, exceptions, or interrupts taken by the processor. NOTE: The MSRs at 680H-68FH, 6C0H-6CfH are not available in processor releases before family 0FH, model 03H. These MSRs replace MSRs previously located at 1DBH-1DEH.which performed the same function for early releases.
681H	1665	MSR_LASTBRANCH _1_FROM_LIP	3	Unique	**Last Branch Record 1.** See description of MSR_LASTBRANCH_0 at 680H.
682H	1666	MSR_LASTBRANCH _2_FROM_LIP	3	Unique	**Last Branch Record 2.** See description of MSR_LASTBRANCH_0 at 680H.

Table B-1. MSRs in the Pentium 4 and Intel Xeon Processors (Continued)

Register Address		Register Name Fields and Flags	Model Avail-ability	Shared/ Unique[1]	Bit Description
Hex	Dec				
683H	1667	MSR_LASTBRANCH _3_FROM_LIP	3	Unique	**Last Branch Record 3.** See description of MSR_LASTBRANCH_0 at 680H.
684H	1668	MSR_LASTBRANCH _4_FROM_LIP	3	Unique	**Last Branch Record 4.** See description of MSR_LASTBRANCH_0 at 680H.
685H	1669	MSR_LASTBRANCH _5_FROM_LIP	3	Unique	**Last Branch Record 5.** See description of MSR_LASTBRANCH_0 at 680H.
686H	1670	MSR_LASTBRANCH _6_FROM_LIP	3	Unique	**Last Branch Record 6.** See description of MSR_LASTBRANCH_0 at 680H.
687H	1671	MSR_LASTBRANCH _7_FROM_LIP	3	Unique	**Last Branch Record 7.** See description of MSR_LASTBRANCH_0 at 680H.
688H	1672	MSR_LASTBRANCH _8_FROM_LIP	3	Unique	**Last Branch Record 8.** See description of MSR_LASTBRANCH_0 at 680H.
689H	1673	MSR_LASTBRANCH _9_FROM_LIP	3	Unique	**Last Branch Record 9.** See description of MSR_LASTBRANCH_0 at 680H.
68AH	1674	MSR_LASTBRANCH _10_FROM_LIP	3	Unique	**Last Branch Record 10.** See description of MSR_LASTBRANCH_0 at 680H.
68BH	1675	MSR_LASTBRANCH _11_FROM_LIP	3	Unique	**Last Branch Record 11.** See description of MSR_LASTBRANCH_0 at 680H.
68CH	1676	MSR_LASTBRANCH _12_FROM_LIP	3	Unique	**Last Branch Record 12.** See description of MSR_LASTBRANCH_0 at 680H.
68DH	1677	MSR_LASTBRANCH _13_FROM_LIP	3	Unique	**Last Branch Record 13.** See description of MSR_LASTBRANCH_0 at 680H.
68EH	1678	MSR_LASTBRANCH _14_FROM_LIP	3	Unique	**Last Branch Record 14.** See description of MSR_LASTBRANCH_0 at 680H.
68FH	1679	MSR_LASTBRANCH _15_FROM_LIP	3	Unique	**Last Branch Record 15.** See description of MSR_LASTBRANCH_0 at 680H.
6C0H	1728	MSR_LASTBRANCH _0_TO_LIP	3	Unique	**Last Branch Record 0.** (R/W) One of 16 pairs of last branch record registers on the last branch record stack (6C0H-6CFH). This part of the stack contains pointers to the **destination instruction** for one of the last 16 branches, exceptions, or interrupts that the processor took.

Table B-1. MSRs in the Pentium 4 and Intel Xeon Processors (Continued)

Register Address		Register Name Fields and Flags	Model Avail-ability	Shared/ Unique[1]	Bit Description
Hex	Dec				
6C1H	1729	MSR_LASTBRANCH _1_TO_LIP	3	Unique	**Last Branch Record 1.** See description of MSR_LASTBRANCH_0 at 6C0H.
6C2H	1730	MSR_LASTBRANCH _2_TO_LIP	3	Unique	**Last Branch Record 2.** See description of MSR_LASTBRANCH_0 at 6C0H.
6C3H	1731	MSR_LASTBRANCH _3_TO_LIP	3	Unique	**Last Branch Record 3.** See description of MSR_LASTBRANCH_0 at 6C0H.
6C4H	1732	MSR_LASTBRANCH _4_TO_LIP	3	Unique	**Last Branch Record 4.** See description of MSR_LASTBRANCH_0 at 6C0H.
6C5H	1733	MSR_LASTBRANCH _5_TO_LIP	3	Unique	**Last Branch Record 5.** See description of MSR_LASTBRANCH_0 at 6C0H.
6C6H	1734	MSR_LASTBRANCH _6_TO_LIP	3	Unique	**Last Branch Record 6.** See description of MSR_LASTBRANCH_0 at 6C0H.
6C7H	1735	MSR_LASTBRANCH _7_TO_LIP	3	Unique	**Last Branch Record 7.** See description of MSR_LASTBRANCH_0 at 6C0H.
6C8H	1736	MSR_LASTBRANCH _8_TO_LIP	3	Unique	**Last Branch Record 8.** See description of MSR_LASTBRANCH_0 at 6C0H.
6C9H	1737	MSR_LASTBRANCH _9_TO_LIP	3	Unique	**Last Branch Record 9.** See description of MSR_LASTBRANCH_0 at 6C0H.
6CAH	1738	MSR_LASTBRANCH _10_TO_LIP	3	Unique	**Last Branch Record 10.** See description of MSR_LASTBRANCH_0 at 6C0H.
6CBH	1739	MSR_LASTBRANCH _11_TO_LIP	3	Unique	**Last Branch Record 11.** See description of MSR_LASTBRANCH_0 at 6C0H.
6CCH	1740	MSR_LASTBRANCH _12_TO_LIP	3	Unique	**Last Branch Record 12.** See description of MSR_LASTBRANCH_0 at 6C0H.
6CDH	1741	MSR_LASTBRANCH _13_TO_LIP	3	Unique	**Last Branch Record 13.** See description of MSR_LASTBRANCH_0 at 6C0H.
6CEH	1742	MSR_LASTBRANCH _14_TO_LIP	3	Unique	**Last Branch Record 14.** See description of MSR_LASTBRANCH_0 at 6C0H.
6CFH	1743	MSR_LASTBRANCH _15_TO_LIP	3	Unique	**Last Branch Record 15.** See description of MSR_LASTBRANCH_0 at 6C0H.

20. Limited Availability of Two MSR's Have Been Documented

A note has been added to Volume 3, Chapter 15, Table 15-4. The note indicates the availability of MSR_IQ_ESCR0 and MSR_IQ_ESCR1. The impacted table cells are reproduced below.

Counter			CCCR		ESCR		
Name	**No.**	**Addr**	**Name**	**Addr**	**Name**	**No.**	**Addr**
MSR_IQ_COUNTER0	12	30CH	MSR_IQ_CCCR0	36CH	MSR_CRU_ESCR0 MSR_CRU_ESCR2 MSR_CRU_ESCR4 MSR_IQ_ESCR0[1] MSR_RAT_ESCR0 MSR_SSU_ESCR0 MSR_ALF_ESCR0	4 5 6 0 2 3 1	3B8H 3CCH 3E0H 3BAH 3BCH 3BEH 3CAH
MSR_IQ_COUNTER1	13	30DH	MSR_IQ_CCCR1	36DH	MSR_CRU_ESCR0 MSR_CRU_ESCR2 MSR_CRU_ESCR4 MSR_IQ_ESCR0[1] MSR_RAT_ESCR0 MSR_SSU_ESCR0 MSR_ALF_ESCR0	4 5 6 0 2 3 1	3B8H 3CCH 3E0H 3BAH 3BCH 3BEH 3CAH
MSR_IQ_COUNTER2	14	30EH	MSR_IQ_CCCR2	36EH	MSR_CRU_ESCR1 MSR_CRU_ESCR3 MSR_CRU_ESCR5 MSR_IQ_ESCR1[1] MSR_RAT_ESCR1 MSR_ALF_ESCR1	4 5 6 0 2 1	3B9H 3CDH 3E1H 3BBH 3BDH 3CBH
MSR_IQ_COUNTER3	15	30FH	MSR_IQ_CCCR3	36FH	MSR_CRU_ESCR1 MSR_CRU_ESCR3 MSR_CRU_ESCR5 MSR_IQ_ESCR1[1] MSR_RAT_ESCR1 MSR_ALF_ESCR1	4 5 6 0 2 1	3B9H 3CDH 3E1H 3BBH 3BDH 3CBH
MSR_IQ_COUNTER4	16	310H	MSR_IQ_CCCR4	370H	MSR_CRU_ESCR0 MSR_CRU_ESCR2 MSR_CRU_ESCR4 MSR_IQ_ESCR0[1] MSR_RAT_ESCR0 MSR_SSU_ESCR0 MSR_ALF_ESCR0	4 5 6 0 2 3 1	3B8H 3CCH 3E0H 3BAH 3BCH 3BEH 3CAH
MSR_IQ_COUNTER5	17	311H	MSR_IQ_CCCR5	371H	MSR_CRU_ESCR1 MSR_CRU_ESCR3 MSR_CRU_ESCR5 MSR_IQ_ESCR1[1] MSR_RAT_ESCR1 MSR_ALF_ESCR1	4 5 6 0 2 1	3B9H 3CDH 3E1H 3BBH 3BDH 3CBH
[1] MSR_IQ_ESCR0 and MSR_IQ_ESCR1 are available only on early processor builds (family 0FH, models 01H-02H). These MSRs are not available on later versions.							

21. The Microcode Update Facilities Section Has Been Updated

Volume 3 of the IA-32 Intel Architecture Software Developer's Manual, Section 9.11 has been updated. The new information has been added that documents the microcode update facilities added on new processors.

9.11. MICROCODE UPDATE FACILITIES

The Pentium 4, Intel Xeon, and P6 family processors have the capability to correct errata by loading an Intel-supplied data block into the processor. The data block is called a microcode update. This section describes the mechanisms the BIOS needs to provide in order to use this feature during system initialization. It also describes a specification that permits the incorporation of future updates into a system BIOS.

Intel considers the release of a microcode update for a silicon revision to be the equivalent of a processor stepping and completes a full-stepping level validation for releases of microcode updates.

A microcode update is used to correct errata in the processor. The BIOS, which has an update loader, is responsible for loading the update on processors during system initialization (Figure 9-7). There are two steps to this process: the first is to incorporate the necessary update data blocks into the BIOS; the second is to load update data blocks into the processor.

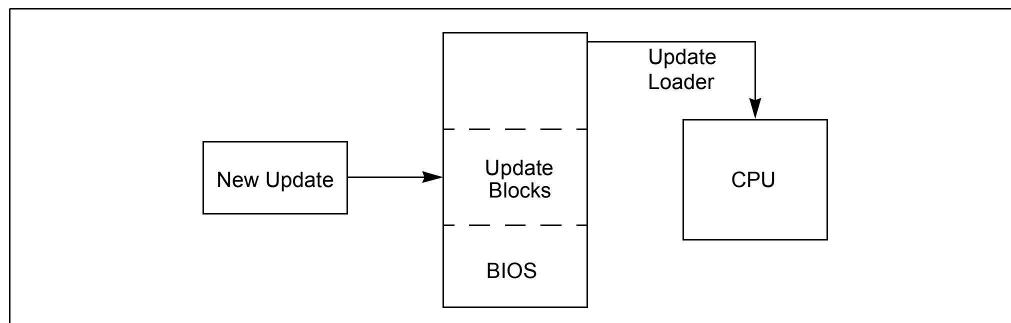

Figure 9-7. Applying Microcode Updates

9.11.1. Microcode Update

A microcode update consists of an Intel-supplied binary that contains a descriptive header and data. No executable code resides within the update. Each microcode update is tailored for a specific list of processor signatures. A mismatch of the processor's signature with the signature contained in the update will result in a failure to load. A processor signature includes the extended family, extended model, type, family, model, and stepping of the processor (starting with processor family 0fH, model 03H, a given microcode update may be associated with one of multiple processor signatures; see Section 9.11.2. for detail).

Microcode updates are composed of a multi-byte header, followed by encrypted data and then by an optional extended signature table. Table 9-1 provides a definition of the fields; Table 9-2 shows the format of an update.

The header is 48 bytes. The first 4 bytes of the header contain the header version. The update header and its reserved fields are interpreted by software based upon the header version. An encoding scheme guards against tampering and provides a means for determining the authenticity of any given

update. For microcode updates with a data size field equal to 00000000H, the size of the microcode update is 2048 bytes. The first 48 bytes contain the microcode update header. The remaining 2000 bytes contain encrypted data.

For microcode updates with a data size not equal to 00000000H, the total size field specifies the size of the microcode update. The first 48 bytes contain the microcode update header. The second part of the microcode update is the encrypted data. The data size field of the microcode update header specifies the encrypted data size, its value must be a multiple of the size of DWORD. The optional extended signature table if implemented follows the encrypted data, and its size is calculated by (Total Size – (Data Size + 48)).

NOTE

The optional extended signature table is supported starting with processor family 0FH, model 03H.

.

Table 9-1. Microcode Update Field Definitions

Field Name	Offset (bytes)	Length (bytes)	Description
Header Version	0	4	Version number of the update header.
Update Revision	4	4	Unique version number for the update, the basis for the update signature provided by the processor to indicate the current update functioning within the processor. Used by the BIOS to authenticate the update and verify that the processor loads successfully. The value in this field cannot be used for processor stepping identification alone. This is a signed 32-bit number.
Date	8	4	Date of the update creation in binary format: mmddyyyy (e.g. 07/18/98 is 07181998H).
Processor Signature	12	4	*Extended family, extended model, type, family, model*, and *stepping* of processor that requires this particular update revision (e.g., 00000650H). Each microcode update is designed specifically for a given extended family, extended model, *type, family, model*, and *stepping* of the processor. The BIOS uses the processor signature field in conjunction with the CPUID instruction to determine whether or not an update is appropriate to load on a processor. The information encoded within this field exactly corresponds to the bit representations returned by the CPUID instruction.
Checksum	16	4	Checksum of update data and header. Used to verify the integrity of the update header and data. Checksum is correct when the summation of the DWORDs that comprise the microcode update results in 00000000H.
Loader Revision	20	4	Version number of the loader program needed to correctly load this update. The initial version is 00000001H.
Processor Flags	24	4	Platform type information is encoded in the lower 8 bits of this 4-byte field. Each bit represents a particular platform type for a given CPUID. The BIOS uses the processor flags field in conjunction with the platform Id bits in MSR (17H) to determine whether or not an update is appropriate to load on a processor. Multiple bits may be set representing support for multiple platform IDs.

Table 9-1. Microcode Update Field Definitions (Continued)

Field Name	**Offset (bytes)**	**Length (bytes)**	**Description**
Data Size	28	4	Specifies the size of the encrypted data in bytes, and must be a multiple of DWORDs. If this value is 00000000H, then the microcode update encrypted data is 2000 bytes (or 500 DWORDs).
Total Size	32	4	Specifies the total size of the microcode update in bytes. It is the summation of the header size, the encrypted data size and the size of the optional extended signature table.
Reserved	36	12	Reserved fields for future expansion
Update Data	48	Data Size or 2000	Update data
Extended Signature Count	Data Size + 48	4	Specifies the number of extended signature structures (Processor Signature[n], processor flags[n] and checksum[n]) that exist in this microcode update.
Extended Checksum	Data Size + 52	4	Checksum of update extended processor signature table. Used to verify the integrity of the extended processor signature table. Checksum is correct when the summation of the DWORDs that comprise the extended processor signature table results in 00000000H.
Reserved	Data Size + 56	12	Reserved fields
Processor Signature[n]	Data Size + 68 + (n * 12)	4	*Extended family, extended model, type, family, model*, and *stepping* of processor that requires this particular update revision (e.g., 00000650H). Each microcode update is designed specifically for a given extended family, extended model, *type, family, model*, and *stepping* of the processor. The BIOS uses the processor signature field in conjunction with the CPUID instruction to determine whether or not an update is appropriate to load on a processor. The information encoded within this field exactly corresponds to the bit representations returned by the CPUID instruction.
Processor Flags[n]	Data Size + 72 + (n * 12)	4	Platform type information is encoded in the lower 8 bits of this 4-byte field. Each bit represents a particular platform type for a given CPUID. The BIOS uses the processor flags field in conjunction with the platform Id bits in MSR (17H) to determine whether or not an update is appropriate to load on a processor. Multiple bits may be set representing support for multiple platform IDs.
Checksum[n]	Data Size + 76 + (n * 12)	4	Used by utility software to decompose a microcode update into multiple microcode updates where each of the new microcode updates is constructed without the optional extended processor signature table.

Table 9-2. Microcode Update Format

<table>
<tr><th colspan="9">31   24   16   8   0</th><th>Bytes</th></tr>
<tr><td colspan="9">Header Version</td><td>0</td></tr>
<tr><td colspan="9">Update Revision</td><td>4</td></tr>
<tr><td colspan="2">Month: 8</td><td colspan="2">Day: 8</td><td colspan="5">Year: 16</td><td>8</td></tr>
<tr><td colspan="9">Processor Signature (CPUID)</td><td rowspan="2">12</td></tr>
<tr><td>Res: 4</td><td colspan="2">Extended Family: 8</td><td>Extended Mode: 4</td><td>Reserved: 2</td><td>Type: 2</td><td>Family: 4</td><td>Model: 4</td><td>Stepping: 4</td></tr>
<tr><td colspan="9">Checksum</td><td>16</td></tr>
<tr><td colspan="9">Loader Revision</td><td>20</td></tr>
<tr><td colspan="9">Processor Flags</td><td rowspan="2">24</td></tr>
<tr><td>Reserved (24 bits)</td><td>P7</td><td>P6</td><td>P5</td><td>P4</td><td>P3</td><td>P2</td><td>P1</td><td>P0</td></tr>
<tr><td colspan="9">Data Size</td><td>28</td></tr>
<tr><td colspan="9">Total Size</td><td>32</td></tr>
<tr><td colspan="9">Reserved (12 Bytes)</td><td>36</td></tr>
<tr><td colspan="9">Update Data (Data Size bytes, or 2000 Bytes if Data Size = 00000000H)</td><td>48</td></tr>
<tr><td colspan="9">Extended Signature Count ‘n’</td><td>Data Size + 48</td></tr>
<tr><td colspan="9">Extended Processor Signature Table Checksum</td><td>Data Size + 52</td></tr>
<tr><td colspan="9">Reserved (12 Bytes)</td><td>Data Size + 56</td></tr>
<tr><td colspan="9">Processor Signature[n]</td><td>Data Size + 68 + (n * 12)</td></tr>
<tr><td colspan="9">Processor Flags[n]</td><td>Data Size + 72 + (n * 12)</td></tr>
<tr><td colspan="9">Checksum[n]</td><td>Data Size + 76 + (n * 12)</td></tr>
</table>

9.11.2. Optional Extended Signature Table

The extended signature table is a structure that may be appended to the end of the encrypted data when the encrypted data only supports a single processor signature (optional case). The extended signature table will always be present when the encrypted data supports multiple processor steppings and/or models (required case).

The extended signature table consists of a 20-byte extended signature header structure, which contains the extended signature count, the extended processor signature table checksum, and 12 reserved bytes (Table 9-3). Following the extended signature header structure, the extended signature table contains 0-to-n extended processor signature structures.

Each processor signature structure consist of the processor signature, processor flags, and a checksum (Table 9-4).

The extended signature count in the extended signature header structure indicates the number of processor signature structures that exist in the extended signature table.

The extended processor signature table checksum is a checksum of all DWORDs that comprise the extended signature table. That includes the extended signature count, extended processor signature table checksum, 12 reserved bytes and the n processor signature structures. A valid extended signature table exists when the result of a DWORD checksum is 00000000H.

Table 9-3. Extended Processor Signature Table Header Structure

Extended Signature Count 'n'	Data Size + 48
Extended Processor Signature Table Checksum	Data Size + 52
Reserved (12 Bytes)	Data Size + 56

Table 9-4. Processor Signature Structure

Processor Signature[n]	Data Size + 68 + (n * 12)
Processor Flags[n]	Data Size + 72 + (n * 12)
Checksum[n]	Data Size + 76 + (n * 12)

9.11.3. Processor Identification

Each microcode update is designed to for a specific processor or set of processors. To determine the correct microcode update to load, software must ensure that one of the processor signatures embedded in the microcode update matches the 32-bit processor signature returned by the CPUID instruction when executed by the target processor with EAX = 1. Attempting to load a microcode update that does not match a processor signature embedded in the microcode update with the processor signature returned by CPUID will cause the processor to reject the update.

Example 9-1 shows how to check for a valid processor signature match between the processor and microcode update.

Example 9-1. Pseudo Code to Validate the Processor Signature

```
ProcessorSignature ← CPUID(1):EAX

If (Update.HeaderVersion == 00000001h)
{
        // first check the ProcessorSignature field
    If (ProcessorSignature == Update.ProcessorSignature)
        Success

        // if extended signature is present
    Else If (Update.TotalSize > (Update.DataSize + 48))
    {

        //
        // Assume the Data Size has been used to calculate the
        // location of Update.ProcessorSignature[0].
        //

        For (N ← 0; ((N < Update.ExtendedSignatureCount) AND
```

```
            (ProcessorSignature != Update.ProcessorSignature[N])); N++);

            // if the loops ended when the iteration count is
            // less than the number of processor signatures in
            // the table, we have a match
        If (N < Update.ExtendedSignatureCount)
            Success
        Else
            Fail
    }
    Else
        Fail
Else
    Fail
```

9.11.4. Platform Identification

In addition to verifying the processor signature, the intended processor platform type must be determined to properly target the microcode update. The intended processor platform type is determined by reading the IA32_PLATFORM_ID register, (MSR 17H). This 64-bit register must be read using the RDMSR instruction.

The three platform ID bits, when read as a binary coded decimal (BCD) number, indicate the bit position in the microcode update header's processor flags field associated with the installed processor. The processor flags in the 48-byte header and the processor flags field associated with the extended processor signature structures may have multiple bits set. Each set bit represents a different platform ID that the update supports.

Register Name: **IA32_PLATFORM_ID**

MSR Address: 017H
Access: Read Only

IA32_PLATFORM_ID is a 64-bit register accessed only when referenced as a Qword through a RDMSR instruction.

Table 9-5. Processor Flags

Bit	Descriptions
63:53	Reserved
52:50	Platform Id Bits (RO). The field gives information concerning the intended platform for the processor. See also Table 9-2. **52 51 50** 0 0 0 Processor Flag 0 0 0 1 Processor Flag 1 0 1 0 Processor Flag 2 0 1 1 Processor Flag 3 1 0 0 Processor Flag 4 1 0 1 Processor Flag 5 1 1 0 Processor Flag 6 1 1 1 Processor Flag 7
49:0	Reserved

To validate the platform information, software may implement an algorithm similar to the algorithms in Example 9-2.

Example 9-2. Pseudo Code Example of Processor Flags Test

```
Flag ← 1 << IA32_PLATFORM_ID[52:50]

If (Update.HeaderVersion == 00000001h)
{
    If (Update.ProcessorFlags & Flag)
    {
        Load Update
    }
    Else
    {

        //
        // Assume the Data Size has been used to calculate the
        // location of Update.ProcessorSignature[N] and a match
        // on Update.ProcessorSignature[N] has already succeeded
        //

        If (Update.ProcessorFlags[n] & Flag)
        {
            Load Update
        }
    }
}
```

9.11.5. Microcode Update Checksum

Each microcode update contains a DWORD checksum located in the update header. It is software's responsibility to ensure that a microcode update is not corrupt. To check for a corrupt microcode update, software must perform a unsigned DWORD (32-bit) checksum of the microcode update. Even though some fields are signed, the checksum procedure treats all DWORDs as unsigned. Microcode updates with a header version equal to 00000001H must sum all DWORDs that comprise the microcode update. A valid checksum check will yield a value of 00000000H. Any other value indicates the microcode update is corrupt and should not be loaded.

The checksum algorithm shown by the pseudo code in Example 9-3 treats the microcode update as an array of unsigned DWORDs. If the data size DWORD field at byte offset 32 equals 00000000H, the size of the encrypted data is 2000 bytes, resulting in 500 DWORDs. Otherwise the microcode update size in DWORDs = (*Total Size / 4*).

Example 9-3. Pseudo Code Example of Checksum Test

```
N ← 512

If (Update.DataSize != 00000000H)
    N ← Update.TotalSize / 4

ChkSum ← 0
For (I ← 0; I < N; I++)
{
    ChkSum ← ChkSum + MicrocodeUpdate[I]
}

If (ChkSum == 00000000H)
    Success
Else
    Fail
```

9.11.6. Microcode Update Loader

This section describes an update loader used to load an update into a Pentium 4, Intel Xeon, or P6 family processor. It also discusses the requirements placed on the BIOS to ensure proper loading. The update loader described contains the minimal instructions needed to load an update. The specific instruction sequence that is required to load an update is dependent upon the loader revision field contained within the update header. This revision is expected to change infrequently (potentially, only when new processor models are introduced).

Example 9-4 below represents the update loader with a loader revision of 00000001H. Note that the microcode update must be aligned on a 16-byte boundary.

Example 9-4. Assembly Code Example of Simple Microcode Update Loader

```
mov ecx,79h     ; MSR to read in ECX
xor eax,eax     ; clear EAX
xor ebx,ebx     ; clear EBX
mov ax,cs       ; Segment of microcode update
shl eax,4
mov bx,offset Update; Offset of microcode update
add eax,ebx     ; Linear Address of Update in EAX
add eax,48d     ; Offset of the Update Data within the Update
xor edx,edx     ; Zero in EDX
WRMSR           ; microcode update trigger
```

The loader shown in Example 9-4 assumes that *update* is the address of a microcode update (header and data) embedded within the code segment of the BIOS. It also assumes that the processor is operating in real mode. The data may reside anywhere in memory, aligned on a 16-byte boundary, that is accessible by the processor within its current operating mode (real, protected).

Before the BIOS executes the microcode update trigger (WRMSR) instruction, the following must be true:

- EAX contains the linear address of the start of the update data
- EDX contains zero
- ECX contains 79H (address of IA32_BIOS_UPDT_TRIG)

Other requirements are:

- If the update is loaded while the processor is in real mode, then the update data may not cross a segment boundary.
- If the update is loaded while the processor is in real mode, then the update data may not exceed a segment limit.
- If paging is enabled, pages that are currently present must map the update data.
- The microcode update data requires a 16-byte boundary alignment.

9.11.6.1. Hard Resets in Update Loading

The effects of a loaded update are cleared from the processor upon a hard reset. Therefore, each time a hard reset is asserted during the BIOS POST, the update must be reloaded on all processors that observed the reset. The effects of a loaded update are, however, maintained across a processor INIT. There are no side effects caused by loading an update into a processor multiple times.

9.11.6.2. Update in a Multiprocessor System

A multiprocessor (MP) system requires loading each processor with update data appropriate for its CPUID and platform ID bits. The BIOS is responsible for ensuring that this requirement is met and that the loader is located in a module executed by all processors in the system. If a system design permits multiple steppings of Pentium 4, Intel Xeon, and P6 family processors to exist concurrently; then the BIOS must verify individual processors against the update header information to ensure appropriate loading. Given these considerations, it is most practical to load the update during MP initialization.

9.11.6.3. Update in a System with Intel HT Technology

Intel Hyper-Threading Technology (HT Technology) has implications on the loading of the microcode update. The update must be loaded for each core in a physical processor. Thus, for a processor with HT Technology, only one logical processor per core is required to load the microcode update. Each individual logical processor can independently load the update. However, MP initialization must provide some mechanism (e.g. a software semaphore) to force serialization of microcode update loads and to prevent simultaneous load attempts to the same core.

9.11.6.4. Update Loader Enhancements

The update loader presented in Section 9.11.6., *Microcode Update Loader* is a minimal implementation that can be enhanced to provide additional functionality. Potential enhancements are described below:

- BIOS can incorporate multiple updates to support multiple steppings of the Pentium 4, Intel Xeon, and P6 family processors. This feature provides for operating in a mixed stepping environment on an MP system and enables a user to upgrade to a later version of the processor. In this case, modify the loader to check the CPUID and platform ID bits of the processor that it is running on against the available headers before loading a particular update. The number of updates is only limited by available BIOS space.
- A loader can load the update and test the processor to determine if the update was loaded correctly. See Section 9.11.7., *Update Signature and Verification*.
- A loader can verify the integrity of the update data by performing a checksum on the double words of the update summing to zero. See Section 9.11.5., *Microcode Update Checksum*.
- A loader can provide power-on messages indicating successful loading of an update.

9.11.7. Update Signature and Verification

The Pentium 4, Intel Xeon, and P6 family processors provide capabilities to verify the authenticity of a particular update and to identify the current update revision. This section describes the model-specific extensions of processors that support this feature. The update verification method below assumes that the BIOS will only verify an update that is more recent than the revision currently loaded in the processor.

CPUID returns a value in a model specific register in addition to its usual register return values. The semantics of CPUID cause it to deposit an update ID value in the 64-bit model-specific register at address 08BH (IA32_BIOS_SIGN_ID). If no update is present in the processor, the value in the MSR remains unmodified. The BIOS must pre-load a zero into the MSR before executing CPUID. If a read of the MSR at 8BH still returns zero after executing CPUID, this indicates that no update is present.

The update ID value returned in the EDX register after RDMSR executes indicates the revision of the update loaded in the processor. This value, in combination with the CPUID value returned in the EAX register, uniquely identifies a particular update. The signature ID can be directly compared with the update revision field in a microcode update header for verification of a correct load. No consecutive updates released for a given stepping of a processor may share the same signature. The processor signature returned by CPUID differentiates updates for different steppings.

9.11.7.1. Determining the Signature

An update that is successfully loaded into the processor provides a signature that matches the update revision of the currently functioning revision. This signature is available any time after the actual update has been loaded. Requesting the signature does not have a negative impact upon a loaded update.

The procedure for determining this signature shown in Example 9-5.

Example 9-5. Assembly Code to Retrieve the Update Revision

```
MOV     ECX, 08BH;IA32_BIOS_SIGN_ID
XOR     EAX, EAX;clear EAX
XOR     EDX, EDX;clear EDX
WRMSR   ;Load 0 to MSR at 8BH
MOV     EAX, 1
cpuid
MOV     ECX, 08BH;IA32_BIOS_SIGN_ID
rdmsr   ;Read Model Specific Register
```

If there is an update active in the processor, its revision is returned in the EDX register after the RDMSR instruction executes.

IA32_BIOS_SIGN_ID	**Microcode Update Signature Register**
MSR Address:	08BH Accessed as a Qword
Default Value:	XXXX XXXX XXXX XXXXh
Access:	Read/Write

The IA32_BIOS_SIGN_ID register is used to report the microcode update signature when CPUID executes. The signature is returned in the upper DWORD (Table 9-6).

Table 9-6. Microcode Update Signature

Bit	Description
63:32	Microcode update signature. This field contains the signature of the currently loaded microcode update when read following the execution of the CPUID instruction, function 1. It is required that this register field be pre-loaded with zero prior to executing the CPUID, function 1. If the field remains equal to zero, then there is no microcode update loaded. Another non-zero value will be the signature.
31:0	Reserved.

9.11.7.2. Authenticating the Update

An update may be authenticated by the BIOS using the signature primitive, described above, and the algorithm in Example 9-6.

Example 9-6. Pseudo Code to Authenticate the Update

```
Z ← Obtain Update Revision from the Update Header to be authenticated;
X ← Obtain Current Update Signature from MSR 8BH;

If (Z > X)
{
    Load Update that is to be authenticated;
    Y ← Obtain New Signature from MSR 8BH;

    If (Z == Y)
        Success
    Else
        Fail
}
Else
    Fail
```

Example 9-6 requires that the BIOS only authenticate updates that contain a numerically larger revision than the currently loaded revision, where Current Signature (X) < New Update Revision (Z). A processor with no loaded update is considered to have a revision equal to zero.

This authentication procedure relies upon the decoding provided by the processor to verify an update from a potentially hostile source. As an example, this mechanism in conjunction with other safeguards provides security for dynamically incorporating field updates into the BIOS.

9.11.8. Pentium 4, Intel Xeon, and P6 Family Processor Microcode Update Specifications

This section describes the interface that an application can use to dynamically integrate processor-specific updates into the system BIOS. In this discussion, the application is referred to as the calling program or caller.

The real mode INT15 call specification described here is an Intel extension to an OEM BIOS. This extension allows an application to read and modify the contents of the microcode update data in NVRAM. The update loader, which is part of the system BIOS, cannot be updated by the interface. All of the functions defined in the specification must be implemented for a system to be considered compliant with the specification. The INT15 functions are accessible only from real mode.

9.11.8.1. Responsibilities of the BIOS

If a BIOS passes the presence test (INT 15H, AX = 0D042H, BL = 0H), it must implement all of the sub-functions defined in the INT 15H, AX = 0D042H specification. There are no optional functions. BIOS must load the appropriate update for each processor during system initialization.

A Header Version of an update block containing the value 0FFFFFFFFH indicates that the update block is unused and available for storing a new update.

The BIOS is responsible for providing a region of non-volatile storage (NVRAM) for each potential processor stepping within a system. This storage unit consists of one or more update blocks. An update block is a contiguous 2048-byte block of memory. The BIOS for a single processor system need only provide update blocks to store one microcode update. If the BIOS for a multiple processor system is intended to support mixed processor steppings, then the BIOS needs to provide enough update blocks to store each unique microcode update or for each processor socket on the OEM's system board.

The BIOS is responsible for managing the NVRAM update blocks. This includes garbage collection, such as removing microcode updates that exist in NVRAM for which a corresponding processor does not exist in the system. This specification only provides the mechanism for ensuring security, the uniqueness of an entry, and that stale entries are not loaded. The actual update block management is implementation specific on a per-BIOS basis.

As an example, the BIOS may use update blocks sequentially in ascending order with CPU signatures sorted versus the first available block. In addition, garbage collection may be implemented as a setup option to clear all NVRAM slots or as BIOS code that searches and eliminates unused entries during boot.

NOTE

For IA-32 processors starting with family 0FH and model 03H, the microcode update may be as large as 16 KBytes. Thus, BIOS must allocate 8 update blocks for each microcode update. In a MP system, a common microcode update may be sufficient for each socket in the system.

For IA-32 processors earlier than family 0FH and model 03H, the microcode update is 2 KBytes. An MP-capable BIOS that supports multiple steppings must allocate a block for each socket in the system.

A single-processor BIOS that supports variable-sized microcode update and fixed-sized microcode update must allocate one 16 KByte region and a second region of at least 2 KBytes.

The following algorithm (Example 9-7) describes the steps performed during BIOS initialization used to load the updates into the processor(s). The algorithm assumes:

- The BIOS ensures that no update contained within NVRAM has a header version or loader version that does not match one currently supported by the BIOS.
- The update contains a correct checksum.
- The BIOS ensures that (at most) one update exists for each processor stepping.
- Older update revisions are not allowed to overwrite more recent ones.

These requirements are checked by the BIOS during the execution of the write update function of this interface. The BIOS sequentially scans through all of the update blocks in NVRAM starting with index 0. The BIOS scans until it finds an update where the processor fields in the header match the processor signature (extended family, extended model, type, family, model, and stepping) as well as the platform bits of the current processor.

Example 9-7. Pseudo Code, Checks Required Prior to Loading an Update

```
For each processor in the system
{
    Determine the Processor Signature via CPUID function 1;
    Determine the Platform Bits ← 1 << IA32_PLATFORM_ID[52:50];

    For (I ← UpdateBlock 0, I < NumOfBlocks; I++)
    {
        If (Update.Header_Version == 0x00000001)
        {
            If ((Update.ProcessorSignature == Processor Signature) &&
                 (Update.ProcessorFlags & Platform Bits))
            {
                Load Update.UpdateData into the Processor;
                Verify update was correctly loaded into the processor
                Go on to next processor
                    Break;
            }
            Else If (Update.TotalSize > (Update.DataSize + 48))
            {
                N ← 0
                While (N < Update.ExtendedSignatureCount)
                {
                    If ((Update.ProcessorSignature[N] ==
                         Processor Signature) &&
                         (Update.ProcessorFlags[N] & Platform Bits))
                    {
                        Load Update.UpdateData into the Processor;
                        Verify update was correctly loaded into the processor
                        Go on to next processor
                            Break;
                    }
                    N ← N + 1
                }
                I ← I + (Update.TotalSize / 2048)
                If ((Update.TotalSize MOD 2048) == 0)
                    I ← I + 1
            }
        }
    }
```

```
}
```

NOTE

The platform Id bits in IA32_PLATFORM_ID are encoded as a three-bit binary coded decimal field. The platform bits in the microcode update header are individually bit encoded. The algorithm must do a translation from one format to the other prior to doing a check.

When performing the INT 15H, 0D042H functions, the BIOS must assume that the caller has no knowledge of platform specific requirements. It is the responsibility of BIOS calls to manage all chipset and platform specific prerequisites for managing the NVRAM device. When writing the update data using the Write Update sub-function, the BIOS must maintain implementation specific data requirements (such as the update of NVRAM checksum). The BIOS should also attempt to verify the success of write operations on the storage device used to record the update.

9.11.8.2. Responsibilities of the Calling Program

This section of the document lists the responsibilities of acalling program using the interface specifications to load microcode update(s) into BIOS NVRAM.

- The calling program should call the INT 15H, 0D042H functions from a pure real mode program and should be executing on a system that is running in pure real mode.
- The caller should issue the presence test function (sub function 0) and verify the signature and return codes of that function.
- It is important that the calling program provides the required scratch RAM buffers for the BIOS and the proper stack size as specified in the interface definition.
- The calling program should read any update data that already exists in the BIOS in order to make decisions about the appropriateness of loading the update. The BIOS must refuse to overwrite a newer update with an older version. The update header contains information about version and processor specifics for the calling program to make an intelligent decision about loading.
- There can be no ambiguous updates. The BIOS must refuse to allow multiple updates for the same CPU to exist at the same time; it also must refuse to load updates for processors that don't exist on the system.
- The calling application should implement a verify function that is run after the update write function successfully completes. This function reads back the update and verifies that the BIOS returned an image identical to the one that was written.

Example 9-8 represents a calling program.

Example 9-8. INT 15 DO42 Calling Program Pseudo-code

```
//
// We must be in real mode
//
If the system is not in Real mode exit
//
// Detect the presence of Genuine Intel processor(s) that can be updated
// using(CPUID)
//
If no Intel processors exist that can be updated exit
```

```
//
// Detect the presence of the Intel microcode update extensions
//
If the BIOS fails the PresenceTestexit
//
// If the APIC is enabled, see if any other processors are out there
//
Read IA32_APICBASE
If APIC enabled
{
    Send Broadcast Message to all processors except self via APIC
    Have all processors execute CPUID and record the Processor Signature
    (i.e.,Extended Family, Extended Model, Type, Family, Model, Stepping)
    Have all processors read IA32_PLATFORM_ID[52:50] and record Platform
     Id Bits

    If current processor cannot be updated
        exit
}
//
// Determine the number of unique update blocks needed for this system
//
NumBlocks = 0
For each processor
{
    If ((this is a unique processor stepping) AND
        (we have a unique update in the database for this processor))
    {
        Checksum the update from the database;
        If Checksum fails
            exit
        NumBlocks ← NumBlocks + size of microcode update / 2048
    }
}

//
// Do we have enough update slots for all CPUs?
//
If there are more blocks required to support the unique processor steppings
than update blocks provided by the BIOS
    exit
//
// Do we need any update blocks at all?  If not, we are done
//
If (NumBlocks == 0)
    exit
//
// Record updates for processors in NVRAM.
//
For (I=0; I<NumBlocks; I++)
{
    //
    // Load each Update
    //
    Issue the WriteUpdate function

    If (STORAGE_FULL) returned
```

```
        {
            Display Error -- BIOS is not managing NVRAM appropriately
            exit
        }

        If (INVALID_REVISION) returned
        {
            Display Message: More recent update already loaded in NVRAM for
              this stepping
            continue
        }

        If any other error returned
        {
            Display Diagnostic
            exit
        }

        //
        //  Verify the update was loaded correctly
        //
        Issue the ReadUpdate function

        If an error occurred
        {
            Display Diagnostic
            exit
        }
        //
        //  Compare the Update read to that written
        //
        If (Update read != Update written)
        {
            Display Diagnostic
            exit
        }

        I ← I + (size of microcode update / 2048)
    }
    //
    //  Enable Update Loading, and inform user
    //
    Issue the Update Control function with Task = Enable.
```

9.11.8.3. Microcode Update Functions

Table 9-7 defines current Pentium 4, Intel Xeon, and P6 family processor microcode update functions.

Table 9-7. Microcode Update Functions

Microcode Update Function	Function Number	Description	Required/Optional
Presence test	00H	Returns information about the supported functions.	Required
Write update data	01H	Writes one of the update data areas (slots).	Required
Update control	02H	Globally controls the loading of updates.	Required
Read update data	03H	Reads one of the update data areas (slots).	Required

9.11.8.4. INT 15H-based Interface

Intel recommends that a BIOS interface be provided that allows additional microcode updates to be added to system flash. The INT15H interface is the Intel-defined method for doing this.

The program that calls this interface is responsible for providing three 64-kilobyte RAM areas for BIOS use during calls to the read and write functions. These RAM scratch pads can be used by the BIOS for any purpose, but only for the duration of the function call. The calling routine places real mode segments pointing to the RAM blocks in the CX, DX and SI registers. Calls to functions in this interface must be made with a minimum of 32 kilobytes of stack available to the BIOS.

In general, each function returns with CF cleared and AH contains the returned status. The general return codes and other constant definitions are listed in Section 9.11.8.9., *Return Codes*.

The OEM error field (AL) is provided for the OEM to return additional error information specific to the platform. If the BIOS provides no additional information about the error, OEM error must be set to SUCCESS. The OEM error field is undefined if AH contains either SUCCESS (00H) or NOT_IMPLEMENTED (86H). In all other cases, it must be set with either SUCCESS or a value meaningful to the OEM.

The following sections describe functions provided by the INT15H-based interface.

9.11.8.5. Function 00H—Presence Test

This function verifies that the BIOS has implemented required microcode update functions. Table 9-8 lists the parameters and return codes for the function.

Table 9-8. Parameters for the Presence Test

Input		
AX	Function Code	0D042H
BL	Sub-function	00H - Presence test
Output		
CF	Carry Flag	Carry Set - Failure - AH contains status Carry Clear - All return values valid
AH	Return Code	
AL	OEM Error	Additional OEM information.
EBX	Signature Part 1	'INTE' - Part one of the signature
ECX	Signature Part 2	'LPEP'- Part two of the signature
EDX	Loader Version	Version number of the microcode update loader
SI	Update Count	Number of 2048 update blocks in NVRAM the BIOS allocated to storing microcode updates
Return Codes (see Table 9-13 for code definitions)		
SUCCESS		The function completed successfully.
NOT_IMPLEMENTED		The function is not implemented.

Description

In order to assure that the BIOS function is present, the caller must verify the carry flag, the return code, and the 64-bit signature. The update count reflects the number of 2048-byte blocks available for storage within one non-volatile RAM.

The loader version number refers to the revision of the update loader program that is included in the system BIOS image.

9.11.8.6. Function 01H—Write Microcode Update Data

This function integrates a new microcode update into the BIOS storage device. Table 9-9 lists the parameters and return codes for the function.

Table 9-9. Parameters for the Write Update Data Function

Input		
AX	Function Code	0D042H
BL	Sub-function	01H - Write update
ES:DI	Update Address	Real Mode pointer to the Intel Update structure. This buffer is 2048 bytes in length if the processor supports only fixed-size microcode update or... Real Mode pointer to the Intel Update structure. This buffer is 64K-bytes in length if the processor supports a variable-size microcode update.
CX	Scratch Pad1	Real mode segment address of 64 kilobytes of RAM block
DX	Scratch Pad2	Real mode segment address of 64 kilobytes of RAM block
SI	Scratch Pad3	Real mode segment address of 64 kilobytes of RAM block
SS:SP	Stack pointer	32 kilobytes of stack minimum
Output		
CF	Carry Flag	Carry Set - Failure - AH Contains status Carry Clear - All return values valid
AH	Return Code	Status of the call
AL	OEM Error	Additional OEM information
Return Codes (see Table 9-13 for code definitions)		
SUCCESS		The function completed successfully.
WRITE_FAILURE		A failure occurred because of the inability to write the storage device.
ERASE_FAILURE		A failure occurred because of the inability to erase the storage device.
READ_FAILURE		A failure occured because of the inability to read the storage device.
STORAGE_FULL		The BIOS non-volatile storage area is unable to accommodate the update because all available update blocks are filled with updates that are needed for processors in the system.
CPU_NOT_PRESENT		The processor stepping does not currently exist in the system.
INVALID_HEADER		The update header contains a header or loader version that is not recognized by the BIOS.
INVALID_HEADER_CS		The update does not checksum correctly.
SECURITY_FAILURE		The processor rejected the update.
INVALID_REVISION		The same or more recent revision of the update exists in the storage device.

Description

The BIOS is responsible for selecting an appropriate update block in the non-volatile storage for storing the new update. This BIOS is also responsible for ensuring the integrity of the information

provided by the caller, including authenticating the proposed update before incorporating it into storage.

Before writing the update block into NVRAM, the BIOS should ensure that the update structure meets the following criteria in the following order:

1. The update header version should be equal to an update header version recognized by the BIOS.
2. The update loader version in the update header should be equal to the update loader version contained within the BIOS image.
3. The update block must checksum. This checksum is computed as a 32-bit summation of all double words in the structure, including the header, data, and processor signature table.

The BIOS selects update block(s) in non-volatile storage for storing the candidate update. The BIOS can select any available update block as long as it guarantees that only a single update exists for any given processor stepping in non-volatile storage. If the update block selected already contains an update, the following additional criteria apply to overwrite it:

- The processor signature in the proposed update must be equal to the processor signature in the header of the current update in NVRAM (Processor Signature + platform ID bits).
- The update revision in the proposed update should be greater than the update revision in the header of the current update in NVRAM.

If no unused update blocks are available and the above criteria are not met, the BIOS can overwrite update block(s) for a processor stepping that is no longer present in the system. This can be done by scanning the update blocks and comparing the processor steppings, identified in the MP Specification table, to the processor steppings that currently exist in the system.

Finally, before storing the proposed update in NVRAM, the BIOS must verify the authenticity of the update via the mechanism described in Section 9.11.6., *Microcode Update Loader*. This includes loading the update into the current processor, executing the CPUID instruction, reading MSR 08Bh, and comparing a calculated value with the update revision in the proposed update header for equality.

When performing the write update function, the BIOS must record the entire update, including the header, the update data, and the extended processor signature table (if applicable). When writing an update, the original contents may be overwritten, assuming the above criteria have been met. It is the responsibility of the BIOS to ensure that more recent updates are not overwritten through the use of this BIOS call, and that only a single update exists within the NVRAM for any processor stepping and platform ID.

Figure 9-8 and Figure 9-9 show the process the BIOS follows to choose an update block and ensure the integrity of the data when it stores the new microcode update.

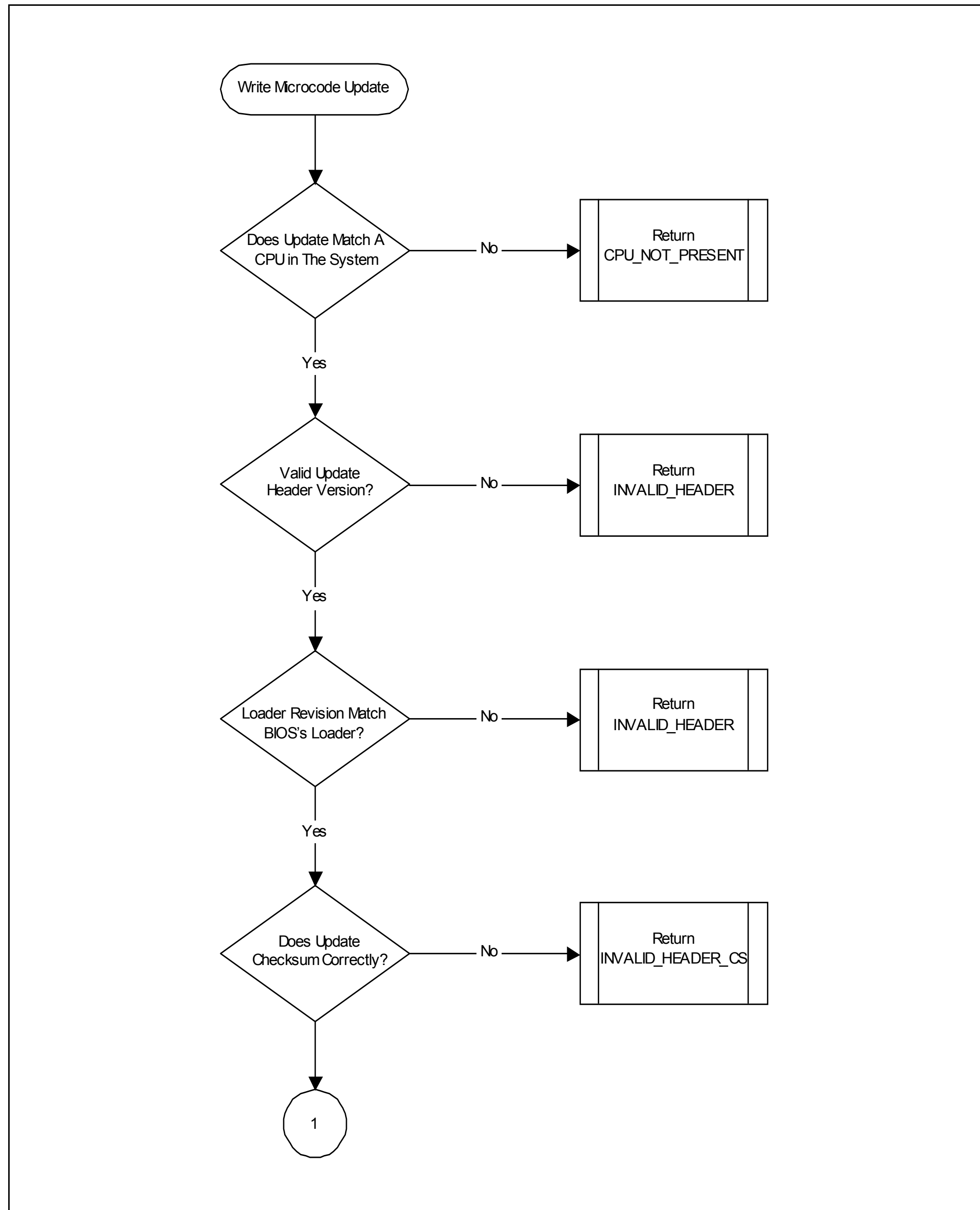

Figure 9-8. Microcode Update Write Operation Flow [1]

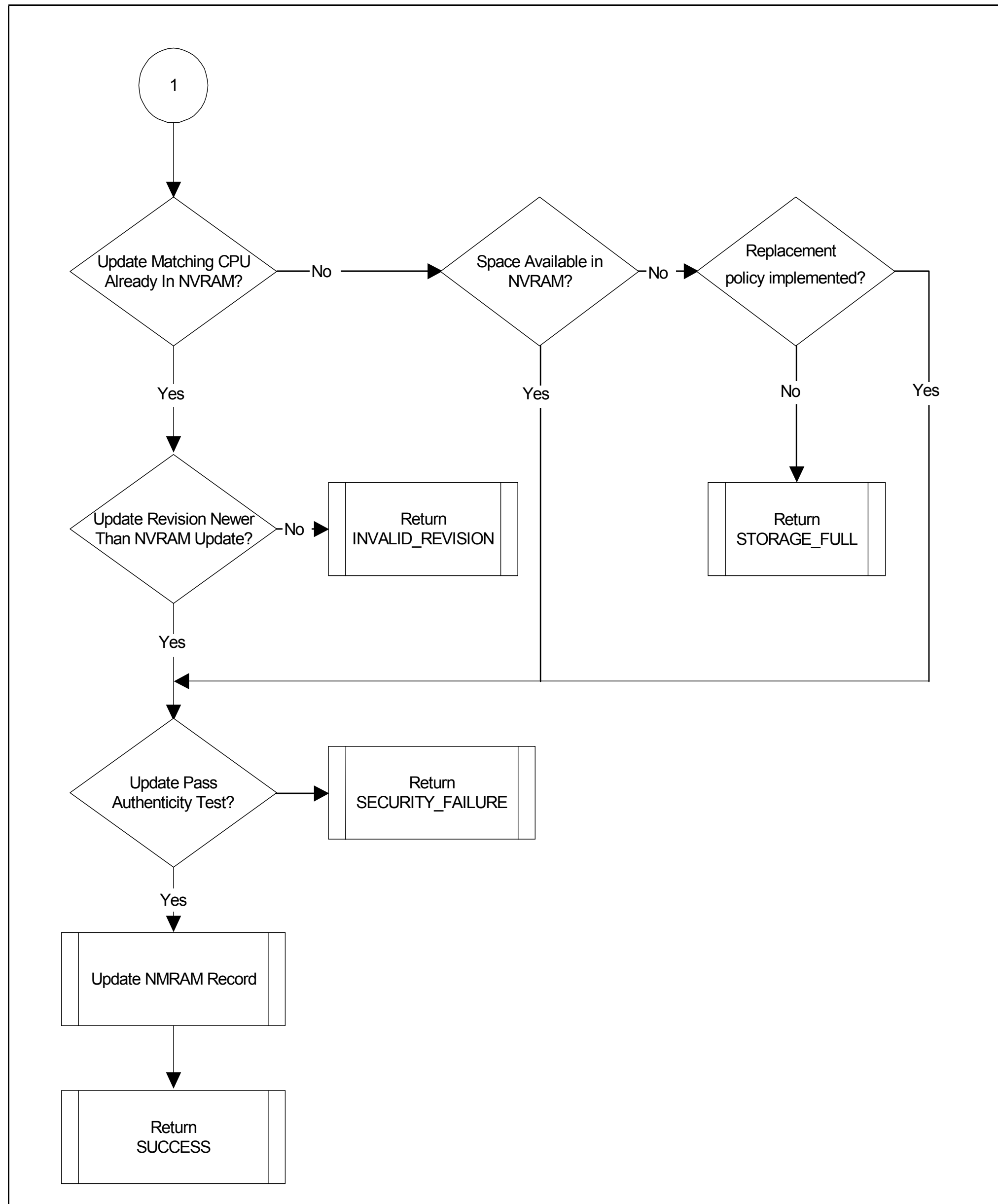

Figure 9-9. Microcode Update Write Operation Flow [2]

9.11.8.7. Function 02H—Microcode Update Control

This function enables loading of binary updates into the processor. Table 9-10 lists the parameters and return codes for the function.

Table 9-10. Parameters for the Control Update Sub-function

Input		
AX	Function Code	0D042H
BL	Sub-function	02H - Control update
BH	Task	See the description below.
CX	Scratch Pad1	Real mode segment of 64 kilobytes of RAM block
DX	Scratch Pad2	Real mode segment of 64 kilobytes of RAM block
SI	Scratch Pad3	Real mode segment of 64 kilobytes of RAM block
SS:SP	Stack pointer	32 kilobytes of stack minimum
Output		
CF	Carry Flag	Carry Set - Failure - AH contains status Carry Clear - All return values valid.
AH	Return Code	Status of the call
AL	OEM Error	Additional OEM Information.
BL	Update Status	Either enable or disable indicator
Return Codes (see Table 9-13 for code definitions)		
SUCCESS		Function completed successfully.
READ_FAILURE		A failure occurred because of the inability to read the storage device.

Description

This control is provided on a global basis for all updates and processors. The caller can determine the current status of update loading (enabled or disabled) without changing the state. The function does not allow the caller to disable loading of binary updates, as this poses a security risk.

The caller specifies the requested operation by placing one of the values from Table 9-11 in the BH register. After successfully completing this function, the BL register contains either the enable or the disable designator. Note that if the function fails, the update status return value is undefined.

Table 9-11. Mnemonic Values

Mnemonic	Value	Meaning
Enable	1	Enable the Update loading at initialization time.
Query	2	Determine the current state of the update control without changing its status.

The READ_FAILURE error code returned by this function has meaning only if the control function is implemented in the BIOS NVRAM. The state of this feature (enabled/disabled) can also be implemented using CMOS RAM bits where READ failure errors cannot occur.

9.11.8.8. Function 03H—Read Microcode Update Data

This function reads a currently installed microcode update from the BIOS storage into a caller-provided RAM buffer. Table 9-12 lists the parameters and return codes for the function.

Table 9-12. Parameters for the Read Microcode Update Data Function

Input		
AX	Function Code	0D042H
BL	Sub-function	03H - Read Update
ES:DI	Buffer Address	Real Mode pointer to the Intel Update structure that will be written with the binary data
ECX	Scratch Pad1	Real Mode Segment address of 64 kilobytes of RAM Block (lower 16 bits)
ECX	Scratch Pad2	Real Mode Segment address of 64 kilobytes of RAM Block (upper 16 bits)
DX	Scratch Pad3	Real Mode Segment address of 64 kilobytes of RAM Block
SS:SP	Stack pointer	32 kilobytes of Stack Minimum
SI	Update Number	This is the index number of the update block to be read. This value is zero based and must be less than the update count returned from the presence test function.
Output		
CF	Carry Flag	Carry Set - Failure - AH contains Status
Carry Clear - All return values are valid.		
AH	Return Code	Status of the Call
AL	OEM Error	Additional OEM Information
Return Codes (see Table 9-13 for code definitions)		
SUCCESS		The function completed successfully.
READ_FAILURE		There was a failure because of the inability to read the storage device.
UPDATE_NUM_INVALID		Update number exceeds the maximum number of update blocks implemented by the BIOS.
NOT_EMPTY		The specified update block is a subsequent block in use to store a valid microcode update that spans multiple blocks. The specified block is not a header block and is not empty.

Description

The read function enables the caller to read any microcode update data that already exists in a BIOS and make decisions about the addition of new updates. As a result of a successful call, the BIOS copies the microcode update into the location pointed to by ES:DI, with the contents of all Update block(s) that are used to store the specified microcode update.

If the specified block is not a header block, but does contain valid data from a microcode update that spans multiple update blocks, then the BIOS must return Failure with the NOT_EMPTY error code in AH.

An update block is considered unused and available for storing a new update if its Header Version contains the value 0FFFFFFFFH after return from this function call. The actual implementation of

NVRAM storage management is not specified here and is BIOS dependent. As an example, the actual data value used to represent an empty block by the BIOS may be zero, rather than 0FFFFFFFFH. The BIOS is responsible for translating this information into the header provided by this function.

9.11.8.9. Return Codes

After the call has been made, the return codes listed in Table 9-13 are available in the AH register.

Table 9-13. Return Code Definitions

Return Code	Value	Description
SUCCESS	00H	The function completed successfully.
NOT_IMPLEMENTED	86H	The function is not implemented
ERASE_FAILURE	90H	A failure because of the inability to erase the storage device.
WRITE_FAILURE	91H	A failure because of the inability to write the storage device.
READ_FAILURE	92H	A failure because of the inability to read the storage device.
STORAGE_FULL	93H	The BIOS non-volatile storage area is unable to accommodate the update because all available update blocks are filled with updates that are needed for processors in the system.
CPU_NOT_PRESENT	94H	The processor stepping does not currently exist in the system.
INVALID_HEADER	95H	The update header contains a header or loader version that is not recognized by the BIOS.
INVALID_HEADER_CS	96H	The update does not checksum correctly.
SECURITY_FAILURE	97H	The update was rejected by the processor.
INVALID_REVISION	98H	The same or more recent revision of the update exists in the storage device.
UPDATE_NUM_INVALID	99H	The update number exceeds the maximum number of update blocks implemented by the BIOS.
NOT_EMPTY	9AH	The specified update block is a subsequent block in use to store a valid microcode update that spans multiple blocks. The specified block is not a header block and is not empty.

22. A Mechanism for Determining Sync/Async SMIs Has Been Documented

In Volume 3, a new section has been added. The section provides information about two new CRs that provide an effective means to determine whether an SMI is synchronous or asynchronous. The new section is reproduced below.

13.7. MANAGING SYNCHRONOUS AND ASYNCHRONOUS SYSTEM MANAGEMENT INTERRUPTS

Particularly when coding for a multiprocessor system or a system with Intel HT Technology, it was not always possible for an SMI handler to distinguish between a synchronous SMI (triggered during an I/O instruction) and an asynchronous SMI. To facilitate the discrimination of these two events, incremental state information has been added to the SMM state save map.

Processors that have an SMM revision ID of 30004H or higher have the incremental state information described below.

13.7.1. I/O State Implementation

Within the extended SMM state save map, a bit (IO_SMI) is provided that is set only when an SMI is either taken immediately after a *successful* I/O instruction or is taken after a *successful* iteration of a REP I/O instruction (note that the *successful* notion pertains to the processor point of view; not necessarily to the corresponding platform function). When set, the IO_SMI bit provides a strong indication that the corresponding SMI was synchronous. In this case, the SMM State Save Map also supplies the port address of the I/O operation. The IO_SMI bit and the I/O Port Address may be used in conjunction with the information logged by the platform to confirm that the SMI was indeed synchronous.

Note that the IO_SMI bit by itself is a strong indication, not a guarantee, that the SMI is synchronous. This is because an asynchronous SMI might coincidentally be taken after an I/O instruction. In such a case, the IO_SMI bit would still be set in the SMM state save map.

Information characterizing the I/O instruction is saved in two locations in the SMM State Save Map (Table 13-14). Note that the IO_SMI bit also serves as a valid bit for the rest of the I/O information fields. The contents of these I/O information fields are not defined when the IO_SMI bit is not set.

Table 13-14. I/O Instruction Information in the SMM State Save Map

State (SMM Rev. ID: 30004H or higher)	Format				
	31 16	15 8	7 4	3 1	0
I/0 State Field SMRAM offset 7FA4	I/O Port	Reserved	I/O Type	I/O Length	IO_SMI
	31				0
I/O Memory Address Field SMRAM offset 7FA0	I/O Memory Address				

IO_SMI is set if an SMI was taken during or immediately following an I/O instruction. When IO_SMI is set, the other fields may be interpreted as follows:

- I/O Length can be:
  - 001 – Byte
  - 010 – Word
  - 100 – Dword
- I/O Instruction Type:

Table 13-15. I/O Instruction Type Encodings

Instruction	Encoding
IN Immediate	1001
IN DX	0001
OUT Immediate	1000
OUT DX	0000
INS	0011
OUTS	0010
REP INS	0111
REP OUTS	0110

- I/O Memory Address is:
  - for OUTS/REP_OUTS: Segment_base + eSI
  - for INS/REP_INS: ES_base + eDI
  - for IN/OUT: 0x0

The SMRAM save map (Table 13-1) has also been updated. It now indicates the location of the relevant CRs. Impacted table cells are reproduced below.

Table 13-1. SMRAM State Save Map

Offset (Added to SMBASE + 8000H)	Register	Writable?
7FA4H	I/O State Field, see Section 13.7.	No
7FA0H	I/O Memory Address Field, see Section 13.7.	No

23. Omitted Debug Data Has Been Restored

In the Volume 3, Chapter 5, Interrupt 1 section; data deleted by mistake has been restored. This material is reproduced below.

Interrupt 1—Debug Exception (#DB)

Exception Class Trap or Fault. The exception handler can distinguish between traps or faults by examining the contents of DR6 and the other debug registers.

Description

Indicates that one or more of several debug-exception conditions has been detected. Whether the exception is a fault or a trap depends on the condition (see Table 13-2). See Chapter 15, *Debugging and Performance Monitoring*, for detailed information about the debug exceptions.

Table 13-2. Debug Exception Conditions and Corresponding Exception Classes

Exception Condition	Exception Class
Instruction fetch breakpoint	Fault
Data read or write breakpoint	Trap
I/O read or write breakpoint	Trap
General detect condition (in conjunction with in-circuit emulation)	Fault
Single-step	Trap
Task-switch	Trap

Exception Error Code

None. An exception handler can examine the debug registers to determine which condition caused the exception.

Saved Instruction Pointer

Fault—Saved contents of CS and EIP registers point to the instruction that generated the exception.

Trap—Saved contents of CS and EIP registers point to the instruction following the instruction that generated the exception.

Program State Change

Fault—A program-state change does not accompany the debug exception, because the exception occurs before the faulting instruction is executed. The program can resume normal execution upon returning from the debug exception handler.

Trap—A program-state change does accompany the debug exception, because the instruction or task switch being executed is allowed to complete before the exception is generated. However, the new state of the program is not corrupted and execution of the program can continue reliably.

24. CLTS Exception Information Updated

In the Volume 2, Chapter 3, CLTS—Clear Task-Switched Flag in CR0 section; missing exception data has been added. The corrected area is reproduced below.

CLTS—Clear Task-Switched Flag in CR0: only updated exception sections reproduced...

Protected Mode Exceptions

#GP(0) If the current privilege level is not 0.

Real-Address Mode Exceptions

None.

Virtual-8086 Mode Exceptions

#GP(0) CLTS is not recognized in virtual-8086 mode.

25. The MOVSS Description Has Been Updated

In the Volume 2, Chapter 3, MOVSS—Move Scalar Single--Precision Floating-Point Values section; incorrect data has been updated. The corrected area is reprinted below. In the incorrect version, the description block below indicated 64-bit memory locations.

MOVSS—Move Scalar Single--Precision Floating-Point Values

Opcode	Instruction	Description
F3 0F 10 /r	MOVSS *xmm1*, *xmm2/m32*	Move scalar single-precision floating-point value from *xmm2/m32* to *xmm1* register.
F3 0F 11 /r	MOVSS *xmm2/m32*, *xmm1*	Move scalar single-precision floating-point value from *xmm1* register to *xmm2/m32*.

Description

Moves a scalar single-precision floating-point value from the source operand (second operand) to the destination operand (first operand). The source and destination operands can be XMM registers or 32-bit memory locations. This instruction can be used to move a single-precision floating-point value to and from the low doubleword of an XMM register and a 32-bit memory location, or to move a single-precision floating-point value between the low doublewords of two XMM registers. The instruction cannot be used to transfer data between memory locations.

When the source and destination operands are XMM registers, the three high-order doublewords of the destination operand remain unchanged. When the source operand is a memory location and

destination operand is an XMM registers, the three high-order doublewords of the destination operand are cleared to all 0s.

----

26. An Instruction Listing (PULLHUW) Has Been Deleted

In Volume 2of the IA-32 Intel Architecture Software Developer's Manual, Appendix B, Table B-13; there was a listing for a 'PULLHUW' instruction. This appears to have been an old corruption of PMULHUW entry (see the SSE/SSE2 entries by that name). The PULLHUW table entry has been deleted. The reproduced table cells (below) indicate the point of deletion.

PMADDWD - Packed multiply add	
mmxreg2 to mmxreg1	0000 1111:11110101: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:11110101: mod mmxreg r/m
PMULHW - Packed multiplication, store high word	
mmxreg2 to mmxreg1	0000 1111:11100101: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:11100101: mod mmxreg r/m

27. Data Entry Errors in Table B-20 Have Been Corrected

In Volume 2 of the IA-32 Intel Architecture Software Developer's Manual, Appendix B, Table B-20. MOVQ encoding information has been changed. The impacted table cells are reproduced below.

Table B-20. (continued)

MOVQ - Move Quadword	
xmmreg2 to xmmreg1	**11110011**:00001111:01111110: 11 xmmreg1 xmmreg2
xmmreg2 from xmmreg1	01100110:00001111:11010110: 11 xmmreg1 xmmreg2
mem to xmmreg	**11110011**:00001111:01111110: mod xmmreg r/m
mem from xmmreg	01100110:00001111:11010110: mod xmmreg r/m

28. Figure 8-22 Has Been Corrected

In Volume 3 of the IA-32 Intel Architecture Software Developer's Manual, Chapter 8, Figure 8-22; an incorrect address has been corrected. The corrected figure has been reproduced below.

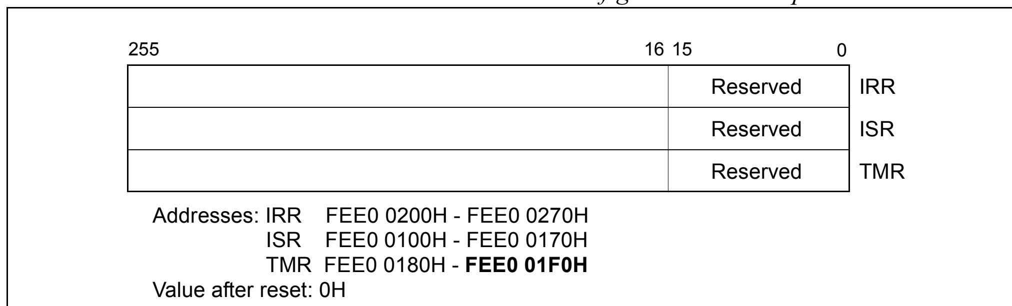

Figure 8-20. IRR, ISR and TMR Registers

29. The Description of Minimum Thermal Monitor Activation Time Has Been Updated

In Volume 3of the IA-32 Intel Architecture Software Developer's Manual, Section 13.15.2.4; a paragraph describing TM1/TM2 has been re-written to provide a more accurate description. The applicable text is reproduced below.

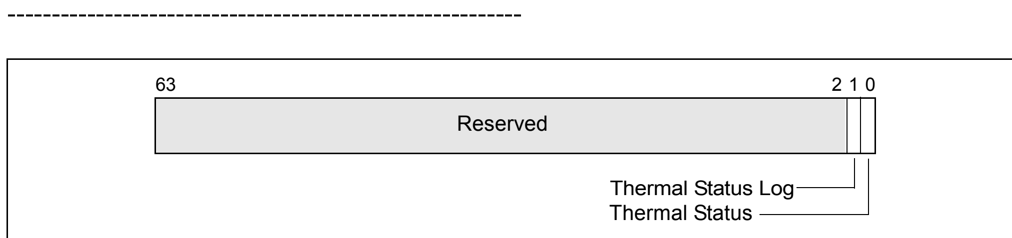

Figure 8-21. IA32_THERM_STATUS MSR

Thermal Status Log flag, bit 1

When set, indicates that the thermal sensor has tripped since the last power-up or reset or since the last time that software cleared this flag. This flag is a sticky bit; once set it remains set until cleared by software or until a power-up or reset of the processor. The default state is clear.

After the second temperature sensor has been tripped, the thermal monitor (TM1/TM2) will remain engaged for a minimum time period (on the order of 1 ms). The thermal monitor will remain engaged until the processor core temperature drops below the preset trip temperature of the temperature sensor, taking hysteresis into account.

30. Corrected Description of Exception- or Interrupt-Handler Procedures

In Volume 3, Section 5.12.1; text describing exception- or interrupt-handler procedures has been re-written. The new text is reproduced below.

When the processor performs a call to the exception- or interrupt-handler procedure:

- If the handler procedure is going to be executed at a numerically lower privilege level, a stack switch occurs. When the stack switch occurs:

a. The segment selector and stack pointer for the stack to be used by the handler are obtained from the TSS for the currently executing task. On this new stack, the processor pushes the stack segment selector and stack pointer of the interrupted procedure.

b. The processor then saves the current state of the EFLAGS, CS, and EIP registers on the new stack (see Figure 5-4).

c. If an exception causes an error code to be saved, it is pushed on the new stack after the EIP value.

- If the handler procedure is going to be executed at the same privilege level as the interrupted procedure:

a. The processor saves the current state of the EFLAGS, CS, and EIP registers on the current stack (see Figure 5-4).

b. If an exception causes an error code to be saved, it is pushed on the current stack after the EIP value.

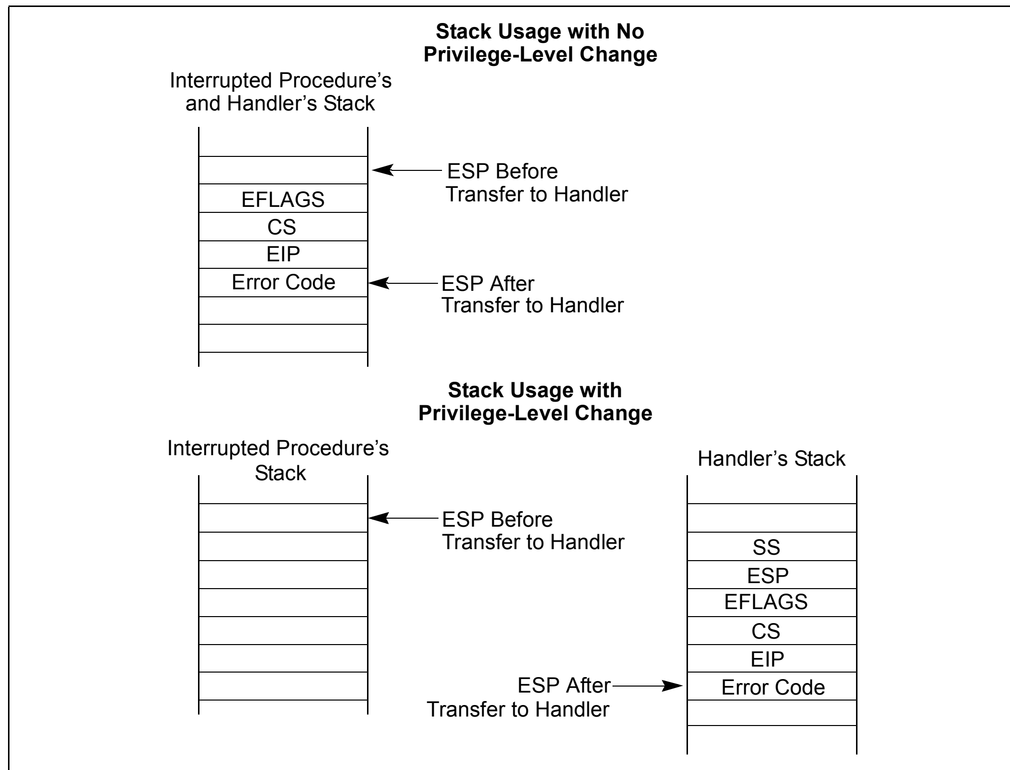

Figure 5-4. Stack Usage on Transfers to Interrupt and Exception-Handling Routines

31. CMPSD and CMPSS Exception Information Updated

Changes were made in the relevant sections of Volume 2of the IA-32 Intel Architecture Software Developer's Manual, Chapter 3. The updated exception sections are reproduced below.

CMPSD — Compare Scalar Double-Precision Floating-Point Values: only updated exception sections reproduced....

Real-Address Mode Exceptions

#GP(0)	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.
	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE2 is 0.

CMPSS — Compare Scalar Single-Precision Floating-Point Values: only updated exception sections reproduced....

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.
	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE is 0.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP(0)	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.

#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.
	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE is 0.

32. PUNPCKHB*/PUNPCKLB* Exception Information Improved

Changes were made in the relevant sections of Volume 2, Chapter 3. The updated exception sections are reproduced below.

PUNPCKHBW/PUNPCKHWD/PUNPCKHDQ/PUNPCKHQDQ — Unpack High Data: only updated exception sections reproduced....

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	(128-bit operations only) If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.
	128-bit operations will generate #UD only if OSFXSR in CR4 is 0. Execution of 128-bit instructions on a non-SSE2 capable processor (one that is MMX technology capable) will result in the instruction operating on the mm registers, not #UD.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP(0)	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
	(128-bit operations only) If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#UD	If EM in CR0 is set.

128-bit operations will generate #UD only if OSFXSR in CR4 is 0. Execution of 128-bit instructions on a non-SSE2 capable processor (one that is MMX technology capable) will result in the instruction operating on the mm registers, not #UD.

#NM If TS in CR0 is set.

#MF (64-bit operations only) If there is a pending x87 FPU exception.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code) For a page fault.

#AC(0) (64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made.

PUNPCKLBW/PUNPCKLWD/PUNPCKLDQ/PUNPCKLQDQ — Unpack Low Data: only updated exception sections reproduced...

Protected Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

(128-bit operations only) If memory operand is not aligned on a 16-byte boundary, regardless of segment.

#SS(0) If a memory operand effective address is outside the SS segment limit.

#UD If EM in CR0 is set.

128-bit operations will generate #UD only if OSFXSR in CR4 is 0. Execution of 128-bit instructions on a non-SSE2 capable processor (one that is MMX technology capable) will result in the instruction operating on the mm registers, not #UD.

#NM If TS in CR0 is set.

#MF (64-bit operations only) If there is a pending x87 FPU exception.

#PF(fault-code) If a page fault occurs.

#AC(0) (64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP(0) If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.

(128-bit operations only) If memory operand is not aligned on a 16-byte boundary, regardless of segment.

#UD If EM in CR0 is set.

128-bit operations will generate #UD only if OSFXSR in CR4 is 0. Execution

of 128-bit instructions on a non-SSE2 capable processor (one that is MMX technology capable) will result in the instruction operating on the mm registers, not #UD.

#NM	If TS in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.

Virtual-8086 Mode Exceptions

Same exceptions as in Real Address Mode

#PF(fault-code)	For a page fault.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made.

33. MOVHPD, MOVLPD, UNPCKHPS, UNPCKLPS Exception Information Updated

Changes were made in the relevant sections of Volume 2 of the *IA-32 Intel Architecture Software Developer's Manual*, Chapter 3. The updated exception sections are reproduced below.

MOVHPD—Move High Packed Double-Precision Floating-Point Value: only updated exception sections reproduced...

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#UD	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE2 is 0.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

GP(0)	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#UD	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.

If CPUID feature flag SSE2 is 0.

MOVHPS — Move High Packed Single-Precision Floating-Point Values: only updated exception sections reproduced...

Real-Address Mode Exceptions

GP(0)	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#UD	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE is 0.

UNPCKHPS—Unpack and Interleave High Packed Single-Precision Floating-Point Values: only updated exception sections reproduced...

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.
	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE is 0.

Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.
	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE is 0.

UNPCKLPS—Unpack and Interleave Low Packed Single-Precision Floating-Point Values: only updated exception sections reproduced...

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#NM	If TS in CR0 is set.
#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.
	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE is 0.

Real-Address Mode Exceptions

#GP(0)	If memory operand is not aligned on a 16-byte boundary, regardless of segment.
	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If TS in CR0 is set.

#XM	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 1.
#UD	If an unmasked SIMD floating-point exception and OSXMMEXCPT in CR4 is 0.
	If EM in CR0 is set.
	If OSFXSR in CR4 is 0.
	If CPUID feature flag SSE is 0.

34. PEXTRW - PINSRW Exception Information Updated

Changes were made in the relevant sections of Volume 2 *of the IA-32 Intel Architecture Software Developer's Manual*, Chapter 3. The updated exception sections are reproduced below.

PEXTRW — Extract Word: only updated exception sections reproduced...

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	#SS(0)If a memory operand effective address is outside the SS segment limit.
#UD	If EM in CR0 is set.
	(128-bit operations only) If OSFXSR in CR4 is 0.
	(128-bit operations only) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	(64-bit operations only) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP(0)	If any part of the operand lies outside of the effective address space from 0 to FFFFH.
#UD	If EM in CR0 is set.
	(128-bit operations only) If OSFXSR in CR4 is 0.
	(128-bit operations only) If CPUID feature flag SSE2 is 0.
#NM	If TS in CR0 is set.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.