Advanced Micro Devices, Inc.
AMD I/O Virtualization Technology (IOMMU) Specification License Agreement

AMD I/O Virtualization Technology (IOMMU) Specification License Agreement (this “Agreement”) is a legal agreement between Advanced Micro Devices, Inc., Sunnyvale CA ("AMD") and the recipient of the AMD IO MMU Specification (any version) (the “Specification”), whether an individual or an entity ("You"). If you have accessed this Agreement as part of the Specification, or in the process of downloading the Specification from an AMD web site, by clicking an “I Accept” or similar button, or otherwise in the process of acquiring the Specification, or by using or providing feedback on the Specification, You agree to these terms. If this Agreement is attached to the Specification, by accessing, using or providing feedback on the Specification, You agree to these terms.

For good and valuable consideration, the receipt and sufficiency of which are acknowledged, You and AMD agree as follows:

1. You may review the Specification only (a) as a reference to assist You in planning and designing Your product, service or technology ("Product") to interface with an AMD or third-party Product as described in the Specification; and (b) to provide Feedback (defined below) on the Specification to AMD. All other rights are retained by AMD; this agreement does not give You rights under any AMD patents. You may not (i) duplicate any part of the Specification, (ii) remove this agreement or any notices from the Specification, or (iii) give any part of the Specification, or assign or otherwise provide Your rights under this Agreement, to anyone else.

2. The Specification may contain preliminary information or inaccuracies. The Specification is provided entirely "AS IS." To the extent permitted by law, AMD MAKES NO WARRANTY OF ANY KIND, DISCLAIMS ALL EXPRESS, IMPLIED AND STATUTORY WARRANTIES, AND ASSUMES NO LIABILITY TO YOU FOR ANY DAMAGES OF ANY TYPE IN CONNECTION WITH THESE MATERIALS OR ANY INTELLECTUAL PROPERTY IN THEM.

3. If You are an entity and (a) merge into another entity or (b) a controlling ownership interest in You changes, Your right to use the Specification automatically terminates and You must destroy it.

4. You have no obligation to give AMD any suggestions, comments or other feedback ("Feedback") relating to the Specification. However, any Feedback you voluntarily provide may be used by AMD without restriction including the use in any revision or update to the Specification. Accordingly, if You do give AMD Feedback on any version of the Specification, You agree: (a) AMD may freely use, reproduce, license, distribute, and otherwise commercialize Your Feedback in any product made or distributed by or for AMD (an “AMD Product”); (b) You also grant third parties, without charge, only those patent rights necessary to enable other products to use or interface with any specific parts of an AMD Product that incorporates Your Feedback or Your Product; and (c) You will not give AMD any Feedback (i) that You have reason to believe is subject to any patent, copyright or other intellectual property claim or right of any third party; or (ii) subject to license terms which seek to require any product incorporating or derived from Your Feedback, any AMD Product or other AMD intellectual property, to be licensed to or otherwise provided to any third party.

5. This Agreement is governed by the laws of the State of Texas without regard to its choice of law principles. Any dispute involving it must be brought in a court having jurisdiction of such dispute in Travis County, Texas, and You waive any defenses allowing the dispute to be litigated elsewhere. If there is litigation, the losing party must pay the other party’s reasonable attorneys’ fees, costs and other expenses. If any part of this agreement is unenforceable, it will be considered modified to the extent necessary to make it enforceable, and the remainder shall continue in effect. This agreement is the entire agreement between You and AMD concerning the Specification; it may be changed only by a written document signed by both You and AMD.

AMD I/O Virtualization Technology (IOMMU) Specification

Publication # **34434**	Revision: **1.20**
Issue Date: **February 2007**	

Advanced Micro Devices

Table of Contents

1 Overview 15
1.1 Intended Audience 15
1.2 Definitions. 15
1.3 Bit Attributes. 16

2 IOMMU Overview 17
2.1 Architecture Summary 17
2.2 Usage Models 18
2.2.1 Replacing the GART 18
2.2.2 Substituting for the DEV 19
2.2.3 32-bit to 64-bit Legacy I/O Device Mapping 20
2.2.4 User Mode Device Accesses 20
2.2.5 Virtual Machine Guest Access to Devices. 20
2.2.6 Virtualizing the IOMMU 21

3 Architecture. 22
3.1 Behavior 22
3.1.1 Normal Operation 22
3.1.2 IOMMU Logical Topology 23
3.1.3 IOMMU Error Reporting. 23
3.1.3.1 IOMMU Error Responses 24
3.1.3.2 I/O Page Faults 24
3.1.3.3 Memory Access Errors 24
3.1.4 Special Conditions 25
3.2 Data Structures 25
3.2.1 Updating Shared Tables. 26
3.2.2 Device Table 26
3.2.2.1 Device Table Entry Format 27
3.2.2.2 Making Device Table Entry Changes 32
3.2.3 I/O Page Tables 33
3.2.4 Sharing AMD64 CPU and IOMMU Page Tables 38
3.2.5 Interrupt Remapping Tables 39
3.3 Commands 42
3.3.1 COMPLETION_WAIT 44
3.3.2 INVALIDATE_DEVTAB_ENTRY 44
3.3.3 INVALIDATE_IOMMU_PAGES 45
3.3.4 INVALIDATE_IOTLB_PAGES. 45
3.3.5 INVALIDATE_INTERRUPT_TABLE 46
3.3.6 IOMMU Ordering Rules 47
3.3.6.1 Invalidation Command Ordering Requirements 47
3.3.6.2 Invalidation Commands Interaction Requirements 47
3.4 Event Logging. 47
3.4.1 ILLEGAL_DEV_TABLE_ENTRY 53
3.4.2 IO_PAGE_FAULT 54
3.4.3 DEV_TAB_HARDWARE_ERROR. 55
3.4.4 PAGE_TAB_HARDWARE_ERROR. 56
3.4.5 ILLEGAL_COMMAND_ERROR 57

3.4.6 COMMAND_HARDWARE_ERROR 58
3.4.7 IOTLB_INV_TIMEOUT. 58
3.4.8 INVALID_DEVICE_REQUEST 59
3.5 IOMMU Interrupt Support 61
3.6 PCI Resources. 61
3.6.1 IOMMU Capability Block Registers. 61
3.6.2 IOMMU Control Registers 64

4 Implementation Considerations 72
4.1 Caching and Invalidation Strategies 72
4.2 Recommended IOMMU Topologies 73
4.3 Issues Specific to the HyperTransport™ Architecture 75
4.4 Chipset Specific Implementation Issues. 75
4.5 Software and BIOS Implementation Issues 75

5 IOMMU Page Walker Pseudo Code. 77

6 Register List. 79

List of Figures

Figure 1: Example Platform Architecture 18
Figure 2: IOMMU Data Structures 26
Figure 3: DeviceID Derived from Peripheral RequesterID 26
Figure 4: DeviceID Derived from Peripheral UnitID 27
Figure 5: Device Table Entry Fields 27
Figure 6: I/O Page Table Entry Not Present (any level) 36
Figure 7: I/O Page Translation Entry (PTE) 36
Figure 8: I/O Page Directory Entry (PDE) 36
Figure 9: Address Translation Example with Skipped Level 37
Figure 10: Address Translation Example with Page Size Larger than Default Size 37
Figure 11: Sharing AMD64 and IOMMU Page Tables with Identical Addressing 39
Figure 12: Interrupt Remapping Table Lookup for Fixed and Arbitrated Interrupts 41
Figure 13: Interrupt Remapping Table Entry 41
Figure 14: Circular Command Buffer in System Memory 42
Figure 15: Generic Command Buffer Entry 43
Figure 16: COMPLETION_WAIT command format 44
Figure 17: INVALIDATE_DEVTAB_ENTRY Command Format 45
Figure 18: INVALIDATE_IOMMU_PAGES Command Encoding 45
Figure 19: INVALIDATE_IOTLB_PAGES 46
Figure 20: INVALIDATE_INTERRUPT_TABLE 46
Figure 21: Circular Event Log in System Memory 48
Figure 22: Generic Event Log Buffer Entry 49
Figure 23: ILLEGAL_DEV_TABLE_ENTRY Event Log Buffer Entry 53
Figure 24: IO_PAGE_FAULT Event Log Buffer Entry 54
Figure 25: DEV_TAB_HARDWARE_ERROR Event Log Buffer Entry 55
Figure 26: PAGE_TAB_HARDWARE_ERROR Event Log Buffer Entry 56
Figure 27: ILLEGAL_COMMAND_ERROR 57
Figure 28: COMMAND_HARDWARE_ERROR Event Log Buffer Entry 58
Figure 29: IOTLB_INV_TIMEOUT Event Log Buffer Entry 59
Figure 30: INVALID_DEVICE_REQUEST Event Log Buffer Entry 60
Figure 31: IOMMU in a Tunnel 73
Figure 32: IOMMU in a Peripheral Bus Bridge 74
Figure 33: Hybrid IOMMU 74
Figure 34: Chained Hybrid IOMMU in a Large System 75

List of Tables

Table 1: Bit Attribute Definitions 16
Table 2: Special Address Controls 23
Table 3: Device Table Entry Field Definitions 28
Table 4: V and TV Fields in Device Table Entry 31
Table 5: IV and IntCtl Fields in Device Table Entry for Fixed and Arbitrated Interrupts 32
Table 6: IV and Pass Fields in Device Table Entry for Selected Interrupts 32
Table 7: Example Page Size Encodings 34
Table 8: Page Table Level Parameters 35
Table 9: IOMMU Interrupt Controls and Actions 40
Table 10: Interrupt Remapping Table Fields 41
Table 11: Event Summary 49
Table 12: ILLEGAL_DEV_TABLE_ENTRY Event Log Buffer Entry Fields 53
Table 13: IO_PAGE_FAULT Event Log Buffer Entry Fields 54
Table 14: Event Log Type Field Encodings 55
Table 15: DEV_TAB_HARDWARE_ERROR Event Log Buffer Entry Fields 55
Table 16: PAGE_TAB_HARDWARE_ERROR Event Log Buffer Entry Fields 56
Table 17: ILLEGAL_COMMAND_ERROR Event Log Buffer Entry Fields 57
Table 18: COMMAND_HARDWARE_ERROR Event Log Buffer Entry Fields 58
Table 19: IOTLB_INV_TIMEOUT Event Log Buffer Entry Fields 59
Table 20: INVALID_DEVICE_REQUEST Type Field Encodings 60
Table 21: INVALID_DEVICE_REQUEST Event Log Buffer Entry Fields 60

Revision History

Date	Rev	Description
February, 2007	1.20	• Throughout document, corrected trademark attributions. • In Section 1.2 [Definitions], defined low and high memory, MMU, PDE, and PTE.
		• In Section 2.1 [Architecture Summary], clarified need for multiple IOMMUs. • In Section 2.1 [Architecture Summary], revised Figure 1. • In Section 2.1 [Architecture Summary], clarified differences from GART and DEV. • In Section 2.1 [Architecture Summary], added interrupt remapping. • In Section 2.2 [Usage Models], text clarifications throughout. • In Section 2.2.1 [Replacing the GART], clarified NPcache. • In Section 2.2.5 [Virtual Machine Guest Access to Devices], Table 10, marked all but Fixed and Arbitrated as Reserved. • In Section 2.2.5 [Virtual Machine Guest Access to Devices], clarified DomainID. • In Section 2.2.6 [Virtualizing the IOMMU] removed performance comments.
		• In Section 3 [Architecture], clarified text throughout. • In Section 3.1 [Behavior], updated IOMMU transaction requirements. • In Section 3.1 [Behavior] and Section 3.1.1 [Normal Operation], added interrupt text. • In Section 3.1 [Behavior], added atomic operations. • In Section 3.1 [Behavior], inserted text relocated from Section 3.1.1 [Normal Operation]. • In Section 3.1 [Behavior], deleted reference to (deleted) Figure 2. • In Section 3.1 [Behavior], Table 2, deleted footnote #1 and renumbered footnotes. • In Section 3.1 [Behavior] and Section 3.1.1 [Normal Operation], updated transactions types. • In Section 3.1 [Behavior] and Section 3.1.1 [Normal Operation], clarified behavior. • In Section 3.1.1 [Normal Operation], clarified enumeration. • In Section 3.1.1 [Normal Operation], specified Port I/O space mappings and added I. • In Section 3.1.1 [Normal Operation], clarified terminology and changed master abort to target abort. • Added Section 3.1.2 [IOMMU Logical Topology]. • In Section 3.1.2 [IOMMU Logical Topology], clarified topology comments. • In Section 3.1.2 [IOMMU Logical Topology], clarified and extended Table 2. • In Section 3.1.3 [IOMMU Error Reporting], clarified SA and SE. • In Section 3.1.3 [IOMMU Error Reporting], clarified event logging. • In Section 3.1.3 [IOMMU Error Reporting], added software note. • Added Section 3.1.3.1 [IOMMU Error Responses]. • In Section 3.1.3.1 [IOMMU Error Responses], Figure 5, added TaErrEn. • In Section 3.1.3.1 [IOMMU Error Responses], added clarifications and references. • In Section 3.1.3.1 [IOMMU Error Responses], clarified error condition. • In Section 3.1.3.1 [IOMMU Error Responses], clarified behavior when not enabled. • In Section 3.1.3.2 [I/O Page Faults], clarified error handling. • In Section 3.1.3.2 [I/O Page Faults], changed “master abort” to “target abort”. • Added Section 3.1.4 [Special Conditions].

Date	Rev	Description
		• In Section 3.2 [Data Structures], added interrupt text. • In Section 3.2 [Data Structures], Figure 2, added Interrupt Remapping Tables. • In Section 3.2.1 [Updating Shared Tables], clarifications. • In Section 3.2.2 [Device Table], enlarged device table entries to 256 bits and added interrupt information. • In Section 3.2.2 [Device Table], expanded discussion of valid bits. • In Section 3.2.2 [Device Table], swapped order of Section 3.2.2.1 [Device Table Entry Format] and Section 3.2.2.2 [Making Device Table Entry Changes]. • In Section 3.2.2 [Device Table], added PCI Phantom functions. • Added Section 3.2.2.1 [Device Table Entry Format]. • In Section 3.2.2.1 [Device Table Entry Format], defined IV bit in Table 3, updated IntTabLen and reserved field [133:132], and clarified V bit. • In Section 3.2.2.1 [Device Table Entry Format], Table 3, required alignment of interrupt table root pointer. • In Section 3.2.2.1 [Device Table Entry Format], Table 3 and Table 5, added TV. • In Section 3.2.2.1 [Device Table Entry Format], Table 3, updated EX, IoCtl, SA, SE. • In Section 3.2.2.1 [Device Table Entry Format], Table 3 and Figure 5, defined IG. • In Section 3.2.2.1 [Device Table Entry Format], Table 3, renamed Lint1En and Lint0En as Lint1Pass and Lint0Pass. • In Section 3.2.2.1 [Device Table Entry Format], Table 3 and Figure 5, defined NMIPass, InitPass, and EIntPass. • In Section 3.2.2.1 [Device Table Entry Format], Table 3, added software note for SE. • In Section 3.2.2.1 [Device Table Entry Format], Table 3, clarified SysMgt, IntCtl, Mode, IoCtl, EX, and Cache. • In Section 3.2.2.1 [Device Table Entry Format], Table 3, replaced H with SysMgt. • In Section 3.2.2.1 [Device Table Entry Format], added Table 4, Table 5, and Table 6. • Added Section 3.2.2.2 [Making Device Table Entry Changes]. • In Section 3.2.2.2 [Making Device Table Entry Changes], revised pseudo-code examples.
		• In Section 3.2.3 [I/O Page Tables], removed S bit definition and modified next level encodings to make page tables compatible with AMD64 page table. • In Section 3.2.3 [I/O Page Tables], Figure 7 and Figure 8, renamed NS bit as FC. • In Section 3.2.3 [I/O Page Tables], clarified Table 7 and added Figure 10. • In Section 3.2.3 [I/O Page Tables], added row to Table 7, clarified software note. • In Section 3.2.3 [I/O Page Tables], corrected typos in Figure 9 and Figure 10. • In Section 3.2.3 [I/O Page Tables], clarified usage of FC bit. • In Section 3.2.3 [I/O Page Tables], corrected FC bit, U bit, and physical address bits [63:53] descriptions. • In Section 3.2.3 [I/O Page Tables], clarified zero-fill. • In Section 3.2.3 [I/O Page Tables], Table 7, changed low-order “Un” bits to be 1s. • In Section 3.2.3 [I/O Page Tables], clarified address bits above 48 bits.

Date	Rev	Description
		• Added Section 3.2.5 [Interrupt Remapping Tables]. • In Section 3.2.5 [Interrupt Remapping Tables], restructured text for clarity. • In Section 3.2.5 [Interrupt Remapping Tables], clarified INIT, Startup, and device table control bits. • In Section 3.2.5 [Interrupt Remapping Tables], Table 9, added to define interrupt controls. • In Section 3.2.5 [Interrupt Remapping Tables], Table 10, updated IntType. • In Section 3.2.5 [Interrupt Remapping Tables], Table 10 and Figure 13, added SupIOPF and redefined RemapEn. • In Section 3.2.5 [Interrupt Remapping Tables], Figure 12, corrected DM. • In Section 3.2.5 [Interrupt Remapping Tables], added software note for EOI. • In Section 3.2.5 [Interrupt Remapping Tables], added EOI reply requirement. • In Section 3.2.5 [Interrupt Remapping Tables], Figure 12, added MSI interrupt fields.
		• In Section 3.3 [Commands], corrected command buffer tail pointer definition to match the definition in the register. • In Section 3.3 [Commands], clarified command buffer head pointer. • In Section 3.3.1 [COMPLETION_WAIT], changed scope of COMPLETION_WAIT. • In Section 3.3.3 [INVALIDATE_IOMMU_PAGES], clarified PDE bit. • In Section 3.3.3 [INVALIDATE_IOMMU_PAGES] and Section 3.3.4 [INVALIDATE_IOTLB_PAGES], corrected Address field. • In Section 3.3.4 [INVALIDATE_IOTLB_PAGES], updated discussion of Maxpend. • In Section 3.3.4 [INVALIDATE_IOTLB_PAGES], added QueueID to support virtual functions. • In Section 3.3.4 [INVALIDATE_IOTLB_PAGES], added software note. • In Section 3.3.4 [INVALIDATE_IOTLB_PAGES], clarified flush of entire IOTLB. • Added Section 3.3.5 [INVALIDATE_INTERRUPT_TABLE]. • In Section 3.3.6 [IOMMU Ordering Rules], added requirements from Section 3.3.5 [INVALIDATE_INTERRUPT_TABLE] and HyperTransport™ tunnel requirements. • In Section 3.3.6.2 [Invalidation Commands Interaction Requirements], clarified Invalidation Completion.
		• In Section 3.4 [Event Logging], corrected event log tail pointer definition to match the definition in the register. • In Section 3.4 [Event Logging], defined “halt command processing”. • In Section 3.4 [Event Logging] updated descriptions to match Section 3.1.3.1 [IOMMU Error Responses]. • In Section 3.4 [Event Logging], clarified error recovery procedures. • In Section 3.4 [Event Logging], clarified logging requirements. • In Section 3.4 [Event Logging], Table 11, reclassified certain errors and renamed ILLEGAL_DEV_TABLE_ENTRY. • In Section 3.4 [Event Logging], Table 11, updated with interrupt remapping and HyperTransport™ special address events. • In Section 3.4 [Event Logging], Table 11, updated illegal level encoding error in IO_PAGE_FAULT event type. • In Section 3.4 [Event Logging], Table 11 updated invalid translation errors, and split out some posted and non-posted writes. • In Section 3.4 [Event Logging], Table 11,updated ILLEGAL_DEV_TABLE_ENTRY, INVALID_DEVICE_REQUEST, and IO_PAGE_FAULT event types.

Date	Rev	Description
		• In Section 3.4.1 [ILLEGAL_DEV_TABLE_ENTRY], updated with interrupt remapping and HyperTransport™ special address events. • In Section 3.4.1 [ILLEGAL_DEV_TABLE_ENTRY], moved text to Table 12. • In Section 3.4.1 [ILLEGAL_DEV_TABLE_ENTRY], added RZ bit to Figure 23 and Table 12. • In Section 3.4.1 [ILLEGAL_DEV_TABLE_ENTRY], Figure 23, corrected low-order address bits. • In Section 3.4.2 [IO_PAGE_FAULT], moved text to Table 13. • In Section 3.4.2 [IO_PAGE_FAULT], updated with interrupt remapping and HyperTransport™ special address events. • In Section 3.4.3 [DEV_TAB_HARDWARE_ERROR], updated with interrupt remapping and HyperTransport™ special address events. • In Section 3.4.3 [DEV_TAB_HARDWARE_ERROR], clarified use of I, TR, and RW. • In Section 3.4.3 [DEV_TAB_HARDWARE_ERROR], moved text to Table 15. • In Section 3.4.4 [PAGE_TAB_HARDWARE_ERROR], updated with interrupt remapping and HyperTransport™ special address events. • In Section 3.4.4 [PAGE_TAB_HARDWARE_ERROR], moved text to Table 16. • In Section 3.4.5 [ILLEGAL_COMMAND_ERROR], moved text to Table 17. • In Section 3.4.6 [COMMAND_HARDWARE_ERROR], moved text to Table 18. • In Section 3.4.7 [IOTLB_INV_TIMEOUT], moved text to Table 19. • In Section 3.4.8 [INVALID_DEVICE_REQUEST], updated with interrupt remapping and HyperTransport™ special address events. • In Section 3.4.8 [INVALID_DEVICE_REQUEST], updated Table 20, added new error code, harmonized with changes to Table 11, and corrected reserved range. • In Section 3.4.8 [INVALID_DEVICE_REQUEST], moved text to Table 21.
		• In Section 3.5 [IOMMU Interrupt Support], updated MSI requirements. • In Section 3.5 [IOMMU Interrupt Support], clarified PassPW.
		• In Section 3.6 [PCI Resources], updated with the requirement that the IOMMU be implemented in an independent function, and the HyperTransport™ UnitId requirements. • In Section 3.6 [PCI Resources], clarified wording and added PCI class and subclass. • In Section 3.6 [PCI Resources], defined behavior of undefined IOMMU registers.

Date	Rev	Description
		• In IOMMU Range Register [Capability Offset 0Ch], added BusNumber, UnitID. • In Section 3.6.1 [IOMMU Capability Block Registers], IOMMU Capability Header [Capability Offset 00h] clarified. • In Section 3.6.1 [IOMMU Capability Block Registers], IOMMU Base Address Low Register [Capability Offset 04h] corrected alignment. • In Section 3.6.1 [IOMMU Capability Block Registers], IOMMU Base Address Low Register [Capability Offset 04h], added locking attribute to Enable; and applied locking to LastDevice, FirstDevice, and BusNumber in IOMMU Range Register [Capability Offset 0Ch]. • In Section 3.6.1 [IOMMU Capability Block Registers], IOMMU Base Address Low Register [Capability Offset 04h], the fields are optionally RO. • In Section 3.6.1 [IOMMU Capability Block Registers], IOMMU Base Address High Register [Capability Offset 08h], updated alignment. • In Section 3.6.1 [IOMMU Capability Block Registers], renamed IOMMU Miscellaneous Information Register [Capability Offset 10h] from IOMMU MSI Message Number Register, added VAsize and PAsize, and corrected MsiNum. • In Section 3.6.1 [IOMMU Capability Block Registers], IOMMU Miscellaneous Information Register [Capability Offset 10h], corrected MsiNum. • In Section 3.6.1 [IOMMU Capability Block Registers], clarified Bus Number field in IOMMU Range Register [Capability Offset 0Ch] description table. • In Section 3.6.1 [IOMMU Capability Block Registers], IOMMU Miscellaneous Information Register [Capability Offset 10h], updated Reserved, VAsize and PAsize. • In Section 3.6.1 [IOMMU Capability Block Registers], clarified NpCache and added implementation note. • In Section 3.6.1 [IOMMU Capability Block Registers], IOMMU Miscellaneous Information Register [Capability Offset 10h], added HtAtsResv.

Date	Rev	Description
		• In Section 3.6.2 [IOMMU Control Registers], added byte-level addressing. • In Section 3.6.2 [IOMMU Control Registers], Device Table Base Address Register [MMIO Offset 0000h], specified RW and reset. • In Section 3.6.2 [IOMMU Control Registers], Device Table Base Address Register [MMIO Offset 0000h], corrected Size and changed Figure 5 [Device Table Entry Fields]. • In Section 3.6.2 [IOMMU Control Registers], Command Buffer Base Address Register [MMIO Offset 0008h], changed buffer alignment. • In Command Buffer Base Address Register [MMIO Offset 0008h], updated ComLen encodings. • In Section 3.6.2 [IOMMU Control Registers], Command Buffer Base Address Register [MMIO Offset 0008h] and Event Log Base Address Register [MMIO Offset 0010h], updated length fields. • In Section 3.6.2 [IOMMU Control Registers], Event Log Base Address Register [MMIO Offset 0010h], changed buffer alignment and clarified write behavior. • In Section 3.6.2 [IOMMU Control Registers], Event Log Base Address Register [MMIO Offset 0010h], corrected EventLen encodings. • In Section 3.6.2 [IOMMU Control Registers], defined IOMMU Control Register [MMIO Offset 0018h] and IOMMU Status Register [MMIO Offset 2020h] to be 64-bit registers. • In Section 3.6.2 [IOMMU Control Registers], IOMMU Control Register [MMIO Offset 0018h], deleted TranCheckDis and expanded InvTimeOut. • In Section 3.6.2 [IOMMU Control Registers], IOMMU Control Register [MMIO Offset 0018h], corrected the assertion state of the Coherent bit, added command fetch, clarified error logging controls. • In Section 3.6.2 [IOMMU Control Registers], IOMMU Control Register [MMIO Offset 0018h], updated target abort terminology. • In Section 3.6.2 [IOMMU Control Registers], IOMMU Control Register [MMIO Offset 0018h], defined 1 and 0 states and clarified ComWaitIntEn behavior. • In Section 3.6.2 [IOMMU Control Registers], IOMMU Exclusion Base Register [MMIO Offset 0020h] and IOMMU Exclusion Limit Register [MMIO Offset 0028h], corrected length of reset value. • In Section 3.6.2 [IOMMU Control Registers], Command Buffer Tail Pointer Register [MMIO Offset 2008h], specified modulo arithmetic. • In Section 3.6.2 [IOMMU Control Registers], Event Log Head Pointer Register [MMIO Offset 2010h] and Event Log Tail Pointer Register [MMIO Offset 2018h], corrected EventHeadPtr and EventTailPtr. • In Section 3.6.2 [IOMMU Control Registers], Command Buffer Head Pointer Register [MMIO Offset 2000h] and Command Buffer Tail Pointer Register [MMIO Offset 2008h], corrected CmdHeadPtr and CmdTailPtr. • In Section 3.6.2 [IOMMU Control Registers], IOMMU Exclusion Limit Register [MMIO Offset 0028h], clarifications. • In Section 3.6.2 [IOMMU Control Registers], IOMMU Status Register [MMIO Offset 2020h] adds CmdBufRun, EventLogRun, and clarified EventOverflow behavior.
		• Deleted Section 4.3 [RequesterID Mapping Capability Block Registers], causing Sections 4.4, 4.5, and 4.6 to be renumbered. • Added Section 4.5 [Software and BIOS Implementation Issues]. • In Section 4.5 [Software and BIOS Implementation Issues], clarified “root device”.

Date	Rev	Description
		• In Section 5 [IOMMU Page Walker Pseudo Code], updated pseudo-code for clarity and completeness.
		• Added Section 6 [Register List].
02/01/06	1.00	• Initial Public Release.

1 Overview

The I/O Memory Management Unit (IOMMU) is a chipset function that translates addresses used in DMA transactions and protects memory from illegal access by I/O devices.

The IOMMU can be used to:

- Replace the existing GART mechanism.
- Remap addresses above 4GB for devices that do not support 64-bit addressing.
- Allow a guest OS running under a VMM to have direct control of a device.
- Provide fine-grain control of device access to system memory.
- Enable a device direct access to user space I/O.

1.1 Intended Audience

This document provides the IOMMU behavioral definition and associated design notes. It is intended for the use chipset designers and programmers involved in the development of low-level BIOS (basic input/output system) functions, drivers, and operating system kernel modules. It assumes prior experience in personal computer chipset design, microprocessor programming, and legacy x86 and AMD64 microprocessor architecture.

1.2 Definitions

- **ATS.** Address translation service.
- **BAR**. PCI defined base address register.
- **Bounce Buffer**. A buffer located in low system memory for DMA traffic from devices that do not support 64-bit addressing. The OS copies the DMA data to or from the buffer to the real buffer in high memory used by the driver.
- **Cold Reset**. A reset generated by removing and reapplying power to the device.
- **Device Exclusion Vector (DEV)**. Contiguous arrays of bits in physical memory. Each bit in the DEV table represents a 4KB page of physical memory (including system memory and MMIO). The DEV table is packed as follows: bit[0] of byte 0 controls the first 4K bytes of physical memory; bit[1] of byte 0 controls the second 4K bytes of physical memory; etc.
- **DeviceID**. A 16 bit device identification number consisting of the Bus number, Device number and Function number.
- **Device Virtual Address**. The untranslated address used by a device in a DMA transaction. If the IOMMU is not enabled this address corresponds to the system physical address.
- **Device Table**. A table in system memory that maps DeviceIDs to DomainIDs and page table root pointers.
- **Domain**. See Protection Domain.
- **DomainID.** A 16-bit number chosen by software to identify a domain.
- **GART**. Graphics Address Remapping Table.
- **Guest**. An application or OS run by the host in its own virtual environment.
- **Guest Physical Address**. An address that is created by using the guest page tables to translate a guest virtual address. The result of the translation is a Guest Physical Address.
- **Guest Virtual Address**. The virtual addresses used by a guest application.
- **High memory.** Memory with addresses at or above 4G bytes.
- **Host**. The system software layer responsible for running guests.
- **Low memory.** Memory with addresses below 4G bytes.
- **IOMMU**. Refers to the I/O Memory Management Unit defined by this specification.
- **MMIO**. Read or write access to memory mapped resources provided by devices.
- **MMU**. Memory Management Unit.

- **Message Signalled Interrupt (MSI)**. An interrupt that is signalled by generating a posted write to a system-defined physical address.
- **Page Tables**. A table structure in main memory used to translate an address from one representation to an alternate representation.
- **PR.** The present bit in the page table entries shown in Figure 6, Figure 7, and Figure 8.
- **Protection Domain**. A set of address mappings and access rights that can be shared by multiple devices.
- **PDE**. Page directory entry for address translation as shown in Figure 8.
- **PTE**. Page translation entry for address translation as shown in Figure 7.
- **System Physical Address**. The address used by the DRAM controller to specify a specific memory location or the address given to a MMIO device to specify a specific MMIO register.

1.3 Bit Attributes

All bit attributes used in this specification are defined in Table 1. These attributes apply to register definitions, device table entries, page table entries, command buffer entries and event log entries.

Table 1: Bit Attribute Definitions

Attribute	Description
HwInit	**Hardware Initialized**: Register bits are initialized by firmware or hardware mechanisms such as pin strapping or serial EEPROM. Bits are read-only after initialization and can only be reset (or write-once by firmware) with a cold reset.
Ignored Ign	**Ignored or Ign:** The state of the bit is a don't care to the IOMMU but is used by the processor MMU.
RO	**Read-only register:** Register bits are read-only and cannot be altered by software.
RW	**Read-Write register:** Register bits are read-write and may be either set or cleared by software to the desired state.
RW1C	**Read-only status, Write-1-to-clear status register:** Register bits indicate status when read, a set bit indicating a status event may be cleared by writing a 1. Writing a 0 to RW1C bits has no effect.
RW1S	**Write-1-to-set register:** Register bits indicate status of an operation when read, setting bit initiates the operation. Hardware clears the bit when the operation completes. Writing a 0 to RW1S bits has no effect.
Reserved Res	**Reserved or Res:** Reserved for future implementations. Bits must be implemented as read only zero.
Unused Un	**Unused or Un:** Bit is not used by hardware. Software is allowed to use the bit for its own purposes.

2 IOMMU Overview

The I/O Memory Management Unit (IOMMU) extends the AMD64 system architecture with support for address translation and access protection on DMA transfers by peripheral devices. The IOMMU enables several significant enhancements to system-level software:

- Legacy 32-bit I/O device support on 64-bit systems (without requiring bounce buffers and expensive memory copies).
- Secure user-level application access to selected I/O devices.
- Secure virtual machine guest operating system access to selected I/O devices.

The IOMMU can be thought of as a combination and generalization of two facilities included in the AMD64 architecture: the Graphics Aperture Remapping Table (GART) and the Device Exclusion Vector (DEV). The GART provides address translation of I/O device accesses to a small range of the system physical address space, and the DEV provides a limited degree of I/O device classification and memory protection. In combination with appropriate software manipulation of host CPU page tables, the IOMMU can provide GART or DEV functionality.

2.1 Architecture Summary

The detailed architecture of the IOMMU is discussed in Chapter 3. The remainder of Chapter 2 consists of a brief summary of the architecture of the IOMMU along with a discussion of some anticipated usage models.

The IOMMU extends the concept of protection domains (domains for short) first introduced with the DEV. The IOMMU allows each I/O device in the system to be assigned to a specific domain and a distinct set of I/O page tables. When an I/O device attempts to read or write system memory, the IOMMU intercepts the access, determines the domain to which the device has been assigned, and uses the TLB entries associated with that domain or the I/O page tables associated with that I/O device to determine whether the access will be permitted as well as the actual location in system memory that will be accessed.

The IOMMU may optionally include support for remote IOTLBs. An I/O device with IOTLB support can cooperate with the IOMMU to maintain its own cache of address translations. This creates a framework for creating scalable systems with an IOMMU in which I/O devices may have different usage models and working set sizes. IOTLB-capable I/O devices contain private TLBs tailored for their own needs, creating a scalable distributed system of TLBs. The performance of IOTLB-capable I/O devices is not limited by the number of TLB entries implemented in the IOMMU.

Major system resources provided by the IOMMU include:

- I/O page tables which the IOMMU uses to provide permission checking and address translation on memory accesses by I/O devices.
- A device table that allows I/O devices to be assigned to specific domains and contains pointers to the I/O devices' page tables.
- An interrupt remapping table which the IOMMU uses to provide permission checking and interrupt remapping for I/O device interrupts.

In summary, the IOMMU is very similar to the processor's MMU, except that it provides address translation and page protection to memory accesses by peripheral devices rather than memory accesses by the processor and that it provides an interrupt remapping capability. However, compared to the processor's MMU, the IOMMU has a few differences.

The first difference is that the IOMMU provides no direct indication to an I/O device of a failed translation when processing an untranslated posted request.

A second difference is related to the organization of the AMD64 system architecture. AMD64 systems can consist of a number of processor and device nodes connected to each other by HyperTransport™ links. The IOMMU can only see and translate memory traffic that is routed through its node in the system fabric. In a system with multiple links and buses to I/O devices (see Figure 1), multiple IOMMUs are required to ensure that each I/O link or bus has appropriate protection and translation applied.

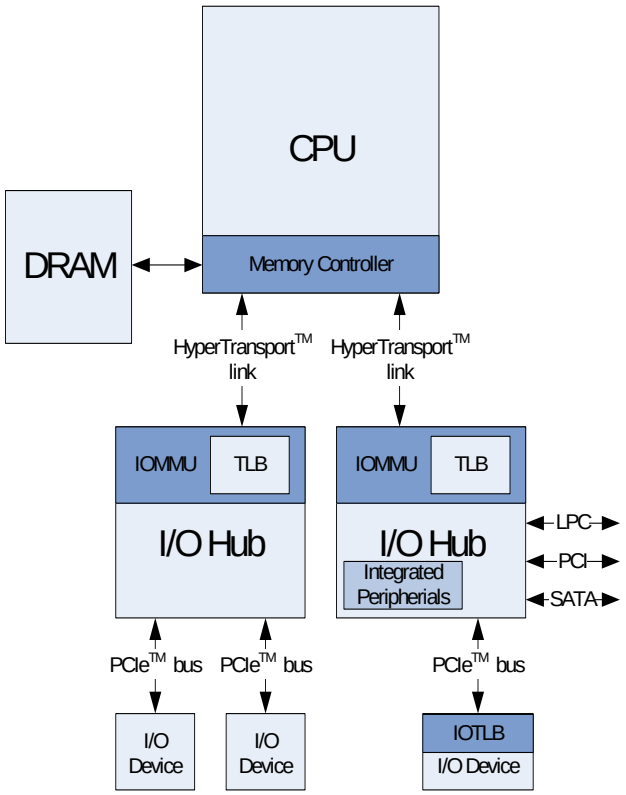


Figure 1: Example Platform Architecture

2.2 Usage Models

Six models are discussed to highlight potential uses of the IOMMU in conventional and virtualized systems. These usage models can enhance system security and stability.

2.2.1 Replacing the GART

The GART is a system facility that performs physical-to-physical translation of memory addresses within a graphics aperture. The GART was defined to allow complex graphical objects, such as texture maps, to appear to a graphics co-processor as if they were located in contiguous pages of memory, even though they are actually scattered across randomly allocated pages by most operating systems. The GART translates all accesses to the graphics aperture, including loads and stores executed by the host CPU as well as memory reads and writes performed by I/O devices. Only accesses whose system physical addresses are within the GART aperture are translated; however, the results of the translation can be any system physical address.

Unlike the GART, the IOMMU translates only memory accesses by I/O devices. However, with appropriate

programming, a host OS can use the IOMMU as a functional equivalent of the GART. First, the host OS must set up its own page tables to perform translations of host CPU accesses formerly translated by the GART. Then, to set up the same translations for I/O device-initiated accesses, the host OS must:

- Construct I/O page tables that specify the desired translations.
- Make an entry in the device table pointing to the newly constructed I/O page tables.
- Notify the IOMMU of the newly updated device table entry if NPcache=1.

At this point, all accesses by both the host CPU and the graphics device will have been mapped to the same pages as they would have been by the GART.

If the host OS wishes to change the page protection or translation, it must update both the processor page tables and, if not shared, the I/O page tables, and issue appropriate page-invalidate commands to both the processor and the IOMMU. Unlike the processor, the IOMMU may require page-invalidate commands after any change to the I/O page tables. (AMD64 processors do not require page-invalidate operations after changes to leaf page table entries that add permission and make no change to translation.) Sharing of page tables is discussed in Section 3.2.1 [Updating Shared Tables] and Section 3.2.4 [Sharing AMD64 CPU and IOMMU Page Tables].

Eventually the host OS may have to tear down the mappings when they are no longer used (e.g., removed from the system). The procedure is similar to setup:

- Mark the device table entry as not valid by setting V=1b, IV=0b, and IntCtl=00b in the device table entry. If transaction pass-through is acceptable, set V=0b and IV=0b.
- Notify the IOMMU of the newly invalidated device table entry.
- Wait for the IOMMU to indicate that the invalidation is complete.
- Finally, de-allocate the I/O page tables.

Since the IOMMU offers no facilities for restarting device accesses to unmapped or protected addresses, all pages that the device might access must be mapped with appropriate permissions. In this respect the IOMMU is no different from the GART.

The IOMMU cannot be used to emulate the GART if processor paging is not enabled; in that case host CPU accesses are not translated. This should not be a problem in practice, however, since historically the GART has only been used by systems that enable paging on the CPU.

In the foregoing procedures for setup and teardown of IOMMU page tables, the order of operations is chosen to prevent the IOMMU from ever looking at device or page table contents before they are initialized. During setup, the I/O page tables are constructed before the pointers are installed, and in teardown the pointers are cleared before the page table is destroyed. Similar principles apply to the other applications in this chapter.

2.2.2 Substituting for the DEV

The Device Exclusion Vector is a simple security mechanism that was introduced with Secure Virtual Machine Architecture. Like the IOMMU, the DEV allows I/O devices to be classified into different domains. Associated with each domain is a bit vector, indexed by physical page address, indicating whether I/O devices in that domain are allowed to access the corresponding physical page.

The IOMMU provides not only protection but also translation. If only protection is needed, software can create identity-mapped I/O page tables that specify the desired protection.

2.2.3 32-bit to 64-bit Legacy I/O Device Mapping

With the advent of large physical memories, legacy 32-bit devices that rely on DMA can no longer access arbitrary system memory. This complicates operating systems, which must introduce a distinction between low memory and high memory, and perform appropriate bookkeeping to ensure that legacy I/O devices are only commanded to perform transfers using low memory. The cost is not just complexity: in order to perform a transfer from a legacy I/O device to high memory, for example, the operating system typically allocates a bounce buffer in low memory, performs the transfer in low memory, and then copies the result to the real destination in high memory. For high-bandwidth I/O devices like disk controllers and network interfaces, the performance cost of bounce buffer allocation and copying can be large.

In some operating systems, the GART has been used to work around this problem. When the OS wishes to perform a transfer between a legacy I/O device and high memory, it allocates a portion of the GART aperture and maps those pages to high memory. It then commands the I/O device to execute the transfer using the address within the GART aperture, which must be located in low memory. Although this approach avoids the cost of bounce buffer copies, it is less than desirable, since the relatively small GART aperture must be shared by all legacy I/O devices and any graphics processors in the system. In the best case, device drivers will have additional locking and synchronization overhead associated with page allocation and de-allocation in the GART aperture; in the worst case, system performance is actually degraded due to serialization waiting for the GART aperture to become available.

The IOMMU creates a better solution. First of all, IOMMU translation applies to the full range of addresses an I/O device can generate, rather than requiring high-memory transfers to be mapped only within the narrow range of GART addresses. Moreover, the IOMMU's ability to assign each I/O device to a different domain means that heavily used I/O devices can be given their own sets of I/O page tables, and do not have to contend with other I/O devices for allocation and de-allocation of I/O pages.

2.2.4 User Mode Device Accesses

The IOMMU plays a crucial role in allowing arbitrary I/O devices to be safely controlled by user-level processes, since I/O devices whose memory accesses are translated by the IOMMU can only access pages that are explicitly mapped by the associated I/O page tables. The I/O devices' access can therefore be limited to only those pages to which the user processes legitimately have access.

Setting up the IOMMU for user-level I/O to an I/O device may be set up similar to GART emulation, with two differences: first, the mappable address range is the entire range of I/O device-generatable addresses, and secondly the operating system is not necessarily required to make exactly equivalent mappings in the CPU page tables (although most likely it will).

Even with the help of the IOMMU, enabling user level I/O device access involves many design considerations. Protecting and remapping DMA is one part of the problem; the other part is interrupt management, for which the IOMMU provides help.

As was the case with GART emulation, system software will have to assess the need to lock in memory all pages that might ever be accessed by an I/O device controlled by a user-level process.

2.2.5 Virtual Machine Guest Access to Devices

The IOMMU can be used to allow unmodified virtual machine guest operating systems to directly access I/O devices. This is really just a special case of allowing user-level access to I/O devices, but there are a few considerations that warrant separate mention.

First of all, a non-VM-aware guest will have no way of informing its Virtual Machine Monitor (VMM) which pages an I/O device might access, so the VMM must lock the entire guest in memory. The VMM's I/O page tables for the guest should then simply map guest physical addresses to system physical addresses. If the VMM is running the guest under nested paging and is using host page tables built to be compatible with the IOMMU, then the IOMMU can directly share the host page tables for the guest.

Often a single VM guest will have direct access to multiple I/O devices. By design, all I/O devices in the guest that need to see exactly the same I/O page translations can share a DomainID. If all the I/O devices belonging to a given VM guest are assigned to the same domain then the IOMMU can share translation cache entries among any of the guest's I/O devices.

Finally, guest I/O throughput will probably be significantly enhanced if guest memory is allocated using large pages on the host system. Then the I/O page tables can similarly use large pages and the IOMMU will be more likely to avoid thrashing in its translation cache.

2.2.6 Virtualizing the IOMMU

The IOMMU has been designed so that it can be emulated in software by a VMM that wishes to present its guests the illusion that they have an IOMMU.

All VMMs that run non-VM-aware guests already intercept and emulate attempts by their guests to access PCI configuration space. Therefore emulation of the IOMMU configuration registers is straightforward; the emulation can be hooked directly to the existing facilities of the VMM for intercepting PCI configuration space accesses.

The VMM must also arrange to intercept and emulate guest accesses to the IOMMU's MMIO-mapped command registers. Since the overhead of each VMM intercept is high, guest operating systems accessing the IOMMU will have better performance if they enqueue batches of commands in the IOMMU's (DRAM-based) command buffer prior to initiating IOMMU command processing via an MMIO register access.

Since an untrusted guest OS cannot be allowed to write in the real device table, the VMM must maintain shadow entries in the real table on behalf of the guest. The IOMMU architecture requires software to issue invalidate-entry commands to the IOMMU after updating device table entries. The VMM can intercept these invalidate commands, look up the corresponding entries in the guest's simulated device table, and make shadow entries in the real device table on behalf of the guest. Note that the DeviceIDs as seen by the guest need not be the same as the real DeviceIDs, and the DomainIDs used by the guest will almost certainly not be the same as the DomainIDs used by the VMM in the real device table.

In addition, for each guest I/O page table, the VMM will have to construct a shadow I/O page table. This shadow I/O page table is the page table that will be given to the real IOMMU. Unfortunately, since a failed I/O device access cannot be restarted, the VMM will have to construct each guest domain's complete shadow I/O page tables eagerly as soon as the guest enables paging for that domain. The VMM will have to write-protect guest I/O page tables from the guest, in order to intercept all guest updates and propagate the updates to the shadow I/O page tables.

3 Architecture

This chapter describes the IOMMU's architecture mainly from a system software point of view. The discussion starts with the normal steady state behavior of the IOMMU once it has been set up, focusing on how the IOMMU handles various device transactions. The following section describes the in-memory data structures used to control the IOMMU, together with the procedures software must follow to correctly update these (shared) data structures. Finally, the chapter concludes with a description of the PCI resources that must be initialized at system startup time to configure the IOMMU.

3.1 Behavior

When the IOMMU is disabled it simply passes all bus traffic through without alteration.

When the IOMMU is enabled, it intercepts requests arriving from downstream devices (which may be HyperTransport™ link or PCI based), performs permission checks and address translation on the requests, and sends translated versions upstream via the HyperTransport™ link to system memory space. Other requests are passed through unaltered (details in Section 3.1.1 [Normal Operation]).

The IOMMU reads a variety of tables in system memory to perform its permission checks, interrupt remapping, and address translations. To ensure deadlock free operation, memory accesses for device tables entries, page table walks, and interrupt remapping tables by the IOMMU use an isochronous virtual channel and may only reference addresses in system memory. Other memory reads originated by the IOMMU to command buffers and event logs use the normal virtual channel. System performance could be substantially reduced if the IOMMU performed the full table lookup process for every device request it handled. Implementations of the IOMMU are therefore expected to maintain internal caches for the contents of the IOMMU's in-memory tables, and correct operation of the IOMMU requires system software to send appropriate invalidation commands when it updates table entries that may have been cached by the IOMMU.

The IOMMU writes to the event log in system memory; these writes use the normal virtual channel.

The IOMMU signals interrupts using standard PCI INTx, MSI, or MSI-X interrupts.

3.1.1 Normal Operation

The normal flow of requests through the IOMMU is as follows:

- Read, write and interrupt transactions generated by the IOMMU are not translated by the IOMMU.
- Transactions arriving from upstream must be passed downstream unaltered.
- Transactions arriving from downstream that are response, fence, or flush commands must be passed upstream unaltered.
- Memory read and write transactions from downstream result in table lookups in the device table to obtain the DomainID of the requesting I/O device and to locate I/O page tables, and then in I/O page tables to perform address translation and permission checking. After performing permission checks and address translation, the IOMMU forwards the resulting transactions upstream if the transaction is allowed from the I/O device.
- Interrupts from downstream result in table lookups in the device table and then in the interrupt remapping tables to remap the interrupt. After performing checks and interrupt remapping, the IOMMU forwards the resulting interrupts upstream if the interrupt is allowed from the I/O device.
- Port I/O space transactions from downstream devices result in a device table lookup to determine if the I/O device is allowed to access port I/O space.
- The IOMMU maintains an event log containing the details of failed transactions.

- The IOMMU does not translate pre-translated memory read and write requests from devices if the I/O device is marked as being able to translate addresses (I=1b in the device table entry, Section 3.2.2.1 [Device Table Entry Format])..

In addition to passing on transactions from downstream devices, the IOMMU will insert transactions of its own to perform reads and writes from memory and to signal interrupts.

3.1.2 IOMMU Logical Topology

Once configured, the IOMMU logically resides on the HyperTransport™ bus between the devices it translates for and the upstream interface. As a result of this logical topology the transactions seen by the IOMMU are HyperTransport™ transactions. Accesses to the HyperTransport™ address range FD_0000_0000h - FF_FFFF_FFFFh have special meaning. The meaning is encoded into various portions of the address as shown in Table 2 and Table 9; complete details are in the *HyperTransport™ I/O Link Specification*. Upstream transactions to these address ranges are controlled by device table control bits, page tables or the interrupt remapping tables.

Table 2: Special Address Controls

Base Address	Top Address	Use	Access controlled by
FD_0000_0000h	FD_F7FF_FFFFh	Reserved interrupt address space	See Section 3.4.8 [INVALID_DEVICE_REQUEST]
FD_F800_0000h	FD_F8FF_FFFFh	Interrupt/EOI	IntCtl, Interrupt Remapping Tables
FD_F900_0000h	FD_F90F_FFFFh	Legacy PIC IACK	Page Tables
FD_F910_0000h	FD_F91F_FFFFh	System Management	SysMgt, Page Tables
FD_F920_0000h	FD_FAFF_FFFFh	Reserved	Page Tables
FD_FB00_0000h	FD_FBFF_FFFFh	Address Translation	HtAtsResv, Page Tables
FD_FC00_0000h	FD_FDFF_FFFFh	I/O Space	IoCtl, Page Tables
FD_FE00_0000h	FD_FFFF_FFFFh	Configuration	Page Tables
FE_0000_0000h	FE_1FFF_FFFFh	Extended Configuration/ Device Messages	Page Tables
FE_2000_0000h	FF_FFFF_FFFFh	Reserved	Page Tables

During configuration, an IOMMU may appear connected in different topologies that are implementation dependent.

3.1.3 IOMMU Error Reporting

The IOMMU must detect and may report several kinds of errors that may arise due to malfunctioning hardware or software. When the IOMMU detects an error of any kind and event logging is enabled, it writes an appropriate error entry into the event log located in system memory. In addition, it may optionally signal an interrupt when the event log is written.

Errors detected by the IOMMU include I/O page faults as well as memory errors due to I/O page table walks.

Software note: the TLB caching behavior of the IOMMU is not defined for an entry causing an error; some implementations may insert an entry in the TLB cache before verifying that it causes no errors. System software should invalidate the entry that caused the error.

3.1.3.1 IOMMU Error Responses

The IOMMU response to errors depends on the type of error detected, the type of transaction that caused the error, and the state of the IOMMU at the time of the error.

If an IOMMU is not enabled or does not support address translation requests, the IOMMU will respond to translation requests with a master abort.

If the IOMMU is enabled, it can have one of three error responses:
- For upstream transactions that are master aborted or target aborted, the PCI/Host bridge that is co-located with the IOMMU is the completer of the transaction. Transactions that are target aborted will set the legacy Signaled Target Abort bit in a manor consistent with the bus specification over which the transaction was received (secondary port). These aborted transactions should not set any AER bits (if implemented and otherwise applicable).
- Errors in transactions that target the IOMMU function will not be logged in the IOMMU event log. They will signal the error following the rules of the bus specification applicable to the primary bus with which the IOMMU function is associated.
- Errors in the transactions originating from the IOMMU function will signal the error following the rules of the bus specification applicable to the primary bus with which the IOMMU is associated. Additionally, errors in command buffer and table walk reads will be logged in the IOMMU event log.

A transaction that attempts to use a device table entry beyond the end of the device table is treated as if it were using a device table entry with the following settings: V=1b, IV=1b, and the remainder set to zero. Section 3.2.2.1 [Device Table Entry Format] defines the control bits and the size of the device table is defined by the Device Table Base Address Register, MMIO Offset 0000h[Size].

3.1.3.2 I/O Page Faults

IOMMU processing of a device request may result in an I/O page fault. These faults can arise for a variety of reasons, such as I/O page table entries lacking sufficient permission or marked not present. In a traditional CPU paging implementation, page faults activate an exception handler that has the option of attempting to correct the underlying problem and retry the faulting instruction. The IOMMU has no such option: the underlying HyperTransport™ and PCI bus protocols provide no means for the IOMMU to signal a device that it should attempt to retry an access. Consequently, when the IOMMU detects an I/O page fault, it target aborts the faulting request. The IOMMU sets the legacy Signaled Target Abort bit, if appropriate, and records I/O page faults in its event log when event logging is enabled.

3.1.3.3 Memory Access Errors

The IOMMU's own memory accesses to its in-memory tables may themselves result in several kinds of errors, including:

- Accesses to nonexistent or non-DRAM addresses (the IOMMU's isochronous virtual channel is restricted to DRAM addresses only).
- Uncorrectable ECC errors.
- Reserved value errors, including invalid or unsupported type codes in device table entries and reserved bits in page table entries.

The IOMMU records all detected memory access errors in its event log when event logging is enabled.

3.1.4 Special Conditions

In some bus architectures, a zero-byte read operation is defined as a special operation with well-defined side effects. Because of these side effects, the IOMMU must permit a zero-byte read operation when a page is marked to allow either read or write access. Further, because the zero-byte read operation returns undefined data in some bus specifications, protecting the contents of a non-readable memory location requires that the IOMMU obscure the returned data for a zero-byte read operation.

Implementation note: methods to obscure the returned data in a zero-byte read operation include returning a constant, a random value, or a predictable value not based on the data contents such as the address.

Accesses to the interrupt address range (Table 2) are defined to go through the interrupt remapping portion of the IOMMU and not through any address translation processing. Therefore, when a transaction is being processed as an interrupt remapping operation, the transaction attribute of pre-translated or untranslated is ignored.

3.2 Data Structures

Host software must maintain five types of in-memory data structures for use by the IOMMU. These data structures are:

1. The *device table* is a table indexed by DeviceIDs. Each device table entry contains mode bits, a pointer to the I/O page tables, a pointer to an interrupt remapping control table, and a 16-bit DomainID. The DomainID acts as an address space identifier, allowing multiple devices sharing the same I/O page tables to share the same translation cache resources on the IOMMU. The DomainID must be the same for all devices that share the same page tables.
2. The *I/O page table(s)*: Each device table entry may specify different I/O page tables, or different device table entries may share the same I/O page tables. Each time the IOMMU processes a device access to memory, it looks up the device virtual address in its translation cache and/or the appropriate I/O page tables to determine whether the device has permission, as well as (if permitted) the system physical address to access.
3. The *command buffer:* The IOMMU accepts commands queued by the CPU through a circular buffer located in system memory.
4. The *event log*: The IOMMU reports errors to the CPU by means of another circular buffer, also located in system memory. The event log is the only data structure in system memory that is written by the IOMMU.
5. The *interrupt remapping table(s):* Each device table entry may specify an interrupt remapping table. Each time the IOMMU processes a device interrupt request, it looks up the interrupt in the interrupt remapping table to redirect the interrupt to the destination with a translated vector.

Figure 2 illustrates the relationships among the IOMMU data structures.

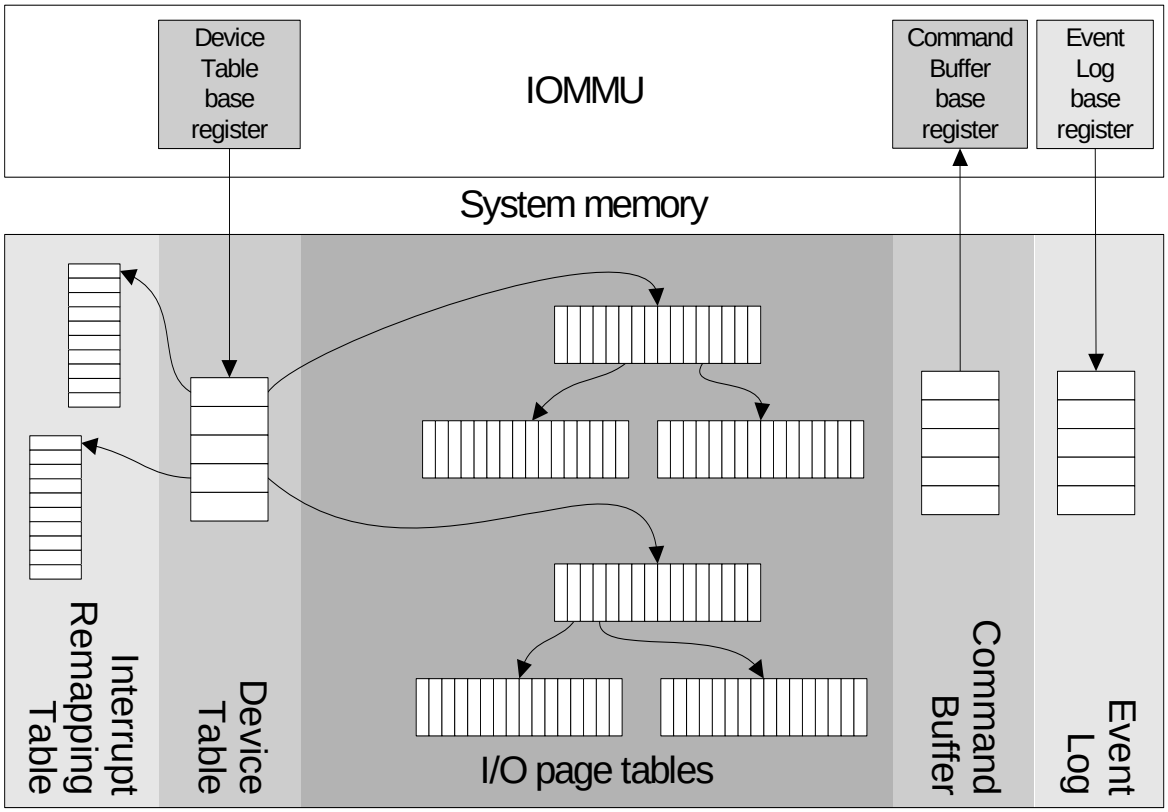


Figure 2: IOMMU Data Structures

3.2.1 Updating Shared Tables

The I/O page table structures have been designed so they can be shared among CPUs and IOMMUs. The table structures (interrupt remapping table, device table, and I/O page tables) can be shared among IOMMUs. Shared tables have similar requirements for safe updates by system software.

When updating a table entry, system software is encouraged to use aligned 64-bit accesses although control bits are defined that allow system software updating a table to use byte accesses.

Each table can also have its contents cached by the IOMMU or downstream IOTLBs. Therefore, after updating a table entry that can be cached, system software must send the IOMMU an appropriate invalidate command.

3.2.2 Device Table

I/O devices that originate transactions are identified by a 16-bit DeviceID that is used to index the device table. The content of the DeviceID is fabric-dependent; for example, Figure 3 shows how PCIe® and PCI-X® RequesterIDs are mapped into IOMMU DeviceIDs, and Figure 4 shows how HyperTransport™ UnitIDs are mapped into IOMMU DeviceIDs.

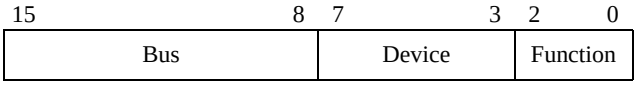
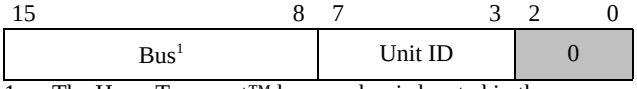


Figure 3: DeviceID Derived from Peripheral RequesterID



1. The HyperTransport™ bus number is located in the Slave/Primary Interface Block associated with the HyperTransport™ link that the traffic was received from.

Figure 4: DeviceID Derived from Peripheral UnitID

The device table is represented as an array of 256-bit entries located in contiguous system memory. Since there are 64K possible DeviceIDs, the device table may be up to 2M bytes in length. 8K2MThe Device Table Base Address Register [MMIO Offset 0000h], controls the system physical address and size of the device table. The device table must be aligned at a 4K byte boundary in system memory and must be a multiple of 4K bytes in length. The IOMMU must read the entire device table entry in two 128-bit transactions (as defined by the scope of the validity indicators) or a single 256-bit transaction.

When the IOMMU is enabled, any I/O device whose DeviceID is beyond the end of the device table is denied I/O permission (the IOMMU will target abort the access), and all attempted accesses by such I/O devices are logged when event logging is enabled.

If an I/O device uses PCI phantom functions, software must replicate device table entries such that index calculations retrieve the correct entries for any phantom function used by the I/O device.

3.2.2.1 Device Table Entry Format

Device table entries take the following form:

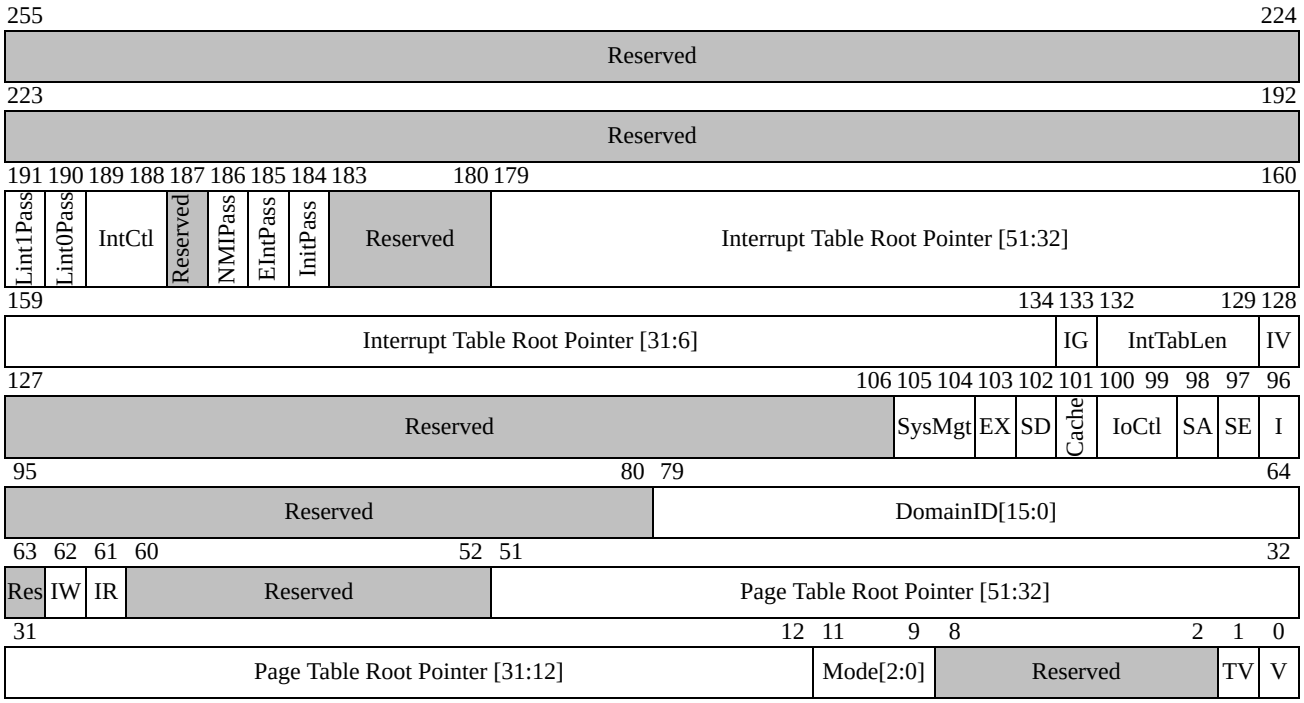


Figure 5: Device Table Entry Fields

Fields of device table entries are:

Table 3: Device Table Entry Field Definitions

Bits	Description
255:192	Reserved. **Note:** Non-zero bits in this field are reported as an error when IV=1
191	**Lint1Pass: LINT1 (legacy PIC NMI) pass-through**. This bit enables device initiated LINT1 interrupts to be forwarded by the IOMMU. 1=Device initiated LINT1 interrupts are forwarded unmapped. 0=Device initiated LINT1 interrupts are target aborted by the IOMMU. See also Table 6.
190	**Lint0Pass: LINT0 (legacy PIC ExtInt) pass-through**. This bit enables device initiated LINT0 interrupts to be forwarded by the IOMMU. 1=Device initiated LINT0 interrupts are forwarded unmapped. 0=Device initiated LINT0 interrupts are target aborted by the IOMMU. See also Table 6.
189:188	**IntCtl: interrupt control**. This field controls how fixed and arbitrated interrupt messages are handled. Fixed and arbitrated interrupt messages use a HyperTransport™ special addresses as shown in Table 2 and Table 9. 00b=Fixed and arbitrated interrupts target aborted 01b=Fixed and arbitrated interrupts are forwarded unmapped 10b=Fixed and arbitrated interrupts remapped 11b=Reserved See also Table 5. If IntCtl=10b, a valid interrupt table root pointer must be present; otherwise the interrupt table root pointer is not used. **Note:** IntCtl=11b is reported as an error when IV=1.
187	**Reserved.** **Note:** Non-zero bits in this field are reported as an error when IV=1**.**
186	**NMIPass: NMI pass-through.** 1=pass through NMI interrupt messages unmapped. 0=NMI interrupt message is target aborted by the IOMMU. See also Table 6.
185	**EIntPass: ExtInt pass-through.** 1=pass through ExtInt interrupt messages unmapped. 0=External interrupt message is target aborted by the IOMMU. See also Table 6.
184	**InitPass: INIT pass-through.** 1=pass through INIT interrupt messages unmapped. 0=INIT interrupt message handling target aborted by the IOMMU. See also Table 6.
183-180	Reserved. **Note:** Non-zero bits in this field are reported as an error when IV=1.
179:134	**Interrupt table root pointer**. The interrupt table root pointer is only used when interrupt translation is enabled (IntCtl=10b). It contains the system physical address of the base address of the interrupt remapping table for the I/O device. The interrupt table must be aligned to start on a 128-byte boundary.
133	**IG: ignore unmapped interrupts.** 1=Supress event logging for interrupt messages causing IO_PAGE_FAULT events. 0=creation of event log entries for IO_PAGE_FAULT events is controlled by SupIOPF in the interrupt remapping table entry (see Section 3.2.5 [Interrupt Remapping Tables]).

Table 3: Device Table Entry Field Definitions

132:129	**IntTabLen: interrupt table length**. This field specifies the length of the interrupt remapping table. 0000b = 1 entry 0001b = 2 entries 0010b = 4 entries 0011b = 8 entries ... 1010b = 1024 entries 1011b = 2048 entries 11xxb = reserved **Note:** IntTabLen=11xxb is reported as an error when IV=1.
128	**IV: interrupt map valid.** 1=Interrupt map information in bits [255:129] is valid. 0=Interrupt remapping information is not valid and interrupts are passed through the IOMMU unmapped. See also Table 5 and Table 6.
127:106	Reserved **Note:** Non-zero bits in this field are reported as an error when V=1.
105:104	**SysMgt: system management message enable**. Specifies whether device-initiated system management messages are blocked, forwarded, or translated by the IOMMU. 00b=Device initiated DMA transactions in the system management address range are target aborted by the IOMMU. 01b=Device initiated system management messages, including INTx messages, are forwarded untranslated by the IOMMU. Upstream reads or non-posted writes are target aborted. 10b=Device initiated INTx messages are forwarded by the IOMMU untranslated; device initiated system management messages other than INTx messages are target aborted. Upstream reads and non-posted writes are target aborted. 11b=Device initiated DMA transactions in the system management address range are translated by the IOMMU.
103	**EX: allow exclusion**. 1=Accesses from this device that address the IOMMU exclusion range are excluded from translation and access checks. 0=Accesses from this device to the IOMMU exclusion range are translated and checked for access rights. See IOMMU Exclusion Base Register [MMIO Offset 0020h] and IOMMU Exclusion Limit Register [MMIO Offset 0028h].
102	**SD: snoop disable**. 1=IOMMU page table walk transactions for this device are not snooped. HyperTransport™ transactions by an IOMMU must not set the coherent bit in page table walk requests for this device. 0=IOMMU page table walk transactions for this device are snooped. HyperTransport™ transactions by an IOMMU must set the coherent bit in page table walk requests for this device. See also the Coherent bit in the IOMMU Control Register [MMIO Offset 0018h].
101	**Cache: IOTLB cache hint**. 1=Caching of translations for explicit translation requests is not recommended. **Implementation Note**: This bit is a recommendation not to cache the page translation entry associated with a translation request when U=0. The caching of page directory entries associated with the translation is recommended.

Table 3: Device Table Entry Field Definitions

100:99	**IoCtl: port I/O control**. Specifies whether device-initiated port I/O space transactions are blocked, forwarded, or translated. 00b= Device-initiated port I/O is not allowed. The IOMMU target aborts the transaction if a port I/O space transaction is received . 01b= Device-initiated port I/O space transactions are allowed. The IOMMU must pass port I/O accesses untranslated. 10b= Transactions in the port I/O space address range are translated by the IOMMU page tables as memory transactions. 11b=Reserved. **Note:** IoCtl=00b and IoCtl=01b control the forwarding upstream of port I/O, if it is implemented. **Note:** IoCtl=11b is reported as an error when V=1.
98	**SA: suppress all page fault errors**. 1=Suppress event logging for all IO_PAGE_FAULT errors caused by memory accesses from this I/O device. **Note:** SA does not affect events logged due to interrupts or IOMMU command processing. **Note:** When V=0 the value of SA is ignored by the IOMMU.
97	**SE: suppress page fault errors**. Suppress event logging for IO_PAGE_FAULT errors if an IO_PAGE_FAULT error has already been logged in the event log for this I/O device. 1=The IOMMU must only update the event log with an IO_PAGE_FAULT for the first page fault seen for the device as long as the DeviceID remains in the device cache. The IOMMU clears all state associated with this bit when an INVALIDATE_DEVTAB_ENTRY command is received for the device or when the DeviceID is replaced in the cache by a different DeviceID. SA does not affect events logged due to interrupts or IOMMU command processing. **Software note:** The SE bit controls a mechanism that reduces the number of event log entries on a per-device basis. The degree of filtering depends on the behavior of the device table cache. As such, software should not assume that only a single entry per device will be made in the event log. **Note:** SE does not affect events logged due to interrupts or IOMMU command processing. **Note:** When V=0 the value of SE is ignored by the IOMMU
96	**I: IOTLB support**. 1=I/O device has its own IOTLB. IOTLB support is an optional feature of the IOMMU. 0=I/O device has no IOTLB support. I/O devices with IOTLBs are capable of performing their own I/O page translation and rely on the IOMMU to perform page table lookups. When I=1, the IOMMU is enabled to process page walk requests from devices. This bit does not affect interrupts from the I/O device.
95:80	Reserved. **Note:** Non-zero bits in this field are reported as an error when V=1.
79:64	**DomainID**. The DomainID is a 16-bit integer chosen by software that the IOMMU must use to tag its internal translation caches and to mark event log entries. I/O devices with different page tables must be given different DomainIDs. I/O devices that share the same page tables may be given the same DomainID. I/O devices that share the same DomainID must have the same settings in the Mode field and have the same page table root pointer, however they may have different values in the I and SysMgt fields. If devices with the same DomainID are given different non-zero modes or different page table root pointers, the behavior of the IOMMU is undefined. The value of the DomainID recorded in an event log entry is undefined when V=0 and IV=1.
63	Reserved. **Note:** A non-zero value in this field is reported as an error when V=1.
62	**IW: I/O write permission**. 1=I/O device is allowed to perform DMA write transactions and 0-byte read transactions (see Section 3.1.4 [Special Conditions]). 0=Device initiated DMA write and atomic transactions are target aborted.

Table 3: Device Table Entry Field Definitions

61	**IR: I/O read permission**. 1=I/O device is allowed to perform DMA read transactions. 0=Device initiated DMA read and atomic transactions are target aborted.
60:52	Reserved. **Note:** Non-zero bits in this field are reported as an error when V=1.
51:12	**Page Table Root Pointer**. The page table root pointer contains the system physical address of the root page table for the I/O device. The pointer is only used in modes where page translation is enabled.
11:9	**Mode: paging mode**. Specify how the IOMMU performs page translation on behalf of the device. If page translation is enabled, the mode specifies the depth of the device's I/O page tables (1 to 6 levels). 000b Translation disabled (Access controlled by IR and IW bits) 001b 1 Level Page Table (provides a 21-bit device virtual address space) 010b 2 Level Page Table (provides a 30-bit device virtual address space) 011b 3 Level Page Table (provides a 39-bit device virtual address space) 100b 4 Level Page Table (provides a 48-bit device virtual address space) 101b 5 Level Page Table (provides a 57-bit device virtual address space) 110b 6 Level Page Table (provides a 64-bit device virtual address space) 111b Reserved **Note:** the page table root pointer is ignored when Mode=000b and when Mode=111b. **Note:** Mode=111b is reported as an error when V=1 and TV=1.
8:2	Reserved. **Note:** Non-zero bits in this field are reported as an error when V=1.
1	**TV: translation information valid.** 1=Page translation information is valid, specifically IW, IR, the page table root pointer, and Mode. TV is not meaningful when V=0.
0	**V: valid**. 1=Device table entry bits [127:1] are valid. 0=Device table entry bits [127:1] are invalid and transactions not intercepted by the interrupt remapping portion are passed through. **Note**: The interrupt remapping portion of the device table entry is controlled by the IV bit. **Software note:** DomainID must be valid when address translation is enabled.

The interaction of the V, TV, IV, and IntCtl control bits are summarized in Table 4 and Table 5. The interaction of IV and the pass control bits is defined in Table 6. The event log entries for operations causing a target abort are defined in Section 3.4 [Event Logging].

Table 4: V and TV Fields in Device Table Entry

V	TV	Description
0	X	All addresses are forwarded without translation; individual control fields invalid.
1	0	The SysMgt, EX, SD, Cache, IoCtl, SA, SE, and I fields are valid. The value of DomainID will be used for event log entries. If the request requires a table walk, the table walk will be terminated.
1	1	All fields in bits [127:2] are valid.

Table 5: IV and IntCtl Fields in Device Table Entry for Fixed and Arbitrated Interrupts

IV	IntCtl	Description
0	X	All interrupts are forwarded without remapping.
1	00b	All fixed and arbitrated interrupts are target aborted.
1	01b	All fixed and arbitrated interrupts are forwarded without remapping.
1	10b	All fixed and arbitrated interrupts are remapped.
1	11b	Behavior undefined

Table 6: IV and Pass Fields in Device Table Entry for Selected Interrupts

IV	Pass Field Name	Pass Field=0b	Pass Field=1b
0	X	LINT0, LINT1, SMI, NMI, INIT, and ExtInt interrupts are passed through unmapped.	
1	X	SMI interrupts are passed through unmapped. There is no pass field to control SMI requests.	
1	Lint0Pass	LINT0 interrupts are target aborted.	LINT0 interrupts are passed through unmapped.
1	Lint1Pass	LINT1 interrupts are target aborted.	LINT1 interrupts are passed through unmapped.
1	NMIPass	NMI interrupts are target aborted.	NMI interrupts are passed through unmapped.
1	INITPass	INIT interrupts are target aborted.	INIT interrupts are passed through unmapped.
1	EIntPass	ExtInt interrupts are target aborted.	ExtInt interrupts are passed through unmapped.

3.2.2.2 Making Device Table Entry Changes

This section contains information for software that changes the IOMMU tables. - Software should issue invalidate commands after certain types of changes to tables and note that I/O device accesses are neither queued nor throttled by the IOMMU. Software may change the interrupt remapping information independently of the address translation information in a device table entry. These operational sequences are general and system conditions may allow optimizations.

Software may change the interrupt remapping information in a device table entry with a single 64-bit write. The change must be followed by an INVALIDATE_DEVTAB_ENTRY command when either the initial IV=1b or the initial V=1b. If a 64-bit operation cannot be used, software may change the interrupt remapping information in the device table entry in the following manner, according to the initial value of IV in the relevant device table entry.

- If IV=0b, changes can be made in any order as long as the last change is to set to IV=1b; an INVALIDATE_DEVTAB_ENTRY command is required when V=1b.
- If IV=1b, the following steps may be followed to change interrupt remapping information for fixed and arbitrated interrupts:
  - Set IntCtl=00b in the device table entry to block interrupts; any device-initiated interrupts for the domain will be target aborted and, when enabled, logged to the event log.

  - Update the interrupt table root pointer, IG, and IntTabLen.
  - Invalidate the interrupt table if the interrupt table root pointer or IntTabLen was changed (see Section 3.3.5 [INVALIDATE_INTERRUPT_TABLE]).
  - Change IntCtl to cease blocking interrupts from the device (set IntCtl=01b or 10b).
  - Invalidate the device table entry (see Section 3.3.2 [INVALIDATE_DEVTAB_ENTRY]).
- If IV=1b, the following steps will change interrupt control information in the device table entry for NMI, Lint0, Lint1, Init, and ExtInt interrupts:
  - Update Lint1Pass, Lint0Pass, IntCtl, NMIPass, EIntPass, and InitPass. The setting of IntCtl can be changed at the same time.
  - Invalidate the device table entry for the device (see Section 3.3.2 [INVALIDATE_DEVTAB_ENTRY]).

Software may change the address translation information in a device table entry with a single 128-bit write operation followed by an INVALIDATE_DEVTAB_ENTRY command when either the initial IV=1b or the initial V=1b. If a 128-bit operation cannot be used, software may change the address translation information in the following ways, according to the initial values of V and TV.
- If V=0b, address translation changes can be made in any order as long as the last change is to set V=1b. An INVALIDATE_DEVTAB_ENTRY command is required if IV=1b.
- If V=1b, software can use the following steps to set the IOMMU to pass addresses untranslated with access controlled by IR and IW, depending on the value of TV.
  - If TV=0b, set values for IW, IR, Mode=000b, and TV=1b (maintaining V=1b), then issue an INVALIDATE_DEVTAB_ENTRY command. If not done as a 64-bit write, the values of TV and V must be in the final change. Note that the DomainID and other values in bits [127:96] are already valid because V=1b.
  - If TV=1b, software must change IW and IR concurrently with or before changing Mode, and the values of TV and V must be in the final change. Software then issues an INVALIDATE_DEVTAB_ENTRY command.

To start the IOMMU, activating table walking, etc., use the following procedure after a system reset.
- If not previously set, initialize the following registers:
  - the Device Table Base Address Register [MMIO Offset 0000h],
  - the Command Buffer Base Address Register [MMIO Offset 0008h],
  - the Command Buffer Head Pointer Register [MMIO Offset 2000h],
  - the Command Buffer Tail Pointer Register [MMIO Offset 2008h],
  - the IOMMU Exclusion Base Register [MMIO Offset 0020h] and the IOMMU Exclusion Limit Register [MMIO Offset 0028h], if used,
  - the Event Log Base Address Register [MMIO Offset 0010h], the Event Log Head Pointer Register [MMIO Offset 2010h] and the Event Log Tail Pointer Register [MMIO Offset 2018h], if used.
- Write the IOMMU Control Register [MMIO Offset 0018h] with EventLogEn=1b (if used), CmdBufEn=1b, and IommuEn=1b. Other IOMMU Control Register [MMIO Offset 0018h] bits should be set as necessary.

The IOMMU is now operational and will process device transactions and fetch command buffer entries. When enabled, the IOMMU will create event log entries as events occur.

3.2.3 I/O Page Tables

The IOMMU uses a new page table structure designed to support a full 64-bit device virtual address space while allowing fast translation in many common cases. The IOMMU page tables are a generalization of AMD64 long mode page tables. The IOMMU page tables are a multi-level tree of 4K tables indexed by groups of 9 virtual address bits (determined by the level within the tree) to obtain 8-byte entries. Each page table entry is either a page directory entry pointing to a lower-level 4K page table, or a page translation entry specifying a system physical page address. A page translation entry is a page table entry with the Next Level field set to 0h or 7h. A page directory entry is a page table entry with the Next Level field not equal to 0h or 7h.

The first generalization in the IOMMU page tables compared to AMD64 CPU page tables is that directory entries, in addition to specifying the address of the lower page table, also specify the level, or grouping of bits within the virtual address, that is used for the next page table lookup step. This allows the IOMMU to skip page translation steps in cases where the virtual address often contains long strings of 0 bits, such as software architectures that allocate virtual memory sparsely.

The second generalization in the IOMMU page tables is that page translation entries can specify the page size of the translation. The default page size of a translation can be overridden by setting the Next Level bits to 7h. When the Next Level bits are 7h, the size of the page is determined by the first zero bit in the page address, starting from bit 12 (illustrated in Table 7). . The page size specified by this method must be larger than the default page size and smaller than the default page size for the next higher level.

Table 7: Example Page Size Encodings

Level	Address Bits																				Page Size	Default Page Size
	63:32*	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	
1	Page Address																				0	8K	4K
1	Page Address																			0	1	16K	4K
1	Page Address													0	1	1	1	1	1	1	1	1M	4K
2	Page Address											0	1	1	1	1	1	1	1	1	1	4M	2M
3	Page Address	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	4G	1G
6	7FFF_FFFFh	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	Entire cache	NA
6	FFFF_FFFFh	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	Undef	Undef
* Address bits 63:32 can be used to encode page sizes greater that 4G.																							

Software Note: The page tables are required to have one PTE for each default page size (see Table 8). not equal When the Next Level bits are equal to 7h, some of the least significant bits of the virtual address indexing the PTE are used for indexing the enlarged physical page, therefore those bits are not unique for indexing the PTE and the PTE must be repeated accordingly. For example, if the physical page is 32K bytes, the 3 least significant bits of the Page Table Level 1 virtual address cannot be used only for indexing within the page table and therefore the PTE will need to be repeated 8 times for each of the 64 unique PTEs given 4K byte page tables. Another example, for 4M byte pages, is illustrated in Figure 10. The PTE in the Level-2 page table is replicated twice and bit 21 of the virtual address is used twice for indexing, first to index the Level-2 table of PTEs and again to index into the 4M byte page for the data. The replicated Level-2 PTEs have identical contents and follow the example in Table 7 for a page size of 4M bytes. For larger page sizes, the PTEs must be replicated an appropriate number of times so that more bits of the virtual address can be used for indexing.

Implementation Note: While IOMMU implementations are not strictly required to include translation caches, it is strongly recommended that they include at least a cache for translations of 4K page table entries. IOMMU implementations are free to cache translations of larger pages by splitting them into multiple 4K cache entries.

The page table pointer for each domain specifies the system physical address and level of the root page table for that domain. Translation of a device virtual address begins by comparing it to the root page table's level. If the address contains any nonzero bits in bit positions higher than the range selected by the root page table's level, translation terminates with an IO_PAGE_FAULT. Otherwise, the appropriate group of virtual address bits is used to fetch a page table entry from the root page table. If this entry is marked not present, translation terminates with an IO_PAGE_FAULT. Otherwise the entry may be a page directory entry pointing to a lower-level page table (in which case the translation process repeats starting at the new page table using the

remaining virtual address bits), or it may be a page translation entry containing the final system physical address (in which case the translation process terminates and the remaining device virtual address bits are concatenated with the translation entry's physical address to obtain a translated address). If a translation skips levels and any of the skipped virtual address bits are non-zero, translation terminates with an IO_PAGE_FAULT.

At each level of the translation process, I/O write permission (IW) and read permission (IR) bits from fetched page table entries are logically ANDed into cumulative I/O write and read permissions for the translation including the IR and IW bits in the device tables. Device accesses to translated addresses are first checked against these cumulative permissions before being allowed to proceed. IW and IR bits from skipped levels are treated as if they were 1s.

Table 8 specifies the virtual address bit groups used for indexing at each level of the page tables, as well as the default page sizes associated with page translation entries fetched from page tables at each level. Figure 6 and Figure 7 illustrate the formats of page table entries. If a page table entry contains nonzero bits in any of the fields marked reserved, if the Next Level field is greater than or equal to the current page table entry table's level, or if a page translation entry's physical address is not aligned to a multiple of the appropriate page size for the current page table entry page table's level, translation terminates with an IO_PAGE_FAULT.

The layout of IOMMU page table entries has been chosen so that the IOMMU can use AMD64 long mode CPU page tables, provided the Next Level fields (which occupy bit positions ignored by AMD64 processors) are properly initialized according to their level within the CPU page tables. (AMD64 processors lack the IOMMU's level skipping facility.) All other page table entry fields used by the IOMMU are either ignored by AMD64 processors, or have the same meaning to both the processor and the IOMMU. For more details on sharing page tables see Section 3.2.4 [Sharing AMD64 CPU and IOMMU Page Tables].

The FC bit in the page translation entry is used to specify if DMA transactions that target the page must clear the PCI-defined No Snoop bit. The state of this bit is returned to a device with an IOTLB on an explicit translation request. If FC=1 for an untranslated access, the IOMMU sets the coherent bit in the upstream HyperTransport™ request packet. If FC=0 for an untranslated access, the IOMMU passes upstream the coherent attribute from the originating request.

The U bit in the page tables is an attribute bit passed to IOTLB devices in translation responses. If the U bit is set in a PTE, the IOTLB Cache hint bit should not be used by the IOMMU to determine if the translation should be cached for translation requests.

IOMMU implementations should zero-fill all high-order physical address.The IOMMU fields are architected to produce a physical address of up to 52 bits, thus physical address bits [63:53] are always zero.

Table 8: Page Table Level Parameters

Page Table Level	Virtual address bits indexing table	Default Page size (bytes) for translation entries
6	63:57	NA
5	56:48	2^{48}
4	47:39	2^{39}
3	38:30	2^{30}

Table 8: Page Table Level Parameters

Page Table Level	Virtual address bits indexing table	Default Page size (bytes) for translation entries
2	29:21	2^{21}
1	20:12	4096

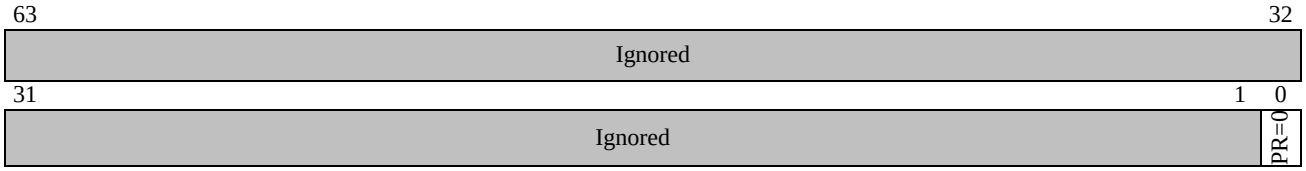


Figure 6: I/O Page Table Entry Not Present (any level)

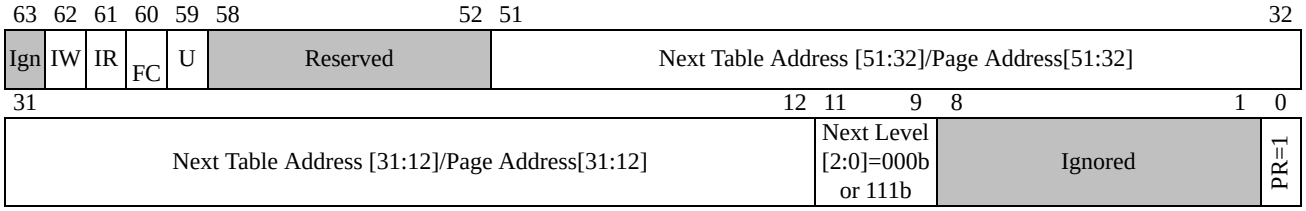


Figure 7: I/O Page Translation Entry (PTE)

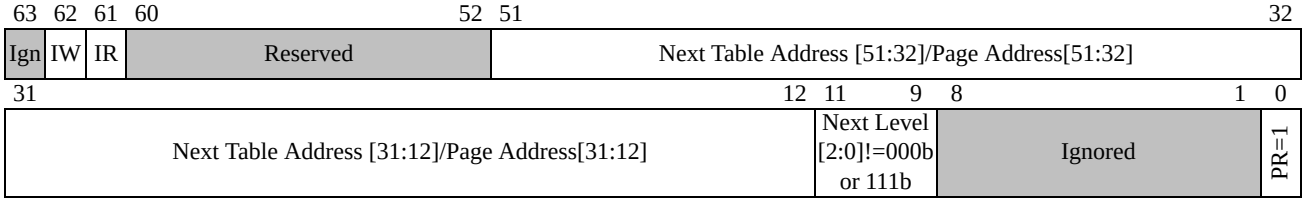
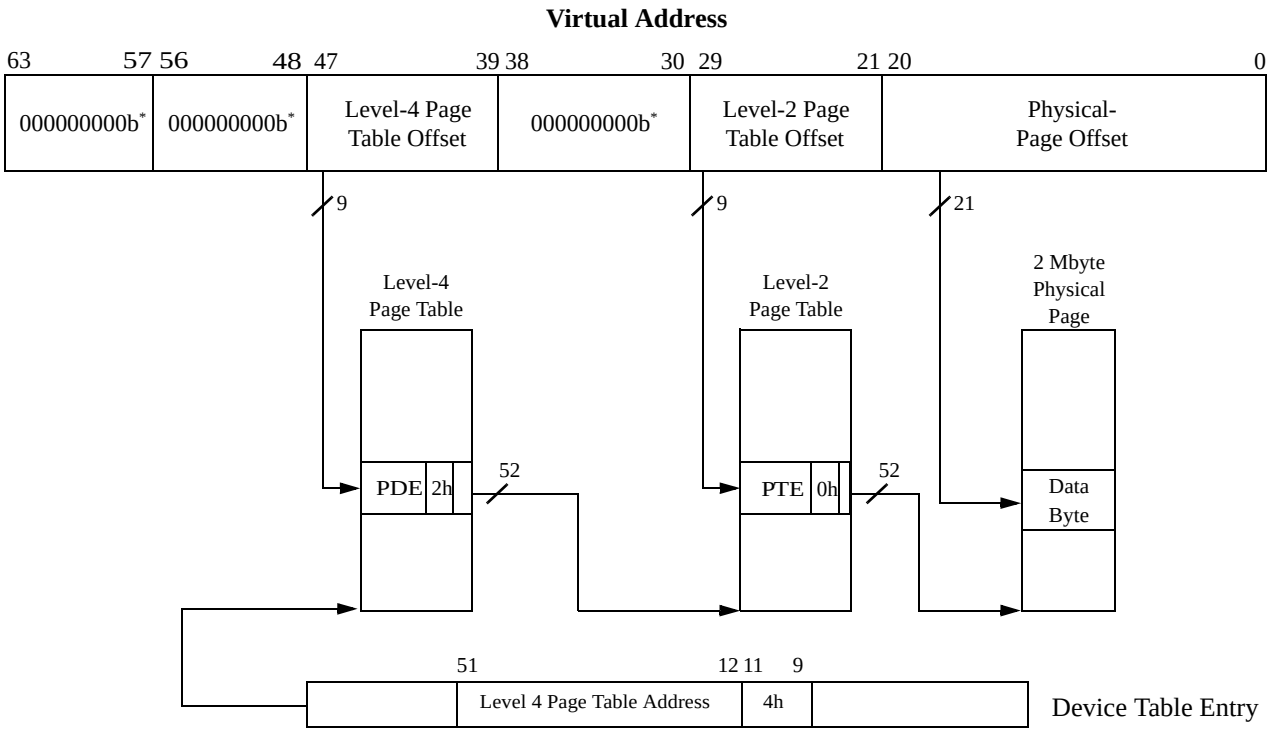
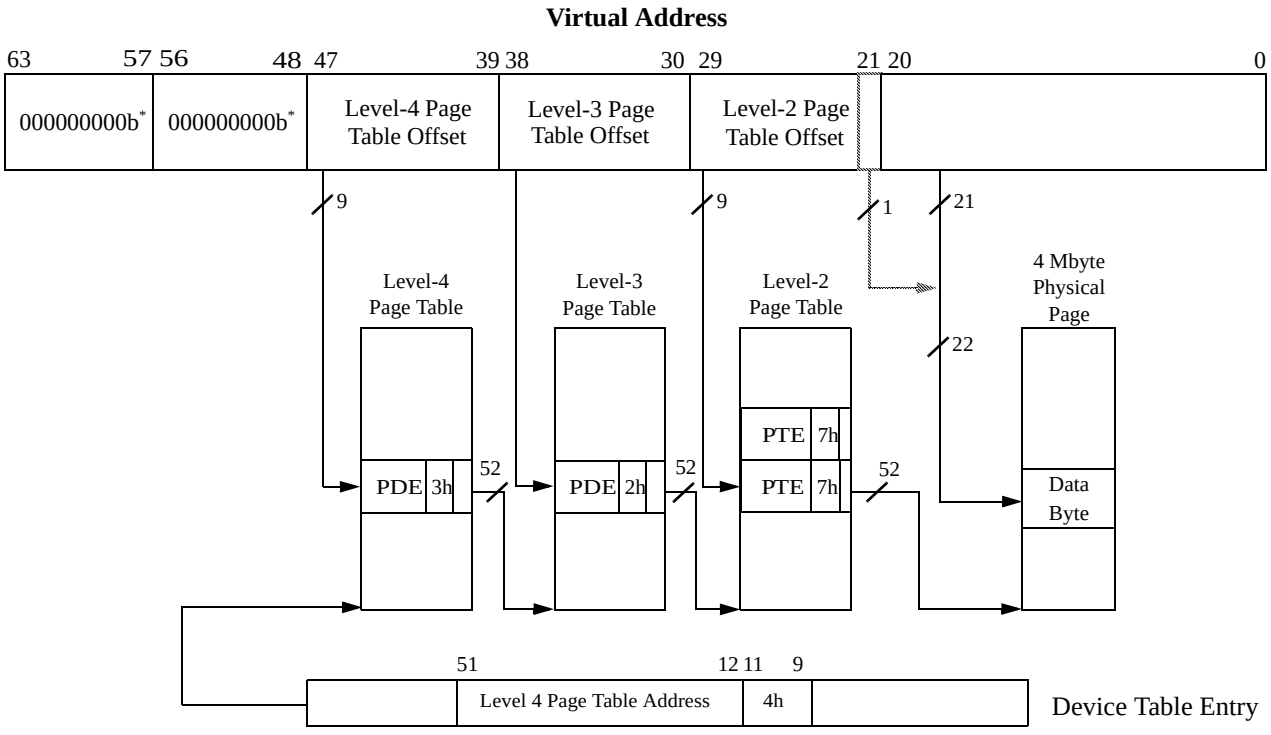


Figure 8: I/O Page Directory Entry (PDE)



* The Virtual Address bits associated with all skipped levels must be zero.

Figure 9: Address Translation Example with Skipped Level



* The Virtual Address bits associated with all skipped levels must be zero.

Figure 10: Address Translation Example with Page Size Larger than Default Size

3.2.4 Sharing AMD64 CPU and IOMMU Page Tables

This section outlines the topics to be considered so that the host or hypervisor page tables may be shared with an IOMMU. A more complete discussion depends on many implementation factors.

AMD64 CPUs and the IOMMU treat upper virtual address bits [63:48] differently. The CPU requires canonical addresses (addresses in which address bits [63:48] are equal to bit 47). By contrast, the IOMMU is designed to support the full PCI 64-bit address space. If 6-level page tables are used, the IOMMU can map any 64-bit address. If fewer than 6 levels are used, the IOMMU requires upper virtual address bits (beyond the range mapped by the page tables) to be 0. This ensures that software can always add levels to page tables without changing the address space as seen by devices.

In AMD64 long mode level 4 page tables, the bottom 256 entries of the root page table correspond to positive virtual addresses with bits [63:47] all 0s, and the top 256 entries correspond to negative virtual addresses with bits [63:47] all 1s.

For the IOMMU to directly share CPU page tables, at a minimum the Next Level fields in all page table entries must be initialized with correct values for the IOMMU.

Once the Next Level fields are initialized, the IOMMU may directly share exactly the same page tables. In 3-level 32-bit PAE mode this is all that's needed. However, in 4-level long mode software should be aware that CPU virtual addresses in the range FFFF_8000_0000_0000h to FFFF_FFFF_FFFF_FFFFh will correspond to I/O virtual addresses in the range 0000_8000_0000_0000h to 0000_FFFF_FFFF_FFFFh.

If software requires 64-bit CPU virtual addresses to be identical to I/O virtual addresses, including negative addresses, software needs to configure the IOMMU with the 6-level paging structure illustrated in Figure 11, where 4 extra 4K byte page tables (shaded) at levels 6, 5, and 4 are used solely by the IOMMU, and sharing with CPU page tables occurs only at levels 3 and below.

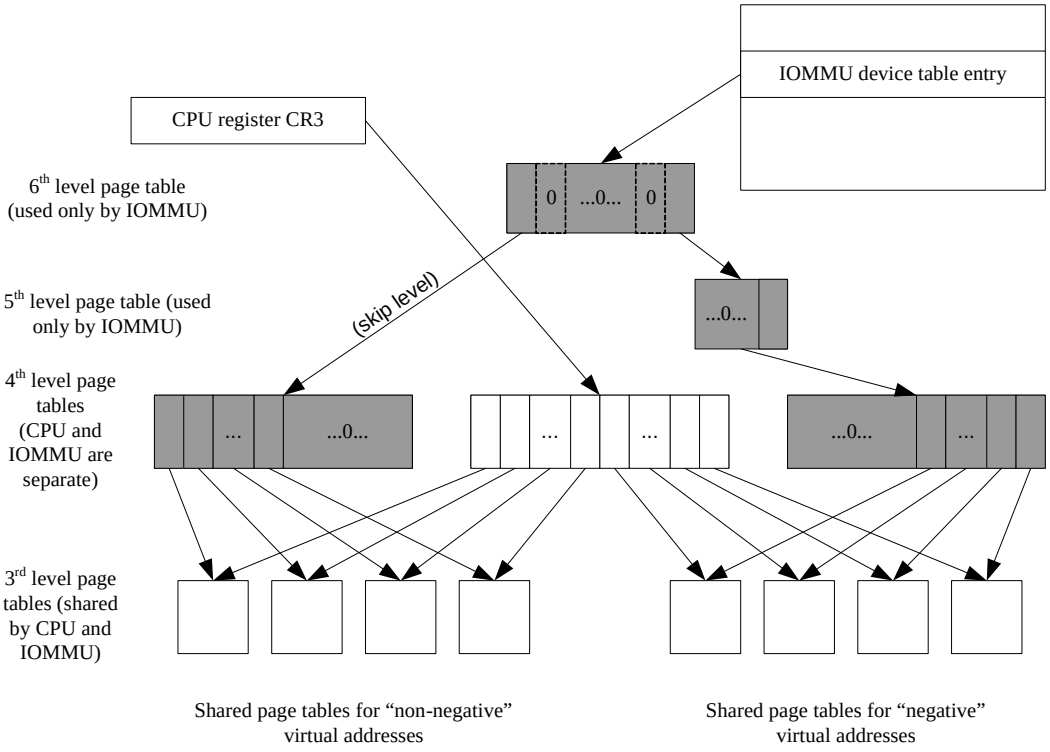


Figure 11: Sharing AMD64 and IOMMU Page Tables with Identical Addressing

3.2.5 Interrupt Remapping Tables

Interrupt messages use a HyperTransport™ interrupt special address range shown in Table 2. All fixed and arbitrated interrupt requests are mapped into the HyperTransport™ address space where they can be remapped by the IOMMU. Other interrupts are handled specially. Startup interrupts cannot originate from I/O devices thus the IOMMU cannot remap them. LINT0, LINT1, NMI, INIT, and External (ExtInt) interrupts are controlled individually using the device table entry control fields (see Table 6 and Table 9). The binary encodings listed in Table 9 are from the HyperTransport™ architecture specification for the MT field.

Table 9: IOMMU Interrupt Controls and Actions

Interrupt type (with MT encoding)		Destination Mode (DM)	Controlled by
0000b	Fixed	0b	Device table entry and interrupt remapping table entry
0001b	Arbitrated		
0010b	SMI		Forward unmapped
0011b	NMI		NMIPass
0100b	INIT		InitPass
0110b	ExtInt		EIntPass
1011b	Lint1		Lint1Pass
1110b	Lint0		Lint0Pass
0101b, 0111b, 1000b, 1001b, 1010b, 1100b, 1101b, 1111b	Startup, EOI, EOI		Target abort
0000b-1111b		1b	Device table entry and interrupt remapping table entry

The IOMMU remaps HyperTransport™ addresses for fixed and arbitrated interrupts as shown in the concatenation in Figure 12. The offset created by this concatenation corresponds directly to data bits 10:0 in the originating MSI interrupt message. After reading the interrupt remapping table entry, the IOMMU creates a new interrupt message address by OR'ing IRTE[23:2] with bits [23:2] of HyperTransport™ interrupt address range base (FD_F800_0000h).

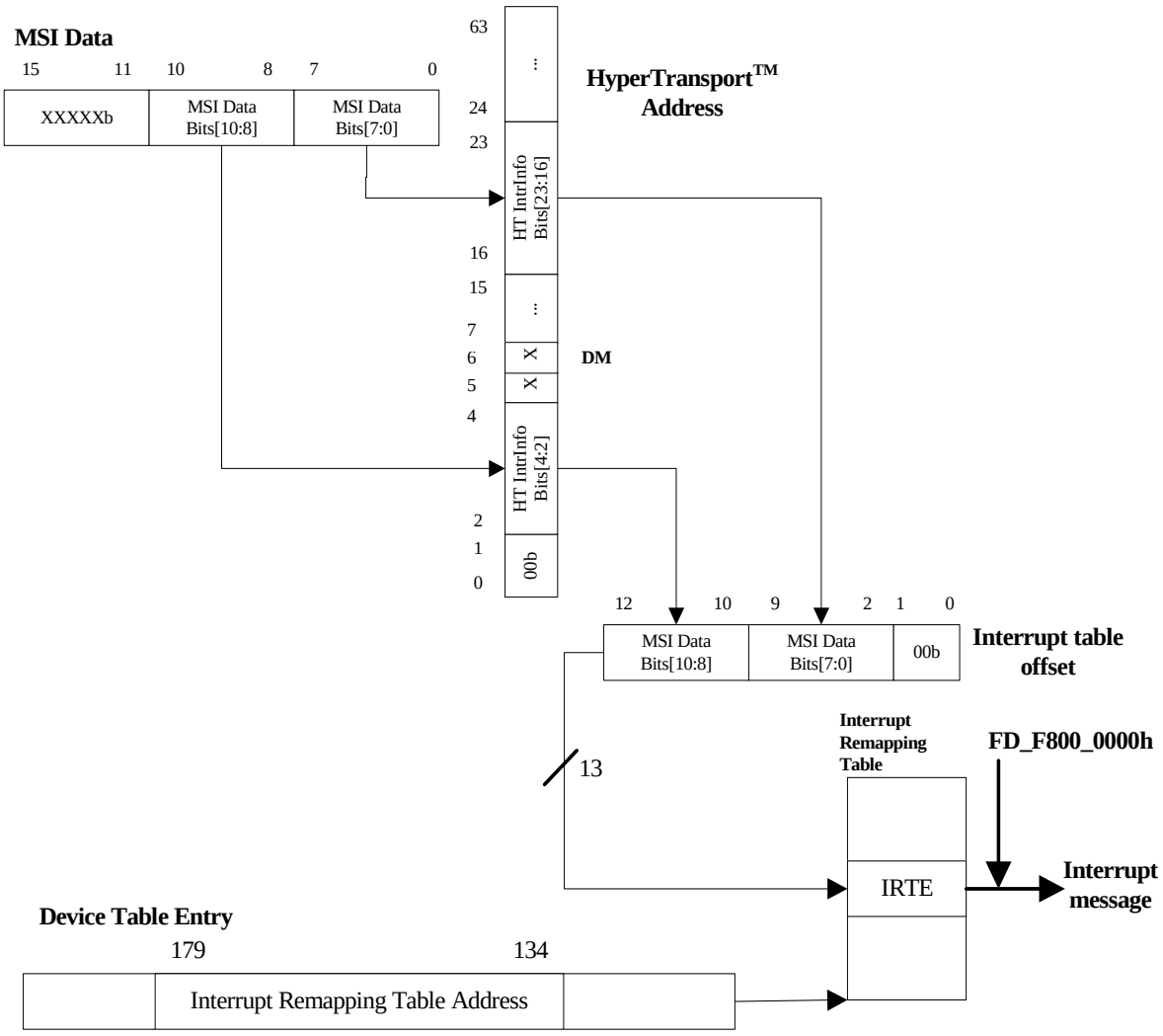


Figure 12: Interrupt Remapping Table Lookup for Fixed and Arbitrated Interrupts

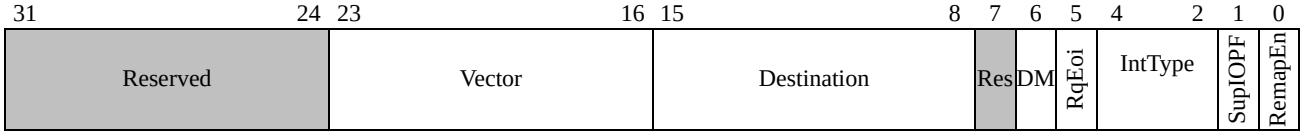


Figure 13: Interrupt Remapping Table Entry

Table 10: Interrupt Remapping Table Fields

Bits	Description
31:24	Reserved.
23:16	**Vector**. Specifies the interrupt vector for the interrupt.
15:8	**Destination**. Specifies the APIC logical or physical address to send the interrupt to.
7	Reserved.
6	**DM: destination mode**. 1=Logical destination mode. 0=Physical destination mode.

Table 10: Interrupt Remapping Table Fields

5	**RqEoi: request EOI**. 1=EOI cycle required. **Software note:** If RqEoi=1, software is responsible for performing the reverse mapping of the vector number.
4:2	**IntType: interrupt type**. This field specifies the type of interrupt message to deliver to the Local APIC. 000b Fixed 001b Arbitrated 010b Reserved 011b Reserved 100b Reserved 101b Reserved 110b Reserved 111b Reserved
1	**SupIOPF:** suppress IO_PAGE_FAULT events. 1=Supress logging when use of this remapping entry causes an IO_PAGE_FAULT. 0=Log event when this entry causes an IO_PAGE_FAULT. See also the IG control bit in the device table entry (Section 3.2.2.1 [Device Table Entry Format]).
0	**RemapEn** . 1=Interrupt is remapped. 0=Interrupt is target aborted. **Note:** SupIOPF is meaningful independent of the value of RemapEn.

3.3 Commands

The host software controls the IOMMU through a shared circular buffer in system memory. The host software writes commands into the buffer and then notifies the IOMMU of their presence by writing a new value to the tail pointer. The IOMMU then reads the commands and executes them at its own pace. The shared command buffer organization was chosen to allow the host software to send commands in batches to the IOMMU, while allowing the IOMMU to set the pace at which commands are actually executed.

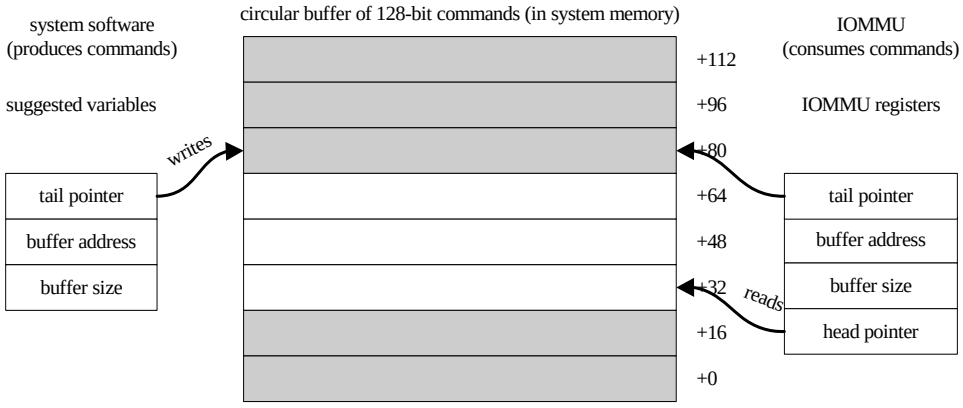


Figure 14: Circular Command Buffer in System Memory

The Command Buffer Base Address Register [MMIO Offset 0008h] is used to program the system physical base address and size of the command buffer. The command buffer occupies contiguous physical memory starting at the programmed base address, up to the programmed size. The size of the command buffer must be a multiple of 4K bytes (to facilitate "mod N" indexing for circularity), and can be as large as 32768 entries (corresponding to a 512 kilobyte buffer). The address of the command buffer must be aligned to a multiple of 4K bytes.

In addition to the Command Buffer Base Address Register [MMIO Offset 0008h], the IOMMU maintains two other registers associated with the command buffer: the Command Buffer Head Pointer Register [MMIO Offset 2000h], an offset from the base address, which points to the next command that the IOMMU will fetch, and the Command Buffer Tail Pointer Register [MMIO Offset 2008h], an offset from the base address, which

points to the next command to be written by software. These registers are located in IOMMU MMIO space. When the Command Buffer Base Address Register [MMIO Offset 0008h] register is written, the Command Buffer Head Pointer Register [MMIO Offset 2000h] and the Command Buffer Tail Pointer Register [MMIO Offset 2008h] are reset to the 0. When the Command Buffer Head Pointer Register [MMIO Offset 2000h] and the Command Buffer Tail Pointer Register [MMIO Offset 2008h] are equal the command buffer is empty. The Command Buffer Head Pointer Register [MMIO Offset 2000h] is incremented by the IOMMU after reading a command from the command buffer.

The IOMMU fetches commands in FIFO order from the command buffer. The IOMMU must never refetch a command. The IOMMU must set the Coherent bit in the HyperTransport™ packet when issuing command buffer read requests. Although the IOMMU fetches commands in order, it may execute them concurrently. Software may use the COMPLETION_WAIT command when synchronization is required.

All commands read by the IOMMU take the form of a 4-bit opcode together with two operands, which may be respectively 60 and 64 bits long, for a total of 128 bits (16 bytes) per command:

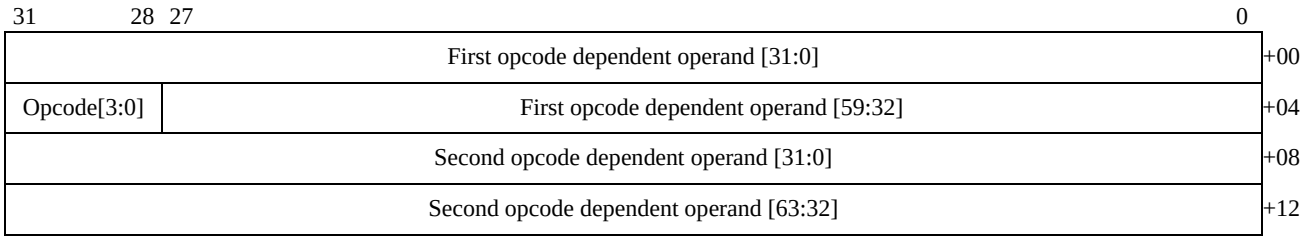


Figure 15: Generic Command Buffer Entry

The COMMAND_HARDWARE_ERROR (Section 3.4.6 [COMMAND_HARDWARE_ERROR]) and ILLEGAL_COMMAND_ERROR (Section 3.4.5 [ILLEGAL_COMMAND_ERROR]) events cause the IOMMU to halt command processing. If a command buffer entry causes one of these errors, the command head pointer freezes and does not advance. Note that the head pointer may have advanced past the command in error. Other activities of the IOMMU, including translations, error logging, and table walks, continue to be processed. Software is required to examine the IOMMU status and event log information to resolve the problem. Command processing is restarted by using the CmdBufEn control bit in the IOMMU Control Register [MMIO Offset 0018h] and status may be determined from CmdBufRun in IOMMU Status Register [MMIO Offset 2020h].

To restart the IOMMU command processing after the IOMMU has halted it, use the following procedure.

- Wait until CmdBufRun=0b in the IOMMU Status Register [MMIO Offset 2020h] so that all commands complete processing as the circumstances allow. CmdBufRun must be 0b to modify the command buffer registers safely.
- Set CmdBufEn=0b in the IOMMU Control Register [MMIO Offset 0018h].
- As necessary, change the following registers (e.g., to relocate the command buffer):
  - the Command Buffer Base Address Register [MMIO Offset 0008h],
  - the Command Buffer Head Pointer Register [MMIO Offset 2000h], and
  - the Command Buffer Tail Pointer Register [MMIO Offset 2008h].
- Any or all command buffer entries may be copied from the old command buffer to the new and software must set the head and tail pointers appropriately.
- Write the IOMMU Control Register [MMIO Offset 0018h] with CmdBufEn=1b and ComWaitIntEn as desired.

The IOMMU will now process command buffer entries.

3.3.1 COMPLETION_WAIT

The COMPLETION_WAIT command allows software to serialize itself with IOMMU command processing. The COMPLETION_WAIT command does not finish until all older commands issued since a prior COMPLETION_WAIT have completely executed. In addition, if f=1, the IOMMU will not begin execution of any younger commands until COMPLETION_WAIT has finished.

Implementation note: The COMPLETION_WAIT command may wait to finish after all older commands complete, including prior COMPLETION_WAIT commands. If there are no prior COMPLETION_WAIT commands in the command buffer, the COMPLETION_WAIT command finishes after all older commands.

For example, system software that wishes to reclaim pages formerly made available to devices should use the following procedure:

- Mark the page table entry (or entries) not present in the IOMMU's tables.
- Issue appropriate page invalidate commands to the IOMMU.
- Issue a COMPLETION_WAIT command to the IOMMU. When the COMPLETION_WAIT has finished, the IOMMU is designed to ensure that there are no transactions in flight anywhere in the system fabric that will read or write the invalidated pages.

The IOMMU may optionally signal that COMPLETION_WAIT has finished by one of two different mechanisms:

- If s=1 the IOMMU will store the specified 64-bit data value to the specified system physical address. Software can use this write to update a semaphore indicating to the waiting process that it can continue execution. The address written by the COMPLETION_WAIT must be located in system memory. The PassPw bit must not be set when performing this write.
- If i=1, the IOMMU sets the ComWaitInt bit in the IOMMU Status Register [MMIO Offset 2020h].

Both s=1 and i=1 may be specified in the same COMPLETION_WAIT command.

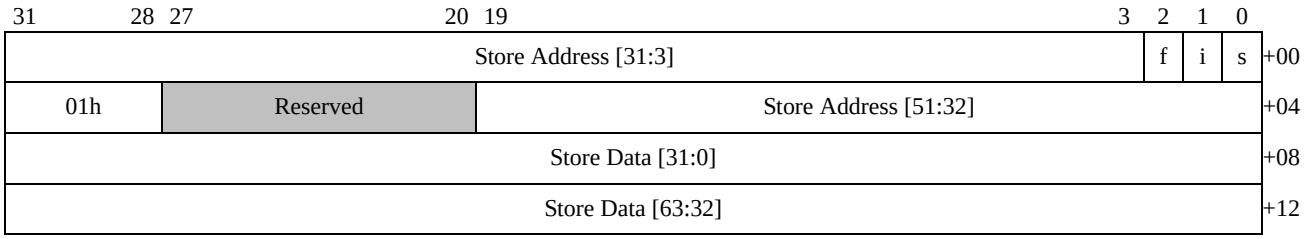


Figure 16: COMPLETION_WAIT command format

3.3.2 INVALIDATE_DEVTAB_ENTRY

When system software changes a device table entry, it must instruct the IOMMU to invalidate that DeviceID from its internal caches. The IOMMU is then forced to reload the device table entry before DMA from the device is allowed. The IOMMU may reload the device table entry any time after the invalidation has completed.

When software invalidates a DeviceID corresponding to an IOMMU-aware device with its own IOTLB, it should immediately follow INVALIDATE_DEVTAB_ENTRY with an INVALIDATE_IOTLB_PAGES targeted at the same DeviceID and sized to invalidate the full 64-bit address space for the given DeviceID. (Note that on a multi-function device this need only invalidate IOTLB entries for the specified function.)

Note that this command does not invalidate translation cache entries, since they may be in use by other devices sharing the same DomainID. If the DomainID is not shared, software should issue INVALIDATE_IOMMU_PAGES for the DomainID.

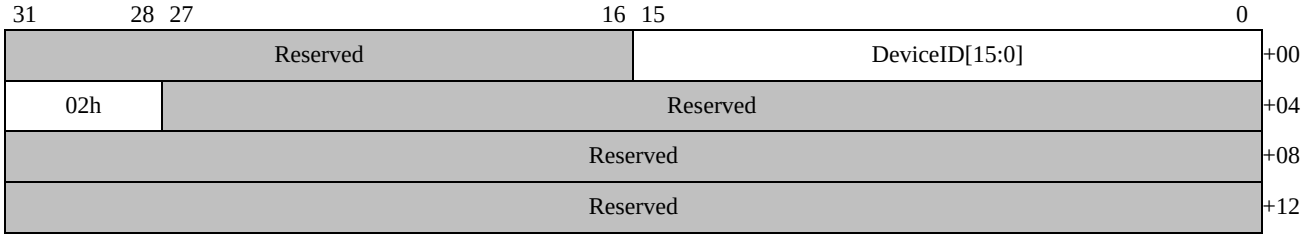


Figure 17: INVALIDATE_DEVTAB_ENTRY Command Format

3.3.3 INVALIDATE_IOMMU_PAGES

The INVALIDATE_IOMMU_PAGES command instructs the IOMMU to invalidate a range of entries in its translation cache for the specified DomainID. The size of the invalidate command is determined by the S bit, and the address. If S=0, size of the invalidate is 4K bytes. If S=1, the size of the invalidate is determined by the first zero bit in the address starting from Address[12]. If S=1, Address[63:12]=7_FFFF_FFFF_FFFFh and PDE=1, all pages associated with the DomainID are invalidated (see also Table 7). If the range of the INVALIDATE_IOMMU_PAGES command covers all of the pages in a page directory entry and PDE=1, the IOMMU must invalidate the page directory entry in the page directory cache. The INVALIDATE_IOMMU_PAGES command must appear as a single atomic operation to the translation engine.

When PDE=0, only the cached page translation entries are flushed.

Software Note: When issuing INVALIDATE_IOMMU_PAGES commands, the size of each invalidate must be greater than or equal to the size of the largest page being invalidated.

Implementation Note: IOMMU implementations are not required to provide optimal support for all of the possible invalidation request sizes. The IOMMU is free to invalidate more than just exactly the requested range of addresses, up to and including its entire translation cache if necessary.

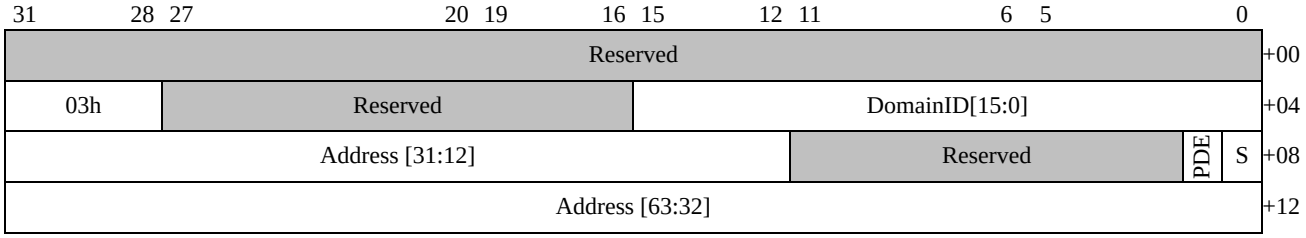


Figure 18: INVALIDATE_IOMMU_PAGES Command Encoding

3.3.4 INVALIDATE_IOTLB_PAGES

The INVALIDATE_IOTLB_PAGES command is only present in IOMMU implementations that support remote IOTLB caching of translations. This command instructs the specified device to invalidate the given range of addresses in its IOTLB. The size of the invalidate command is determined by the S bit, the level bits and the address. If S=0, the size of the invalidate is 4K bytes. If S=1, the size of the invalidate is determined by the first zero bit in the address starting from Address[12]. To invalidate the entire contents of an IOTLB, set S=1 and Address[63:32]=7FFF_FFFFh and Address[31:12]=F_FFFFh in the INVALIDATE_IOTLB_PAGES command. When Address[63:32]=FFFF_FFFFh, the IOMMU behavior is undefined.

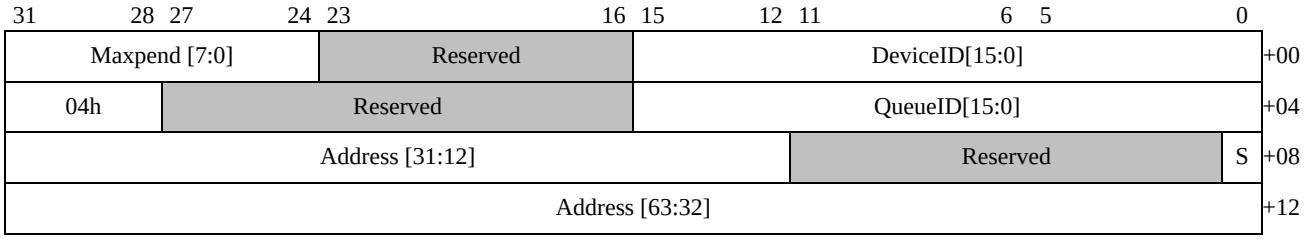


Figure 19: INVALIDATE_IOTLB_PAGES

Since both the IOMMU and the remote IOTLB(s) may contain cached translations for a domain, software must take care to perform invalidations in an order that ensures that no stale translations persist anywhere in the system. After updating a domain's page tables, software should first issue an INVALIDATE_IOMMU_PAGES command for the domain; then, if the domain contains any devices with their own IOTLBs, software should follow with INVALIDATE_IOTLB_PAGES commands for each such device.

The Maxpend field allows software to control the maximum number of simultaneously in-flight INVALIDATE_IOTLB_PAGES transactions that the IOMMU attempts to initiate with any one particular QueueID. The appropriate value for Maxpend is device-dependent, and can be obtained from the device's IOTLB capability. For devices that implement multiple virtual functions sharing a single invalidation queue, the QueueID is used to limit the outstanding invalidations for all virtual devices sharing the queue. Some devices will implement a physical function and multiple virtual functions. Each physical function and virtual function have a unique DeviceID. When the IOMMU receives an invalidate command, the command targets the DeviceID. An implementation may have a single queue (likely associated with the physical function) to receive invalidates for the physical function or any of its virtual functions. To manage the flow control of the unified device invalidate-queue, it is not sufficient to track the outstanding entries based on DeviceID. The QueueID is an abstract number representing the shared queue.

Software Note: In order to flow-control invalidations to functions that share a common invalidation queue, software must set the QueueID to a unique identifier that represents the shared queue. The DeviceID of the physical function associated with the virtual functions may be used as the QueueID to insure the IOMMU will issue a limited number of outstanding invalidates to the given queue.

Software Note: To completely tear down a domain, software should:
- update the IOMMU's in-memory data structures, INVALIDATE_DEVTAB_ENTRY for all devices in the domain, INVALIDATE_IOMMU_PAGES for the domain, and
- INVALIDATE_IOTLB_PAGES for any IOTLB-capable devices that had been assigned to the domain.

3.3.5 INVALIDATE_INTERRUPT_TABLE

The INVALIDATE_INTERRUPT_TABLE command instructs the IOMMU to invalidate all cached interrupt remapping table entries for the device.

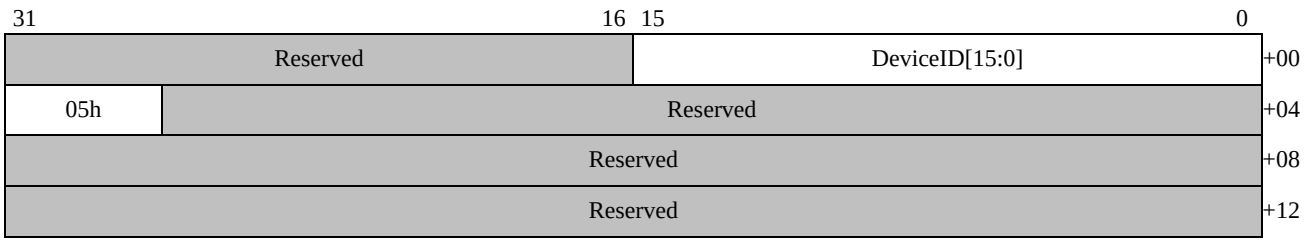


Figure 20: INVALIDATE_INTERRUPT_TABLE

3.3.6 IOMMU Ordering Rules

The IOMMU must ensure that proper ordering is maintained between invalidation command types and between invalidation commands and the translation process.

3.3.6.1 Invalidation Command Ordering Requirements

The IOMMU must ensure that the following command ordering rules are followed for invalidation commands:

- When an INVALIDATE_IOMMU_PAGES or INVALIDATE_INTERRUPT_TABLE command is received, the IOMMU must ensure that all cache entries associated with any prior INVALIDATE_DEVTAB_ENTRY commands are invalidated from the cache before executing the command.
- When an INVALIDATE_IOTLB_PAGES command is received, the IOMMU must ensure that all cache entries associated with any prior INVALIDATE_DEVTAB_ENTRY or INVALIDATE_IOMMU_PAGES commands are invalidated from the cache before executing the command.

3.3.6.2 Invalidation Commands Interaction Requirements

Invalidation commands are considered completed only when the IOMMU can guarantee that there are no DMA transactions in flight anywhere in the system fabric that relied on translation cache contents prior to the invalidation. To ensure that this property is achieved, the IOMMU must follow the following rules:

- The IOMMU must ensure that read responses for all DMA outstanding read transactions that match the invalidation command have been received by the IOMMU.
  - HyperTransport™ tunnels that support address translation can achieve this property by maintaining a counter that is incremented when a non-posted transaction is forwarded to the processor through the tunnel and is decremented when a response is forwarded from the processor through the tunnel. The invalidation command can be considered complete when the counter reaches zero.
    - The tunnel may temporarily block upstream traffic to cause the counter to resolve to zero in a timely manner, ensuring that forward progress of the invalidation command is made.
- The IOMMU must ensure that all DMA write transactions that have already been translated have been pushed to the host bridge by:
  - Prior to sending the invalidation completion indication (interrupt or status write) the IOMMU must:
    - Send an upstream Fence command in the base channel if the IOMMU supports translating requests for more than one upstream stream (more than one unitID is in use).
    - Send an upstream Fence command followed by a Flush command in the isochronous channel if the IOMMU supports translating requests in both the isochronous and the base channels. The invalidation completion must wait for the Flush response to be received.

The IOMMU must ensure that both of these requirements are met prior to executing a subsequent COMPLETION_WAIT command.

An invalidation command matches an outstanding translation if the command:
- Invalidates the device table entry for the I/O device that caused a translation to be initiated, or
- Invalidates the virtual address range being translated for a device.

3.4 Event Logging

The IOMMU reports events to the host software by means of another shared circular buffer in system memory. The IOMMU writes event records into the buffer. If the IOMMU needs to report an error but finds that the event log is already full, it sets MMIO Offset 2020h[EventOverflow]. The IOMMU can be configured to signal an interrupt whenever the event log is written. The host software increments the IOMMU's head pointer to

indicate to the IOMMU that it has consumed event log entries.

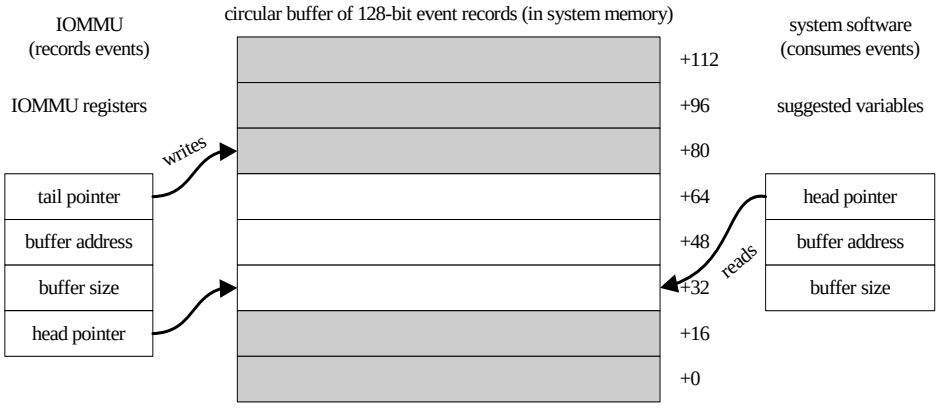


Figure 21: Circular Event Log in System Memory

The Event Log Base Address Register [MMIO Offset 0010h] is used to program the system physical address and size of the event log. The event log occupies contiguous physical memory starting at the programmed base address, up to the programmed size. The size of the event log must be a multiple of 4K bytes (to facilitate "mod N" indexing for circularity), and can be as large as 32768 entries (corresponding to a 512 kilobyte buffer). The address of the event log must be aligned to a multiple of 4K bytes.

In addition to the Event Log Base Address Register [MMIO Offset 0010h], the IOMMU maintains two other registers associated with the event log: the Event Log Head Pointer Register [MMIO Offset 2010h] which points to the next event that software will read, and the Event Log Tail Pointer Register [MMIO Offset 2018h] which points to the next event to be written by the IOMMU. These registers are located in MMIO space. When the Event Log Base Address Register [MMIO Offset 0010h] register is written, the Event Log Head Pointer Register [MMIO Offset 2010h] and the Event Log Tail Pointer Register [MMIO Offset 2018h] are cleared to 0. When the Event Log Head Pointer Register [MMIO Offset 2010h] and the Event Log Tail Pointer Register [MMIO Offset 2018h] are equal, the event log is empty. The Event Log Tail Pointer Register [MMIO Offset 2018h] is incremented by the IOMMU after writing an event to the event log. The event log is full when all slots but one are used. The event log has overflowed when an event occurs that is to be logged and would otherwise consume the last unused slot. When the event log has overflowed, the EventOverflow bit is set in the IOMMU Status Register [MMIO Offset 2020h] and any data for new events is discarded. The host software must make space in the event log by reading entries (by adjusting the head pointer) or resizing the log. Event logging may then be restarted.

The IOMMU event logging is disabled after system reset and when the event log overflows. The IOMMU discards event reports until event logging is enabled, setting the EventOverflow bit in the IOMMU Status Register [MMIO Offset 2020h] to indicate the loss of event information. To restart the IOMMU event logging after the event log overflows, use the following procedure.

- Wait until EventLogRun=0b in the IOMMU Status Register [MMIO Offset 2020h] so that all log entries are completed as circumstances allow. EventLogRun must be 0b to modify the event log registers safely.
- Write EventLogEn=0b in the IOMMU Control Register [MMIO Offset 0018h].
- As necessary, change the following registers (e.g., to relocate or resize the event log).
  - the Event Log Base Address Register [MMIO Offset 0010h],
  - the Event Log Head Pointer Register [MMIO Offset 2010h], and
  - the Event Log Tail Pointer Register [MMIO Offset 2018h].
- Write the IOMMU Status Register [MMIO Offset 2020h] with EventOverflow=0b to clear the bit.
- Write the IOMMU Control Register [MMIO Offset 0018h] with EventLogEn=1b and EventIntEn as desired.

The IOMMU will now create event log entries for new events.

All events recorded by the IOMMU consist of a 4-bit EventCode together with two operands, which may be respectively 60 and 64 bits long, for a total of 128 bits (16 bytes) per record. Events that are logged because of errors that occur while performing device table or page table walks always record the DeviceID and address from the transaction that was being translated.

The IOMMU must set the Coherent bit in the HyperTransport™ packet when generating writes to the event log.

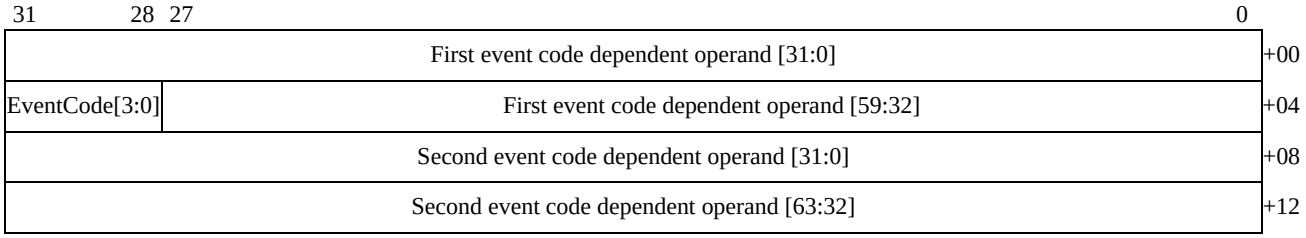


Figure 22: Generic Event Log Buffer Entry

In Figure 11, an error type that can be caused by the I/O device or system hardware are marked with a double dagger (‡) and error types caused only by host software are marked with a dagger (†).

Table 11: Event Summary

Event Type	Error Type	IOMMU Response
ILLEGAL_DEV_TABLE_ENTRY	Non-zero reserved bit in a device table entry. †	Target Abort transaction
	Reserved encoding in the IntTabLen field for a device table entry with IntCtl=10b. †	
	Reserved encoding in the IoCtl field. †	
	Reserved encoding in the IntCtl field. †	
1. An IO_PAGE_FAULT is not logged if this event occurs because of a translation request. 2. Translation requests forward the state of the IR and IW bits in the translation response with a normal completion.		

Table 11: Event Summary

Event Type	Error Type	IOMMU Response
IO_PAGE_FAULT	Reserved paging mode in device table entry. †	Target Abort transaction if the request is untranslated. Return response with data and with R and W bits set to 0 if the request is a translation request.
	Page size encoding in a PTE that is smaller than the default page size of the PTE. †	
	Page size encoding in a PTE that is larger than the default page size of the PTE. †	
	Illegal level encoding in a page table entry or non-zero bit higher than root page table's level. †	
	Non-zero reserved bit in a PTE. †	
	Valid bit not set in page table entry[1]. ‡	
	TV bit not set in device table entry for untranslated non-interrupt transaction. ‡	
	Device attempts a read transaction to a read protected page (IR=0)[2]. ‡	Target Abort transaction if the request is untranslated. Return response with data and with R and W bits from the page translation information if the request is a translation request.
	Device attempts a non-posted write transaction to a write protected page (IW=0)[2]. ‡	
	Device attempts a posted write transaction to a write protected page (IW=0). ‡	
	DeviceID not in the range specified by the device table size. ‡	Master abort if a translation request. Otherwise, target abort.
	Interrupt request that addresses an IRTE that is beyond the end of the table. ‡	Target Abort transaction.
	Non-zero reserved bit in an IRTE. †	
	Interrupt request that targets an IRTE with RemapEn=0. ‡	
	Interrupt request that targets an IRTE with reserved IntType. †	
	Interrupt request aborted by entry in Table 6 (Pass fields) or Table 9 (entries causing target abort). ‡	

1. An IO_PAGE_FAULT is not logged if this event occurs because of a translation request.
2. Translation requests forward the state of the IR and IW bits in the translation response with a normal completion.

Table 11: Event Summary

Event Type	Error Type	IOMMU Response
DEV_TAB_HARDWARE_ERROR	Master abort received on device table read. ‡	Target abort transaction.
	Target abort received on device table read. ‡	
	Poisoned data received on device table read. ‡	
PAGE_TAB_HARDWARE_ERROR	Master abort received on page table read ‡.	
	Target abort received on page table read. ‡	
	Poisoned data received on page table read. ‡	
COMMAND_HARDWARE_ERROR	Master abort received on command buffer read. ‡	Halt command processing.
	Target abort received on command buffer read. ‡	
	Poisoned data received on command buffer read. ‡	
ILLEGAL_COMMAND_ERROR	Non-zero reserved bit in a command buffer entry. †	
	Unsupported command code in a command buffer entry. †	
	IOMMU implementation receives INVALIDATE_IOTLB_PAGES and does not support IOTLB commands. †	
IOTLB_INV_TIMEOUT	Invalidation response not received from IOTLB device. ‡	Log the error if logging is enabled.

1. An IO_PAGE_FAULT is not logged if this event occurs because of a translation request.
2. Translation requests forward the state of the IR and IW bits in the translation response with a normal completion.

Table 11: Event Summary

Event Type	Error Type	IOMMU Response
INVALID_DEVICE_REQUEST (see also Table 20)	Read request or non-posted write in the interrupt address range. ‡	Target abort transaction.
	Pre-translated transaction received from an I/O device with I=0 or V=0. ‡	
	Port I/O Space request from an I/O device with IoCtl=00b. ‡	
	Write to the system management address space from an I/O device with SysMgt=00b, or with SysMgt=10b and the message is not an INTx message. ‡	
	Read request or non-posted write in the system management address range (if SysMgt = 10b or 0xb). ‡	
	Posted write to the Interrupt/EOI interrupt address range from an I/O device with IntCtl=00. ‡	
	Posted write to a reserved interrupt address range (see Table 2). ‡	
	Access to the system management address range when SysMgt=11b or to the port I/O space range when IoCtl=10b, while V=1 and TV=0. ‡	
	Translation request from an I/O device with I=0 or V=0. ‡	Master abort transaction.
	Translation request in the interrupt space, port I/O space (if IoCtl=0xb), or system management address range (if SysMgt = 0xb or 10b). Or translation request in the system management address range when SysMgt=11b or in the port I/O space range when IoCtl=10b, while V=1 and TV=0. ‡	Target abort transaction.

1. An IO_PAGE_FAULT is not logged if this event occurs because of a translation request.
2. Translation requests forward the state of the IR and IW bits in the translation response with a normal completion.

3.4.1 ILLEGAL_DEV_TABLE_ENTRY

When the IOMMU performs a lookup in the device table and encounters a device table entry that it does not support or that is formatted incorrectly, the IOMMU writes an ILLEGAL_DEV_TABLE_ENTRY event to the event log.

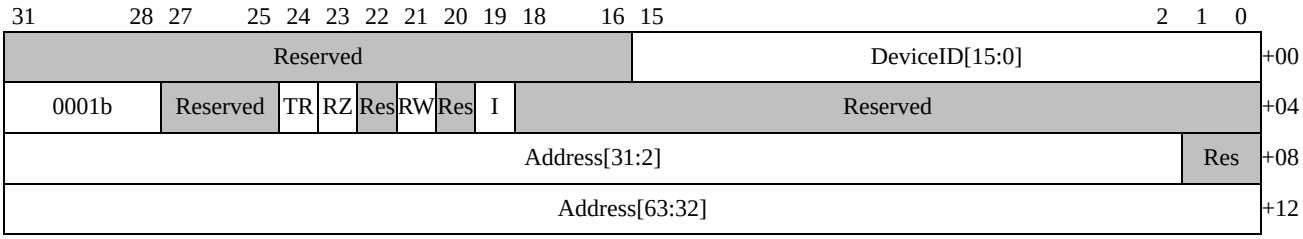


Figure 23: ILLEGAL_DEV_TABLE_ENTRY Event Log Buffer Entry

Table 12: ILLEGAL_DEV_TABLE_ENTRY Event Log Buffer Entry Fields

Bits	Description
31:16 +00	Reserved.
15:0 +00	**DeviceID**. Specifies the DeviceID that caused the device table lookup. The address of the malformed device table entry can be determined using the DeviceID field.
31:28 +04	**0001b**. Specifies an ILLEGAL_DEV_TABLE_ENTRY.
27:25 +04	Reserved.
24 +04	**TR: translation**. 1=transaction that caused the device table lookup was a translation request. 0=transaction that caused the device table lookup was a transaction request.
23 +04	**RZ: reserved bit not zero.** 1=page fault was caused by a non-zero reserved bit in the device entry. 0=illegal level encoding.
22 +04	Reserved.
21 +04	**RW: read-write.** 1=transaction that caused the device table lookup was a write. 0=transaction that caused the device table lookup was a read. RW is only meaningful when TR=0 and I=0.
20 +04	Reserved.
19 +04	**I: interrupt.** 1=transaction that caused the device table lookup was an interrupt request. 0=transaction that caused the device table lookup was a memory request.
18:0 +04	Reserved.
31:2 +08	**Address[31:2]**. The Address field contains the device virtual address that the device was attempting to access.
1:0 +08	Reserved.
31:0 +12	**Address[63:32]**. The Address field contains the device virtual address that the device was attempting to access.

3.4.2 IO_PAGE_FAULT

When the IOMMU performs a lookup in the page tables for a device and encounters an error condition in Table 11, the IOMMU writes the event log with an IO_PAGE_FAULT event as controlled by the SA, SE, IG, and SupIOPF bits (see Figure 5 and Table 3 or Figure 13 and Table 6).I/O page faults detected for translation requests return a translation-not-present response to the device and are not logged in the event log.

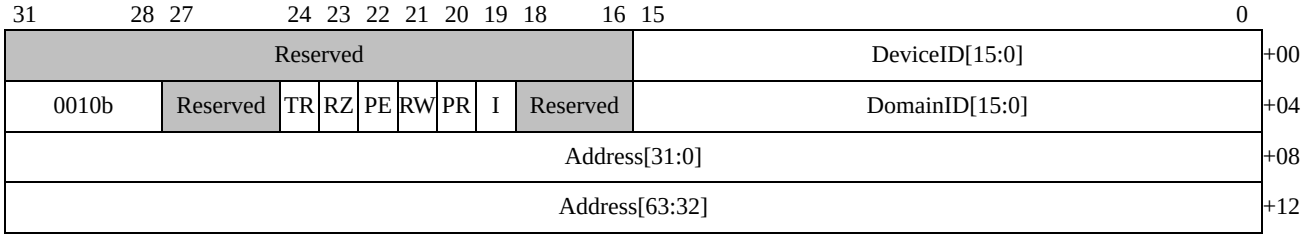


Figure 24: IO_PAGE_FAULT Event Log Buffer Entry

Table 13: IO_PAGE_FAULT Event Log Buffer Entry Fields

Bits	Description
31:16 +00	Reserved.
15:0 +00	**DeviceID**. Specifies the DeviceID that caused the device table lookup. The address of the device table entry can be determined using the DeviceID field.
31:28 +04	**0010b**. Specifies an IO_PAGE_FAULT entry.
27:24 +04	Reserved.
24 +04	**TR: translation**. 1=transaction that caused the device table lookup was a translation request. 0=transaction that caused the device table lookup was a transaction request.
23 +04	**RZ: reserved bit not zero.** 1=page fault was caused by a non-zero reserved bit in the entry. 0=illegal level encoding. RZ is only meaningful when PR=1.
22 +04	**PE: permission indicator.** 1=peripheral did not have the permissions required to perform the transaction. 0=peripheral had the necessary permissions. PE is only meaningful when PR=1.
21 +04	**RW: read-write.** 1=transaction was a write. 0=transaction was a read. RW is only meaningful when PR=1, TR=0, and I=0.
20 +04	**PR: present.** 1=transaction was to a page marked as present or interrupt marked as remapped (RemapEn=1). 0=transaction was to a page marked not present or interrupt marked as blocked (RemapEn=0).
19 +04	**I: interrupt.** 1=transaction was an interrupt request. 0=transaction was a memory request.
18:16 +04	Reserved.
16:0 +04	**DomainID.** The DomainID from the Device Table Entry.

Table 13: IO_PAGE_FAULT Event Log Buffer Entry Fields

31:0 +08	**Address[31:0]**. The Address field contains the device virtual address that the peripheral was attempting to access.
31:0 +12	**Address[63:32]**. The Address field contains the device virtual address that the peripheral was attempting to access.

3.4.3 DEV_TAB_HARDWARE_ERROR

If the IOMMU detects a hardware error (master abort, target abort, poisoned data, etc.) while accessing the device table, the IOMMU writes the event log with a DEV_TAB_HARDWARE_ERROR event.

In this case the Address field does *not* contain the device virtual address the device was attempting to access, but instead contains the system physical address of the failed device table access.

Table 14: Event Log Type Field Encodings

Type	Description
00b	Reserved
01b	Master Abort
10b	Target Abort
11b	Data Error

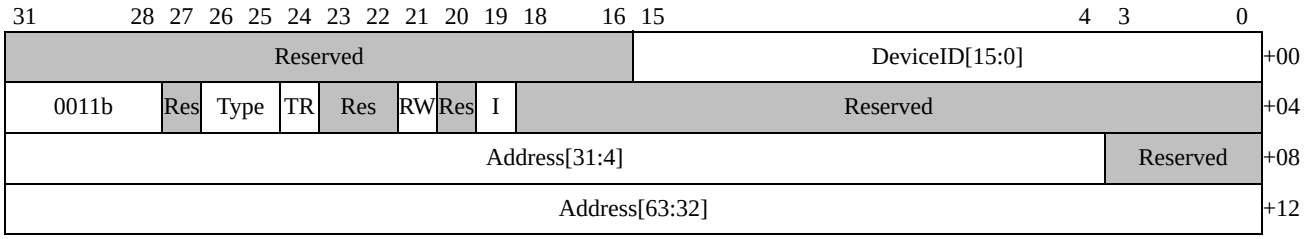


Figure 25: DEV_TAB_HARDWARE_ERROR Event Log Buffer Entry

Table 15: DEV_TAB_HARDWARE_ERROR Event Log Buffer Entry Fields

Bits	Description
31:16 +00	Reserved.
15:0 +00	**DeviceID**. Specifies the DeviceID that caused the device table lookup. The address of the device table entry can be determined using the DeviceID field.
31:28 +04	**0011b**. Specifies a DEV_TAB_HARDWARE_ERROR entry.
27 +04	Reserved.
26:25 +04	**Type.** The Type field indicates the type of hardware error that occurred as listed in Table 14.
24 +04	**TR: translation**. 1=transaction that caused the device table lookup was a translation request. 0=transaction that caused the device table lookup was a transaction request.

Table 15: DEV_TAB_HARDWARE_ERROR Event Log Buffer Entry Fields

23:22 +04	Reserved.
21 +04	**RW: read-write.** 1=transaction was a write. 0=transaction was a read. RW is only meaningful when TR=0 and I=0.
20 +04	Reserved.
19 +04	**I: interrupt.** 1=transaction was an interrupt request. 0=transaction was a memory request.
18:0 +04	Reserved.
31:4 +08	**Address[31:4]**. The system physical address of the failed device table access. In this case the Address field does *not* contain the device virtual address the device was attempting to access.
3:0 +08	Reserved.
31:0 +12	**Address[63:32]**. The system physical address of the failed device table access. In this case the Address field does *not* contain the device virtual address the device was attempting to access.

3.4.4 PAGE_TAB_HARDWARE_ERROR

If the IOMMU detects a hardware error (master abort, target abort, poisoned data, etc.) while accessing the I/O page tables, the IOMMU writes the event log with a PAGE_TAB_HARDWARE_ERROR event.

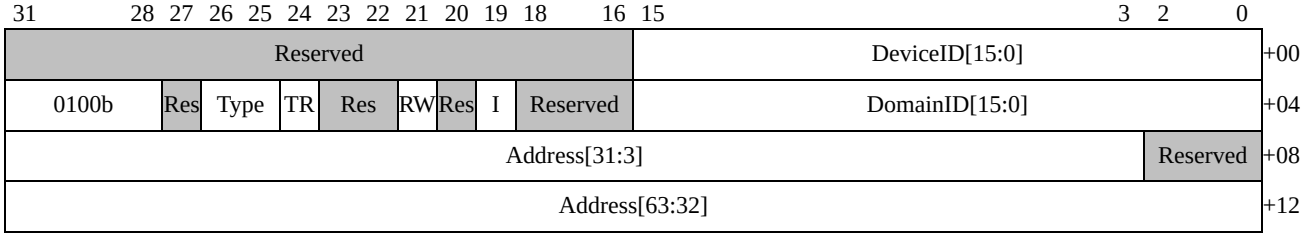


Figure 26: PAGE_TAB_HARDWARE_ERROR Event Log Buffer Entry

Table 16: PAGE_TAB_HARDWARE_ERROR Event Log Buffer Entry Fields

Bits	Description
31:16 +00	Reserved.
15:0 +00	**DeviceID**. Specifies the DeviceID that caused the page table lookup. The address of the device table entry can be determined using the DeviceID field.
31:28 +04	**0100b**. Specifies a PAGE_TAB_HARDWARE_ERROR entry.
27 +04	Reserved.
26:25 +04	**Type.** The Type field indicates the type of hardware error that occurred as listed in Table 14.

Table 16: PAGE_TAB_HARDWARE_ERROR Event Log Buffer Entry Fields

24 +04	**TR: translation**. 1=transaction that caused the page table lookup was a translation request. 0=transaction that caused the page table lookup was an untranslated request.
23:22 +04	Reserved.
21 +04	**RW: read-write.** 1=transaction was a write. 0=transaction was a read. RW is only meaningful when TR=0 and I=0.
20 +04	Reserved.
19 +04	**I: interrupt.** 1=transaction was an interrupt request. 0=transaction was a memory request.
18:16 +04	Reserved.
15:0 +04	**DomainID.** The DomainID of the peripheral that caused the page table lookup.
31:4 +08	**Address[31:4]**. The system physical address of the failed page table entry. In this case the Address field does *not* contain the device virtual address the device was attempting to access.
3:0 +08	Reserved.
31:0 +12	**Address[63:32]**. The system physical address of the failed page table access. In this case the Address field does *not* contain the device virtual address that the device attempted to access.

3.4.5 ILLEGAL_COMMAND_ERROR

If the IOMMU reads an illegal command (including an unsupported command code, or a command that incorrectly has reserved bits set), the IOMMU writes the event log with an ILLEGAL_COMMAND_ERROR event. The IOMMU must stop fetching new commands from the command buffer if a ILLEGAL_COMMAND_ERROR event is detected.

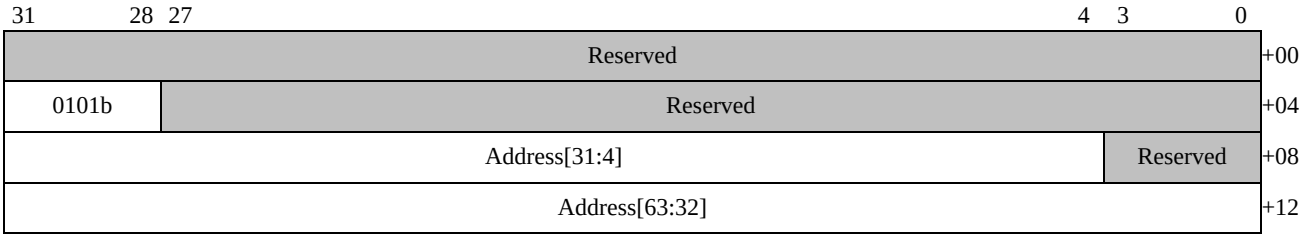


Figure 27: ILLEGAL_COMMAND_ERROR

Table 17: ILLEGAL_COMMAND_ERROR Event Log Buffer Entry Fields

Bits	Description
31:0 +00	Reserved.
31:28 +04	**0101b**. Specifies an ILLEGAL_COMMAND_ERROR entry.
27:0 +04	Reserved.

Table 17: ILLEGAL_COMMAND_ERROR Event Log Buffer Entry Fields

31:4 +08	**Address[31:4]**. The system physical address of the illegal command.
3:0 +08	Reserved.
31:0 +12	**Address[63:32]**. The system physical address of the illegal command.

3.4.6 COMMAND_HARDWARE_ERROR

If the IOMMU detects a hardware error (master abort, target abort, poisoned data, etc.) while accessing the command buffer, the IOMMU writes the event log with a COMMAND_HARDWARE_ERROR event. The IOMMU must stop fetching new commands from the command buffer if a COMMAND_HARDWARE_ERROR event is detected.

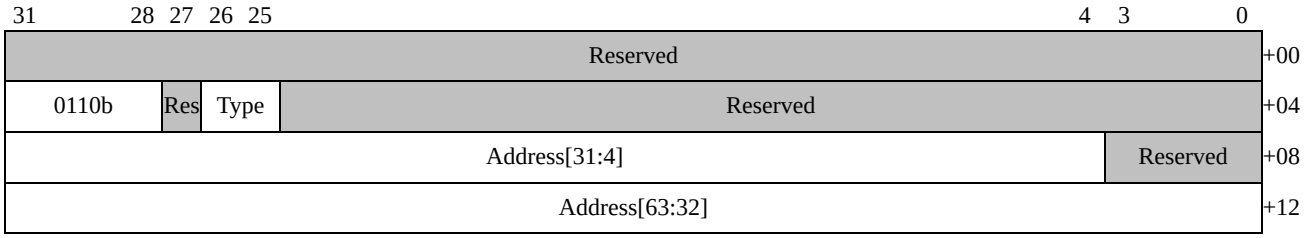


Figure 28: COMMAND_HARDWARE_ERROR Event Log Buffer Entry

Table 18: COMMAND_HARDWARE_ERROR Event Log Buffer Entry Fields

Bits	Description
31:0 +00	Reserved.
31:28 +04	**0110b**. Specifies a COMMAND_HARDWARE_ERROR entry.
27 +04	Reserved.
26:25 +04	**Type.** The Type field indicates the type of hardware error that occurred as listed in Table 14.
24:0 +04	Reserved.
31:4 +08	**Address[31:4]**. The system physical address that the IOMMU attempted to access.
3:0 +08	Reserved.
31:0 +12	**Address[63:32]**. The system physical address that the IOMMU attempted to access.

3.4.7 IOTLB_INV_TIMEOUT

If the IOMMU sends an invalidation request to a device and does not receive a response before the invalidation

timeout timer expires, the IOMMU writes the event log with a IOTLB_INV_TIMEOUT event.

The Address field contains the system physical address of the invalidation command that timed out.

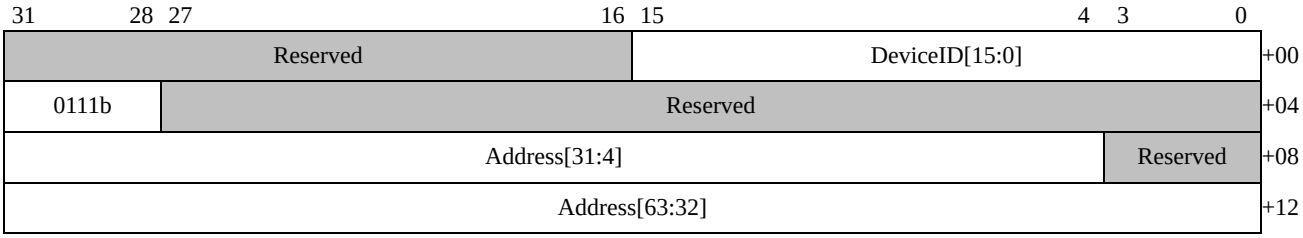


Figure 29: IOTLB_INV_TIMEOUT Event Log Buffer Entry

Table 19: IOTLB_INV_TIMEOUT Event Log Buffer Entry Fields

Bits	Description
31:16 +00	Reserved.
15:0 +00	**DeviceID.** Specifies the DeviceID that caused the invalidation timeout. The identity of the device causing the error can be determined using the DeviceID field.
31:28 +04	**0111b**. Specifies a IOTLB_INV_TIMEOUT entry.
27 +04	Reserved.
26:25 +04	**Type.** The Type field indicates the type of hardware error that occurred as listed in Table 14.
24:0 +04	Reserved.
31:4 +08	**Address[31:4]**. The system physical address of the invalidation command that timed out.
3:0 +08	Reserved.
31:0 +12	**Address[63:32]**. The system physical address of the invalidation command that timed out.

3.4.8 INVALID_DEVICE_REQUEST

If the IOMMU receives a request from a device that the device is not allowed to perform, the IOMMU writes the event log with a INVALID_DEVICE_REQUEST event. Creation of event log entries for INVALID_DEVICE_REQUEST events is controlled by the IG bit in the device table entry (Figure 5 and Table 3).

Table 20: INVALID_DEVICE_REQUEST Type Field Encodings

Type	TR	Description
000b	0b	Read request or non-posted write in the interrupt address range (see Table 2).
001b	0b	Pre-translated transaction received from an I/O device that has I=0 or V=0 in the devices' device table.
010b	0b	Port I/O space transaction received from an I/O device that has IoCtl=00b in the devices' device table.
011b	0b	Posted write to the system management address range received from an I/O device that has SysMgt=00b, or with SysMgt=10b and the message is not a INTx message in the devices' device table, or a posted write to the address translation range when HtAtsResv=1 (see Table 2).
100b	0b	Read request or non-posted write in the system management address range (if SysMgt=10b or 0xb), or a read request or a non-posted write in the address translation range when HtAtsResv=1 (see Table 2).
101b	0b	Posted write to the Interrupt/EOI range from an I/O device that has IntCtl=00b in the devices' device table (see Table 2).
110b	0b	Posted write to a reserved interrupt address range (see Table 2).
111b	0b	Transaction to the system management address range when SysMgt=11b or to the port I/O space range when IoCtl=10b, while V=1 and TV=0.
000b	1b	Translation request received from an I/O device that has I=0, or has V=0, or has V=1 and TV=0 in the devices' device table.
001b	1b	Translation request in the interrupt, port I/O space (if IoCtl=0xb), or system management address range (if SysMgt=0xb or 10b); or translation request in the system management address range when SysMgt=11b or in the port I/O space range when IoCtl=10b, while V=1 and TV=0.
010b-111b	1b	Reserved.

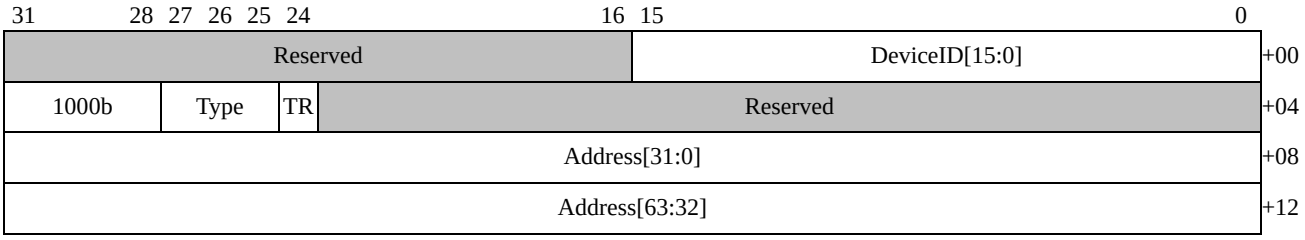


Figure 30: INVALID_DEVICE_REQUEST Event Log Buffer Entry

Table 21: INVALID_DEVICE_REQUEST Event Log Buffer Entry Fields

Bits	Description
31:16 +00	Reserved.
15:0 +00	**DeviceID**. Specifies the DeviceID that caused the page table lookup. The address of the device table entry can be determined using the DeviceID field.
31:28 +04	**1000b**. Specifies an INVALID_DEVICE_REQUEST entry.

Table 21: INVALID_DEVICE_REQUEST Event Log Buffer Entry Fields

27:25 +04	**Type.** The Type field indicates the type of hardware error that occurred as listed in Table 20.
24 +04	**TR: translation**. 1=transaction that caused the page table lookup was a translation request. 0=transaction that caused the page table lookup was a transaction request. See Table 20.
23:0 +04	Reserved.
31:0 +08	**Address[31:0]**. The address that the device attempted to translate or access.
31:0 +12	**Address[63:32]**. The address that the device attempted to translate or access.

3.5 IOMMU Interrupt Support

The IOMMU uses standard PCI interrupt mechanisms to generate interrupts. The IOMMU must support signaling of either MSI or MSI-X interrupts. The MSI capability must support 64-bit addressing. The IOMMU must not set the PassPW bit when sending interrupts associated with the IOMMU over HyperTransport™ links.

The IOMMU supports generation of an interrupt when the event log is updated and when a completion wait command completes (see Capability Offset 10h[MsiNum], MMIO Offset 0018h[ComWaitEn], and MMIO Offset 0018h[EventIntEn]).

3.6 PCI Resources

The IOMMU must be implemented as an independent function. A device may implement more than one IOMMU within the single function. Configuration and status information for the IOMMU are mapped into PCI configuration space using a PCI capability block. One or more IOMMU capability blocks may be implemented in a function. If more than one IOMMU capability block is implemented in a function the IOMMU must support generating MSI-X interrupts with one table entry per IOMMU capability block. The PCI class is System Base Peripheral (08h) with a subclass of IOMMU (06h) and a programming interface code of 00h, as issued by the PCI-SIG.

The HyperTransport™ UnitID used when an IOMMU generates requests must not be used for any other traffic. A HyperTransport™ UnitID can be shared by multiple IOMMUs within a physical component.

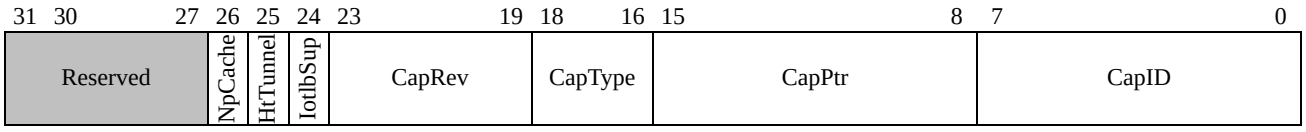
IOMMU registers not specified here return 0s when read and ignore the data when written.

3.6.1 IOMMU Capability Block Registers

A new PCI capability block indicates the presence of the IOMMU and the location of the IOMMUs control registers.

Capability Offset 00h IOMMU Capability Header

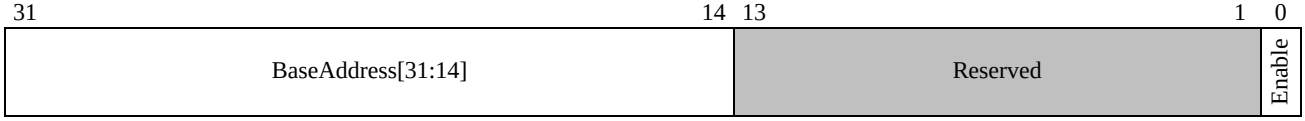
This register indicates that this is an IOMMU capability block.



Bits	Description
31:27	Reserved.
26	**NpCache: not present table entries cached**. RO. Reset Xb. 1=Indicates that the IOMMU caches page table entries that are marked as not present. When this bit is set, software must issue an invalidate after any change to a PDE or PTE . 0=Indicates that the IOMMU caches only page table entries that are marked as present. When NpCache is clear, software must issue an invalidate after any change to a PDE or PTE marked present before the change. **Implementation note:** For hardware implementations of the IOMMU, this bit must be 0b.
25	**HtTunnel: HyperTransport™ tunnel translation support**. RO. Reset Xb. Indicates that the device contains a HyperTransport™ tunnel that supports address translation on the HyperTransport™ interface.
24	**IotlbSup: IOTLB Support**. RO. Reset Xb. Indicates support for remote IOTLBs.
23:19	**CapRev: capability revision**. RO. Reset 00001b. Specifies the IOMMU specification revision.
18:16	**CapType: IOMMU capability block type**. RO. Reset 011b. Specifies the layout of the Capability Block as an IOMMU capability block.
15:8	**CapPtr: capability pointer**. RO. Reset XXh. Indicates the location of the next capability block if one is present.
7:0	**CapId: capability ID**. RO. Reset 0Fh. Indicates a Secure Device capability block.

Capability Offset 04h IOMMU Base Address Low Register

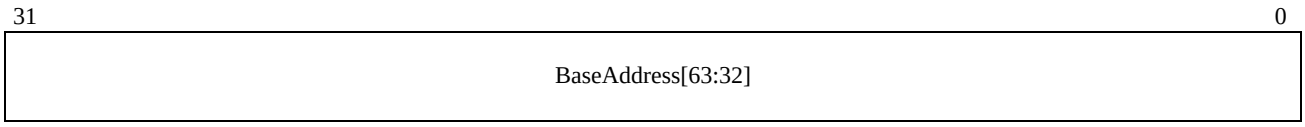
This register specifies the lower 32 bits of the base address of the IOMMU control registers. This register is locked when IOMMU Base Address Low[Enable] is written with a 1.



Bits	Description
31:14	**BaseAddress[31:14]**. RW when Capability Offset 04h[Enable]=0. RO when Capability Offset 04h[Enable]=1. Reset 0000_0000h. Specifies the lower 32 bits of the 16K-byte-aligned base address of the IOMMU control registers.
13:1	Reserved.
0	**Enable**. RW1S. Reset 0b. 1=IOMMU accepts memory accesses to the address specified in the Base Address Register. When Enable is written with a 1, all RW capability registers defined in Section 3.6.1 [IOMMU Capability Block Registers] are locked until the next system reset.

Capability Offset 08h IOMMU Base Address High Register

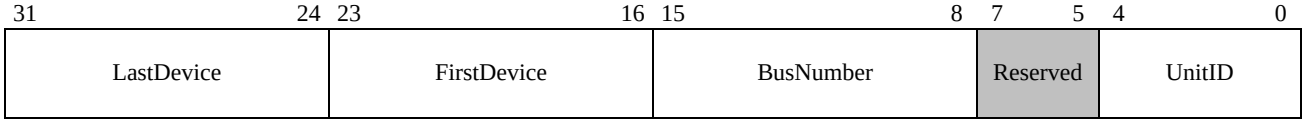
This register specifies the upper 32 bits of the base address of the IOMMU control registers. This register is locked when IOMMU Base Address Low[Enable] is written with a 1.



Bits	Description
31:0	**BaseAddress[63:32]**. RW when Capability Offset 04h[Enable]=0. RO when Capability Offset 04h[Enable]=1. Reset 0000_0000h. Specifies the upper 32 bits of the 16Kbyte-aligned base address of the IOMMU control registers.

Capability Offset 0Ch IOMMU Range Register

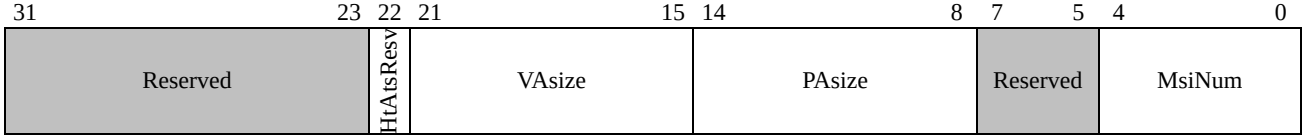
This register indicates the device and function numbers of the first and last devices associated with the IOMMU. This register is locked when IOMMU Base Address Low[Enable] is written with a 1. All root port devices that have device and function numbers between the first and last device numbers inclusive are supported by the IOMMU and provide full source identification to the IOMMU. All non-root port devices that have device and function numbers between the first and last device numbers inclusive are devices integrated in with the IOMMU and support address translation using the IOMMU. All integrated devices associated with the IOMMU must be located on the same logical bus.



Bits	Description
31:24	**LastDevice: last device**. RW when Capability Offset 04h[Enable]=0. RO when Capability Offset 04h[Enable]=1. Reset XXh. Indicates device and function number of the last integrated device associated with the IOMMU. **Note:** an implementation may define this value as RO.
23:16	**FirstDevice: first device**. RW when Capability Offset 04h[Enable]=0. RO when Capability Offset 04h[Enable]=1. Reset XXh. Indicates device and function number of the first integrated device associated with the IOMMU. **Note:** an implementation may define this value as RO.
15:8	**BusNumber: Device range bus number**. RW when Capability Offset 04h[Enable]=0. RO when Capability Offset 04h[Enable]=1. Reset XXh. Indicates the bus number that FirstDevice and LastDevice reside on. **Note:** an implementation may define this value as RO.
7:5	Reserved.
4:0	**UnitID: IOMMU HyperTransport™ UnitID**. RO. Reset X_XXXXb. This field returns the HyperTransport™ UnitID used by the IOMMU. **Note:** an implementation may define this value as RO.

Capability Offset 10h IOMMU Miscellaneous Information Register

This register returns the size of virtual and physical addresses supported by the IOMMU, and the message number for MSI or MSI-X interrupts associated with the IOMMU.



Bits	Description
31:23	Reserved.

22	**HtAtsResv: ATS response address range reserved.** RW when Capability Offset 04h[Enable]=0b. RO when Capability Offset 04h[Enable]=1b. Reset 0b. 1=The HyperTransport™ Address Translation address range for ATS responses is reserved and cannot be translated by the IOMMU. 0=The Address Translation address range can be translated by the IOMMU. See Table 2. Implementation note: This bit may be RO if ATS is not supported.
21:15	**VAsize: Virtual Address size.** RO. Reset XXXXXXb. This field must indicate the size of the maximum virtual address processed by the IOMMU. The value is the (unsigned) binary log of the maximum address size. Allowed values are 32, 40, 48, and 64; all other values are reserved. 010_0000b = 32 bits 010_1000b = 40 bits 011_0000b = 48 bits 100_0000b = 64 bits
14:8	**PAsize: Physical Address size.** RO. Reset XXXXXXb. This field must indicate the size of the maximum physical address generated by the IOMMU. The value is the (unsigned) binary log of the maximum address size. Allowed values are 48 and 52; all other values are reserved. 011_0000b = 48 bits 011_0100b = 52 bits
7:5	Reserved.
4:0	**MsiNum: MSI message number**. RO. Reset XXXXXb. This field must indicate which MSI/MSI-X vector is used for the interrupt message generated by the IOMMU. For MSI there can only be one IOMMU so this field must be zero (Section 3.6 [PCI Resources]). For MSI-X, the value in this field indicates which MSI-X Table entry is used to generate the interrupt message. Either MSI or MSI-X can be implemented, but not both. If the MSI-X capability is not enabled, INTx is used to deliver the interrupt. This interrupt is not remapped by the IOMMU.

3.6.2 IOMMU Control Registers

The IOMMU control registers are mapped using the IOMMU Base Address Low Register [Capability Capability Offset 04h] and IOMMU Base Address High Register [Capability Capability Offset 08h] specified in the IOMMU capability block. Software access to IOMMU registers may not be larger than 64 bits. All accesses must be aligned to the size of the access and the size in bytes must be a power of two. Software may use accesses as small as one byte.

MMIO Offset 0000h Device Table Base Address Register

This register specifies the system physical address of the device table.

Bits	Description
63:52	Reserved.

51:12	**DevTabBase: device table base address**. RW. Reset 00_0000_0000h. Specifies the 4Kbyte-aligned base address of the first level device table.
11:9	Reserved.
8:0	**Size: size of the device table**. RW. Reset 000h. This field contains 1 less than the length of the device table, in multiples of 4K bytes. A minimum size of 0 corresponds to a 4K byte device table and a maximum size of 1FFh corresponds to a 2M byte device table.

MMIO Offset 0008h Command Buffer Base Address Register

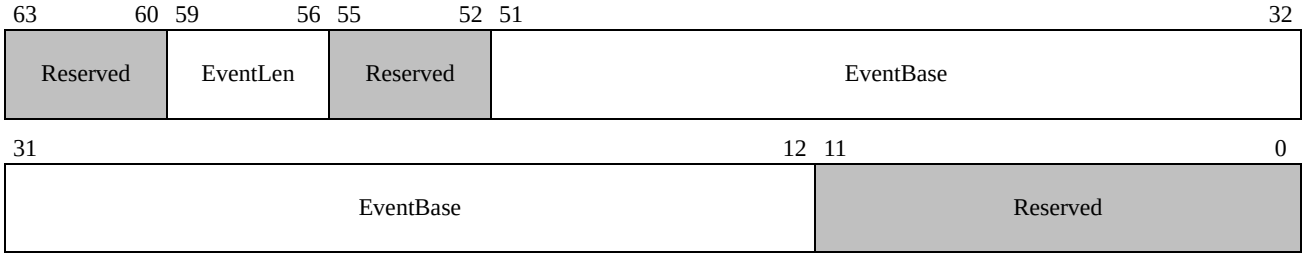
This register specifies the system physical address and length of the command buffer.

Bits	Description
63:60	Reserved.
59:56	**ComLen: command buffer length**. RW. Reset 1000b. Specifies the length of the command buffer in power of 2 increments. The minimum size is 256 entries (4K bytes); values less than 1000b are reserved. 0000b - 0111b = Reserved 1000b = 256 entries (4K bytes) 1001b = 512 entries (8K bytes) ... 1111b = 32768 entries (512K bytes)
55:52	Reserved
51:12	**ComBase: command buffer base address**. RW. Reset 00_0000_0000h. Specifies the base address of the command buffer. The base address programmed must be aligned to 4K bytes.
11:0	Reserved

MMIO Offset 0010h Event Log Base Address Register

This register specifies the system physical address and length of the event log.

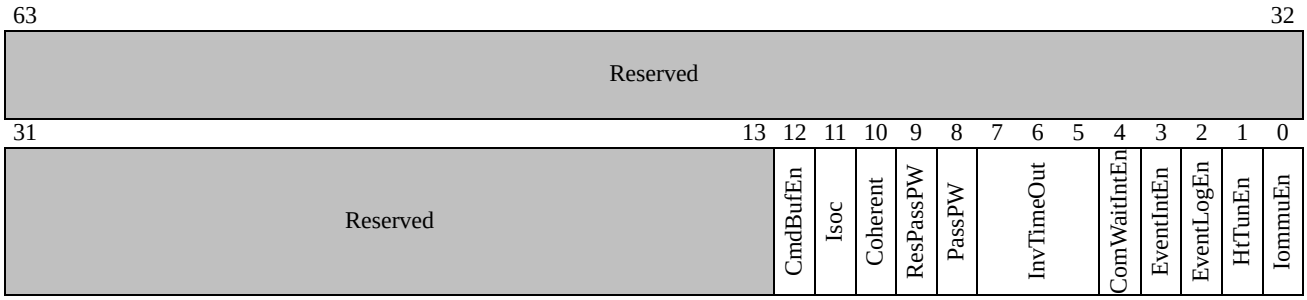
Software Note: If EventLen or EventBase is changed while the EventLogRun=1, the IOMMU behavior is undefined.



Bits	Description
63:60	Reserved.
59:56	**EventLen: event log length**. RW. Reset 1000b. Specifies the length of the event log in power of 2 increments. The minimum size is 256 entries (4K bytes); values less than 1000b are reserved. 0000b - 0111b = Reserved 1000b = 256 entries (4K bytes) 1001b = 512 entries (8K bytes) ... 1111b = 32768 entries (512K bytes)
55:52	Reserved
51:12	**EventBase: event log base address**. RW. Reset 00_0000_0000h. Specifies the base address of the event log. The base address programmed must be aligned to 4K bytes .
11:0	Reserved

MMIO Offset 0018h IOMMU Control Register

This register controls the behavior of the IOMMU.



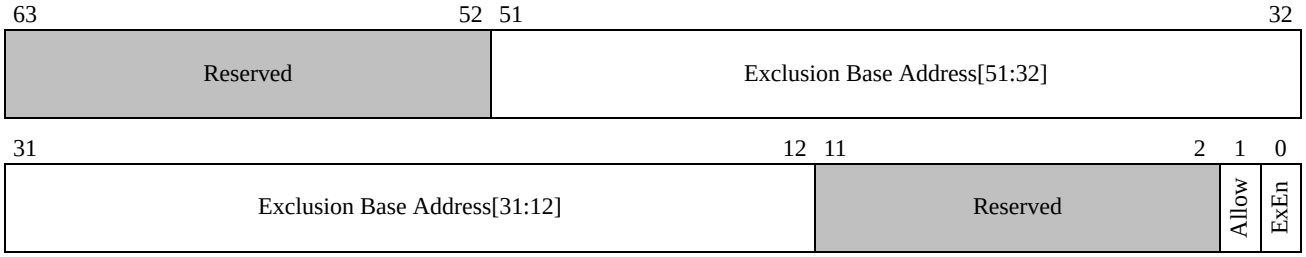
Bits	Description
63:13	Reserved.
12	**CmdBufEn: command buffer enable.** RW. Reset 0b. 1=Start or restart command buffer processing. When CmdBufEn=1b and IommuEn=1b, the IOMMU starts fetching commands and sets MMIO Offset 2020h[CmdBufRun] to 1b. Writing a 1b to this bit when CmdBufRun=1b has no effect. 0=Halt command buffer processing. Writing a 0 to this bit causes the IOMMU to cease fetching new commands although commands previously fetched are completed. The IOMMU stops fetching commands upon reset and after errors as specified in Table 11. See also MMIO Offset 2020h[CmdBufRun]. **Note:** see IOMMU Status Register [MMIO Offset 2020h] to determine the status of command buffer processing. **Note:** writing of event log entries is independently controlled by EventLogEn. **Software note:** the Command Buffer Base Address Register [MMIO Offset 0008h], the Command Buffer Head Pointer Register [MMIO Offset 2000h], and the Command Buffer Tail Pointer Register [MMIO Offset 2008h] must be set prior to enabling the IOMMU command buffer processor.
11	**Isoc: isochronous**. RW. Reset 0b. This bit controls the state of the isochronous bit in the HyperTransport™ read request packet when the IOMMU issues I/O page table reads and device table reads on the HyperTransport™ link. 1=Request packet to use isochronous channel. 0=Request packet to use standard channel. **Note**: Platform firmware should set this bit to 1b for processors that support the isochronous channel.

10	**Coherent: coherent**. RW. Reset 1b. This bit controls the state of the coherent bit in the HyperTransport™ read request packet when the IOMMU issues device table reads on the HyperTransport™ link. 1=Device table requests are snooped by the processor. 0=Device table requests are not snooped by the processor. See also SD in Table 3.
9	**ResPassPW: response pass posted write**. RW. Reset 0b. This bit controls the state of the ResPassPW bit in the HyperTransport™ read request packet when the IOMMU issues I/O page table reads and device table reads on the HyperTransport™ link. 1=Response may pass posted requests. 0=Response may not pass posted requests.
8	**PassPW: pass posted write**. RW. Reset 0b. This bit controls the state of the PassPW bit in the HyperTransport™ read request packet when the IOMMU issues I/O page table reads and device table reads on the HyperTransport™ link. 1=Request packet may pass posted requests. 0=Request packet may not pass posted requests.
7:5	**InvTimeOut: invalidation time-out**. RW. Reset 000b. This field specifies the invalidation time-out for IOTLB invalidation requests. 000b=No time-out 001b=1 ms 010b=10 ms 011b=100 ms 100b=1 sec. 101b=10 sec. 110b, 111b=Reserved
4	**ComWaitIntEn: completion wait interrupt enable**. RW. Reset 0b. 1=An interrupt is signalled when MMIO Offset 2020h[ComWaitInt]=1.
3	**EventIntEn: event log interrupt enable**. RW. Reset 0b. 1=An interrupt is signalled when the EventLogInt bit is set in the IOMMU Status Register [MMIO Offset 2020h].
2	**EventLogEn: event log enable**. RW. Reset 0b. 1=The Event Log Base Address Register [MMIO Offset 0010h] has been configured and all events detected are written to the event log when IommuEn has also been set. Writing a 1b to this bit when EventLogEn=1b has no effect. 0=Event logging is not enabled. Events are discarded when the event log is not enabled. When IommuEn=1b and software writes EventLogEn with 1b, the IOMMU clears the EventOverflow bit, and sets the EventLogRun bit in the IOMMU Status Register [MMIO Offset 2020h]. The IOMMU can now write new entries to the event log if there are usable entries available. **Note:** software can read MMIO Offset 2020h[EventLogRun] to determine the status of event log writing by the IOMMU. **Note:** the fetching of command is independently controlled by CmdBufEn. **Software note:** the Event Log Base Address Register [MMIO Offset 0010h], the Event Log Head Pointer Register [MMIO Offset 2010h], and the Event Log Tail Pointer Register [MMIO Offset 2018h] must be set prior to enabling the event log.

1	**HtTunEn: HyperTransport™ tunnel translation enable**. RW. Reset 0b. 1= Upstream traffic received by the HyperTransport™ tunnel is translated by the IOMMU. 0=Upstream traffic received by the HyperTransport™ tunnel is not translated by the IOMMU. The IOMMU ignores the state of this bit while IommuEn=0. See also the HtTunnel bit in the IOMMU Capability Header [Capability Offset 00h].
0	**IommuEn: IOMMU enable**. RW. Reset 0b. 1=IOMMU enabled. All upstream transactions are translated by the IOMMU. The Device Table Base Address Register [MMIO Offset 0000h] must be configured by software before setting this bit. 0=IOMMU is disabled and no upstream transactions are translated or remapped by the IOMMU. When disabled, the IOMMU reads no commands and creates no event log entries. **Software note**: Software must configure EventLogEn and CmdBufEn.

MMIO Offset 0020h IOMMU Exclusion Base Register

This register specifies the base device virtual address of the IOMMU exclusion range. Transactions that target addresses in the exclusion range are neither translated nor access checked if the EX bit in the device table is set for the device or if the Allow bit is set in this register.



Bits	Description
63:52	Reserved.
51:12	**Exclusion range base address**. RW. Reset 00_0000_0000h . Specifies the 4Kbyte-aligned base address of the exclusion range.
11:2	Reserved.
1	**Allow: allow all devices**. RW. Reset 0b. 1=All accesses to the exclusion range are forwarded untranslated. 0=The EX bit in the device table entry specifies if accesses to the exclusion range are translated.
0	**ExEn: exclusion range enable**. RW. Reset 0b. 1=The exclusion range is enabled. 0=the exclusion range is disabled.

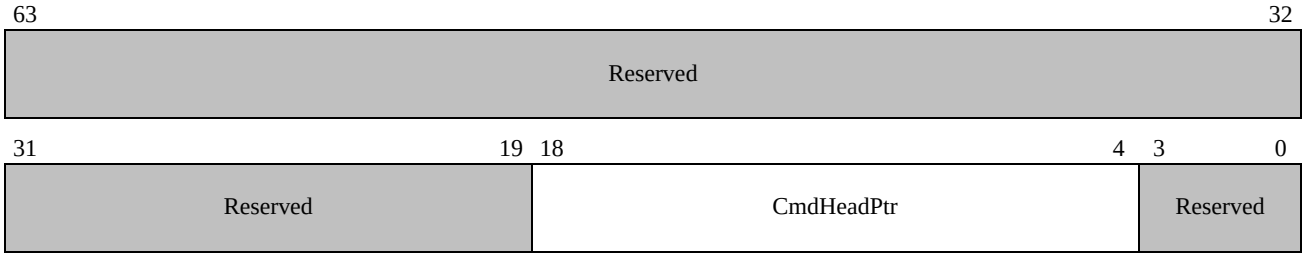
MMIO Offset 0028h IOMMU Exclusion Limit Register

This register specifies the limit of the IOMMU exclusion range. The lower 12 bits of the limit are treated as FFFh for range comparisons.
Note: when the exclusion base address equals the exclusion limit address, the exclusion range is 4K bytes.

Bits	Description
63:52	Reserved.
51:12	**Exclusion range limit**. RW. Reset 00_0000_0000h . Specifies the 4K byte limit of the exclusion range.
11:0	Reserved.

MMIO Offset 2000h Command Buffer Head Pointer Register

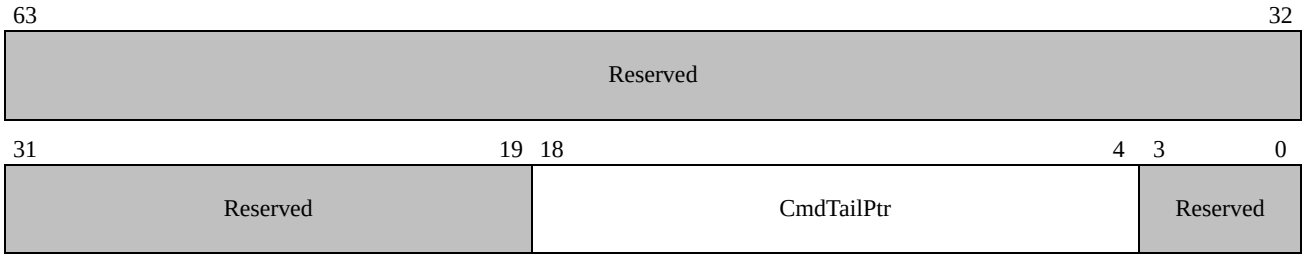
This register points to the offset in the command buffer that will be read next by the IOMMU.



Bits	Description
63:19	Reserved.
18:4	**CmdHeadPtr: command buffer head pointer**. RW. Reset 0000h. Specifies the 128-bit aligned offset from the command buffer base address register of the next command to be fetched by the IOMMU. The IOMMU increments this register, rolling over to zero at the end of the buffer, after fetching and validating the command in the command buffer. After incrementing this register, the IOMMU cannot re-fetch the command from the buffer. If this register is written to by software while CmdBufRun=1b, the IOMMU behavior is undefined. If this register is set by software to a value outside the length specified by MMIO Offset 0008h[ComLen], the IOMMU behavior is undefined.
3:0	Reserved.

MMIO Offset 2008h Command Buffer Tail Pointer Register

This register points to the offset in the command buffer that will be written next by the software.

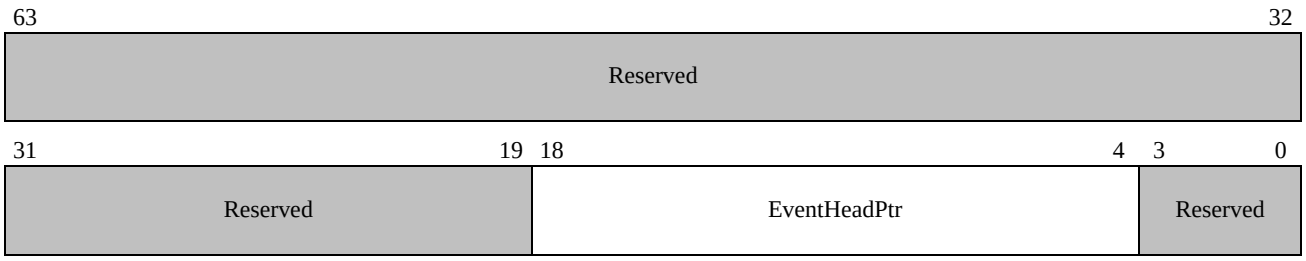


Bits	Description
63:19	Reserved.

18:4	**CmdTailPtr: command buffer tail pointer**. RW. Reset 0000h. Specifies the 128-bit aligned offset from the command buffer base address register of the next command to be written by the software. Software must increment this field, rolling over to zero at the end of the buffer, after writing a command to the command buffer. If software advances the tail pointer equal to or beyond the head pointer after adding one or more commands to the buffer, the IOMMU behavior is undefined. If software sets the command buffer tail pointer to an offset beyond the length of the command buffer, the IOMMU behavior is undefined.
3:0	Reserved.

MMIO Offset 2010h Event Log Head Pointer Register

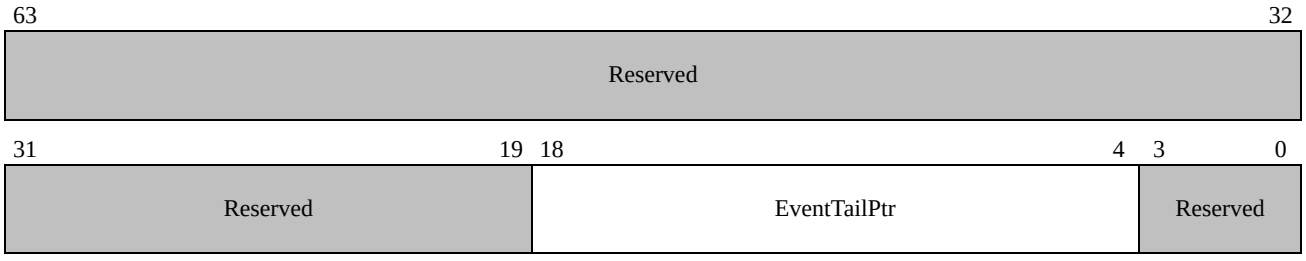
This register points to the offset in the event buffer that will be read next by the software.



Bits	Description
63:19	Reserved.
18:4	**EventHeadPtr: event log head pointer**. RW. Reset 0000h. Specifies the 128 bit aligned offset from the event log base address register that will be read next by software. Software must increment this field, rolling over at the end of the buffer, after reading an event from the event log. If software advances the head pointer beyond the tail pointer, the IOMMU behavior is undefined. If software sets the event log head pointer to an offset beyond the length of the event log, the IOMMU behavior is undefined.
3:0	Reserved.

MMIO Offset 2018h Event Log Tail Pointer Register

This register points to the offset in the event buffer that will be written next by the IOMMU.

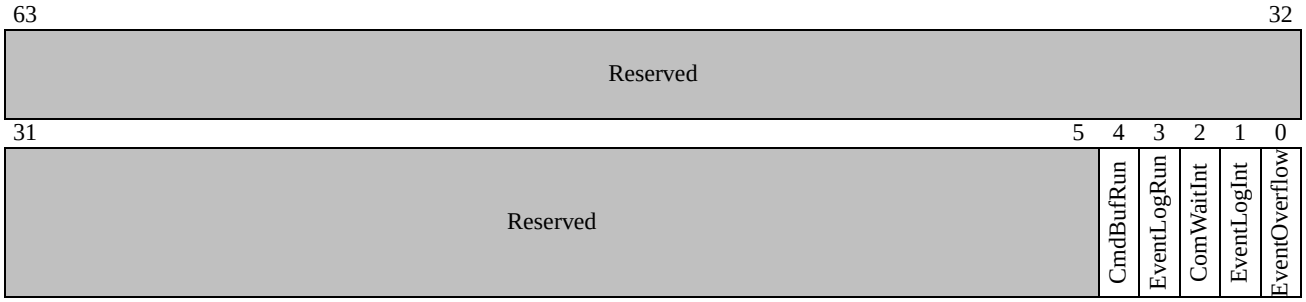


Bits	Description
63:19	Reserved.

18:4	**EventTailPtr: event log tail pointer**. RW. Reset 0000h. Specifies the 128-bit aligned offset from the event log base address register that will be written next by the IOMMU when an event is detected. The IOMMU increments this register, rolling over at the end of the buffer, after writing an event to the event log. If this register is written while EventLogRun=1, the IOMMU behavior is undefined. If this register is set by software to a value outside the length specified by MMIO Offset 0010h[EventLen], the IOMMU behavior is undefined
3:0	Reserved.

MMIO Offset 2020h IOMMU Status Register

This register indicates the current status of the IOMMU interrupt sources. If interrupts are enabled the IOMMU signals an interrupt when one of the status bits is set by hardware and no other bits are set.



Bits	Description
63:5	Reserved.
4	**CmdBufRun: command buffer running.** RO. Reset 0b. 1=commands may be fetched from the command buffer. 0=IOMMU has stopped fetching new commands. The IOMMU freezes command processing after COMMAND_HARDWARE_ERROR (Section 3.4.6 [COMMAND_HARDWARE_ERROR]) and ILLEGAL_COMMAND_ERROR (Section 3.4.5 [ILLEGAL_COMMAND_ERROR]) errors. When frozen, command fetching is restarted by using MMIO Offset 0018h[CmdBufEn].
3	**EventLogRun: event logging is running.** RO. Reset 0b. 1=events are logged as they occur. 0=event reports are discarded without logging. When EventOverflow=1b, the IOMMU will not write new event log entries even when EventLogRun=1b. When halted, event logging is restarted by using MMIO Offset 0018h[EventLogEn].
2	**ComWaitInt: completion wait interrupt**. RW1C. Reset 0b. 1=COMPLETION_WAIT command completed. This bit is only set if the i bit is set in the COMPLETION_WAIT command. An interrupt is generated when ComWaitInt=1b and MMIO Offset 0018h[ComWaitIntEn]=1b.
1	**EventLogInt: event log interrupt**. RW1C. Reset 0b. 1=Event entry written to the event log by the IOMMU. 0=No event entry written to the event log by the IOMMU. An interrupt is generated when EventLogInt=1b and MMIO Offset 0018h[EventIntEn]=1b.
0	**EventOverflow: event log overflow**. RW1C. Reset 0b. 1=IOMMU event log overflow has occurred. This bit is set when a new event is to be written to the event log and there is no usable entry in the event log, causing the new event information to be discarded. An interrupt is generated when EventOverflow=1b and MMIO Offset 0018h[EventIntEn]=1b. No new event log entries are written while this bit is set.

4 Implementation Considerations

This chapter discusses issues that are primarily of concern to IOMMU implementers.

The IOMMU specification is intended to allow a wide range of implementations with different cost and performance trade-offs. Potential implementation technology may range from ASIC to full custom. Capacity and organization of the IOMMU's translation caches can vary substantially depending on technology, die budgets, and product requirements. The IOMMU can be integrated with a chipset (typically as part of some existing HyperTransport™ bridge) or built as a standalone component (which can act as a HyperTransport™ bridge or tunnel).

4.1 Caching and Invalidation Strategies

All IOMMU implementations should have some form of translation cache that allows the IOMMU to determine the disposition of device accesses quickly without having to re-walk the IOMMU tables for each separate device access. The translation cache will likely be the largest portion of the IOMMU's die area budget in all but the smallest implementations. Consequently the IOMMU specification has been written to allow considerable flexibility in the design of the translation cache.

Plausible implementations range from direct mapped RAM structures to fully associative CAM structures, with the expectation that most implementations will be set associative. Furthermore, implementers may choose to flatten the multi-stage IOMMU table walk into a single cache array lookup, or, alternatively, may choose to use a similar multi-stage organization for internal translation cache lookups.

The IOMMU's translation cache must support the following operations:

- Lookup — when the IOMMU processes an access by a particular device to a specified device virtual address, what protection and translation should apply? The lookup process must be keyed by DeviceID and device virtual address.
- Invalidate device — discard any translation cache contents that depend on a specific device table entry.
- Invalidate virtual address (within domain) — discard any cached translations for a virtual address within the specified domain.

Typical IOMMU implementations are likely to be built with ASIC design flows, where CAM cells are very expensive compared to RAM cells. The main implication of this is that direct support for different page sizes is likely to require a combination of separate arrays and/or multiple entries within arrays, and therefore both fills and invalidations may require time-consuming search-and-destroy algorithms.

The IOMMU is designed to support three main usage models:

- Direct user process access to a single device like a graphics controller,
- Direct virtual machine guest access to a collection of devices that have been dedicated to that guest, and
- A single non-virtualized OS using the IOMMU to enforce device to system memory access controls.

When a user process directly controls a single device, the total memory footprint for the device's accesses is likely to be a modest fraction of the process's own memory footprint. Moreover the user process has direct knowledge of the specific device, so there is a good chance that the device's access pattern will be controllable for good locality. In this case the main consideration for achieving good performance is to ensure that the IOMMU's translation cache is large enough.

By contrast, the potential memory footprint of a virtual machine guest's devices is the entire memory of the

guest. Worse, the access pattern is poorly controlled; it is determined by the guest operating system's workload (of which the VMM likely has no specific knowledge), and, moreover, consists of interactions with a variety of devices under the control of different guest device drivers and subsystems, with diverse memory allocation strategies. In the case of a non-paravirtualized guest, a VMM's best strategy for good performance is probably to set up I/O page tables using the largest available page size and assume that the IOMMU can share the same translation cache entries among multiple devices. It is for this reason that the IOMMU's table structure includes a DomainID that can be shared for multiple DeviceIDs: since the IOMMU uses translation cache entries tagged by *{DomainID, I/O virtual address}* it will automatically share translations among multiple devices assigned to the same domain.

Based on these considerations, AMD recommends the following two-stage organization for the IOMMU's translation cache:

- The first stage should map *DeviceID* to {*DomainID, I/O page table base address}*. Most systems have only a few distinct DeviceIDs, so the capacity of the first stage can be small. The one complication is that DeviceIDs are not very random and tend to be clustered, so, to avoid conflicts, this stage should either be highly associative or use a good DeviceID hash function.
- The second stage should map *{DomainID, device virtual address}* to *{system physical address, protection}*. This stage should have (at least) hundreds of entries. This stage should explicitly include the DomainID in set index hashing (rather than just using the DomainID as a tag), so that different domains with similar memory layouts will not compete for the same translation cache entries. (Server consolidation environments are likely to create many domains with very similar memory layouts.)

In addition, since the latency of IOMMU access to system memory is high, it is further recommended that implementers include a *page directory cache (PDC)* to accelerate processing of translation cache misses. This cache should map *{DomainID, device virtual address}* to *page directory entry (PDE)*, so that the IOMMU can quickly calculate the address of the final PTE needed to resolve a translation cache miss. This way, most translation cache misses can be resolved in a single memory access by the IOMMU, rather than requiring a full multi-stage table walk. The page directory cache could also double as a large-page translation cache, since for large pages the PDE is also the PTE.

4.2 Recommended IOMMU Topologies

The IOMMU's architecture is designed to accommodate a variety of system fabrics and topologies. There can be multiple IOMMUs, located at a variety of places in the system fabric. Some requestor ID information can be lost at bridges between busses or bus types, so it is advantageous to locate IOMMUs in bridges. The mapping of bus requesterIDs to IOMMU DeviceIDs depends on both the bus type as well as the IOMMU's location in the system fabric. In most other respects, the IOMMU's behavior is bus-independent.

The simplest possible implementation of the IOMMU takes the form of a HyperTransport™ tunnel.

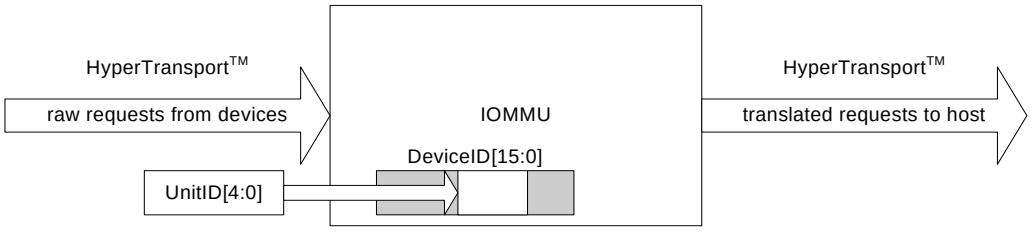


Figure 31: IOMMU in a Tunnel

The advantage of this approach is that it can be easily retrofitted to an existing system design. The main

limitation of this approach is that the HyperTransport™ specification defines only 5 bits of UnitID information to identify the originators of requests, so the IOMMU can provide distinct translations for at most 31 downstream devices. If downstream nodes include any bridges, the IOMMU is unable to distinguish between different devices beyond the bridges, since bridged requests use the UnitID of the bridge. One possible solution to would be to include a separate IOMMU on each downstream bus; each IOMMU can then be programmed not to rewrite transactions whose UnitID proves they have already passed through another IOMMU. Software must understand the system topology to correctly coordinate multiple IOMMUs. If a downstream HyperTransport™ device is a PCIe® root complex or a PCI-X® host bridge, the device can implement the RequesterID mapping capability to assign specific UnitIDs to PCIe® or PCI-X® devices.

An IOMMU implemented in a PCIe®- or PCI-X®-to-HyperTransport™ bridge can exploit the larger PCIe® or PCI-X® RequesterID namespace to provide better discrimination between downstream devices when translating requests:

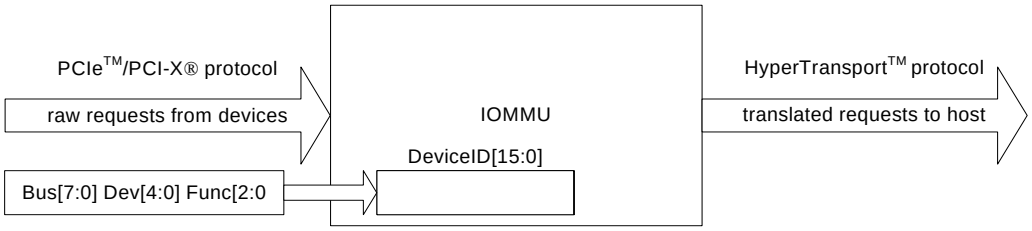


Figure 32: IOMMU in a Peripheral Bus Bridge

Since most future commodity devices will be based on the PCIe® bus, this is likely to be the most common implementation of the IOMMU for low-cost systems.

Large systems may want a scalable IOMMU building block. Such systems may choose to implement a hybrid HyperTransport™ tunnel / PCIe® root complex component or a HyperTransport™ tunnel / PCI-X® host bridge component combining the above ideas:

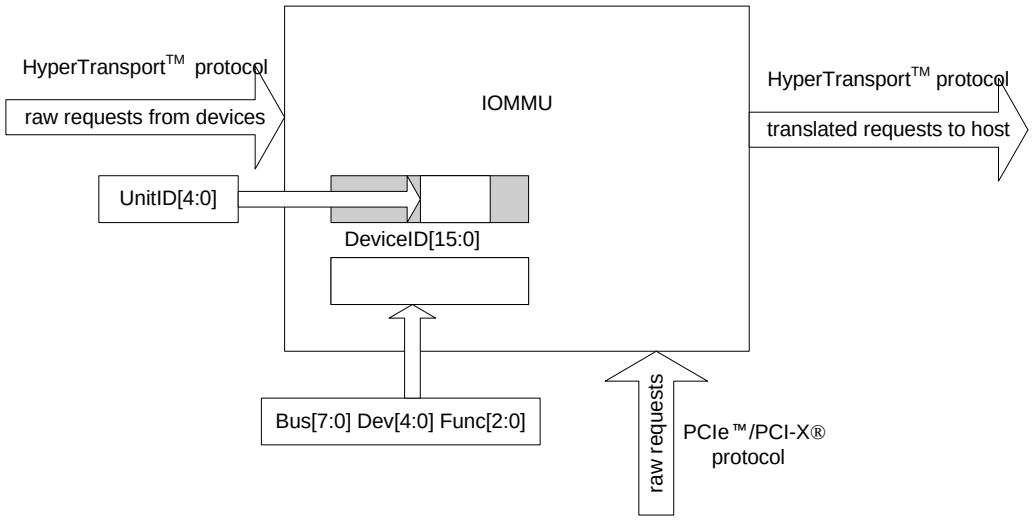


Figure 33: Hybrid IOMMU

Hybrid IOMMUs can be chained together to build large systems:

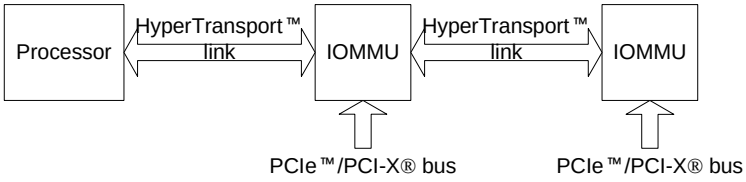


Figure 34: Chained Hybrid IOMMU in a Large System

4.3 Issues Specific to the HyperTransport™ Architecture

This section discusses implementation considerations that are specific to IOMMUs attached to a HyperTransport™ link.

The HyperTransport™ specification requires devices (especially tunnels and bridges) to interoperate with other devices in ways that ensure correctness and maintain performance. Among other requirements, HyperTransport™ devices must make certain transaction ordering guarantees and must ensure they will operate without deadlocks.

A key requirement in the HyperTransport™ specification is that posted requests must be able to pass non-posted requests. The introduction of the IOMMU, however, means that posted requests (e.g. writes to memory) may spawn non-posted requests (I/O page table walks) that must complete before the posted request can be allowed to progress further.

To ensure deadlock free operation, the IOMMU requires a dedicated virtual channel for its I/O page table walk requests. This ensures that, the IOMMU's page table walks on behalf of posted requests can complete, regardless of the completion status of other non-posted traffic in the fabric. The IOMMU also requires that the host bridge process its requests without spawning any requests to other devices. In other words, the IOMMU's table structures must be located solely in system memory.

The IOMMU can share its virtual channel with other traffic as long the other traffic is also guaranteed to make forward progress. In practice, this means that any other devices sharing the IOMMU's page walk channel must also restrict their non-posted traffic solely to accessing system memory.

To allow the IOMMU to support different AMD processors with different isochronous capabilities the IOMMU control registers contain bits that control the state of the PassPW bits, the coherent bit and the isochronous bit in the HyperTransport™ link read request issued by the IOMMU.

4.4 Chipset Specific Implementation Issues

Chipsets that implement both an IOMMU and a legacy PCI or AGP bridge must provide source identification to identify uniquely DMA traffic as originating from the PCI or AGP bus.To provide this identification, the IOMMU must use the requesterID of the PCI or AGP bridge to perform translations for DMA transactions from the legacy bus.

4.5 Software and BIOS Implementation Issues

Because of the flexible architecture of the IOMMU, it is unlikely that any single system software implementation will use all the features, topologies, or options. The following constraints are strongly recommended.

- An IOMMU should be a root-complex device (i.e., appear directly on the bus at the top of the PCI tree hierarchy).
- Some system software may prohibit an IOMMU to appear under a PCI-to-PCI bridge.
- To ensure the IOMMU is recognized and configured properly, the BIOS should perform the initial configuration of the IOMMU so that it is accessible to system software when control is handed off by the BIOS.
- The BIOS should describe the IOMMU in an ACPI table. The table must include all information necessary to identify, configure, and access the IOMMU.

5 IOMMU Page Walker Pseudo Code

```
//
// Page table walker for IOMMU.
//
// Inputs:
// {dte} is partial device table entry (lower 64-bits),
// {dva} is device virtual address.
//
// Return value is a (possibly synthetic) 64-bit "pte" suitable for storing
// in a TLB, with the following fields valid:
//    [62]       (cumulative) I/O write permission
//    [61]       (cumulative) I/O read permission
//    [60]       FC bit
//    [59]       U bit
//    [51:12]    system physical page address
//    [5:0]      how many VA bits to append
//
// The caller of this routine is responsible for read and write permission
// checks, and for checks that the dte is valid for translation.
// The caller is responsible for special access permission check in the
// case of a 0-length read.
// This routine performs all other checks, and exits by raising an
// exception (instead of returning a value) if any problem is found.
//


#define LARGEST_VA(LEVEL)    ((0x1000ull << ((LEVEL) * 9)) - 1)
#define VABITS(LEVEL)            (((LEVEL) * 9) + 3)

#define IOPERM      0x6000000000000000       // 6000_0000_0000_0000h - IR and IW bits in PTE
#define RESV_BITS 0x1FF0000000000000      // 1FF0_0000_0000_0000h - Reserved bits in PTE
#define U_FC_BITS 0x1800000000000000      // 1800_0000_0000_0000h - U, FC bits in PTE
#define BITS_51_12 0xFFFFFFFFFF000        // 000F_FFFF_FFFF_F000h - bit mask [51:12]

uint64
iopagewalk(uint64 dte, uint64 dva)
{
   uint64 pdte = dte;
   uint64 ioperm = pdte & IOPERM;
   uint64 pa = pdte & BITS_51_12;           // dte bits [51:12]
   uint oldlevel = 7, level = (pdte >> 9) & 7, vabits = 63;

   if (level == 7)
      raise DEVTAB_RESERVED_LEVEL;

   if (level == 0)
      return ioperm | pa | vabits;

   while (level != 0) {
      uint64 skipbits = LARGEST_VA(oldlevel - 1) - LARGEST_VA(level);
```

```
      if ((dva & skipbits) != 0)
         raise PAGE_NOT_PRESENT;
      uint offset = (dva >> (level * 9)) & 0xFF8;
      pdte = read_memory_qword(pa + offset);
      if ((pdte & 1) == 0)
         raise PAGE_NOT_PRESENT;

      oldlevel = level;
      level = (pdte >> 9) & 7;
      uint64 reserved_bits = RESV_BITS;
      if (level == 0 || level ==7) {
         reserved_bits &= ~U_FC_BITS;  // U and FC bits are not reserved for PTEs
         ioperm |= pdte & U_FC_BITS;    // merge U and FC bits into result
      }
      if ((pdte & reserved_bits ) != 0)
         raise PDTE_RESERVED_BITS;
      ioperm &= pdte;
      pa = pdte & BITS_51_12;           // pte bits [51:12]
      oldlevel = level;
      level = (pdte >> 9) & 7;
      if (level == 0x7) {
         uint64 tmp = pa ^ BITS_51_12;
         vabits = bsf(tmp) + 1;                    // find first 0 in pa[51:12] - see BSF instruction
         if ((vabits >= VABITS(oldlevel + 1)) || (vabits <= VABITS(oldlevel)))
            raise PDTE_RESERVED_BITS;
         pa &= ~((1ull << vabits) - 1 );
         return ioperm | pa | vabits
      }
      if (level >= oldlevel)
         raise PDTE_RESERVED_BITS;
   }

   if ((pa & LARGEST_VA(oldlevel - 1) != 0)
      raise PDTE_RESERVED_BITS;


   return ioperm | pa | VABITS(oldlevel);
}
```

6 Register List

The following is a list of all storage elements, context, and registers provided in this document. Page numbers, register mnemonics, and register names are provided.

61 Capability Offset 00h: IOMMU Capability Header
62 Capability Offset 04h: IOMMU Base Address Low Register
62 Capability Offset 08h: IOMMU Base Address High Register
63 Capability Offset 0Ch: IOMMU Range Register
63 Capability Offset 10h: IOMMU Miscellaneous Information Register
64 MMIO Offset 0000h: Device Table Base Address Register
65 MMIO Offset 0008h: Command Buffer Base Address Register
65 MMIO Offset 0010h: Event Log Base Address Register
66 MMIO Offset 0018h: IOMMU Control Register
68 MMIO Offset 0020h: IOMMU Exclusion Base Register
68 MMIO Offset 0028h: IOMMU Exclusion Limit Register
69 MMIO Offset 2000h: Command Buffer Head Pointer Register
69 MMIO Offset 2008h: Command Buffer Tail Pointer Register
70 MMIO Offset 2010h: Event Log Head Pointer Register
70 MMIO Offset 2018h: Event Log Tail Pointer Register
71 MMIO Offset 2020h: IOMMU Status Register