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

© 2005, 2006 Advanced Micro Devices, Inc.

All rights reserved.The contents of this document are provided in connection with Advanced Micro Devices, Inc. ("AMD") products. AMD makes no representations or warranties with respect to the accuracy or completeness of the contents of this publication and reserves the right to make changes to specifications and product descriptions at any time without notice. No license, whether express, implied, arising by estoppel or otherwise, to any intellectual property rights is granted by this publication. Except as set forth in AMD's Standard Terms and Conditions of Sale, AMD assumes no liability whatsoever, and disclaims any express or implied warranty, relating to its products including, but not limited to, the implied warranty of merchantability, fitness for a particular purpose, or infringement of any intellectual property right. AMD's products are not designed, intended, authorized or warranted for use as components in systems intended for surgical implant into the body, or in other applications intended to support or sustain life, or in any other application in which the failure of AMD's product could create a situation where personal injury, death, or severe property or environmental damage may occur. AMD reserves the right to discontinue or make changes to its products at any time without notice.

Trademarks

AMD, the AMD Arrow logo, and combinations thereof, 3DNow!, and AMD PowerNow! are trademarks of Advanced Micro Devices, Inc.

HyperTransport is a licenced trademark of the HyperTransport Technology Consortium.

PCI Express and PCIe are licenced trademarks of the PCI Special Interest Group.

PCI-X is registered trademark of the PCI Special Interest Group.

Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

Table of Contents

1 Overview . . . 9
1.1 Intended Audience . . . 9
1.2 Definitions. . . . 9
1.3 Bit Attributes. . . . 10

2 IOMMU Overview . . . 11
2.1 Architecture Summary . . . 11
2.2 Usage Models . . . 12
2.2.1 Replacing the GART . . . 12
2.2.2 Replacing the DEV . . . 13
2.2.3 32-bit to 64-bit Legacy I/O Device Mapping . . . 13
2.2.4 User Mode Device Accesses . . . 14
2.2.5 Virtual Machine Guest Access to Devices. . . . 14
2.2.6 Virtualizing the IOMMU . . . 15

3 Architecture. . . . 16
3.1 Behavior . . . 16
3.1.1 Normal Operation . . . 16
3.1.2 Error Reporting . . . 17
3.1.2.1 I/O Page Faults . . . 17
3.1.2.2 Memory Access Errors . . . 17
3.2 Data Structures . . . 17
3.2.1 Updating Shared Tables. . . . 18
3.2.2 Device Table . . . 19
3.2.3 I/O Page Tables . . . 21
3.2.4 Sharing AMD64 CPU and IOMMU Page Tables . . . 24
3.3 Commands . . . 25
3.3.1 COMPLETION_WAIT . . . 27
3.3.2 INVALIDATE_DEVTAB_ENTRY . . . 27
3.3.3 INVALIDATE_IOMMU_PAGES . . . 28
3.3.4 INVALIDATE_IOTLB_PAGES. . . . 28
3.3.5 IOMMU Ordering Rules . . . 29
3.3.5.1 Invalidation Command Ordering Requirements . . . 29
3.3.5.2 Invalidation Commands Interaction Requirements . . . 29
3.4 Event Logging. . . . 30
3.4.1 ILLEGAL_DEV_TABLE_ENTRY . . . 32
3.4.2 IO_PAGE_FAULT . . . 33
3.4.3 DEV_TAB_HARDWARE_ERROR. . . . 33
3.4.4 PAGE_TAB_HARDWARE_ERROR. . . . 34
3.4.5 ILLEGAL_COMMAND_ERROR . . . 34
3.4.6 COMMAND_HARDWARE_ERROR . . . 35
3.4.7 IOTLB_INV_TIMEOUT. . . . 35
3.4.8 INVALID_DEVICE_REQUEST . . . 35
3.5 IOMMU Interrupt Support . . . 36
3.6 PCI Resources. . . . 36
3.6.1 IOMMU Capability Block Registers . . . 36
3.6.2 IOMMU Control Registers . . . 39

4 Implementation Considerations 45
4.1 Caching and Invalidation Strategies 45
4.2 Recommended IOMMU Topologies 46
4.3 RequesterID Mapping Capability Block Registers 48
4.3.1 RequesterID Mapping Registers 49
4.4 HyperTransport™ Specific Issues 50
4.5 Chipset Specific Implementation Issues 50
5 IOMMU Page Walker Pseudo Code 51

List of Figures

Figure 1: Example Platform Architecture 12
Figure 2: IOMMU Data Structures 18
Figure 3: DeviceID Derived from PCI Express™ RequesterID 19
Figure 4: DeviceID Derived from HyperTransport™ UnitID 19
Figure 5: Device Table Entry 20
Figure 6: I/O Page Table Entry Not Present (any level) 23
Figure 7: I/O Page Translation Entry (PTE) 23
Figure 8: I/O Page Directory Entry (PDE) 24
Figure 9: Address Translation Example 24
Figure 10: Sharing AMD64 and IOMMU Page Tables with Identical Addressing 25
Figure 11: Circular Command Buffer in System Memory 26
Figure 12: Generic Command Buffer Entry 26
Figure 13: COMPLETION_WAIT command format 27
Figure 14: INVALIDATE_DEVTAB_ENTRY Command Format 28
Figure 15: INVALIDATE_IOMMU_PAGES Command Encoding 28
Figure 16: INVALIDATE_IOTLB_PAGES 28
Figure 17: Circular Event Log in System Memory 30
Figure 18: Generic Event Log Buffer Entry 31
Figure 19: ILLEGAL_DEV_TABLE_ENTRY Event Log Buffer Entry 33
Figure 20: IO_PAGE_FAULT Event Log Buffer Entry 33
Figure 21: DEV_TAB_HARDWARE_ERROR Event Log Buffer Entry 34
Figure 22: PAGE_TAB_HARDWARE_ERROR Event Log Buffer Entry 34
Figure 23: ILLEGAL_COMMAND_ERROR 35
Figure 24: COMMAND_HARDWARE_ERROR Event Log Buffer Entry 35
Figure 25: IOTLB_INV_TIMEOUT Event Log Buffer Entry 35
Figure 26: INVALID_DEVICE_REQUEST Event Log Buffer Entry 36
Figure 27: IOMMU in a HyperTransportTM Tunnel 46
Figure 28: IOMMU in a PCIe™/PCI-X®-to-HyperTransportTM Bridge 47
Figure 29: Hybrid IOMMU 47
Figure 30: Chained Hybrid IOMMU in a Large System 48

List of Tables

Table 1: Bit Attribute Definitions 10
Table 2: Example Page Size Encodings 22
Table 3: Page Table Level Parameters 23
Table 4: Event Summary 32
Table 5: Type Field Encodings 34
Table 6: INVALID_DEVICE_REQUEST Type Field Encodings 36

Revision History

Date	Rev	Description
02/01/06	1.00	• Initial Public Release.

1 Overview

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

- **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.
- **Host**. The system software layer responsible for running guests.
- **IOMMU**. Refers to the I/O Memory Mapping Unit defined by this specification.
- **MMIO**. Read or write access to memory mapped resources provided by devices.
- **MMU**. Memory Mapping Unit.
- **Message Signalled Interrupt (MSI)**. An interrupt that is signalled by generating a posted write to a OS defined physical address.
- **Page Tables**. A table structure in main memory used to translate an address from one representation to an

alternate representation.
- **Protection Domain**. A set of address mappings and access rights that can be shared by multiple devices.
- **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.

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 (for write-once by firmware) with a cold reset.
Ignored Ign	**Ignored:** 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:** Reserved for future implementations. Bits must be implemented as read only zero.
Unused Un	**Unused:** Bit is not used by hardware. Software is allowed to use the bit for its own purposes.

Table 1: Bit Attribute Definitions

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

The IOMMU can be thought of as a combination and generalization of two facilities previously included in the AMD64 architecture: the Graphics Aperture Remapping Table (GART) and the Device Exclusion Vector (DEV). In older systems, the GART provided address translation of device accesses to a small range of the system physical address space, and the DEV provided a limited degree of 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 device in the system to be assigned to a specific domain and a distinct set of I/O page tables. When a 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 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. A 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 devices may have different usage models and working set sizes. IOTLB-capable devices contain private TLBs tailored for their own needs, creating a scalable distributed system of TLBs. The performance of IOTLB-capable 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 devices.
- A device table that allows devices to be assigned to specific domains and contains pointers to the devices' page tables.

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. However, compared to the processor's MMU, the IOMMU has a few limitations.

The first limitation is that the IOMMU provides no direct indication to a device of a failed translation.

The second limitation of the IOMMU is related to the multi-path 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, and between any two nodes there might be multiple routes. The IOMMU can only see and translate memory traffic that is routed through its node in the system fabric. In a large system with multiple paths to devices, multiple IOMMUs are required to ensure that all device accesses have appropriate protection and translation applied to them, and system software must to understand the system topology to correctly configure the IOMMUs.

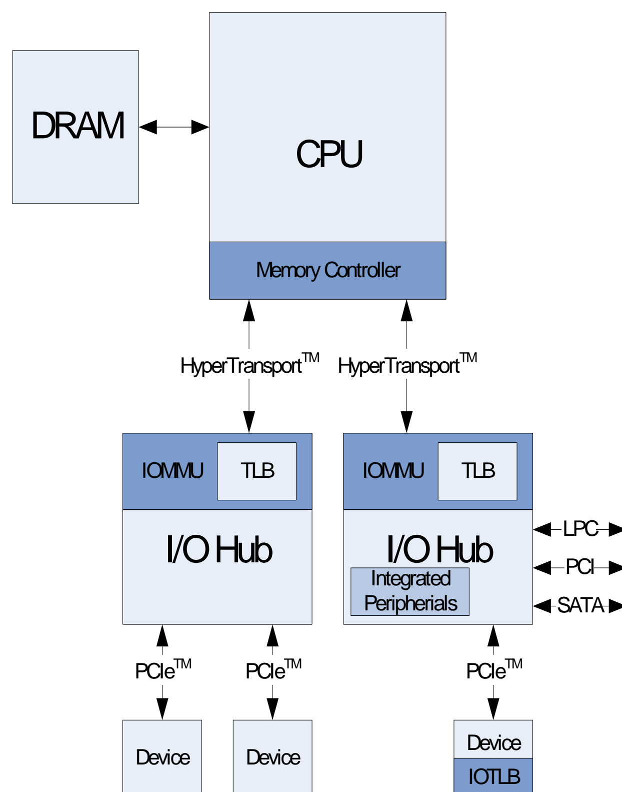


Figure 1: Example Platform Architecture

2.2 Usage Models

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 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 devices. However, with appropriate programming, a host OS can use the IOMMU as a replacement for 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 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.

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 the I/O page tables, and issue appropriate page-invalidate commands to both the processor and the IOMMU. Unlike the processor, the IOMMU requires 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.)

Eventually the host OS may have to tear down the mappings. The procedure is similar to setup:

- Clear the device table entry.
- Notify the IOMMU of the newly cleared table entry.
- 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 be 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 Replacing 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 devices to be classified into different domains. Associated with each domain is a bit vector, indexed by physical page address, indicating whether 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 yet another flavor of distinction between low memory and high memory, and perform appropriate bookkeeping to ensure that legacy 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 device to high memory, 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 devices like disk controllers and network interfaces, the performance cost of bounce buffer allocation and copying can be large.

In some recent systems, the GART has been used to work around this problem. When the OS wishes to perform a transfer between a legacy device and high memory, it allocates a portion of the GART aperture and maps those pages to high memory. It then commands the 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 frame allocation and de-allocation in the GART aperture; in the worst case, system performance is actually degraded due to serialization waiting for GART frames to become available.

The IOMMU allows a much better solution. First of all, IOMMU translation applies to the full range of addresses a 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 device to a different domain means that heavily used devices can be given their own sets of I/O page tables, and not have to contend with other devices for allocation and de-allocation of I/O frames.

2.2.4 User Mode Device Accesses

The IOMMU plays a crucial role in allowing arbitrary devices to be safely controlled by user-level processes, since 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 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 a device may be set up similar to GART emulation, with two differences: first, the mappable address range is the entire range of 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 device access can be a tricky proposition. Protecting and remapping DMA is only part of the problem; the other part is interrupt management, for which the IOMMU provides no help. A variety of strategies for user-level device interrupt management are possible, ranging from hybrid drivers with interrupt service routines in the kernel but all other code at user level, to interrupt-aware process scheduling together with careful management of the interrupt controller's mask bits. Such strategies are beyond the scope of this specification.

As was the case with GART emulation, system software will have to lock in memory all pages that might ever be accessed by a 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 devices. This is really just a special case of allowing user-level access to 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 a 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 numerous devices. Fortunately, in this case, all devices

given to the guest need to see the exactly the same I/O page translations. If all the devices belonging to a given VM guest are assigned to the same domain then the IOMMU can share translation cache entries between any of the guest's 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 allow 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 queue 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 device IDs as seen by the guest need not be the same as the real device IDs, and the domain IDs used by the guest will almost certainly not be the same as the domain IDs 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 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.

Due to the eagerness requirement, shadow I/O page table management is likely to be even more expensive than regular shadow CPU page table management. A future revision of this specification may introduce hardware support for nested I/O page tables, so that a VMM emulating the IOMMU could avoid constructing 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 read and write requests arriving from downstream devices (which may be HyperTransport™ or PCI based), performs permission checks and address translation on the requests, and sends translated versions upstream to system memory. Requests other than reads and writes are passed through unaltered. Read and write requests arriving from upstream (which include requests originated by CPUs as well as peer-to-peer traffic from devices) are similarly passed unaltered by the IOMMU. Request responses are passed unaltered in both directions.

The IOMMU consults a variety of tables in system memory to perform its permission checks and address translation. 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.

3.1.1 Normal Operation

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

- Transactions arriving from upstream must be passed downstream unaltered.
- Transactions arriving from downstream that are not memory or I/O reads or writes must be passed upstream unaltered.
- Memory read and write transactions (including peer-to-peer traffic) from downstream result in (potentially cached) table lookups in a device table (to classify the requesting device and 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 sends appropriately rewritten versions of the transactions upstream.
- I/O space read and write transactions from downstream devices result in a device table lookup to determine if the device is allowed to perform DMA I/O space transactions.
  - I/O space reads and writes from devices that are allowed to perform DMA I/O space transactions must be passed unaltered.
  - I/O space reads and writes from devices that are not allowed to perform DMA I/O space transactions must be master aborted.
- The IOMMU passes all completions unaltered.
- The IOMMU passes all pre-translated memory read and write requests from devices with IOTLBs unaltered.

Implementation Note: Message Signaled Interrupts (MSIs), which take the form of write requests to pre-programmed device virtual addresses, are subject to the same address translation process as any other write requests. The component that recognizes MSI addresses and turns MSIs into interrupt requests understood by

the CPU must be located upstream of the IOMMU. System software is responsible for ensuring that MSIs' translated addresses are, in fact, recognized as MSIs.

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. To ensure dead-lock free operation, page table and devices table reads originated by the IOMMU use an isochronous virtual channel, and may only reference addresses in system memory. The IOMMU signals interrupts using standard PCI INTx, MSI, or MSI-X interrupts.

3.1.2 Error Reporting

The IOMMU must detect and report several kinds of errors that may arise due to malfunctioning hardware or software. When the IOMMU detects an error of any kind, 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.

3.1.2.1 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, as described later in this chapter. 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 simply aborts the attempted operation:

- The IOMMU sends a master abort to the faulting request if the request was of a type that requires an acknowledgement (reads and non-posted writes. Non-posted writes are only supported on HyperTransport™)
- The IOMMU simply discards faulting posted writes, since there is no way for it to indicate any kind of failure to the device.

The IOMMU records I/O page faults in its event log when event logging is enabled.

3.1.2.2 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 errors in its event log when event logging is enabled.

3.2 Data Structures

The host software must maintain four types of in-memory data structures for use by the IOMMU. These data structures are:

- The *device table* is a table indexed by device ID. Each device table entry contains mode bits, a pointer to the I/O page tables, and a 16-bit domain ID. The domain ID 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 domain ID should be chosen by software to be the same for all devices that share the same page tables and must be different for devices that do not share page tables. (If software never shares page tables between devices, it can ensure unique domain IDs simply by making the domain ID equal to the device ID.)
- The *I/O page table(s)*: Each device may specify different I/O page tables, or different devices 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.
- The *command queue*: the IOMMU accepts commands queued by the CPU through a circular buffer located in system memory.
- 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.

Figure 2 illustrates the relationships among the IOMMU's data structures.

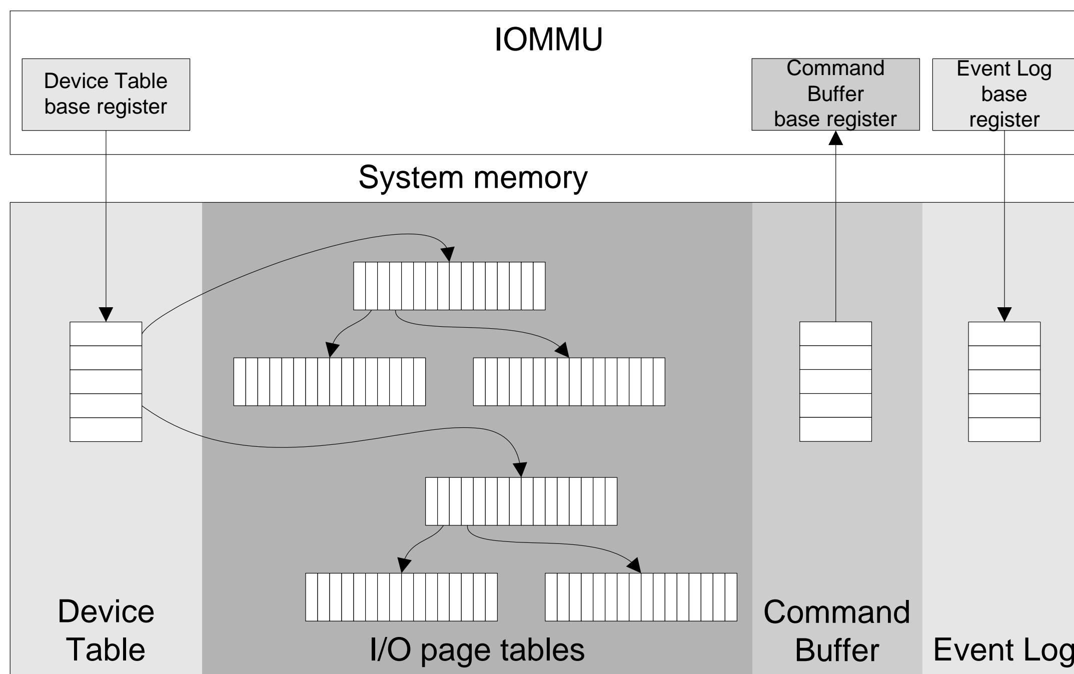


Figure 2: IOMMU Data Structures

3.2.1 Updating Shared Tables

Both of the shared table structures (device table and I/O page tables) have similar requirements for safe updates by system software.

Each table has a natural entry size that is the smallest portion of the table that must be written atomically. When updating a table entry, system software should always use store instructions whose data width is an integer multiple of the table's natural data size, aligned to an address that is a multiple of the table’s natural data size. If system software updating a table were to use narrower stores than the table's natural data size, the IOMMU could read a partially-updated entry that might cause unintended behavior.

Each table can also have its contents cached by the IOMMU or downstream IOTLBs. Therefore, after updating

a table entry, system software must send the IOMMU an appropriate invalidate command. (These invalidate commands also serve as hooks by which a VMM could intercept and virtualize a guest's attempts to control the IOMMU.) System software is not required to send an invalidate when upgrading a page table entry from not-present to present if NpCache=0 (IOMMU Capability Header [Capability Capability Offset 00h]).

3.2.2 Device Table

Devices that originate transactions are identified by a 16-bit device ID that is used to index the Device Table. The content of the device ID is fabric-dependent; for example, Figure 3 shows how PCIe™ and PCI-X® RequesterIDs are embedded in IOMMU device IDs, and Figure 4 shows how HyperTransport™ UnitIDs are embedded in deviceIDs.

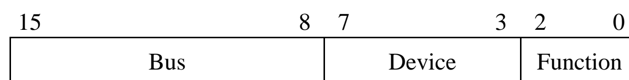
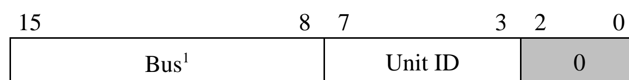


Figure 3: DeviceID Derived from PCI Express™ RequesterID



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

Figure 4: DeviceID Derived from HyperTransport™ UnitID

The device table is represented as an array of 128-bit entries located in contiguous system memory. Since there are 64K possible device IDs, the device table may be up to 1M byte in length. However, bus numbers are sequentially assigned starting at 0, and in all but the very largest systems there are only a few busses, so (at 4K bytes per bus) the device table can typically be substantially smaller than 1M bytes. The 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 boundary in system memory, and must be a multiple of 4K bytes in length.

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

Device table entries are 128 bits in length and should be updated using the following steps if 128 bit atomic writes are not supported:

- Clear the valid bit.
- Update the entry.
- Set the valid bit.

The IOMMU must read the entire device table entry in a single 128 bit transaction.

Device table entries take the following form:

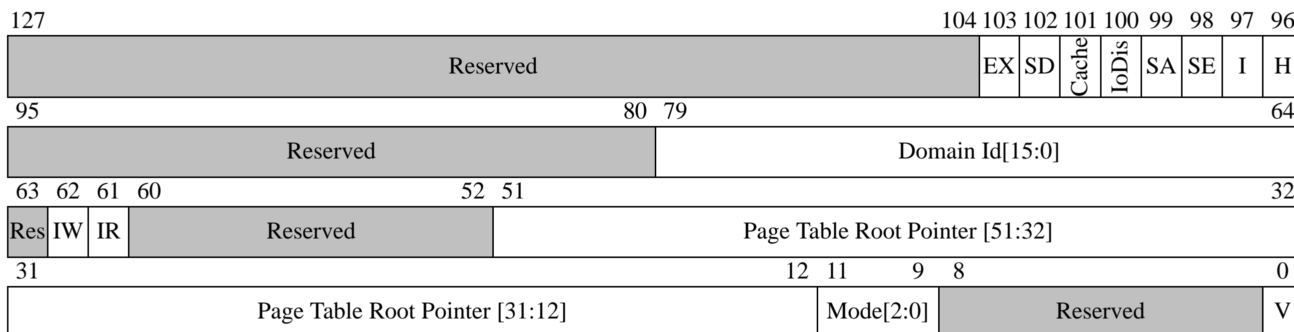


Figure 5: Device Table Entry

Fields of device table entries include:

Bits	Description
127:104	Reserved
103	**EX: allow exclusion**. 1=Accesses from this device that address the exclusion range are not translated.
102	**SD: snoop disable**. 1=IOMMU table walk transactions for this device are not snooped. HyperTransport™ based IOMMU's must not set the coherent bit in table walk requests for this device. 0=IOMMU table walk transactions for this device are snooped. HyperTransport™ based IOMMU's must set the coherent bit in table walk requests for this device.
101	**Cache: IOTLB cache hint**. 1=Caching of translations for explicit translation requests is not recommended. **Implementation Note**: This bit is a recommendation to not cache the final address associated with a translation request. This bit should not be used to prevent caching of page directory entries associated with the translation.
100	**IoDis: I/O disable**. Specifies whether DMA I/O space transactions are disallowed from the device. 1=DMA I/O is not allowed from the device an the IOMMU returns a master abort if a DMA I/O space transaction is received from the device. 0=DMA I/O space transactions are allowed from the device and the IOMMU must pass DMA I/O from the device unaltered.
99	**SA: suppress all page fault errors**. Specifies whether the IOMMU will suppress all page fault errors caused by a device. 1=The IOMMU must not update the event log with any page fault errors associated with the device.
98	**SE: suppress page fault errors**. Specifies whether the IOMMU will suppress errors caused by page faults if a page fault error has already been logged in the event log. 1=The IOMMU must only update the event log with a page fault error for the first page fault seen for the device as long as the Device ID remains in the device cache. The IOMMU will clear all state associated with this bit when an INVALIDATE_DEVTAB_ENTRY command is received for the device or when the Device ID is replaced in the cache by a different Device ID.
97	**I: IOTLB Support**. Indicates that a device has its own IOTLB. IOTLB support is an optional feature of the IOMMU. Devices with IOTLBs are capable of performing their own I/O page translation, and rely on the IOMMU only to perform page table lookups. When I=1 the IOMMU is enabled to process page walk requests from devices.

Bits	Description
96	**H: HyperTransport™ address translation disable**. Specifies whether device-initiated read and write transactions with certain reserved addresses (FD00000000h through FFFFFFFFFFh) with special meaning in HyperTransport™ are subject to regular I/O page translation (H=0) or passed through the IOMMU without I/O page translation or permission checking (H=1). Note that H=1 should only be used with trusted devices.
79:64	**Domain ID**. The domain ID is a 16-bit integer chosen by software that the IOMMU must use to tag its internal translation caches. Devices with different page tables must be given different domain IDs. Devices that share the same page tables may be given the same domain ID. Devices that share the same domain ID 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 H fields. (If devices with the same domain ID are erroneously given different non-zero modes or different page table root pointers, the behavior of the IOMMU is undefined.)
63	Reserved
62	**IW: I/O write permission**. 1=Device is allowed to perform DMA write transactions.
61	**IR: I/O read permission**. 1=Device is allowed to perform DMA read transactions.
51:12	**Page Table Root Pointer**. The page table root pointer is only used in modes where page translation is enabled; in those modes, it contains the system physical address of the root page table for the device.
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
0	**V: valid**. 1=Device table entry is valid.

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's page tables are a generalization of AMD64 long mode page tables: 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, where each table entry is either a directory entry (pointing to a lower-level 4K page table) or a translation entry (specifying a system physical page address). A translation entry is a page table entry with the Next Level field set to 0h. A directory entry is a page table entry with the Next Level field not equal to 0h. The IOMMU provides separate I/O read and write permission bits for devices.

The first generalization in the IOMMU's 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's 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 S bit. When the S bit is set, the size of the page is determined by the first zero bit in the page address, starting from the lowest bit of the default page size. 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.

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	Un	Un	Un	Un	Un	Un	Un	Un	4M	2M
3	Page Address	0	1	1	Un	Un	Un	Un	Un	Un	Un	Un	Un	Un	Un	Un	Un	Un	Un	Un	Un	4G	1G
* Address bits 63:32 can be used to encode page sizes greater that 4G.																							

Table 2: Example Page Size Encodings

Software Note: The page tables are required to have one entry for each default page size (see Table 3) to support the S=0 case. For the S=1 case 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 32 KB the 3 lsbs of the Page Table Level 1 virtual address will not be used for indexing and therefore the PTE will need to be repeated 8 times for each of the 64 unique PTEs given 4 KB page tables.

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.

Software Note: When invalidating pages in the translation cache, the size of the invalidate must be greater than or equal to the size of the largest page being invalidated.

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 a not-present 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 a not-present page fault. Otherwise the entry may be a directory entry (Next Level != 0) 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 translation entry (Next Level = 0) 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 (*i.e.* a directory entry in a level *N* table specifies a next level lower than *N-1*) and any of the skipped virtual address bits are non-zero, translation terminates with a not-present 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. 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 3 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 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, the Next Level field is greater than or equal to the containing table's level, or if a translation entry's physical address is not aligned to a multiple of the appropriate page size for the containing page table's level, translation terminates with a reserved-bits 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 CPUs 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 "Sharing AMD64 CPU and IOMMU Page Tables" on page 24.

The NS bit in the page tables is used to specify if DMA transactions that target the page can set the PCI defined No Snoop bit. The state of this bit is returned to a device with an IOTLB on an explicit translation request. This IOMMU does not use this bit.

The U bit in the page tables is an attribute bit passed to IOTLB devices in translation responses. This IOMMU does not use this bit for untranslated requests.

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}
2	29:21	2^{21}
1	20:12	4096

Table 3: Page Table Level Parameters

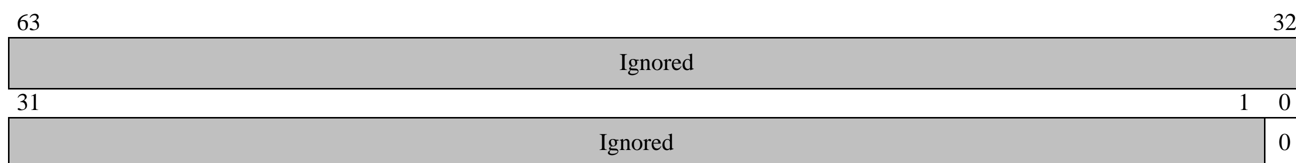


Figure 6: I/O Page Table Entry Not Present (any level)

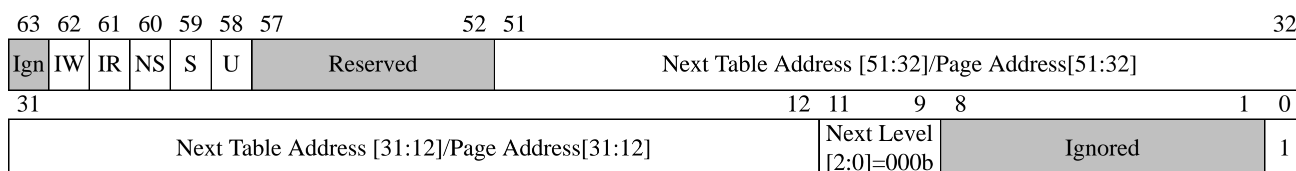


Figure 7: I/O Page Translation Entry (PTE)

63	62	61	60	59 58 ... 52	51 ... 32
Ign	IW	IR	NS	Reserved	Next Table Address [51:32]/Page Address[51:32]

31 ... 12	11 ... 9	8 ... 1	0
Next Table Address [31:12]/Page Address[31:12]	Next Level [2:0]!=000b	Ignored	1

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

Virtual Address

63 ... 58	57 ... 48	47 ... 39	38 ... 30	29 ... 21	20 ... 0
0000000b*	000000000b*	Level-4 Page Table Offset	000000000b*	Level-2 Page Table Offset	Physical-Page Offset

9 / 9 / 21

Level-4 Page Table: PDE 2h — 52

Level-2 Page Table: PTE 0h — 52

2 Mbyte Physical Page: Data Byte

Device Table Entry: 51 ... 12 Level 4 Page Table Address | 11 ... 9 4h

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

Figure 9: Address Translation Example

3.2.4 Sharing AMD64 CPU and IOMMU Page Tables

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 (which occupy bits unused by the CPU) 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 0xFFFF800000000000 to 0xFFFFFFFFFFFFFFFF will correspond to I/O virtual addresses in the range 0x800000000000 to 0xFFFFFFFFFFFF.

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 10, where 4 extra 4K 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.

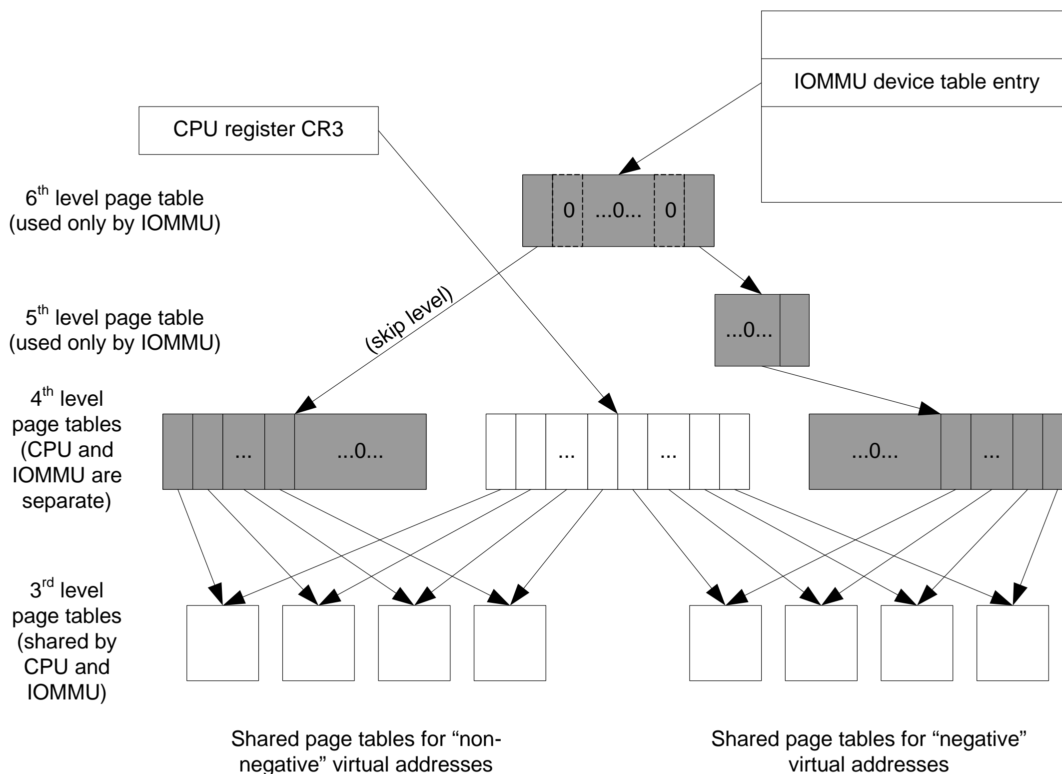


Figure 10: Sharing AMD64 and IOMMU Page Tables with Identical Addressing

3.3 Commands

The host CPU controls the IOMMU through a shared circular buffer in system memory. The CPU 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 execute them at its own pace. The shared command buffer organization was chosen to allow the CPU to send commands in batches to the IOMMU, while still allowing the IOMMU to set the pace at which commands are actually executed.

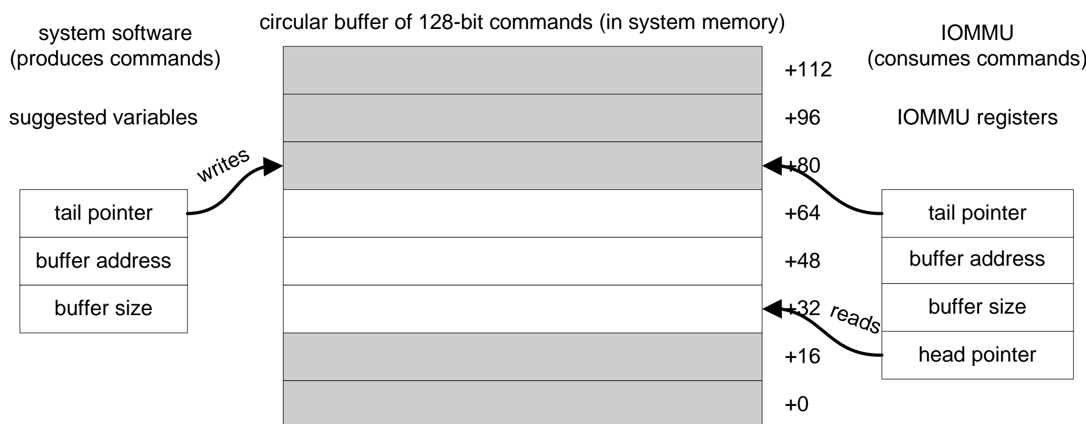


Figure 11: Circular Command Buffer in System Memory

The Command Buffer Base Address Register [MMIO Offset 0008h] is used to program the system physical 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 power of 2 (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 its size.

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], which points to the next command that the IOMMU will execute, and the Command Buffer Tail Pointer Register [MMIO Offset 2008h], which points to the last command written by software. These registers are located in 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 accepted 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:

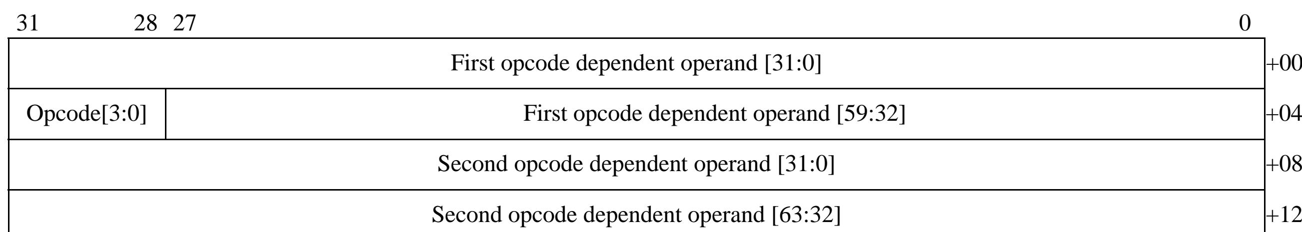


Figure 12: Generic Command Buffer Entry

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 have completely executed. In addition, if f=1, the IOMMU will not begin execution of any younger commands until COMPLETION_WAIT has finished.

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.

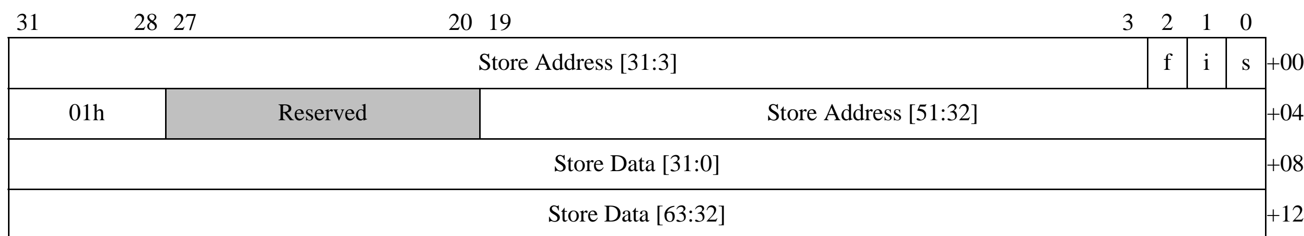


Figure 13: COMPLETION_WAIT command format

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 eagerly (as soon as it's invalidated) or lazily (when the device next attempts a DMA operation) or any time in between.

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 domain ID.

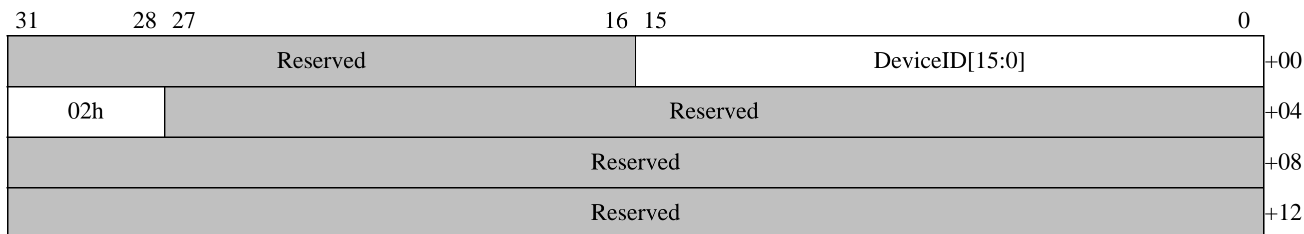


Figure 14: INVALIDATE_DEVTAB_ENTRY Command Format

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 domain ID. The size of the invalidate command is determined by the S bit, and the address. If S=0, size of the invalidate is 4Kbytes. 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]=7F_FFFF_FFFF_FFFFh and PDE=1, all pages associated with the Domain ID are invalidated. 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.

Software Note: When issuing INVALIDATE_IOMMU_PAGES commands, the size of the 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 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.

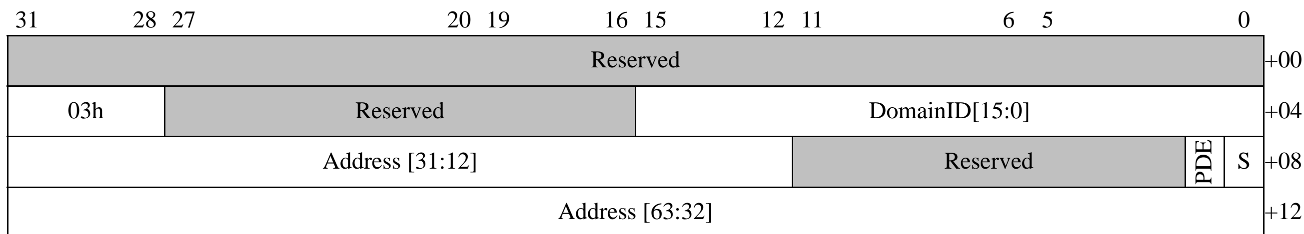


Figure 15: INVALIDATE_IOMMU_PAGES Command Encoding

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, size of the invalidate is 4Kbytes. If S=1, the size of the invalidate is determined by the first zero bit in the address starting from Address[12].

Figure 16: 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 IOMMU must ensure that all INVALIDATE_IOMMU_PAGES commands received prior to an INVALIDATE_IOTLB_PAGES command are completed before the IOMMU forwards the invalidation request to the IOTLB.

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 device ID. The appropriate value for Maxpend is device-dependent, and can be obtained from the device's IOTLB capability.

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

3.3.5 IOMMU Ordering Rules

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

3.3.5.1 Invalidation Command Ordering Requirements

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

- When an INVALIDATE_IOMMU_PAGES 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.

3.3.5.2 Invalidation Commands Interaction Requirements

Invalidation commands are considered completely executed 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.
- 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.
    - 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 IOMMU must ensure that both of these requirements are met prior to executing a subsequent

COMPLETION_WAIT command or updating the Command Buffer Head Pointer Register [MMIO Offset 2000h].

Software Note: Host software must ensure that ResPassPw bit is never set for reads to the IOMMU MMIO registers.

An invalidation command matches an outstanding translation if the command:
- Invalidates the device table entry for the 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 CPU 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 an overflow bit in the IOMMU Status Register [MMIO Offset 2020h]. The IOMMU can be configured to signal an interrupt whenever the event log is written. The CPU increments the IOMMU's head pointer to indicate to the IOMMU that it has consumed event log entries.

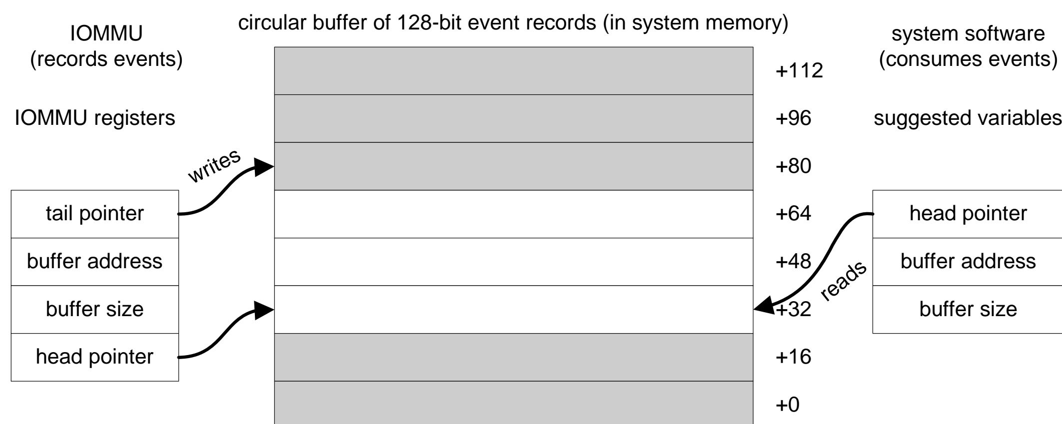


Figure 17: Circular Event Log in System Memory

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 power of 2 (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 its size.

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 last event 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.

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 device ID 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.

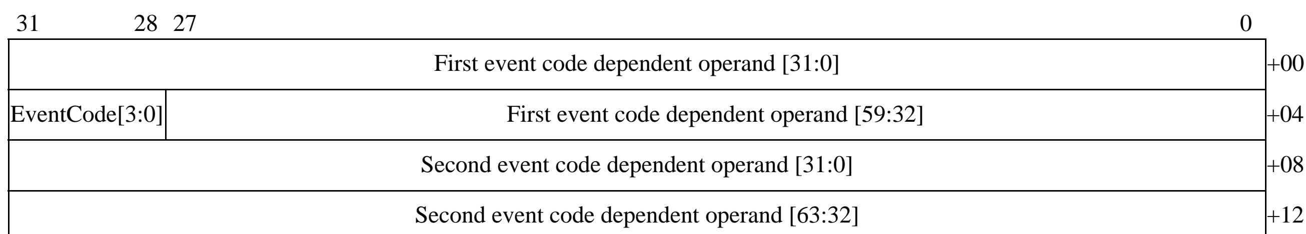


Figure 18: Generic Event Log Buffer Entry

Event Type	Error Type	IOMMU Response
INVALID_DEV_TABLE_ENTRY	Non-zero reserved bit in a device table entry	Target Abort transaction if the request is untranslated. Return completion without data if the request is translation request.
IO_PAGE_FAULT	Reserved paging mode in device table entry	Target Abort transaction if the request is untranslated. Return completion without data if the request is translation request.
	Page size encoding in a PTE that smaller than the default page size of the PTE	
	Page size encoding in a PTE that larger than the default page size of the PTE	
	Illegal level encoding in a page table entry	
	Non-zero reserved bit in a page table entry	
	Valid bit not set in page table entry[1]	
	Valid bit not set in device table entry[1]	
	Device ID not in the range specified by the device table size	
	Device attempts a write transaction to a write protected page (IW=0)[2]	Do not forward write.
	Device attempts a read transaction to a read protected page (IR=0)[2]	Target Abort transaction

1. An IO_PAGE_FAULT is not logged if this event occurs because of a translation request.
2. Translation requests forward that state of the IR/IW bits in the translation response and do not signal an error.

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	
IOTLB_INV_TIMEOUT	Invalidation response not received from IOTLB device	
INVALID_DEVICE_REQUEST	Pre-translated transaction received from a device with I=0	Target Abort transaction if the transaction is non-posted.
	Translation request from a device with I=0	
	Upstream I/O Space request from a device with IODis=1	

1. An IO_PAGE_FAULT is not logged if this event occurs because of a translation request.
2. Translation requests forward that state of the IR/IW bits in the translation response and do not signal an error.

Table 4: Event Summary

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 the event log with an ILLEGAL_DEV_TABLE_ENTRY event.

- The RW bit indicates if the transaction that caused the lookup was a read (RW=0) or a write (RW=1).
- The TR bit indicates if the transaction that caused the device table look-up was a translation request (TR=1).
- The Address field contains the device virtual address that the device was attempting to access. (The address of the malformed device table entry can be determined using the DeviceID field.)

Figure 19: ILLEGAL_DEV_TABLE_ENTRY Event Log Buffer Entry

3.4.2 IO_PAGE_FAULT

When the IOMMU performs a lookup in the page tables for a device and finds an invalid page table format or a page not present, the IOMMU writes the event log with an IO_PAGE_FAULT event.

- The PR bit indicates if the page fault was caused by a page marked as not present (PR=0).
- The RW bit indicates if the transaction that caused the page fault was a read (RW=0) or a write (RW=1).
- The PE bit indicates if the page fault was caused by the device not having permission to perform the transaction (PE=1).
- The RZ bit indicates if the page fault was caused by a reserved bit in the entry being set (RZ=1) or by an illegal level encoding.
- The RW, PE, and RZ bits are only meaningful if PR=1 or IT=1.
- The Address field contains the device virtual address that the device was attempting to access.

IO page faults detected for translation requests return a translation not present response to the device and are not logged in the event log.

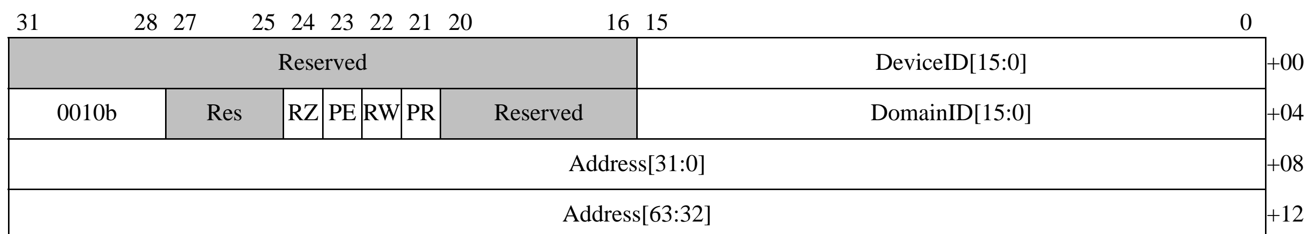


Figure 20: IO_PAGE_FAULT Event Log Buffer Entry

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.

- The Type field indicates the type of hardware error that occurred.

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

Table 5: Type Field Encodings

- The RW bit indicates that the transaction that caused the lookup was a read (RW=0) or a write (RW=1).
- The TR bit indicates that the transaction that caused the page fault was a translation request (TR=1).

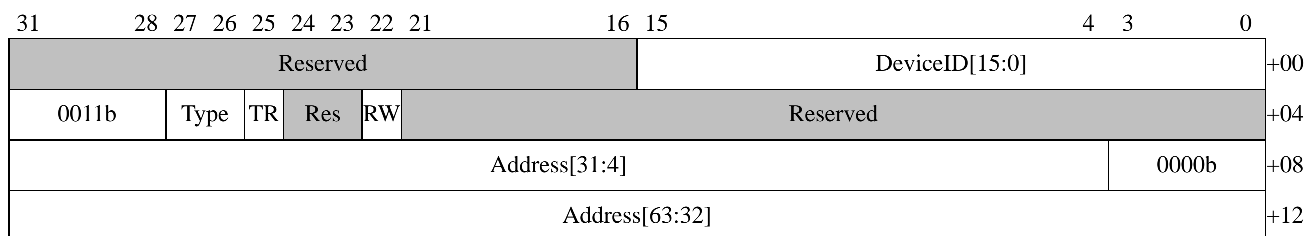


Figure 21: DEV_TAB_HARDWARE_ERROR Event Log Buffer Entry

3.4.4 PAGE_TAB_HARDWARE_ERROR

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

- In this case the Address field does *not* contain the device virtual address that the device attempted to access, but instead contains the system physical address of the failed page table access.
- The RW bit indicates that the transaction that caused the lookup was a read (RW=0) or a write (RW=1).
- The TR bit indicates that the transaction that caused the hardware error was a translation request (TR=1).
- The Type field encodings are defined in Table 5.

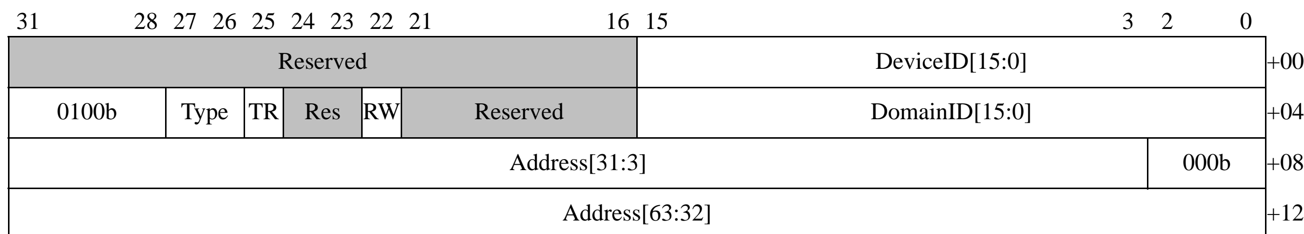


Figure 22: PAGE_TAB_HARDWARE_ERROR Event Log Buffer Entry

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.

The Address field contains the system physical address of the illegal command.

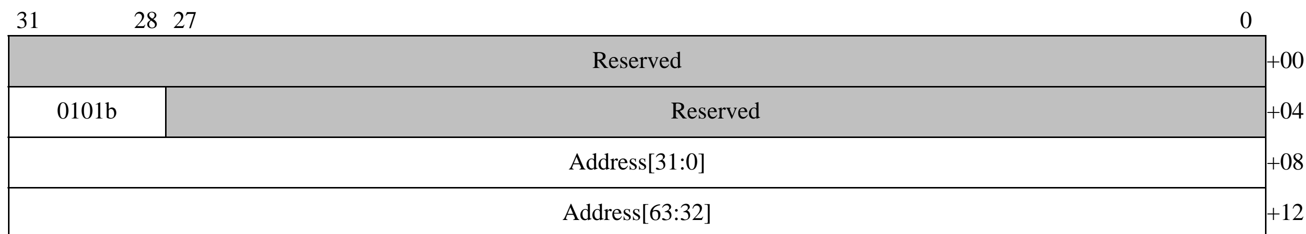


Figure 23: ILLEGAL_COMMAND_ERROR

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.

The Address field contains the system physical address that the IOMMU attempted to access.

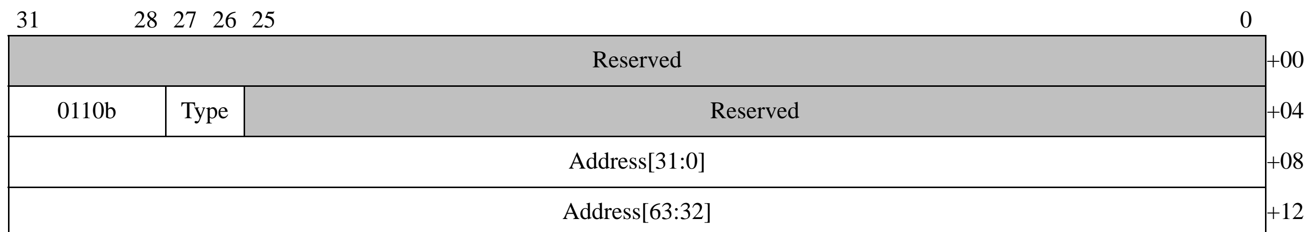


Figure 24: COMMAND_HARDWARE_ERROR Event Log Buffer Entry

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.

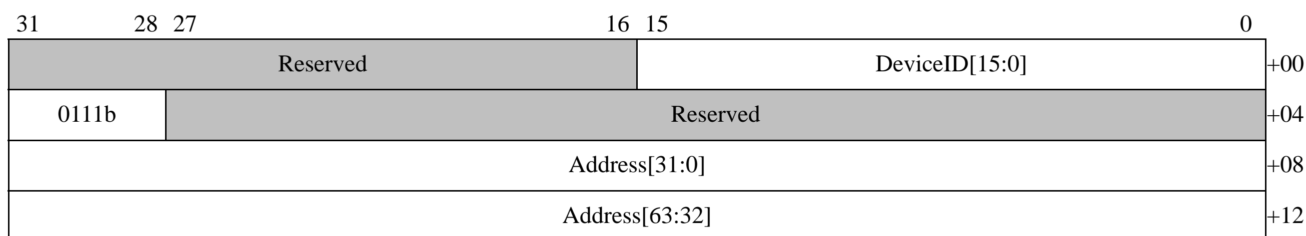


Figure 25: IOTLB_INV_TIMEOUT Event Log Buffer Entry

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.

- The Type field indicates the type of hardware error occurred.

Type	Description
00b	Reserved
01b	Translation request received from a device that has I=0 in the devices' device table.
10b	Pre-translated request received from a device that has I=0 in the devices' device table.
11b	Upstream I/O space request received from a device that has IODis=1 in the devices' device table.

Table 6: INVALID_DEVICE_REQUEST Type Field Encodings

Figure 26: INVALID_DEVICE_REQUEST Event Log Buffer Entry

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 IOMMU must not set the PassPW bit when sending interrupts associated with the IOMMU over HyperTransport™.

The IOMMU supports generation of interrupts when the event log is updated and when a completion wait command completes.

3.6 PCI Resources

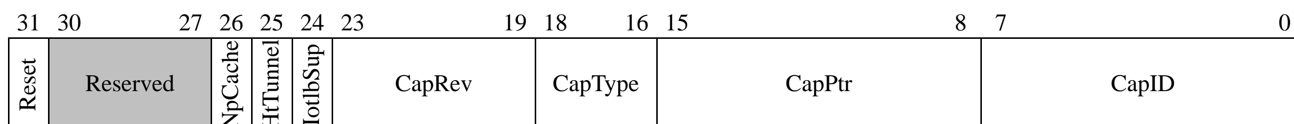
Rather than identifying itself as a separate device, the IOMMU is a capability that can be present on any HyperTransport™ device, including HyperTransport™ bridges and tunnels as well as PCIe root complexes and PCI-X host bridges. Configuration and status information for the IOMMU are mapped into PCI configuration space using a PCI capability block.

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	**Reset: soft reset**. RW1S. Reset 0b. Writing a one to this bit causes all IOMMU internal state to be reset (including clearing all caches in the IOMMU) and all IOMMU MMIO registers to be reset to their default state. After the reset completes, this bit is cleared by hardware.
30:27	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 before changing a page from not present to present.
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.

31 … 12	11 … 0
BaseAddress[31:12]	Reserved

Bits	Description
31:12	**BaseAddress[31:12]**. RW. Reset 0000_0000h. Specifies the lower 32 bits of the 4Kbyte aligned base address of the IOMMU control registers.
11:0	Reserved.

Capability Offset 08h IOMMU Base Address High Register

This register specifies the upper 32 bits of the base address of the IOMMU control registers.

31 … 0
BaseAddress[63:32]

Bits	Description
31:0	**BaseAddress[63:32]**. RW. Reset 0000_0000h. Specifies the upper 32 bits of the 4Kbyte 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. 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.

31 – 24	23 – 16	15 – 0
LastDevice	FirstDevice	Reserved

Bits	Description
31:24	**LastDevice: last device**. RO. Reset Xb. Indicates device and function number of the last integrated device associated with the IOMMU.
23:16	**FirstDevice: first device**. RO. Reset Xb. Indicates device and function number of the first integrated device associated with the IOMMU.
15:0	Reserved.

Capability Offset 10h IOMMU MSI Message Number Register

This register returns the message number for MSI or MSI-X interrupts associated with the IOMMU.

31 – 5	4 – 0
Reserved	MsiNum

Bits	Description
31:0	**MsiNum: MSI message number**. RO. Reset XXXXXb. This register must indicate which MSI/MSI-X vector is used for the interrupt message generated by the IOMMU. For MSI, the value in this register indicates the offset between the base Message Data and the interrupt message that is generated. Hardware is required to update this field so that it is correct if the number of MSI Messages assigned to the device changes when software writes to the Multiple Message Enable field in the MSI Message Control register. For MSI-X, the value in this register indicates which MSI-X Table entry is used to generate the interrupt message. The entry must be one of the first 32 entries even if the function implements more than 32 entries. For a given MSI-X implementation, the entry must remain constant. If both MSI and MSI-X are implemented, they are permitted to use different vectors, though software is permitted to enable only one mechanism at a time. If MSI-X is enabled, the value in this register must indicate the vector for MSI-X. If MSI is enabled or neither is enabled, the value in this register must indicate the vector for MSI. If software enables both MSI and MSI-X at the same time, the value in this register is undefined.

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.

MMIO Offset 0000h Device Table Base Address Register

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

63 … 52	51 … 32
Reserved	DevTabBase

31 … 12	11 … 8	7 … 0
DevTabBase	Reserved	Size[7:0]

Bits	Description
63:52	Reserved.
51:12	**DevTabBase: device table base address**. RW. Reset 00_0000_0000h. Specifies the 4 Kbyte aligned base address of the first level device table.
11:7	Reserved.
6:0	**Size: size of the device table**. 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 device table, and a maximum size of 7Fh corresponds to a 1Mbyte device table.

MMIO Offset 0008h Command Buffer Base Address Register

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

63 … 60	59 … 56	55 … 52	51 … 32
Reserved	ComLen	Reserved	ComBase

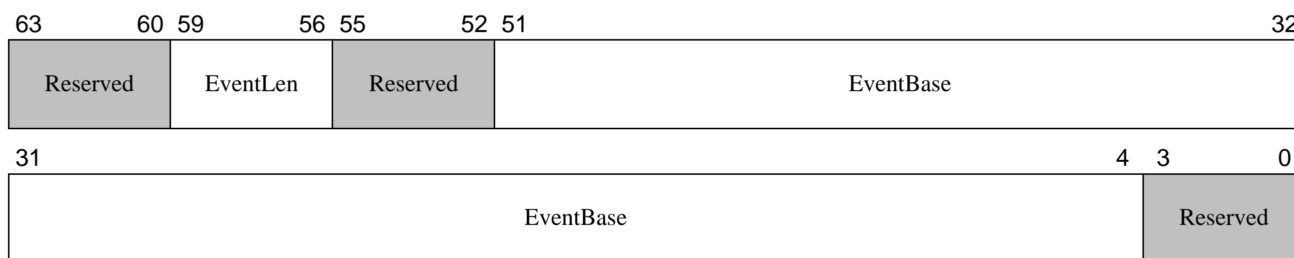
31 … 4	3 … 0
ComBase	Reserved

Bits	Description
63:60	Reserved.
59:56	**ComLen: command buffer length**. RW. Reset 0000b. Specifies the length of the command buffer in power of 2 increments. 0000b = 1 entry 0001b = 2 entries 0010b = 4 entries 0011b = 8 entries ... 1111b = 32768 entries

55:52	Reserved
51:4	**ComBase: command buffer base address**. RW. Reset 0000_0000_0000h. Specifies the base address of the command buffer. The base address programmed must be aligned to the length programmed in the ComLen field.
3:0	Reserved

MMIO Offset 0010h Event Log Base Address Register

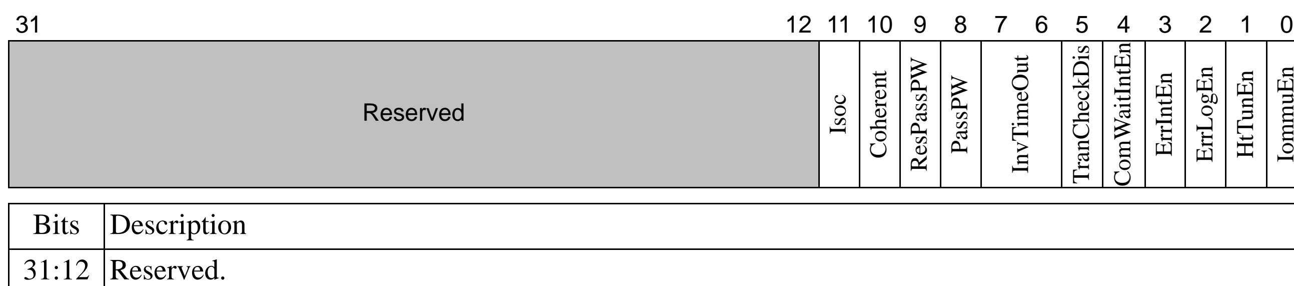
This register specifies the system physical address and length of the event log.



Bits	Description
63:60	Reserved.
59:56	**EventLen: event log length**. RW. Reset 0000b. Specifies the length of the event log in power of 2 increments. 0000b = 1 entry 0001b = 2 entries 0010b = 4 entries 0011b = 8 entries ... 1111b = 32768 entries
55:52	Reserved
51:4	**EventBase: event log base address**. RW. Reset 0000_0000_0000h. Specifies the base address of the event log. The base address programmed must be aligned to the length programmed in the EventLen field.
3:0	Reserved

MMIO Offset 0018h IOMMU Control Register

This register controls the behavior of the IOMMU.

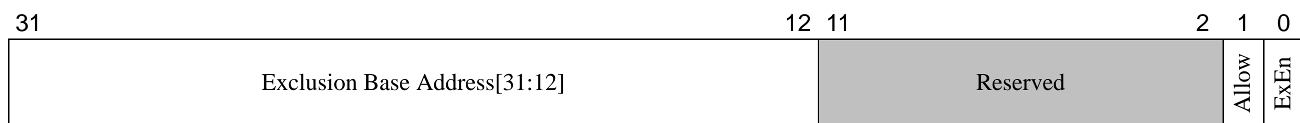


Bits	Description
31:12	Reserved.

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 HyperTransport™.
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 HyperTransport™. 1=Device table requests are not snooped by the processor. 1=Device table requests are snooped by the processor.
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 HyperTransport™.
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 HyperTransport™.
7:6	**InvTimeOut: HyperTransport™ invalidation time-out**.RW. Reset 00b.Ths field specifies the invalidation time-out for IOTLB invalidation requests that are sent on HyperTransport™. Invalidation requests sent on PCIe use the PCIe defined completion time-out. 00b=No time-out 10b=100 us 01b=10 us 11b=1 ms
5	**TranCheckDis: translated transaction check disable**. RW. Reset 0b. This bit disables checking the I bit in the device table entry for pre-translated transactions. 1=The state of the I bit is not checked for pre-translated transactions. 0=The state of the I bit is checked for pre-translated transactions. If I=0, pre-translated transactions are not forwarded, a target abort is returned if the transaction was not a posted write, and an error is logged in the event log.
4	**ComWaitIntEn: completion wait interrupt enable**. RW. Reset 0b. 1=An interrupt is signalled when the ComWaitInt bit is set in the IOMMU Status Register [MMIO Offset 2020h].
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. 0=Event logging is not enabled.
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 if IommuEn=0.
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 by the IOMMU.

MMIO Offset 0020h IOMMU Exclusion Base Register

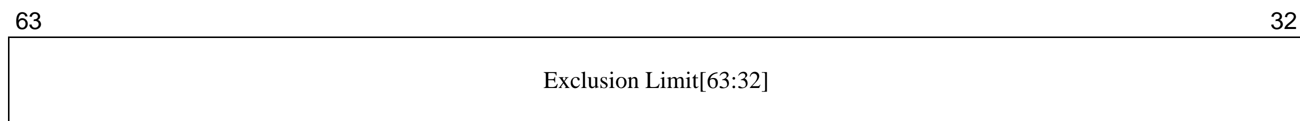
This register specifies the base device virtual address of the IOMMU exclusion range. Transactions that target addresses in the exclusion range are not translated if the EX bit in the device table is set for the device or if the Allow bit is set in this register.

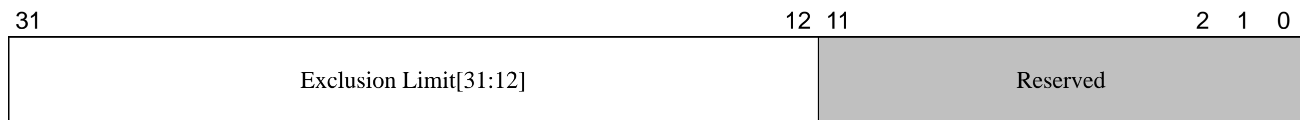


Bits	Description
63:12	**Exclusion range base address**. RW. Reset 0000_000_0000h. Specifies the 4Kbyte aligned base address of the exclusion range.
11:2	Reserved.
1	**Allow: allow all devices**. RW. Reset 0b. 1=All accesses to the exclusion range are untranslated. 0=The EX bit in the device table entry 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 always FFFh.





Bits	Description
63:12	**Exclusion range limit**. RW. Reset 0000_000_0000h. Specifies the 4Kbyte 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
63:20	Reserved.
19:4	**CmdHeadPtr: command buffer head pointer**. RO. Reset 000h. Specifies the 128-bit aligned offset from the command buffer base address register of the next command to be executed by the IOMMU. The IOMMU increments this register completing execution of the command in the command buffer.
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.

63–32
Reserved

31–20	19–4	3–0
Reserved	CmdTailPtr	Reserved

Bits	Description
63:16	Reserved.
19:4	**CmdTailPtr: command buffer tail pointer**. RW. Reset 000h. 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 after writing a commands to the command buffer.
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.

63–32
Reserved

31–20	19–4	3–0
Reserved	EventHeadPtr	Reserved

Bits	Description
63:20	Reserved.
19:4	**EventHeadPtr: event log head pointer**. RW. Reset 000h. 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 after reading an event from the event log.
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.

63–32
Reserved

31–20	19–4	3–0
Reserved	EventTailPtr	Reserved

Bits	Description
63:16	Reserved.
19:4	**EventTailPtr: event log tail pointer**. RO. Reset 000h. 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 after writing an event to the event log.
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.

31–3	2	1	0
Reserved	ComWaitInt	EventLogInt	EventOverflow

Bits	Description
31:1	Reserved.
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 this bit is set if ComWaitIntEn=1 (IOMMU Control Register [MMIO Offset 0018h]).
1	**EventLogInt: event log interrupt**. RW1C. Reset 0b. 1=Error entry written to the event log by the IOMMU. An interrupt is generated when this bit is set if EventIntEn=1 (IOMMU Control Register [MMIO Offset 0018h]).
0	**EventOverflow: event log overflow**. RW1C. Reset 0b. 1=IOMMU event log overflow has occurred. An interrupt is generated when this bit is set if EventIntEn=1 (IOMMU Control Register [MMIO Offset 0018h]).

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 will 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's 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 device ID 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 this case, 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 domain ID that can be shared for multiple device IDs: since the IOMMU uses translation cache entries tagged by *{domain ID, 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 *device ID* to {*domain ID, I/O page table base address}*. Most systems have only a few distinct device IDs, so the capacity of the first stage can be small. The one complication is that device IDs are not very random and tend to be clustered, so, to avoid conflicts, this stage should either be highly associative or use a good device ID hash function.
- The second stage should map *{domain ID, device virtual address*} to *{system physical address, protection}*. This stage should have (at least) hundreds of entries. This stage should explicitly include the domain ID in set index hashing (rather than just using the domain ID 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 *{domain ID, 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 device IDs 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.

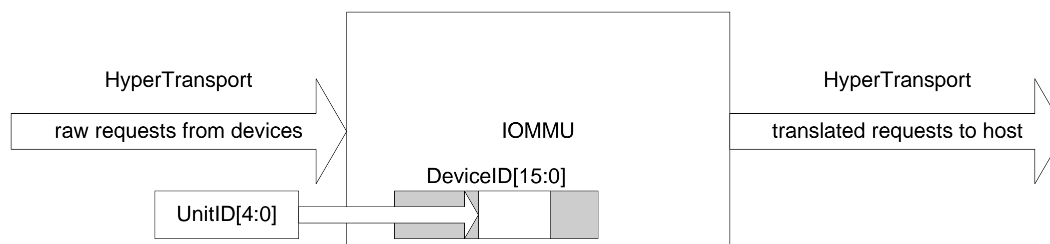


Figure 27: IOMMU in a HyperTransport™ Tunnel

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 HyperTransport™ offers only 5 bits worth 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/PCI-X devices.

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

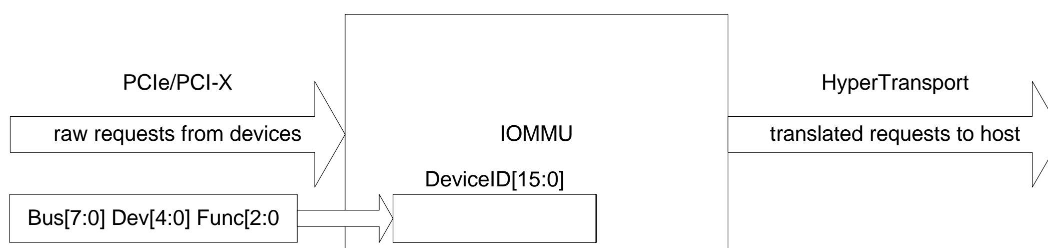


Figure 28: IOMMU in a PCIe™/PCI-X®-to-HyperTransport™ Bridge

Since most future commodity devices will be PCIe-based, 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:

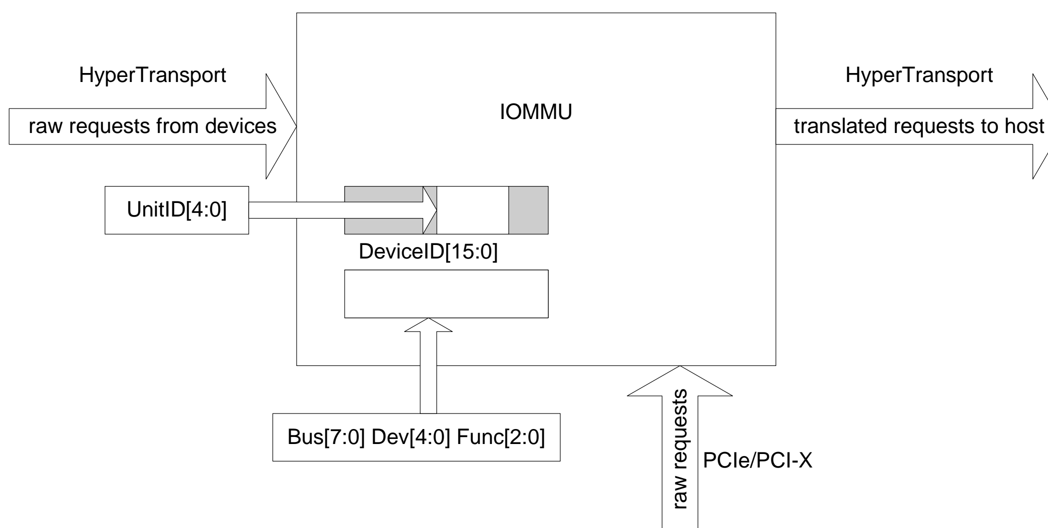


Figure 29: Hybrid IOMMU

Hybrid IOMMUs can be chained together to build large systems:

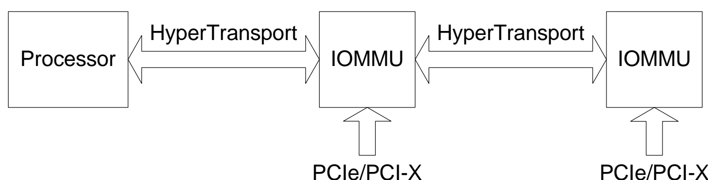


Figure 30: Chained Hybrid IOMMU in a Large System

4.3 RequesterID Mapping Capability Block Registers

A new PCI capability block indicates the presence of the RequesterId mapping capability and the location of RequesterId mapping registers. This capability will be used primarily in systems that have multiple PCIe root complexes or PCI-X host bridges on the same HyperTransport™ chain, but only have one IOMMU on the bus similar to Figure 30.

Capability Offset 00h RequesterID Mapping Capability Header

This register indicates that this is an RequesterID mapping capability block.

31 … 24	23 … 20	19	18 … 16	15 … 8	7 … 0
NMaps	CapRev	MapEn	CapType	CapPtr	CapID

Bits	Description
31:24	**NMaps: number of maps**. RO. Reset XXh. These bits specify the number of RequesterID map registers implemented.
23:20	**CapRev: capability revision**. RO. Reset 00001b. Specifies the RequesterID mapping capability revision.
19	**MapEn: RequesterID mapping enable**. RW. Reset 0b. 1=Mapping of RequesterIDs to UnitIDs enabled.
18:16	**CapType: RequesterID mapping capability block type**. RO. Reset 110b. Specifies the layout of the Capability Block as a RequesterID mapping capability block.
15:8	**CapPtr: capability pointer**. RO. Reset XXh. Indicates the location of the next capability block if one ia present.
7:0	**CapId: capability ID**. RO. Reset 0Fh. Indicates a Secure Device capability block.

Capability Offset 04h RequesterID Mapping Base Offset Register

This register specifies the PCI BAR in which the RequesterID mapping registers are located and the offset within the PCI BAR where the RequesterID mapping registers begin.

31 … 12	11 … 3	2 … 0
BaseOffset	Reserved	BaseReg

Bits	Description
31:12	**BaseOffset**. RO. Reset X_XXXXh. Indicates the 4Kbyte offset within the BAR where the RequesterID mapping registers start.
11:3	Reserved.
2:0	**BaseReg**. RO. Reset XXXb. Indicates the BAR into which the RequesterID mapping registers are mapped.

Capability Offset 0Ch RequesterID Mapping Range Register

This register indicates the device and function numbers of the first and last devices associated with the RequesterID mapping capability. All port devices that have device and function numbers between the first and last device numbers inclusive are supported by RequesterID mapping capability. All devices associated with the RequesterID mapping capability must be located on the same logical bus.This capability must be implemented in the same device and function as that contains the HyperTransport™ capability.

31 – 24	23 – 16	15 – 5	4 – 0
LastDevice	FirstDevice	Reserved	NumIds

Bits	Description
31:24	**LastDevice: last device**. RO. Reset Xb. Indicates device and function number of the last integrated device associated with the RequesterID mapping capability.
23:16	**FirstDevice: first device**. RO. Reset Xb. Indicates device and function number of the first integrated device associated with the RequesterID mapping capability.
15:5	Reserved.
4:0	**NumIds: number of mapping IDs**.RO. Reset XXXXXb. This field indicates the number of HyperTransport™ UnitIDs that the device reserves for use as mapping IDs. The specific UnitIDs reserved for use as mapping IDs must always be the last UnitIDs in the pool of UnitIDs reserved using the Unit Count field in the HyperTransport™ Slave Command CSR. The first UnitID reserved as a mapping ID can be determined using the following formula: Mapping ID 1 = Base UnitID + Unit Count - Number of mapping IDs

4.3.1 RequesterID Mapping Registers

The RequesterID mapping registers are mapped using the PCI BAR specified in the RequesterID mapping capability block. The starting address of the RequesterID mapping registers can be determined by adding the BaseOffset field in the RequesterID Mapping Base Offset Register [Capability Offset 04h] to the base address programmed in the PCI BAR specified by the BaseReg field. The number of mapping registers is defined by the NMaps field in the RequesterID Mapping Capability Header [Capability Offset 00h].

MMIO Offset 0000h+N RequesterID Map Register N

31 – 16	15 – 13	12 – 8	7 – 1	0
RequesterID	Reserved	UnitID	Reserved	Valid

Bits	Description
31:16	**RequesterID: PCI RequesterID**. RW. Reset 0000h. These bits specify the PCI RequesterID that is to be translated.
15:13	Reserved.
12:8	**UnitID: HyperTransport™ UnitId**. RW. Reset 00000b. This field specifies the UnitID of the transaction forwarded to the HyperTransport™ link. This field must be programmed with one of the mapping IDs.
7:1	Reserved.
0	**Valid: translation valid**. RW. Reset 0b. This bit indicates that a valid RequesterID and UnitID have been programmed to the map register. When this bit is set, the root complex uses the UnitID field specified when issuing transactions from the RequesterID specified.

4.4 HyperTransport™ Specific Issues

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

HyperTransport™ 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 HyperTransport™ 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™ read request issued by the IOMMU.

4.5 Chipset Specific Implementation Issues

Chipsets that implement both an IOMMU and a legacy PCI or AGP bridge must provide source identification to identify DMA traffic 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.

5 IOMMU Page Walker Pseudo Code

```
//
// Page table walker for IOMMU.
//
// Inputs: {dte} is device table entry, {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:
//    [61]       (cumulative) I/O write permission
//    [60]       (cumulative) I/O read permission
//    [51:12]    system physical page address
//    [2:0]      level of PTE (so caller knows how many VA bits to append)
// The caller of this routine is responsible for read and write permission
// checks. 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) ((0x1000 << (LEVEL * 9)) - 1)

uint64
iopagewalk(uint64 dte, uint64 dva)
{
   uint64 pdte = dte;
   uint64 ioperm = pdte & 0x6000000000000000;
   uint64 pa = pdte & 0xFFFFFFFFFF000;          // dte bits [51:12]
   uint oldlevel = 7, level = (pdte >> 9) & 7;

   if (level == 7)
      raise DEVTAB_RESERVED_LEVEL;

   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;
      if ((pdte & 0x1FF0000000000000) != 0)
         raise PDTE_RESERVED_BITS;
      ioperm &= pdte;
      pa = pdte & 0xFFFFFFFFFF000;          // pte bits [51:12]
      oldlevel = level;
      level = (pdte >> 9) & 7;
      if (level >= oldlevel)
         raise PDTE_RESERVED_BITS;
   }

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

```
    return ioperm | pa | oldlevel;
}
```