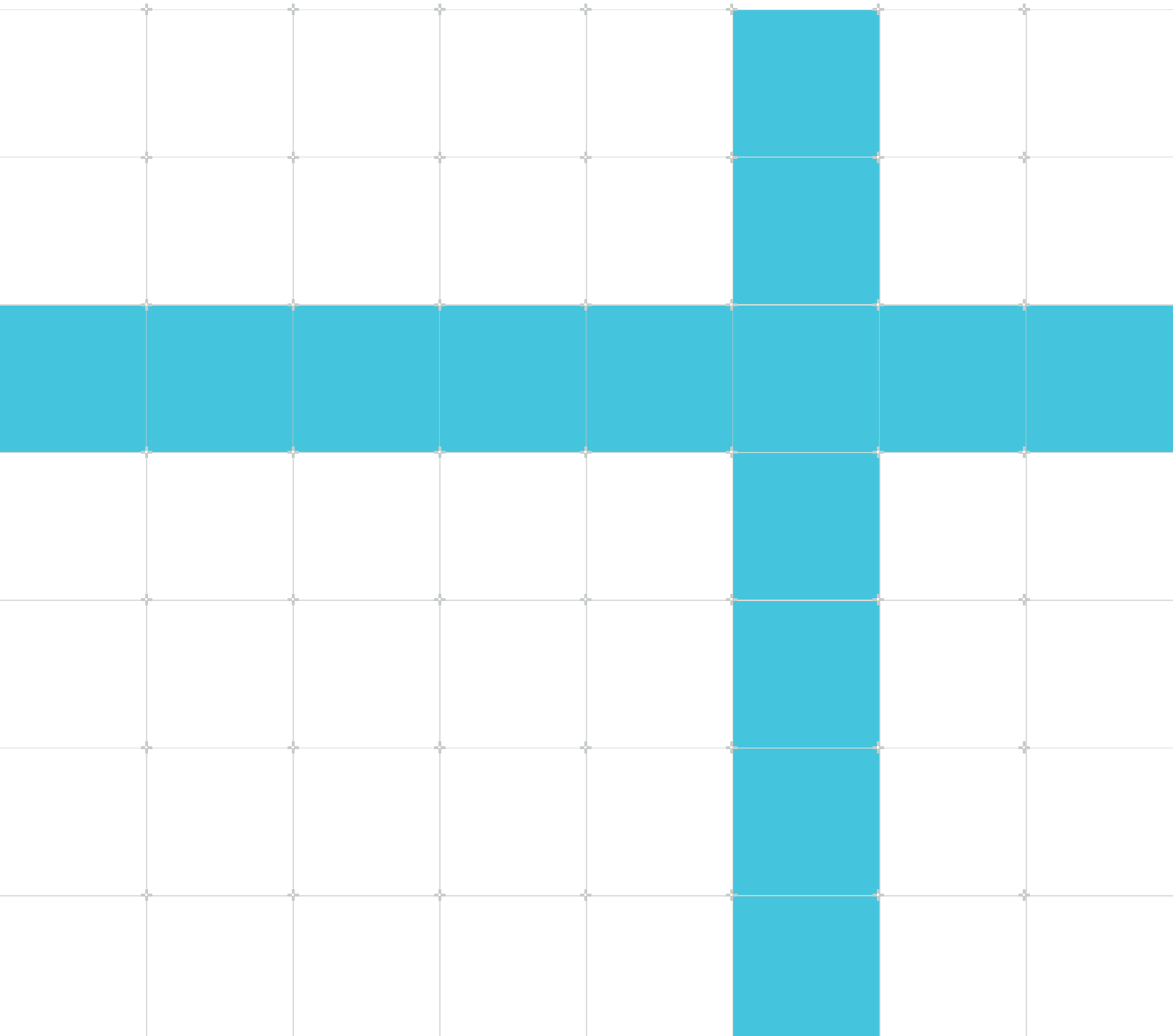
arm

Learn the Architecture - SMMU Software Guide

Version 1.0

Issue 01
109242_0100_01_en



Learn the Architecture - SMMU Software Guide

Release information

Document history

Issue	Date	Confidentiality	Change
0100-01	4 September 2023	Non-Confidential	Initial release

Proprietary Notice

This document is protected by copyright and other related rights and the practice or implementation of the information contained in this document may be protected by one or more patents or pending patent applications. No part of this document may be reproduced in any form by any means without the express prior written permission of Arm. No license, express or implied, by estoppel or otherwise to any intellectual property rights is granted by this document unless specifically stated.

Your access to the information in this document is conditional upon your acceptance that you will not use or permit others to use the information for the purposes of determining whether implementations infringe any third party patents.

THIS DOCUMENT IS PROVIDED "AS IS". ARM PROVIDES NO REPRESENTATIONS AND NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, SATISFACTORY QUALITY, NON-INFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO THE DOCUMENT. For the avoidance of doubt, Arm makes no representation with respect to, and has undertaken no analysis to identify or understand the scope and content of, patents, copyrights, trade secrets, or other rights.

This document may include technical inaccuracies or typographical errors.

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

This document consists solely of commercial items. You shall be responsible for ensuring that any use, duplication or disclosure of this document complies fully with any relevant export laws and regulations to assure that this document or any portion thereof is not exported, directly

or indirectly, in violation of such export laws. Use of the word “partner” in reference to Arm’s customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of the Agreement shall prevail.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349|version 21.0)

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Unrestricted Access is an Arm internal classification.

Product Status

The information in this document is Final, that is for a developed product.

Feedback

Arm welcomes feedback on this product and its documentation. To provide feedback on the product, create a ticket on https://support.developer.arm.com

To provide feedback on the document, fill the following survey: https://developer.arm.com/documentation-feedback-survey.

Inclusive language commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used language that can be offensive. Arm strives to lead the industry and create change.

We believe that this document contains no offensive language. To report offensive language in this document, email terms@arm.com.

Contents

1. Overview

This guide describes the basic operation of the Arm System Memory Management Unit version 3 (SMMUv3) and use cases of the SMMUv3. It includes:

- The SMMU architecture concepts, terminology and operation
- The system-level consideration relevant to what the SMMU does
- Knowledge of typical SMMU use cases

1.1 Before you begin

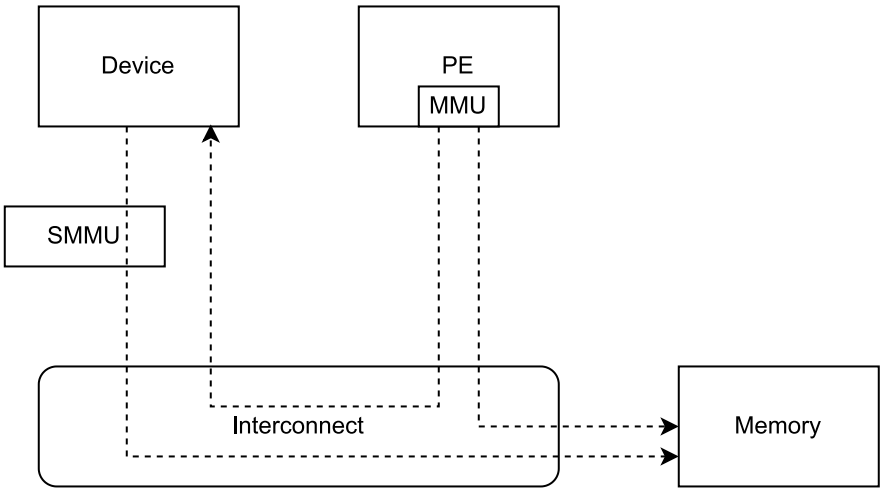
This guide complements the Arm System Memory Management Unit Architecture Specification version 3. It is not a replacement or an alternative. See the Arm System Memory Management Unit Architecture Specification version 3 for detailed descriptions of registers, data structures, and behaviors.

This guide assumes that you are familiar with the AArch64 Virtual Memory System Architecture (VMSA). If you want to learn about the AArch64 VMSA, see the Learn the architecture - AArch64 memory management guide and Learn the architecture - AArch64 memory attributes and properties guide.

2. What an SMMU does

An SMMU performs a task like that of an MMU in a PE. It translates addresses for DMA requests from system I/O devices before the requests are passed into the system interconnect. The SMMU only provides translation services for transactions from the client device, not for transactions to the client device. Transactions from the system or PE to the client device are managed by other means, for example, the PE MMUs. Figure 2-1: The role of an SMMU on page 8 shows the role of an SMMU in a system.

Figure 2-1: The role of an SMMU



Arm SMMUs provide:

Translation

The addresses supplied by the client device are translated from the virtual address space into the system's physical address space.

Protection

Operations from the client device might be prevented by the permissions held in the translation tables. You can prohibit a device to read, write, execute, or make any access to particular regions of memory.

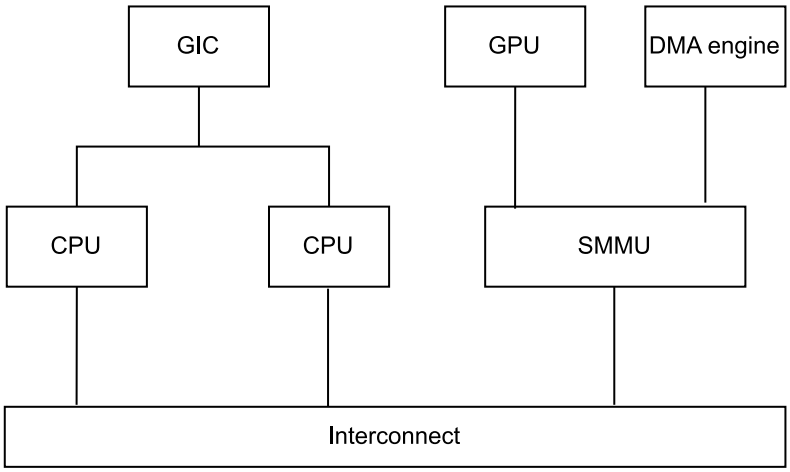
Isolation

Transactions from one device can be differentiated from those of another device, even if both devices share a connection to the SMMU. This means that the translation and protection properties can be applied differently for each device. Each device might have its own private translation tables, or might share them with other devices, as appropriate to the application.

To associate device transactions with translations and to identify different devices connected to an SMMU, DMA requests from a client device have an extra property. A StreamID uniquely identifies a stream of transactions. SMMU can perform different translations or checks for each stream. See What is a StreamID?.

The SMMU provides a flexible and scalable approach to memory management for I/O devices. It supports systems ranging from just one device to large system with many devices. Figure 2-2: Simplified example of a system topology including an SMMU on page 9 shows an SMMU connected to two devices: a GPU and a DMA engine. You can configure each device to have its own set of translation tables.

Figure 2-2: Simplified example of a system topology including an SMMU



An SMMU can optionally support two stages of translation in a similar way to PEs supporting the Virtualization Extensions. You can enable each stage of translation independently. An incoming address is logically translated from virtual address (VA) to intermediate physical address (IPA) in stage 1, then the IPA is input to stage 2 which translates the IPA to the output physical address (PA). Stage 1 is for use by a software entity, for example, an OS or a userspace application. It provides isolation or translation to buffers within the physical address space of the entity, for example DMA isolation within the physical address space of an OS. Stage 2 is for systems supporting the Virtualization Extensions and is intended to virtualize device DMA to guest VM address spaces.

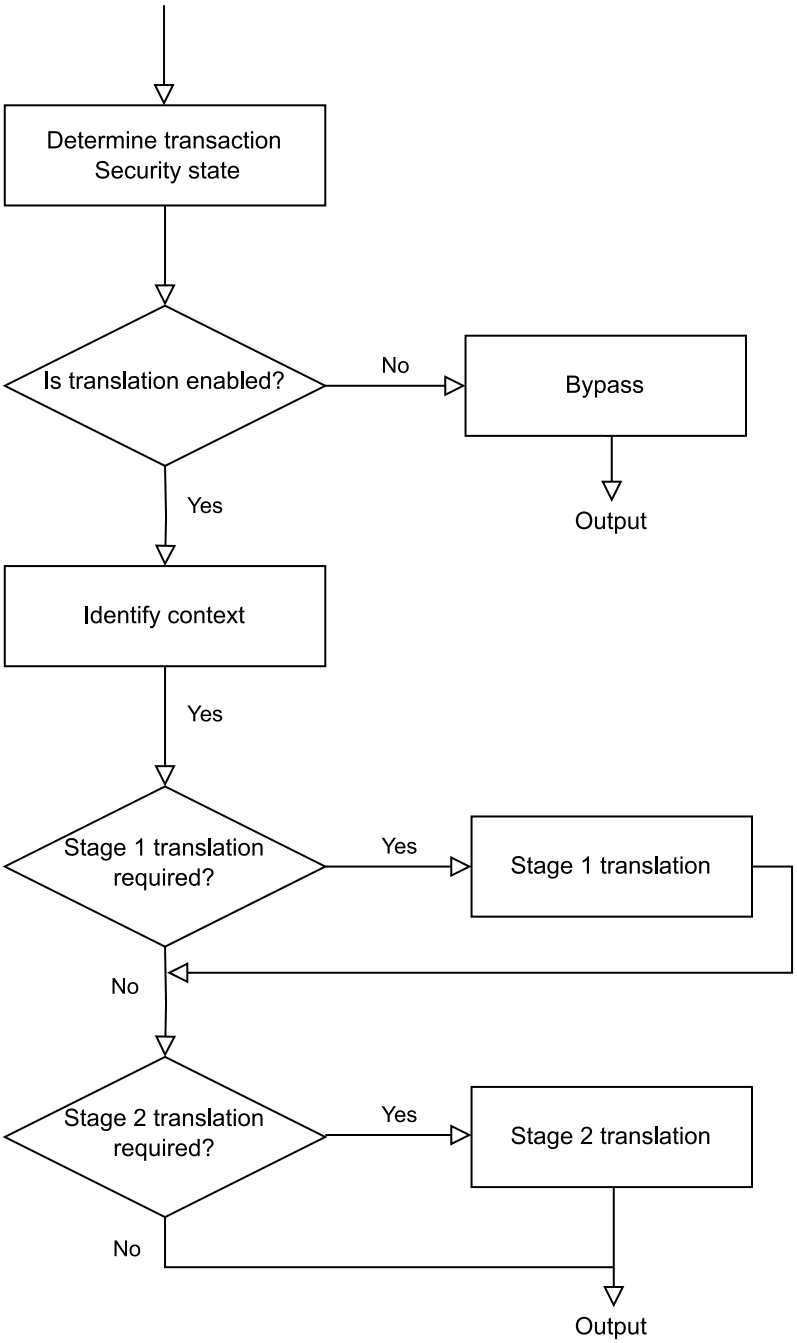
3. Operation of an SMMU

This chapter describes the operation of an SMMU. It contains the following sections:

3.1 Translation process overview

Figure 3-1: Simplified SMMU translation process on page 11 shows a simplified process that each incoming transaction goes through. This section describes the top-level translation process.

Figure 3-1: Simplified SMMU translation process



An incoming transaction follows these steps:

1. If the SMMU is globally disabled, the transaction passes through the SMMU without any address modification. Global attributes, such as memory type or Shareability, might be applied from the SMMU_GBPA register of the SMMU. Or, the SMMU_GBPA register might be configured to abort all transactions.
2. If the global bypass does not apply, the configuration is determined:
    a. A Stream Table Entry (STE) is located.
    b. If the STE enables stage 2 translation, the STE contains the stage 2 translation table base.
    c. If the STE enables stage 1 translation, a Context Descriptor (CD) is located. If stage 2 translation is also enabled by the STE, the CD is fetched from IPA space which uses the stage 2 translations. Otherwise, the CD is fetched from PA space.
3. Translations are performed if the configuration is valid.
    a. If stage 1 is configured to translate, the CD contains a stage 1 translation table base pointing to the base address of a table which is walked. This might require stage 2 translations, if stage 2 is enabled for the STE. If stage 1 is configured to bypass, the input address is provided directly to stage 2.
    b. If stage 2 is configured to translate, the STE contains a stage 2 translation table base which is used to perform the stage 2 translation. Stage 2 translates the output of stage 1, if stage 1 is enabled, or translates the input address if stage 1 is bypassed. If stage 2 is configured to bypass, the stage 2 input address is provided as the output address.
4. When a transaction passes all translation stages then the translated address with the relevant memory attributes is forwarded into the system.

An SMMU that implements the Realm Management Extension (RME) is subject to Granule Protection Checks (GPC) when it accesses to all physical addresses if GPC is enabled. However, fetching of Granule Protection Table (GPT) information is not subject to GPC. Access by all client devices to a physical address must be checked against the GPT. This behavior of the SMMU is the counterpart of FEAT_RME of A-profile architecture. For more details about RME, see Learn the architecture - Realm Management Extension.

This sequence shows the path of a transaction on a Non-secure stream. If Secure state or Realm state is supported, the path of a transaction on a Secure stream or Realm stream is similar, except SMMU_S_CR0.SMMUEN and SMMU_S_GBPA control bypass or SMMU_R_CR0.SMMUEN and SMMU_R_GBPA control bypass.

3.2 Stream Security

The SMMUv3 architecture has optional support for two Security states if RME Device Assignment is not implemented, reported by SMMU_S_IDR1.SECURE_IMPL. If RME Device Assignment is implemented, reported by SMMU_ROOT_IDR0.REALM_IMPL, additional Realm state is supported. For each Security state, there are separate registers and Stream tables. A stream can be Secure, Non-secure, or Realm, which is determined by the input signal SEC_SID. Table 3-1: Stream Security determination on page 13 shows how the input signal SEC_SID determines the Security state of a stream.

Table 3-1: Stream Security determination

SEC_SID value	Description
0b00	The stream is a Non-secure stream and uses the Non-secure registers and the Non-secure Stream Table
0b01	The stream is a Secure stream and uses the Secure registers and the Secure Stream Table
0b10	The stream is a Realm stream and uses the Realm registers and the Realm Stream Table

The way in which SEC_SID is signalled is **IMPLEMENTATION DEFINED**. This can be, but is not limited to, either a side-band signal or tied off for a particular device.

A Non-secure stream can only generate Non-secure downstream transactions. The incoming NS attribute and NS attribute in page tables are ignored. A Secure stream can generate both Secure and Non-secure downstream transactions. A Realm stream can generate both Realm and Non-secure downstream transactions.


Note

There is a difference between a Secure stream and a Secure transaction:

- A Secure transaction means an access to the Secure Physical Address space.
- A Secure stream is the programming interface for a device in the Secure state.

3.3 Stream identification

A system might have multiple devices sharing a single SMMU. Usually different translations are applied to different devices. For example, in the system shown in Figure 2-2: Simplified example of a system topology including an SMMU on page 9, virtual machine A can use the DMA engine and virtual machine B can use the GPU. The SMMU needs to identify different devices connected to it and to associate device traffic with translations.

3.3.1 What is a StreamID?

The StreamID is how the SMMU distinguishes different client devices. At its simplest, one device can have one StreamID. However, a device might be able to generate multiple StreamIDs, with different translations applied based on the StreamID. For example, for a DMA engine that supports multiple channels, different StreamIDs might be applied to different channels.

How the StreamID is formed is **IMPLEMENTATION DEFINED**. Typically it is a combination of side-band signals.

There is a separate StreamID namespace per Security state, that is:

- Secure StreamID 0 is a different StreamID to Non-secure StreamID 0.
- Realm StreamID 0 is a different StreamID to Secure StreamID 0.

However, the Realm and Non-secure StreamID namespaces are shared. A device that can operate in Non-secure state and Realm state must have the same StreamID in both states.

The StreamID selects a STE in a Stream table, which contains per-device configuration settings. See Stream table.

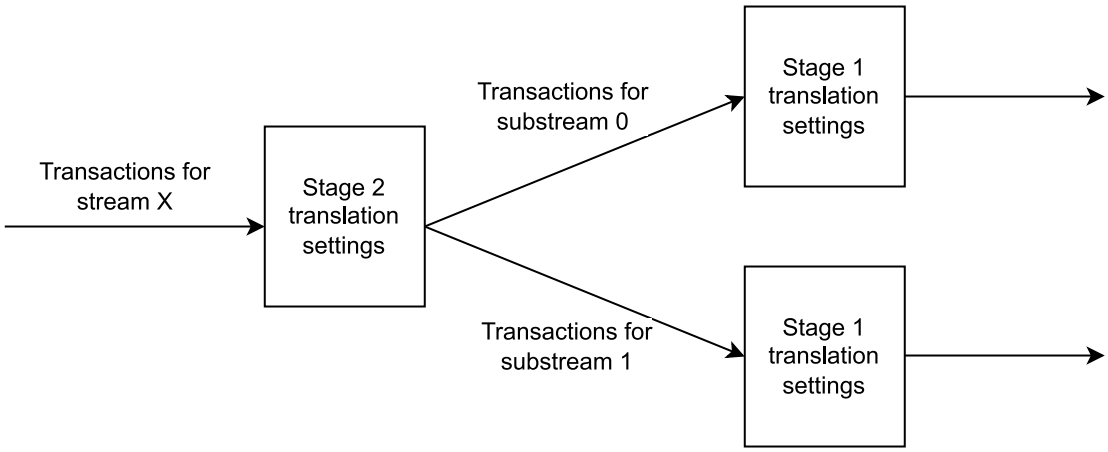
RME extends the SMMU architecture to support transactions without a StreamID. These transactions are subject to GPC, but not stage 1 or stage 2 translation. Arm expects this support to be used for devices like the GIC and the Debug Access Port. That is, this support is used for devices that would not traditionally be connected to an SMMU but whose accesses need Granule Protection Checks.

3.3.2 The role of the SubstreamID

The SubstreamID, might optionally be provided to an SMMU implementing stage 1 translation. Substreams enable transactions from the same device to share the same stage 2 translation, but have different stage 1 translations.

Consider a VM running multiple applications. You might want to have one DMA channel used by one application, and another DMA channel used by a different application. These applications are within the same VM, so they have the same stage 2 translation. However they have different stage 1 translation. SMMUv3 enables each stream to have multiple substreams. All the substreams share the same stage 2 translation, but each substream has it own stage 1 translation. For example, if a DMA engine with a fixed StreamID has multiple channels, different SubstreamIDs might be applied to different channels.

Figure 3-2: The role of the SubstreamID

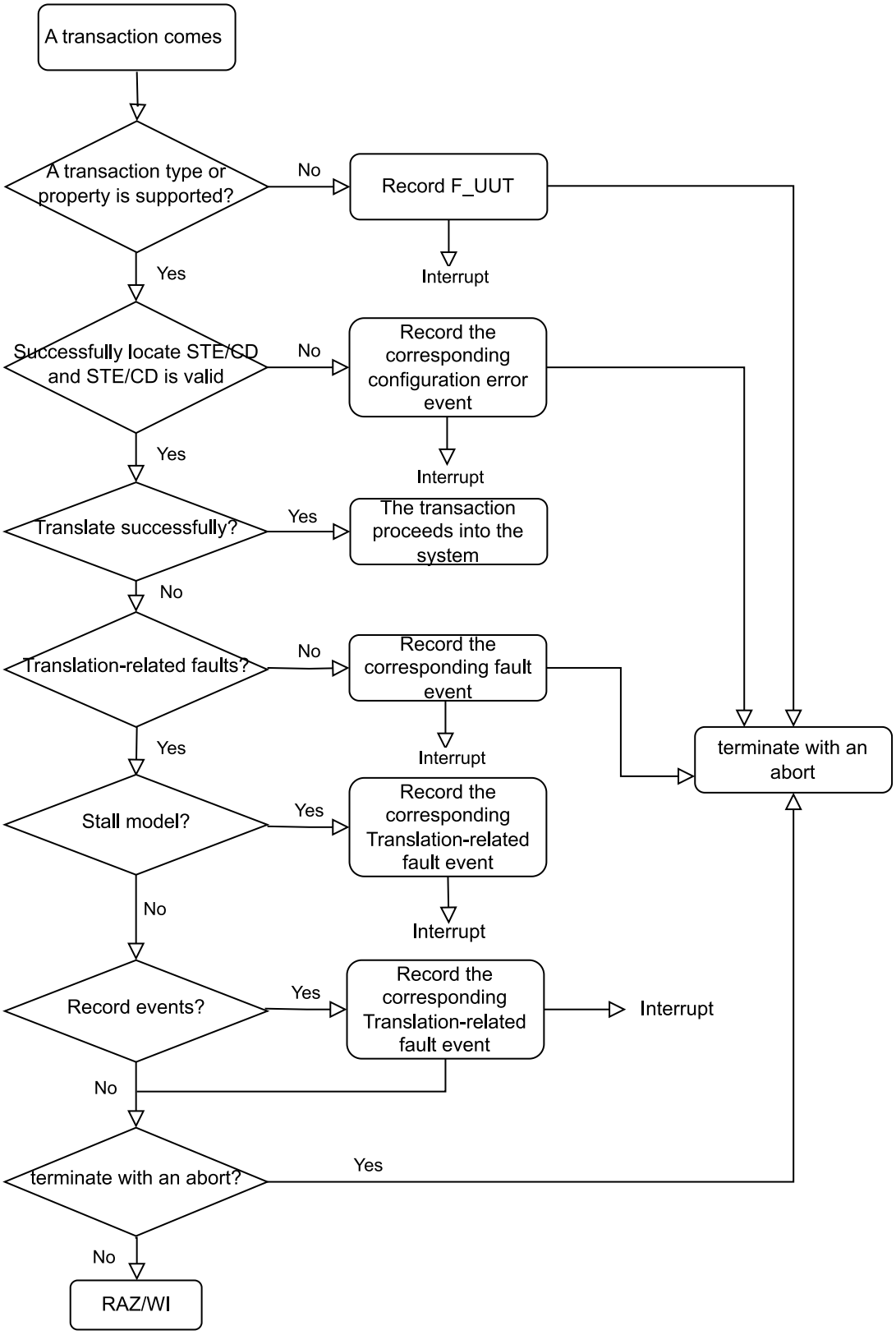


When a SubstreamID is supplied with a transaction and the configuration enables substreams, the SubstreamID indexes the CD table to select a stage 1 translation context. See CDs.

3.4 Fault model

Figure 3-3: SMMU fault recording and reporting on page 16 shows the flow how an SMMU records and reports a fault.

Figure 3-3: SMMU fault recording and reporting



An incoming transaction goes through several logical stages before continuing into the system. If a transaction type or property is unsupported by an SMMU for **IMPLEMENTATION DEFINED** reasons, an Unsupported Upstream Transaction fault (F_UUT) event is recorded and the transaction is terminated with an abort. See Event queue.

Otherwise, the StreamID and SubstreamID, if supplied, are used to locate a configuration for the transaction. If any of the required STE and CD cannot be located or are invalid, a configuration error event is recorded, and the transaction is terminated with an abort.

If a valid configuration is located so that the translation tables can be accessed, the translation process begins. Other faults can occur during this phase. When a transaction progresses as far as translation, the behavior on encountering a fault becomes configurable. The following fault types that constitute Translation-related faults when they are generated at either stage 1 or stage 2:

- F_TRANSLATION
- F_ADDR_SIZE
- F_ACCESS
- F_PERMISSION

These fault types correspond to the fault types in VMSA as:

- Translation fault
- Address Size fault
- Access Flag fault
- Permission fault

You can switch the behavior for Translation-related faults between the Terminate and Stall model as determined by the `CD.{A,R,S}` flags for stage 1 and the `STE.{S2R,S2S}` flags for stage 2.

3.4.1 Terminate model

When stage 1 is configured to terminate faults, a transaction that faults at stage 1 is either:

- Terminated with an abort reported to the client device that is making the access, when `SMMU_IDR0.TERM_MODEL` = 1 or `CD.A` = 1.
- **RAZ/WI** when `SMMU_IDR0.TERM_MODEL` = 0 and `CD.A` = 0.

When stage 2 is configured to terminate faults, a transaction that faults at stage 2 is terminated with an abort.

The behavior of the client device after termination is specific to the device.

If a stage that is configured to terminate faults is also configured with `CD.R == 1` or `STE.S2R == 1`, as appropriate to the stage of the fault, the SMMU records the details of the failed access into one Event record in the Event queue.

3.4.2 Stall model

The Stall model allows for a demand-paging type model. If the Stall model is enabled, then the SMMU records the details of the stalled transaction and raises an interrupt, so that software is aware of it. On receipt of the interrupt, software pages in the memory, and updates the translation tables, then sends a Resume command to tell the SMMU to retry the translation. The upstream device seems to experience a long latency.

Alternatively, software can choose to terminate the transaction rather than restarting it. For example, when a device accesses an address range which is not allowed to be accessed for security reasons.

The retry or terminate command is issued via the command queue of SMMU. See Command queue.

3.5 Bypass

In bypass mode, the SMMU does not have to translate the addresses of incoming transactions. A stream or entire Security state can be set to bypass. This means the output address is equivalent to the input address. However, the other input transaction attributes, for example cacheability, can still optionally be overridden. There are three types of bypass:

Global bypass

When translation is bypassed for one Security state, all transactions are bypassed translation for that Security state. Software can set `SMMU_(*_)CR0.SMMUEN` to 0 to enable global bypass. Software can also optionally set `SMMU_(*_)GBPA` to override input attributes.

Stream bypass

This type of SMMU bypass is available when `SMMU_(*_)CR0.SMMUEN = 1`. This is when a StreamID selects an STE configured to bypass (`STE.Config == 0b100`). For a stream-based bypass, the attributes are configured using STE fields. `STE.{MTCFG,MemAttr,ALLOCCFG,SHCFG,NSCFG,PRIVCFG,INSTCFG}` can override any attribute before the transaction is passed to the memory system

Stage bypass

When translation is enabled, bypass is configured separately for each stage of translation, this is also controlled by `STE.Config`:

- Stage 1 bypass: `STE.Config[0] == 0`
- Stage 2 bypass: `STE.Config[1] == 0`

This is almost the same as disabling a stage of translation in the VMSA architecture.

3.6 Address Translation Services

Address Translation Services (ATS) extends the PCIe protocol to support an SMMU that translates DMA addresses in advance. The translated addresses are subsequently cached in the local TLB of a PCIe device. The local TLB is named as Address Translation Cache (ATC). Locating translated addresses in the device aims to reduce latency and provides a scalable, distributed caching system that improves I/O performance. See the PCI Express Base Specification for more information about the mechanism of ATS.

Figure 3-4: ATS mechanism

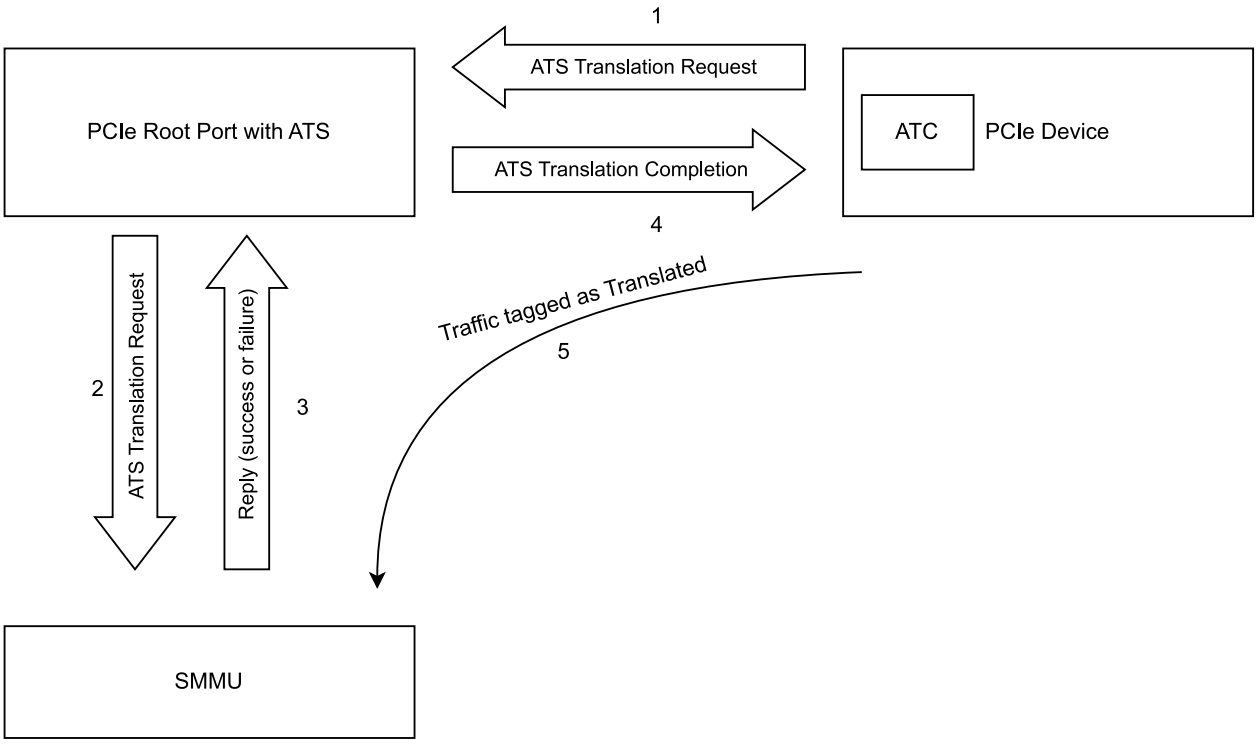


Figure 3-4: ATS mechanism on page 19 shows the flow how ATS works. A PCIe Function generates an ATS Translation Request which is sent through the PCIe hierarchy to the Root Port which then forwards it to SMMU. When it receives an ATS Translation Request, the SMMU performs the following basic steps:

1. Validates that the Function has been configured to enable ATS translation.
2. Determines whether the Function might access the memory indicated by the ATS Translation Request and has the associated access rights.
3. Determines whether a full translation or partial translation can be provided to the Function. If yes, the SMMU issues a translation to the Function.
4. The SMMU communicates the success or failure of the request to the Root Port which generates an ATS Translation Completion and transmits via a Response TLP through a Root Port to the Function.

When the Function receives the ATS Translation Completion, then it either updates its ATC to reflect the translation or notes that a translation does not exist. The Function generates subsequent requests using either a translated address or an untranslated address on the result of the Completion. SMMU_CR0.ATSCHK controls whether the SMMU allows translated transactions to bypass with no further checks.


Note

In an SMMU with RME DA, a Device Permission Table (DPT) performs checks on translated transactions.

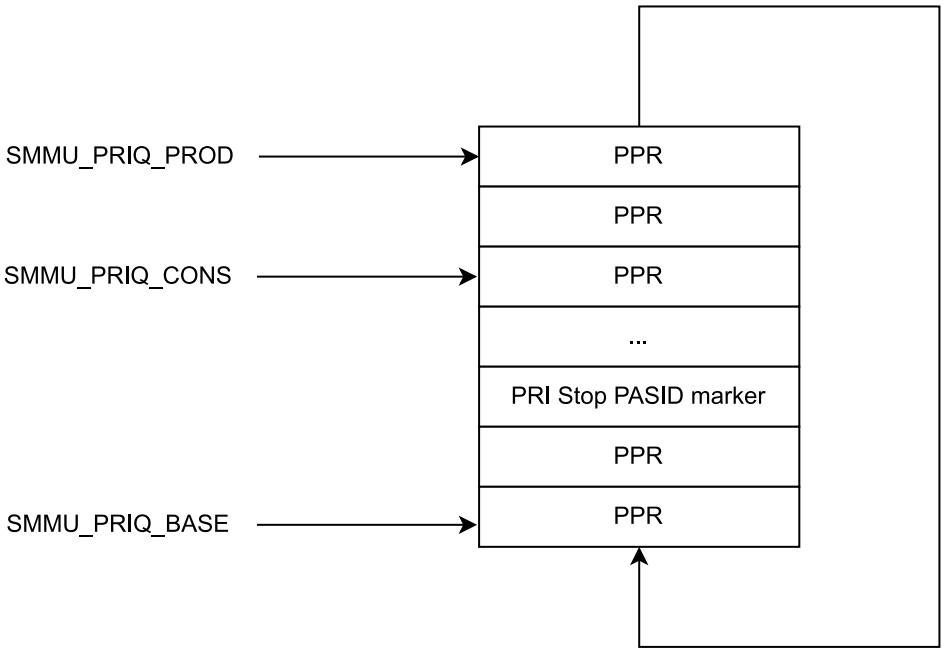
An SMMU implemention that supports PCIe ATS might provide an optional extra hardware interface. For example, the MMU-600/MMU-700 provides DTI-ATS interface which supports ATS translation requests and ATS translation responses. SMMU_IDR0.ATS shows whether the SMMU implements ATS.

3.7 Page Request Interface

If there is only an ATS interface, to improve the efficiency of DMA, the software needs to pin the memory used by DMA. That is, the memory is locked in place so that it cannot be swapped out by the system's dynamic paging mechanism. The overhead associated with pinning memory might be modest. However, the negative impact on system performance of removing a large portion of memory from a pageable pool can be significant. An associated Page Request Interface (PRI) adds the ability for PCIe functions to target DMA at unpinned, dynamically-paged memory. See the PCI Express Base Specification for more information about the mechanism of PRI.

To support PRI, the SMMUv3 architecture introduced an optional PRI queue to store the PRI Page Requests (PPRs) received from PCIe Root Ports. Figure 3-5: PRI queue on page 21 shows the data structure of the PRI queue. The PPRs contain information, such as StreamID, SubstreamID, and page address. Software can use this information to locate the target pages.

Figure 3-5: PRI queue



The following procedure describes how PRI works:

1. A PCIe Function can issue PPRs to ask the software to make the requested pages available when an ATS request fails due to the pages not being present.
2. SMMU stores PPRs in the PRI queue and increments `SMMU_PRIQ_PROD`.
3. Software receives these PPRs on the PRI queue and increments `SMMU_PRIQ_CONS`.
4. Software can issue `CMD_PRI_RESP` using the command queue to send a PRI response to the endpoint to indicate success or failure of the PPRs. See Commands.
    - Software issues a positive PRI response command to the SMMU after making pages present. If a requested address is unavailable, the software must issue a negative PRI response command. For example, when a programming failure has caused the device to request an illegal address.

4. Programming the SMMU

This section introduces the programming interface of SMMUv3:

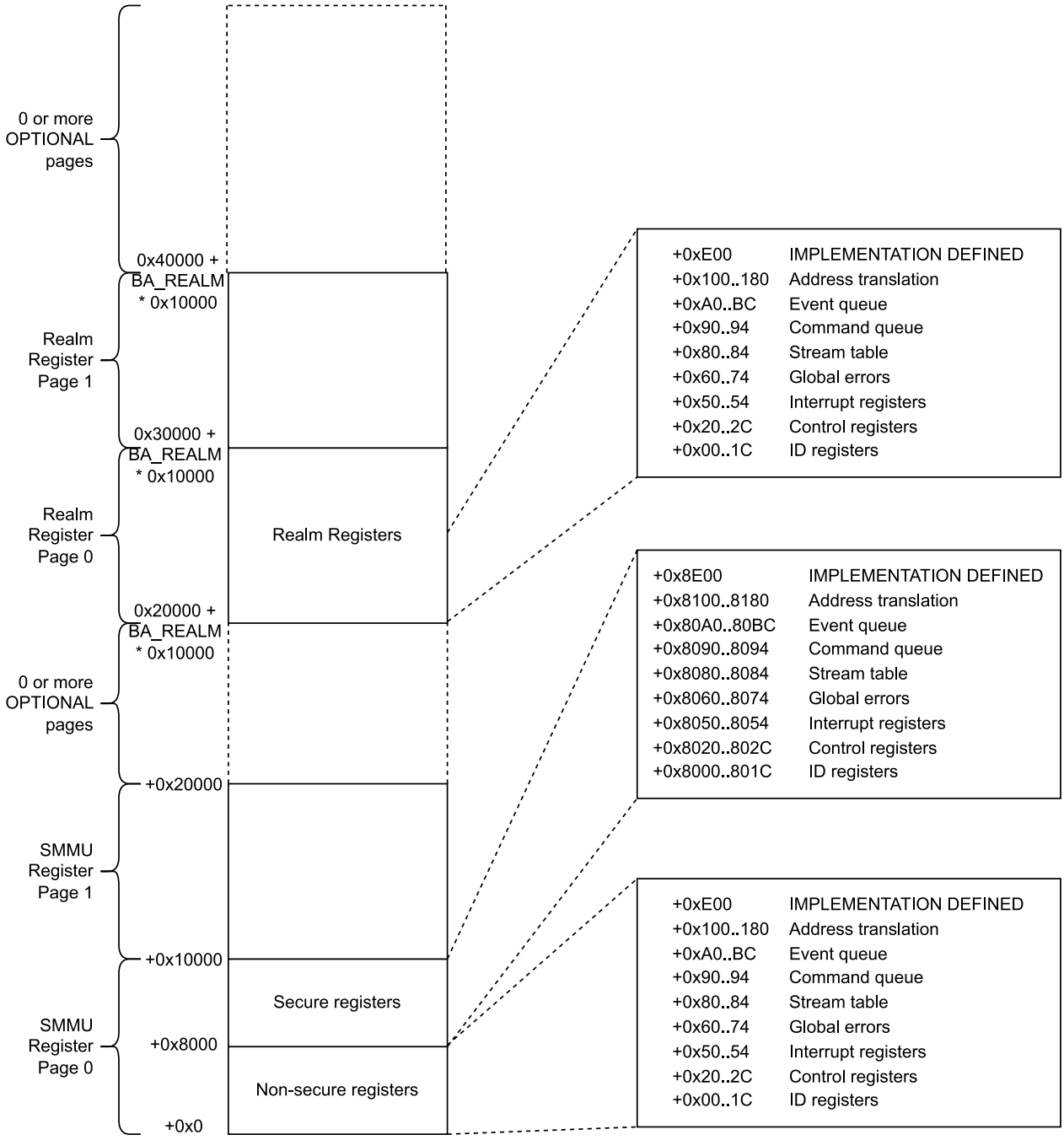
- SMMU registers
- Stream table
- CD
- Event queue
- Command queue

Most configuration is stored in memory structures, so software must allocate the memory for these structures. However, some configuration is stored in SMMU registers, such as the location and size of each of these structures.

4.1 SMMU registers

Figure 4-1: SMMU register map pages on page 23 shows the SMMU register map pages.

Figure 4-1: SMMU register map pages



The SMMU registers occupy two consecutive 64K pages, SMMU Register Page 0 and SMMU Register Page 1, that are architecturally required. Additional pages might be present for optional or **IMPLEMENTATION DEFINED** features:

- If the VATOS interface is supported, one 64KB page is present containing the VATOS registers.
- If the Secure VATOS interface is supported, one 64KB page is present containing the S_VATOS registers.
- If Enhanced Command queue interfaces are supported, one or more Command queue control pages might be present.
- If Enhanced Command queue interfaces are supported for Secure state, one or more Secure Command queue control pages might be present.
- Any number of **IMPLEMENTATION DEFINED** pages might be present.

An SMMU with RME adds a Root Control Page which is accessible only in the Root PA space. Its base address is distinct from addresses of registers accessible in other PA spaces.

An SMMU with RME DA adds two consecutive 64K Realm Register Pages, Realm Register Pages 0 and Realm Register Pages 1, which are only accessible in Realm and Root PA spaces. The base address of Realm Register Pages 0 = the base address of SMMU Register Page 0 + `0x20000` + (`SMMU_ROOT_IDR0.BA_REALM` * `0x10000`).

The registers in SMMU Register Page 0 are split into:

- Non-secure registers, `SMMU_*`, starting at offset `0x0`
- Secure registers, `SMMU_S_*`, starting at offset `0x8000`

The equivalent Realm registers, `SMMU_R_*`, are in Realm Register Page 0.

Most of the sets of registers are the same, but the same register in different security states might have a different address offset. For example:

SMMU_CMDQ_BASE
: Offset `0x90` in SMMU Register Page 0, sets the base address of the Non-secure Command queue.

SMMU_S_CMDQ_BASE
: Offset `0x8090` in SMMU Register Page 0, sets the base address of the Secure Command queue.

SMMU_R_CMDQ_BASE
: Offset `0x90` in Realm Register Pages 0, sets the base address of the Realm Command queue.

As well as the registers used for the data structures, there are other registers that control other functions of the SMMU:

- Identification fields that report implemented features
- Top level control, such as enabling the SMMU and the queues
- Interrupt configuration
- Global error reporting

- Address translation operations

4.2 Stream table

The SMMU uses a set of data structures in memory to locate translation data. See Translation process overview. SMMU_(*_)STRTAB_BASE hold the base address of the initial structure, the Stream table. A Stream Table Entry (STE) contains stage 2 translation table base pointers. It also locates stage 1 configuration structures, which contain translation table base pointers.

The StreamID of an incoming transaction, qualified by SEC_SID, determines which Stream table is used for lookup, and locates the STE.

Two formats of Stream table can be supported:

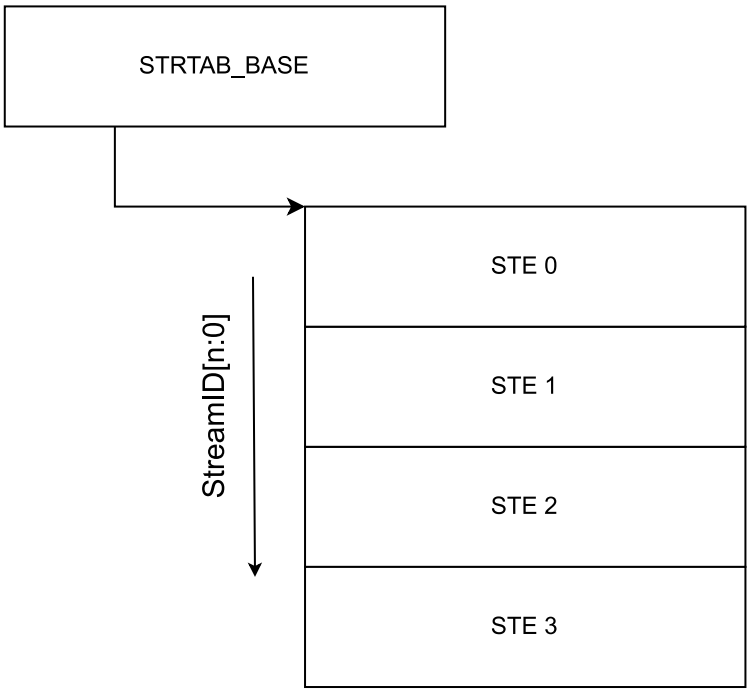
- Linear Stream table
- 2-level Stream table

Support for the 2-level Stream table format is discoverable using the SMMU_IDR0.ST_LEVEL field. Software configures SMMU_(*_)STRTAB_BASE_CFG to specify the format of the Stream table used.

4.2.1 Linear Stream table

A linear Stream table is a contiguous array of STEs, indexed from 0 by StreamID. The size of the array is configurable as a 2^n multiple of STE size up to the maximum number of StreamID bits supported in hardware by the SMMU.

Figure 4-2: Linear Stream table



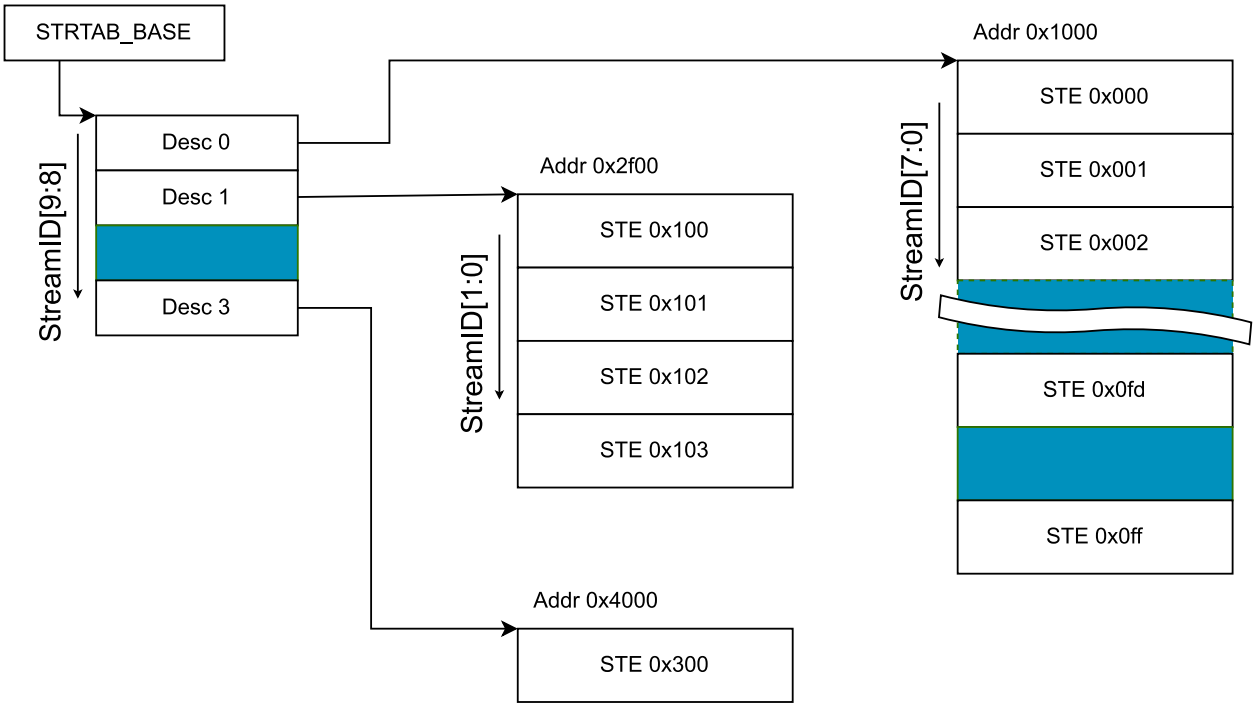
To locate the STE for a transaction, the Stream tables are indexed by the StreamID of the transaction:

```
STE_addr = STRTAB_BASE.ADDR + StreamID * sizeof(STE)
```

4.2.2 2-level Stream table

A 2-level Stream table consists one top-level table that contains descriptors pointing to multiple second-level tables that contain linear arrays of STEs. You can configure the span of StreamIDs covered by the entire structure up to the maximum StreamID supported by the SMMU. However, the second-level tables do not have to be fully populated and might vary in size. This saves memory and avoids the requirement of large physically-contiguous allocations for very large StreamID spaces.

Figure 4-3: Example Two level Stream table with SPLIT == 8



The 2-level Stream table is useful to software where the StreamID width is large, and either it cannot easily allocate that much contiguous memory or the distribution of StreamIDs is relatively sparse. For example, for a use-case like PCIe, the maximum of 256 buses are supported, and the RequesterID or StreamID space is at least 16-bits. See PCIe considerations. However, because there is usually one PCIe bus for each PCIe link and potentially one device for each bus, in the worst case a effective StreamID might only appear once every 256 possible StreamIDs. For example, if the number of StreamID is 16 bits, the first effective StreamID is 0, the second effective StreamID is 256, the third effective StreamID is 512, and so on.

The top-level table is indexed by StreamID[n:x], where *n* is the uppermost StreamID bit covered, and *x* is a configurable Split point given by `SMMU_(*_)STRTAB_BASE_CFG.SPLIT`. The second-level tables are indexed by up to StreamID[(x - 1):0], depending on the span of each table.

```
L1STD_addr = STRTAB_BASE.ADDR + StreamID[n:x] * sizeof(L1STD)
STE_addr = L1STD.L2Ptr + StreamID[(x - 1):0] * sizeof(STE)
```

Figure 4-3: Example Two level Stream table with SPLIT == 8 on page 26 shows an example of a 2-level Stream table with SPLIT == 8.

Software initially allocates the L1 Stream table, which sets the maximum StreamID width the SMMU is configured to accept. Each entry in the L1 table represents a block of StreamIDs, the size of which is set via `SMMU_(*_)STRTAB_BASE_CFG.SPLIT`. `SMMU_(*_)STRTAB_BASE_CFG.SPLIT` sets the maximum size of the L2 Stream tables.

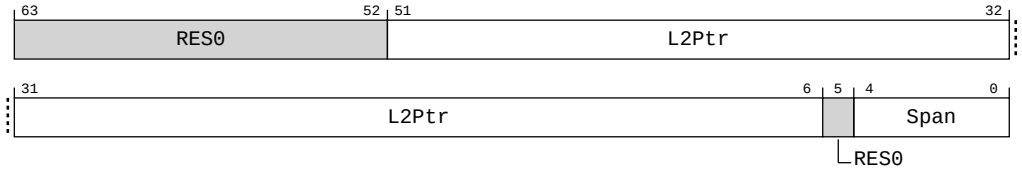
Then L1 Stream Table Descriptor (L1STD) can either be invalid, or point at a L2 Stream table. Software can allocate these L2 Stream tables on demand.

The L2 Stream tables do not have to cover the full range of StreamIDs represented by the L1STD. The L1STD contains a `Span` field which sets the size of the table pointed at. Any StreamID which falls within the L1 entry but outside of the `Span` is treated as invalid.

4.3 L1 Stream Table Descriptor

Figure 4-4: L1 Stream table descriptor on page 27 shows the fields of the L1STD.

Figure 4-4: L1 Stream table descriptor



Field	Description
L2Ptr	Pointer to start of Level-2 array
Span	Level 2 array contains 2^{Span-1} STEs

4.4 STEs

An STE stores the context information for that Stream. Each STE is 64 bytes.

STE fields follow the convention as follows:

- An S1 prefix for fields related to stage 1 translation
- An S2 prefix for fields related to stage 2 translation
- Neither for fields unrelated to a specific translation stage

The following list lists some commonly used fields:

- Common configuration
  - Valid: STE is valid or not
  - Config: stage 1/stage 2 translation enabled or bypass
  - CONT: contiguous Hint
  - EATS: enables PCIe ATS translation and traffic
  - STRW: StreamWorld control corresponding to translation regime in VMSA
- Stage 1 translation settings
  - S1Fmt: format of the CD table
  - S1ContextPtr: stage 1 Context descriptor pointer
  - S1CDMax: number of CDs pointed by S1ContextPtr, $2^{S1CDMax}$
  - S1DSS: default substream
  - S1CIR/S1COR/S1CSH: CD table memory attributes
- Stage 2 translation settings
  - S2T0SZ: size of IPA covered by stage 2 translation table
  - S2SL0: starting level of stage 2 translation table walk
  - S2IR0/S2OR0/S2SH0: memory attributes of stage 2 translation table walk
  - S2TG: stage 2 Translation Granule size
  - S2PS: physical address size

4.5 CDs

A CD stores all the settings related to stage 1 translation. Each CD is 64-bytes. The pointer to a CD comes from an STE, not from a register. A CD is only required when stage 1 translation is performed. For a stream with only stage 2 translation, or bypass, only the STE is needed.

The CD associates the StreamID with stage 1 translation table base pointers, to translate a VA into an IPA, for each stage 2 configuration. If substreams are in use, multiple CDs indicate multiple stage 1 translations, one for each substream. Transactions with a SubstreamID are terminated when stage 1 translation is not enabled.

When stage 1 translation is used, the `STE.S1ContextPtr` field gives the address of one of the following, configured by `STE.S1Fmt` and `STE.S1CDMax`:

- A single CD
- The start address of a single-level table of CDs

- The start address of a first-level, L1, table of L1CDs

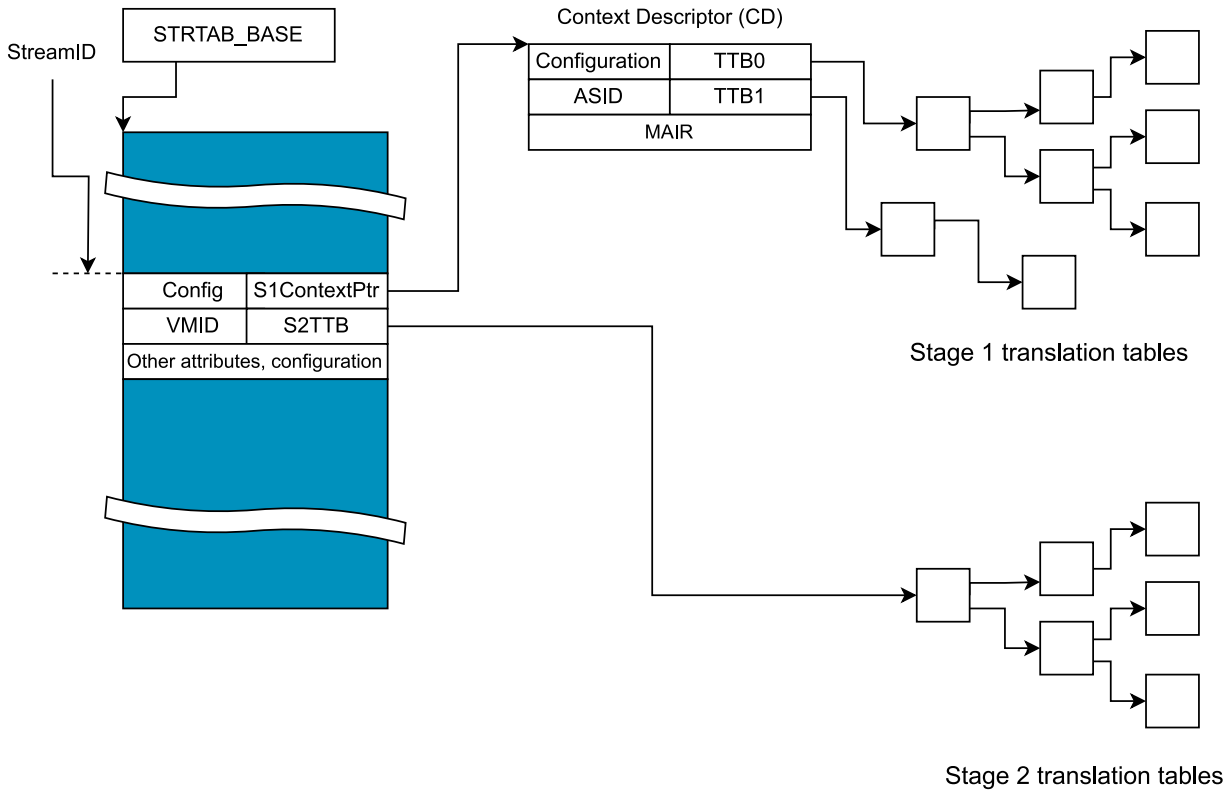
It is expected that, where hypervisor software is present:

- The Stream table and stage 2 translation table are managed by the hypervisor.
- The CD tables and stage 1 translation table associated with devices under guest control are managed by the guest OS.

4.5.1 Single CD

Figure 4-6: Multiple Context Descriptors for Substreams on page 30 shows an example configuration in which a StreamID selects an STE from a linear Stream table. The STE points to a translation table for stage 2 and points to a single CD for stage 1 configuration. The CD points to translation tables for stage 1.

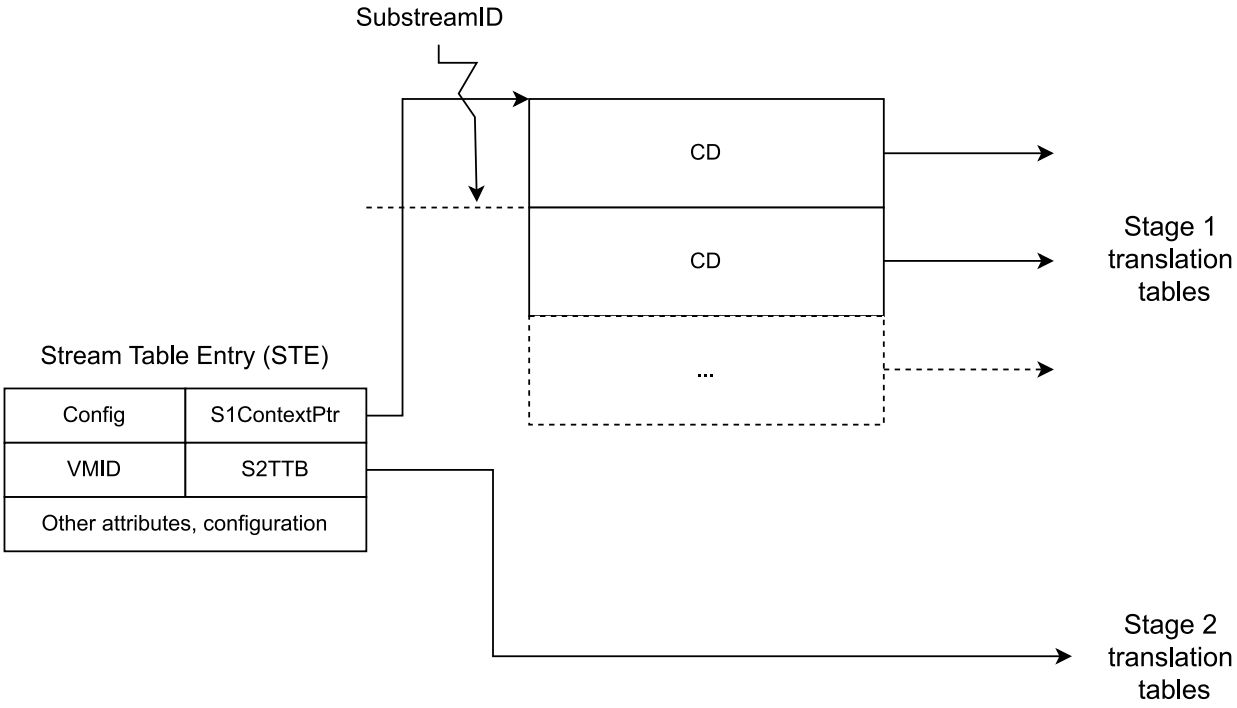
Figure 4-5: Configuration structure example



4.5.2 Single-level CD table

Figure 4-6: Multiple Context Descriptors for Substreams on page 30 shows a configuration in which an STE points to an array of several CDs. An incoming SubstreamID selects one of the CDs and therefore the SubstreamID determines which stage 1 translations are used by a transaction.

Figure 4-6: Multiple Context Descriptors for Substreams

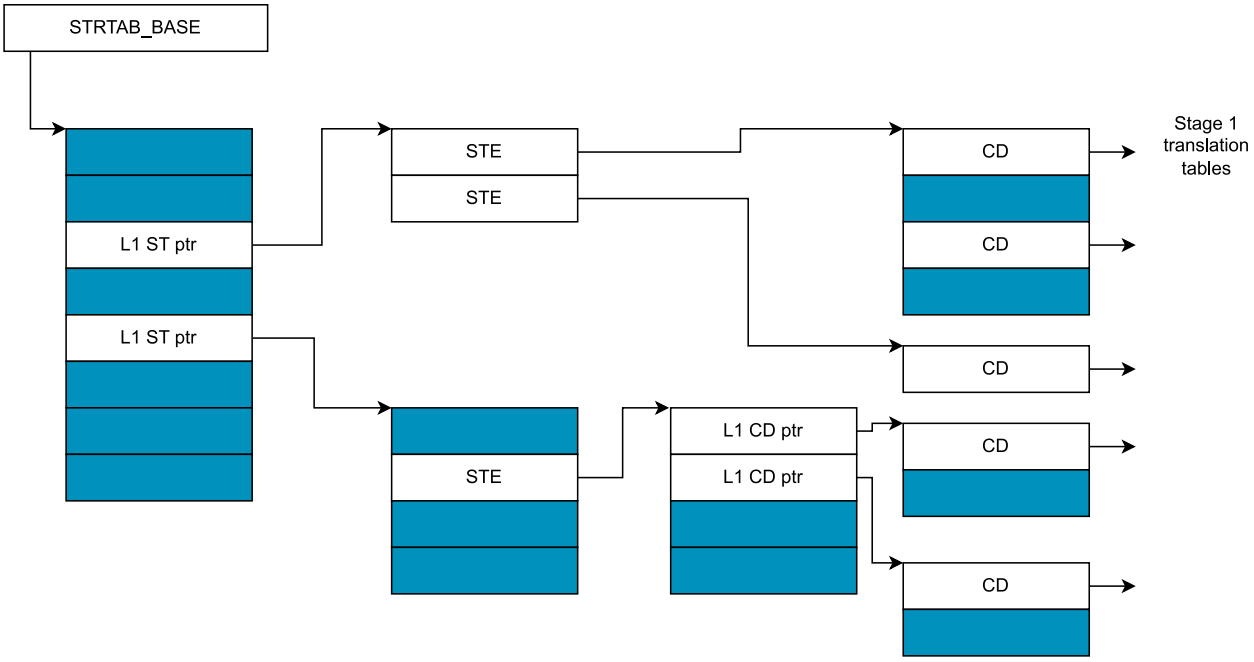


4.5.3 2-level CD table

When Substreams are used, then the array of CDs can be 2-level in much the same way as the Stream table.

Figure 4-7: Configuration structure example on page 31 shows a complex layout in which a 2-level Stream table is used. Two of the STEs point to a single CD, or a flat array of CDs. The third STE points to a 2-level CD table. With multiple levels, many streams and many substreams might be supported without large physically-contiguous tables.

Figure 4-7: Configuration structure example



The L1 table is a contiguous array of L1CDs indexed by the upper bits of the SubstreamID. Each `L1CD.L2Ptr` in the L1 table is configured with the address of a linear level two, L2, table of CDs. The L2 table is a contiguous array of CDs indexed by the lower bits of the SubstreamID. The ranges of SubstreamID bits that are used for the L1 and L2 indices are configured by `STE.S1Fmt`.

4.6 Virtual Machine Structure

The Virtual Machine Structure (VMS) is an SMMU concept and is a structure that is located from the pointer field `STE.VMSPtr`. The VMS holds per-VM configuration settings. Multiple STEs within a Security state with the same VMID must point to the same VMS. Currently the VMS only supports the feature of Memory System Resource Partitioning and Monitoring (MPAM). It maps from virtual `CD.PARTID` values which are configured by the guest OS to physical PARTID values. For more information about the MPAM feature, see Learn the architecture - Memory System Resource Partitioning and Monitoring (MPAM) Overview.

4.7 Caching

An SMMU implementation is not required to implement caching of any kind. However, it is expected that performance requirements require caching of at least some configuration or translation information.

Caching of configuration or translations might manifest as separate caches for each type of structure, or some combination of structures into a smaller number of caches.

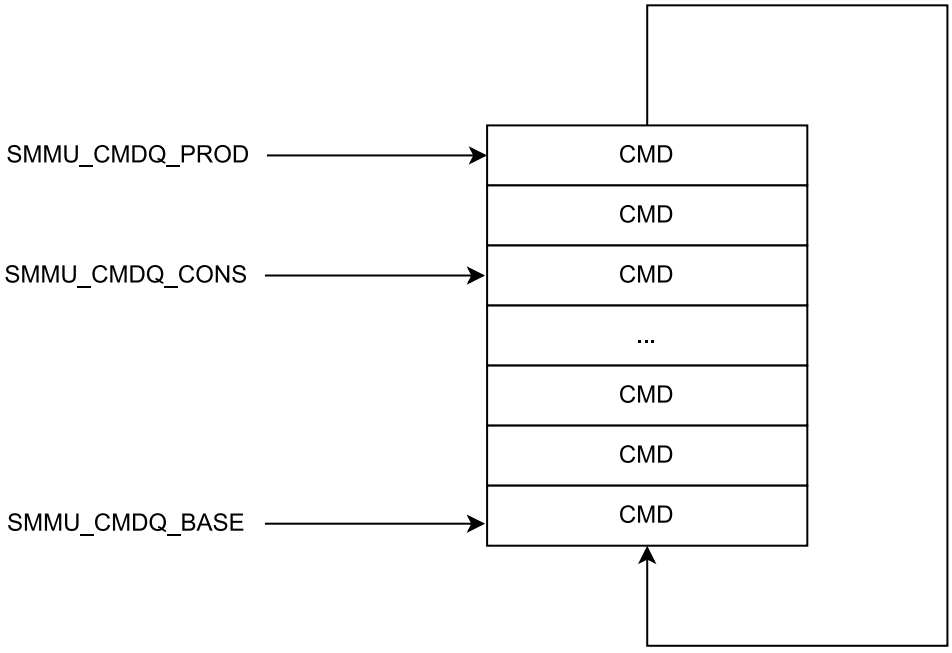
When software wants to change the configuration, it needs to invalidate any caching in the SMMU. It does this by issuing configuration cache invalidation commands to the Command queue. See Command queue. When software wants to change the translations, it also needs to invalidate any caching in the SMMU. It does this either by issuing TLB invalidation commands to the Command queue or sending TLB invalidations via broadcast.

4.8 Command queue

The SMMU is controlled via a circular Command queue in memory. For example, when software changes an STE or a translation, it needs to invalidate the related caching in the SMMU. This can be done by issuing corresponding invalidation commands to the Command queue. See Commands for the classes of commands.

Prior to SMMUv3.3, there is one Command queue per Security state. An SMMU that implements SMMUv3.3 can optionally support multiple Command queues for reducing contention between multiple PEs submitting Commands to the SMMU simultaneously.

Figure 4-8: Command queue



SMMU_CMDQ_BASE stores the base address and size of the Command queue. Software needs to check that there is space on the command queue before adding new commands. When software adds one or more commands to the queue, it updates the SMMU_CMDQ_PROD pointer to tell the SMMU that new commands are available. As the SMMU processes commands, it updates the SMMU_CMDQ_CONS pointer. Software reads the SMMU_CMDQ_CONS pointer to determine that commands are consumed and space is free.

If SMMU_CMDQ_PROD.WR == SMMU_CMDQ_CONS.RD and SMMU_CMDQ_PROD.WR_WRAP != SMMU_CMDQ_CONS.RD_WRAP, the queue is full.

SMMU_CMDQ_CONS updating only shows that a command has been consumed. It might not mean that the effects of the command are visible. The CMD_SYNC command is available to synchronize. It is only be consumed when the effects of previous commands are visible. For example:

```
CMD_TLBI_EL3_ALL
CMD_SYNC
```

When CMD_SYNC is consumed, the effect of CMD_TLBI_EL3_ALL is guaranteed to be visible.

4.8.1 Commands

All entries in the Command queue are 16 bytes long. All Command queue entries are little-endian. Each command begins with an 8-bit command opcode. The following list shows the classes of commands:

Prefetch commands

Prefetch translation or configuration.

TLB invalidation commands

Invalidates all the TLB entries at the given Exception level that match the tag (VMID, ASID, VA or All).

Configuration cache invalidation commands

Invalidates all the configuration cache entries within the specified range <All, RANGE> for a given StreamID (in case of STE) or both StreamID and SubstreamID (in case of CD).

Synchronization command

Provides a synchronization mechanism for preceding commands that were issued to same Command queue as the Synchronization command.

PRI response command

Notifies a device corresponding to StreamID, and SubstreamID when SSV == 1, that page request group indicated by PRGIndex has completed with given response.

ATC invalidation command

Invalidates all the ATC entries that match the tag (StreamID, SubstreamID, Address range).

Stall resume/termination command

Resumes or terminates processing of the stalled transaction identified with the given StreamID and STAG parameter.

DPT invalidation commands

These commands remove cached DPT information from DPT TLBs.

For a complete description of the commands, see Arm System Memory Management Unit version 3

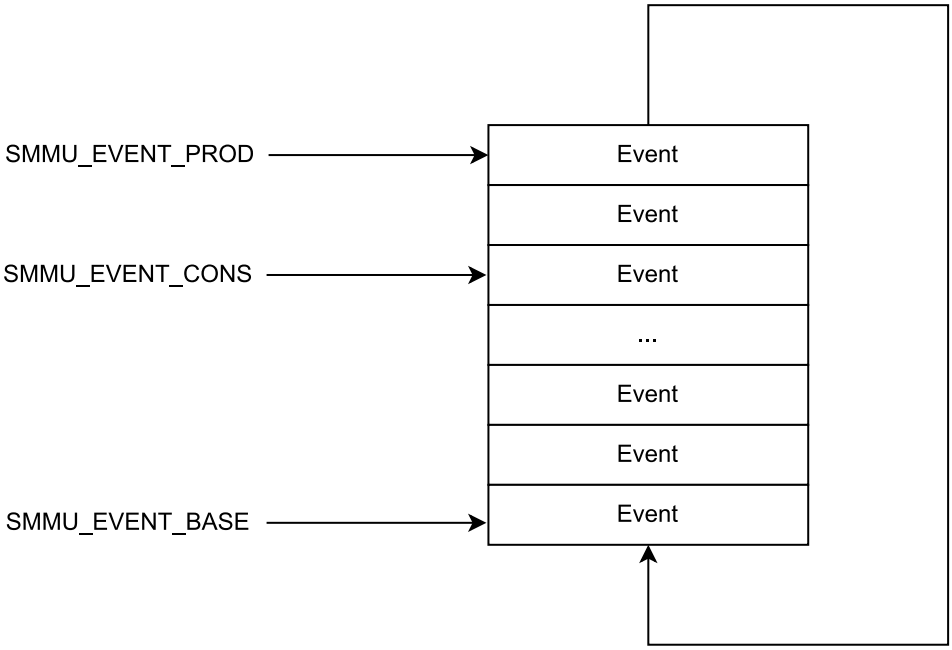
4.9 Event queue

A set of configuration errors and faults are recorded in the Event queue if they occur. These include events that arise from incoming device traffic, such as:

- A configuration error discovered when a configuration is fetched on receipt of device traffic
- A page fault arising from the device traffic address

There is one Event queue per Security state. The SMMU generates an interrupt when the Event queue transitions from empty to non-empty. The structure of the queue is the same as for the Command queue, except that the roles of the producer and the consumer are reversed. For the Command queue, the SMMU is the consumer, whereas for the Event queue, the SMMU is the producer.

Figure 4-9: Event queue



A sequence of faults or errors caused by incoming transactions can fill the Event queue and cause it to overflow if the events are not consumed fast enough. Events resulting from stalled faulting transactions are never discarded if the Event queue is full. They are recorded when entries are consumed from the Event queue and space next becomes available. Other types of events are discarded if the Event queue is full. The system software is expected to consume entries from the Event queue quickly to avoid overflow during normal operation.

4.9.1 Event records

Event records are 32 bytes in size, and all Event records are little-endian. Three categories of events might be recorded into the Event queue:

- Configuration errors
- Faults from the translation process
- Miscellaneous

The following list shows some examples:

- `CERROR_ILL`: Illegal or unrecognised command
- `C_BAD_STREAMID`: Transaction StreamID out of range
- `C_BAD_STE`: Used STE invalid
- `F_TRANSLATION`: Translation fault

For a complete description of Event records, see Arm System Memory Management Unit version 3

4.10 Global errors

Global Errors pertaining to a programming interface are reported into the appropriate `SMMU_(*_)GERROR` register instead of into the memory-based Event queue. These errors tend to be serious, such as stopping the SMMU from making forward progress. For example like an external abort when accessing one of the configuration data structures. The following list shows all the Global errors:

- Command queue error
- Event queue access aborted, delivery into the Event queue stops when an abort is flagged
- PRI queue access aborted, delivery into the PRI queue stops when an abort is flagged
- `CMD_SYNC` Message Signaled Interrupts (MSI) write aborted
- Event queue MSI write aborted
- PRI queue MSI write aborted (Non-secure GERROR only)
- GERROR MSI write aborted
- SMMU entered Service Failure Mode

`SMMU_(*_)GERROR` provides a one-bit flag for each Global error. When an error condition is triggered, the error is activated by toggling the corresponding flag in GERROR. In some cases, the SMMU behavior changes while the error is active. For example, commands are not consumed from the Command queue while the Command queue error is active.

When `SMMU_(*_)IRQ_CTRL.GERROR_IRQEN == 1`, a GERROR interrupt is triggered when the SMMU activates an error which is not GERROR MSI write aborted.

4.11 Minimum configuration

The following sequence shows the minimum configuration for SMMU initialization:

1. Allocate Stream table
   - Allocate memory for Stream table.
   - Configure the format and size of the Stream table by writing to SMMU_STRTAB_BASE_CFG.
   - Configure the base address for the Stream table by writing to SMMU_STRTAB_BASE.
   - Prevent uninitialized memory being interpreted as a valid configuration by setting STE.V = 0 for each STE to mark it as invalid.
   - Ensure that written data is observable to the SMMU by performing a DSB operation.
     - If SMMU_IDR0.COHACC = 0, the system does not support coherent access to memory for the SMMU. In these cases, you might require extra steps, including data cache maintenance to ensure that the SMMU can observe the written data.
2. Allocate Command and Event queues
   - Allocate memory for the command queue and event queue.
   - Specify the base address, the size, the producer pointer and the consumer pointer by configuring SMMU_CMDQ_BASE, SMMU_CMDQ_PROD, SMMU_CMDQ_CONS, SMMU_EVENTQ_BASE, SMMU_EVENTQ_PROD, and SMMU_EVENTQ_CONS.
3. Set memory attributes for accessing the Stream Table, Command Queue, and Event Queue
   - Configure SMMU_CR1.
4. Enable IRQs for the Event queue and GERROR
   - Configure SMMU_IRQ_CTRL.
5. Enable the Command and Event queues
   - Enable the Command queue by setting the SMMU_CR0.CMDQEN bit to 1.
   - Check that the enable operation is complete by polling SMMU_CR0ACK until CMDQEN reads as 1.
   - Enable the Event queue by setting the SMMU_CR0.EVENTQEN bit to 1.
   - Check that the enable operation is complete by polling SMMU_CR0ACK until EVENTQEN reads as 1.
6. Invalidate the TLBs and configuration cache structures
   - Issue commands to the Command queue
     - To invalidate TLB entries, ensure that the software issues the appropriate command for the translation context. For example, to invalidate TLB entries for Non-secure EL1 contexts, issue CMD_TLBI_NSNH_ALL, for EL2 contexts, issue CMD_TLBI_EL2_ALL.
     - To invalidate the SMMU configuration cache, issue the CMD_CFGI_ALL command.
     - To force all previous commands to complete, issue CMD_SYNC.
   - Alternatively, Secure software can invalidate all TLBs and caches with a single write.

    - Set SMMU_S_INIT.INV_ALL to 1.
    - Poll SMMU_S_INIT.INV_ALL to check it is set to 0 before continuing the SMMU configuration.

7. Enable translation
    - Set the SMMU_CR0.SMMUEN bit to 1.
    - Check that the enable operation is complete by polling SMMU_CR0ACK until SMMUEN reads as 1.

This sequence shows the Non-secure SMMU programming when either the SMMU does not implement the RME extension, or GPC is not enabled. The Secure or Realm SMMU programming is similar. If the SMMU implements the RME extension, and GPC is enabled, the software that runs in Root world needs to initialize the GPT first.

5. System architecture considerations

This chapter introduces some system considerations related to SMMU.

5.1 I/O coherency

A device is I/O coherent with the PE caches if its transactions snoop the PE caches for cacheable regions of memory. This improves performance by avoiding Cache Maintenance Operation (CMO). The device does not need to access the external memory. The PE does not snoop the device cache.

If a device is I/O coherent with the PE caches, no CMO is required for memory that is shared with the device. This memory is mapped on CPUs as Inner Write-Back (iWB), Outer Write-back (oWB), and Inner shareable (ISH). If a device is not I/O coherent with the PE caches, CMO is required for memory that is shared with the device that is mapped on CPUs as iWB-oWB-ISH.

These types of SMMU access can be I/O coherent:

- SMMU-originated transactions
  - Translation table walks
  - Fetch of L1STD, STE, L1CD, and CD
  - Command queue, Event queue, and PRI queue access
  - MSI interrupt writes
- Device-originated transactions
  - Simple Non-coherent devices can be made I/O coherent if the SMMU's interface to the system supports I/O coherency. The cacheability and shareability attributes of the transactions from a client device can be overridden by the SMMU, so the transactions output into the interconnect can snoop the PE caches.

Whether an SMMU supports issuing coherent accesses is indicated by `SMMU_IDR0.COHACC`.


Note

I/O coherent translation table and configuration structure access is required by Arm Base System Architecture.

To support I/O coherent accesses, the SMMU requires an interconnect port that provides the correct coherency guarantees. For example, an ACE-Lite port in an AMBA interconnect can support I/O coherent accesses.

5.2 Client device

This section describes some requirements for a client device, such as:

- Address size
- Cache
- Requirements for a PCIe device or Root Port
- StreamID assignment
- Message Signalled Interrupts

5.2.1 Address size

The architectural input address size of the SMMU is 64 bits. If any of the following occurs:

- A client device outputs an address smaller than 64 bits.
- The interconnect between a client device and the SMMU input supports fewer than 64 bits of address.

The smaller address is converted to a 64-bit SMMU input address in a system-specific manner. The SMMU performs the input range checks on the expanded 64-bit address:

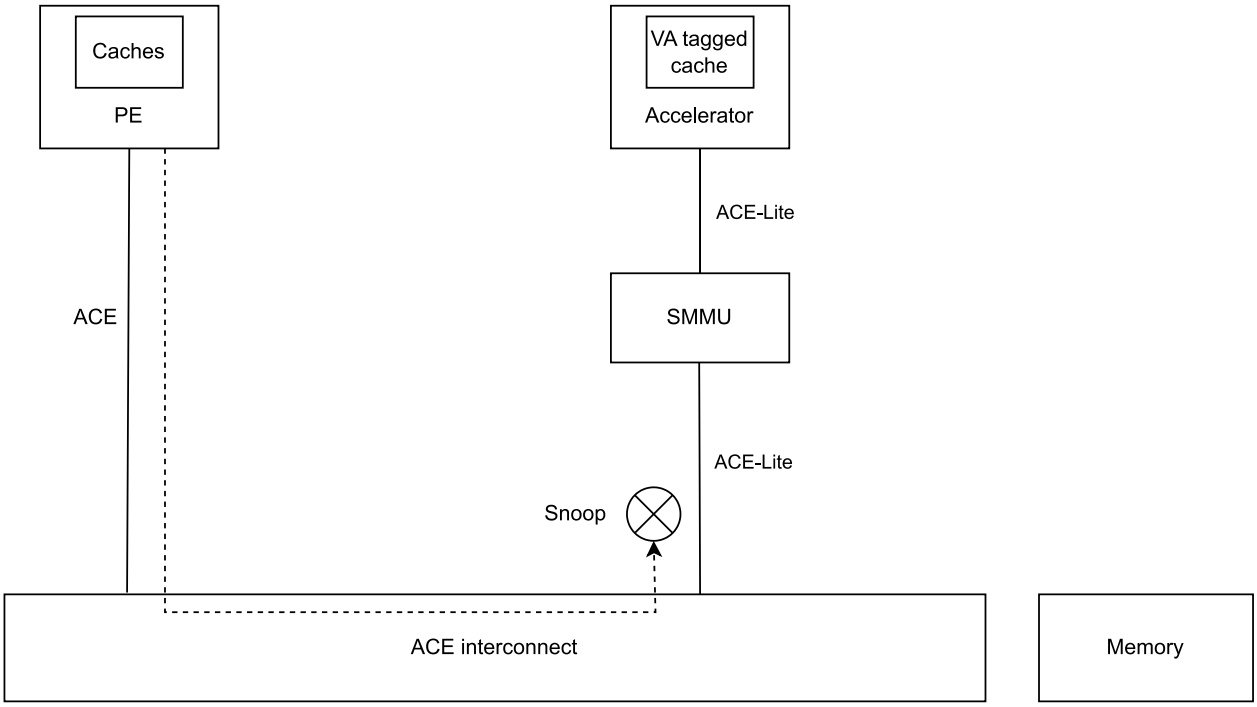
- N is defined as the VA region size which is controlled by `CD.T0SZ` or `CD.T1SZ`, for example 40 bits.
- If Top Byte Ignore, `CD.TBI0` or `CD.TBI1` respectively, is used, VA[55:N-1] are all identical.
- If Top Byte Ignore is not used, VA[63:N-1] are all identical.

5.2.2 Cache

Devices connected behind an SMMU cannot contain caches that are fully-coherent with the rest of the system because snoops relate to physical cache lines, but the SMMU cannot do reverse

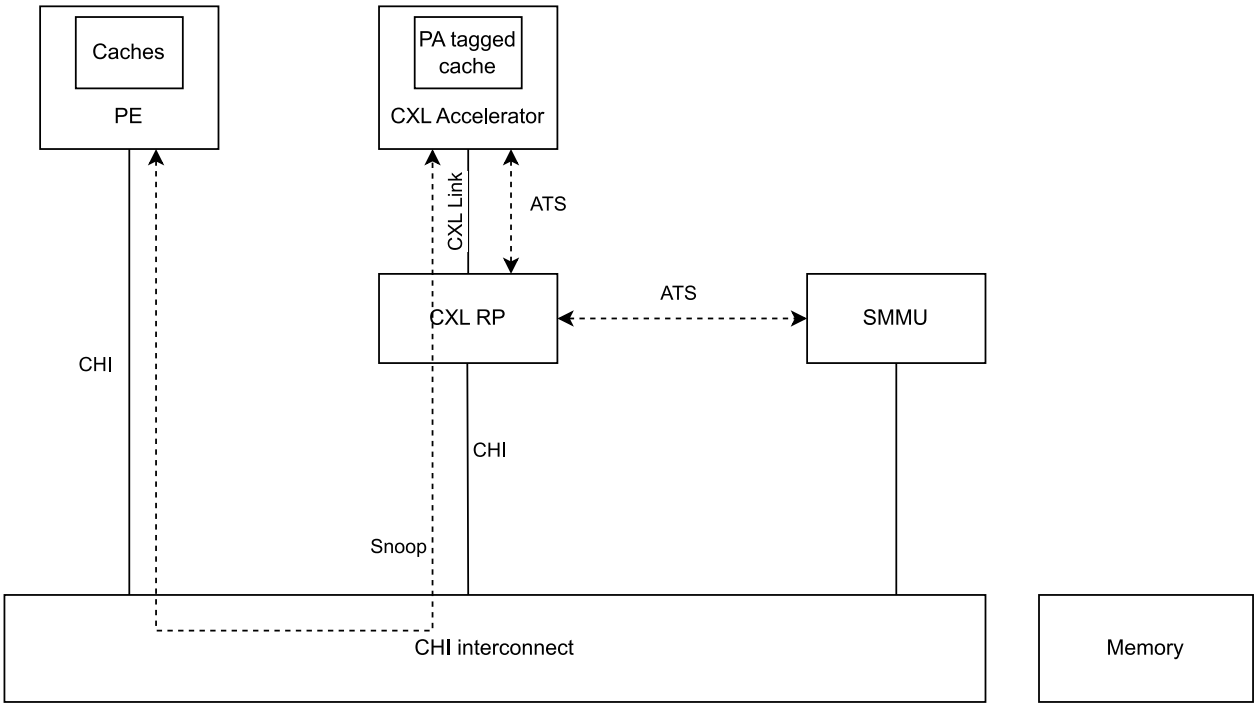
translation from PA to VA. Devices might contain caches that do not support hardware coherency, such caches must be software-maintained.

Figure 5-1: Simplified example of a system topology including an accelerator with a non-coherent cache



However, a client device that contains a TLB filled from the SMMU using ATS might maintain a fully-coherent physically-addressed cache, using the TLBs to translate internal addresses to physical addresses before performing cache accesses. Any physically-addressed caches upstream of the SMMU must be coherent. For example, a CXL.cache or CCIX client device that uses ATS to translate virtual addresses and a coherent physically-addressed cache.

Figure 5-2: Simplified example of a system topology including a CXL accelerator with a coherent cache



5.3 PCIe considerations

When used with a PCIe subsystem, an SMMU implementation must support at least the full 16-bit range of PCIe RequesterIDs. The system must ensure that a Root Port generates StreamIDs from PCI RequesterIDs in a one-to-one or linear fashion so that StreamID[15:0] == RequesterID[15:0]. A larger StreamID might be constructed by concatenating the RequesterIDs from multiple PCI domains (or “segments”), for example:

- StreamID[17:0] == { pci_rp_id[1:0], pci_bus[7:0], pci_dev[4:0], pci_fn[2:0] }; that is, StreamID[17:0] == { pci_domain[1:0], RequesterID[15:0] };

When used with a PCIe system supporting PASIDs, the system must ensure that a Root Port generates SubstreamIDs from PCI PASIDs in a one-to-one fashion. We recommend that the SMMU supports the same number of or fewer PASID bits supported by client Root Ports so that software is able to detect end-to-end SubstreamID capabilities through the SMMU. The motivation is that software can then look at the `SMMU_IDR1.SSIDSIZE` register field and understand how many bits of PASID are available. If the Root Port supports fewer bits than `SMMU_IDR1.SSIDSIZE`, then the software needs to be told this somewhere else, for example firmware tables.

Arm Base System Architecture requires that if the system supports PCIe PASID, then at least 16 bits of PASID must be supported. This support must be full system support, from the root port through to the SMMU for which PASID support is required.

Streams belonging to PCIe endpoints must not be stalled. The Terminate model is the only viable option. Stalling PCIe transactions risks either timeouts from the PCIe endpoint (which might be difficult to recover from), or deadlock in certain scenarios. A system is permitted to enforce the Terminate model for safety reasons. For example, the LTI protocol includes the LAFLOW signals, AXI/ACE-Lite Issue H includes the AxMMUFLOW signals and the DTI protocol includes DTI_TBU_TRANS_REQ.FLOW. These signals and message all support NoStall mode which can enforce the Terminate model. If a translation fault occurs, then even if the SMMU has enabled stall faulting for this translation context, a fault response is returned without depending on the SMMU software configuration.

Specifically, PCIe traffic must not be held up waiting for any PE action, including draining the Event queue or restarting stalled transactions. PCIe traffic must always make forward progress without unbounded delays dependent on software.

5.3.1 Peer-to-peer

PCIe peer-to-peer communication (P2P) enables two PCIe devices to directly transfer data between each other without using host memory as a temporary storage. It is system-specific whether P2P traffic through the system is supported. In a system where the PCIe hierarchy enables P2P communication without going through the SMMU, the SMMU cannot isolate the PCIe devices.

To solve this problem, the PCIe specification includes support for PCIe Access Control Services (ACS):

- With ACS, when a switch port sees a request to a peer switch port, it forwards the P2P request upstream to the Root Port to do a request validation. It checks if the transaction is permitted to be targeted to the peer device as the completer.
- The Root Port makes a decision whether this request can be forwarded to its intended target device.
- The PCIe specification indicates that this decision is made with the help of redirected request validation logic.
- The SMMU is the only agent that can enforce this isolation, so it plays the role of redirected request validation logic.
- A P2P request has an ACS violation error if its SMMU lookup results in an error response. This might be caused by one or more of the following:
    - The request does not have the necessary permissions for accessing the target location.
    - The VA of the request does not have a valid VA to IPA or VA to PA translation existing in the translation table structures corresponding to the requester device context.
    - There is a misconfiguration or some transient error.

5.3.2 No_snoop

PCIe transactions contain a No_snoop attribute. If the No_snoop attribute is set in the PCIe transaction, it indicates that the transaction is allowed to “opt-out” of hardware cache coherency. Software cache coherency ensures the access would not hit in a cache, so the I/O access is

permitted to avoid snooping caches. The memory attributes associated with the transaction must be replaced with Normal-iNC-oNC-OSH.

Support for No_snoop is system-dependent. If it is implemented, No_snoop transforms a final access attribute of a Normal cacheable type to Normal-iNC-oNC-OSH downstream of the SMMU.

5.3.3 ATS

A PCIe Function might determine that caching a translation within its Address Translation Cache is beneficial. A Function or software can consider the following, for example:

- Memory address ranges that are frequently accessed over an extended period of time or whose associated buffer content is subject to a significant update rate
- Memory address ranges such as:
  - Work and completion queue structures
  - Data buffers for low latency communications
  - Graphic frame buffers
  - Host memory that is used to cache Function-specific content

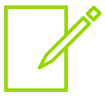
5.4 StreamID assignment

The system designer assigns a requester a unique StreamID to input to the SMMU. The StreamID namespace is per-SMMU, therefore the StreamID has to be unique in each SMMU.

In a Confidential Compute Architecture (CCA) system with SMMU extension for RME DA, a device interface might operate in a trusted or untrusted mode:

- When operating in a untrusted mode, SEC_SID = Non-secure
- When operating in a trusted mode, SEC_SID = Realm

The StreamID presented to the SMMU is the same across the two modes. For example, a PCIe device with RequesterID `0x100` inputs to the SMMU with StreamID `0x100`, but the SEC_SID might be changed because of changes in device configuration.


Note

In Linux, the SMMUv3 driver does not support multiple devices behind the same SMMU using a same StreamID.

For StreamID generation for PCIe devices, see PCIe considerations.

Because the StreamIDs associated with a physical device are system-specific, the system software presents StreamIDs to the OS as part of firmware descriptions for each device. Both the ACPI table and devicetree can present StreamIDs to the OS for each device.

5.5 MSIs

In the Arm GICv3 architecture, the GIC Interrupt Translation Service (ITS) isolates the MSIs. See Locality-Specific Peripheral Interrupts Arm Generic Interrupt Controller v3 and v4.

The ITS receives an MSI write containing EventID and DeviceID input. It uses this information to select the correct PE, or virtual PE, and IRQ number to trigger an interrupt.

The ITS requires the DeviceID input to isolate interrupt sources. The Arm Base System Architecture provides rules for how the DeviceID is generated from the StreamID, which is passed through the SMMU. The simplest way to achieve the same granularity of interrupt source differentiation and SMMU DMA differentiation is for the device's DeviceID to be generated from the device's SMMU StreamID. It is beneficial for high-level software and firmware system descriptions to ensure that this relationship is as simple as possible. DeviceID is derived from a StreamID one-to-one or with a simple linear offset.

The ITS register map provides a page containing one MSI target register. This page can be safely exposed to a device using the SMMU, for example:

- When the device is assigned to a userspace driver, the page can be mapped to the device by the stage 1 translation, so that the userspace driver programs MSIs with a VA destination.
- When the device is assigned to a VM, the page can be mapped to the device by the stage 2 translation, so that the guest OS programs MSIs with an IPA destination.

In a non-CCA system, devices always issue MSIs presented to the SMMU with SEC_SID = Non-secure.

In a CCA system with SMMU for RME DA, a device compatible with the TEE Device Interface Security Protocol (TDISP) might issue MSIs in two ways:

- If the MSI is configured via the MSI capability in configuration space, it is sent to the host SoC with T = 0 and therefore presented to the SMMU with SEC_SID = Non-secure.
- If the MSI is configured via the MSI-X capability in the protected MMIO region of the device interface, it is sent to the host SoC with T = 1 and therefore presented to the SMMU with SEC_SID = Realm.

MSIs from a single device interface are presented to the GIC ITS interface with the same DeviceID regardless of which MSI mechanism is used. The target PA space of an MSI is determined from configuration in translation tables.

6. SMMU use cases

This section describes the following use cases:

- Stage 1 use cases
- Stage 2 use cases

The descriptions about Linux are based on version 6.4 in this section. Some use cases have not been implemented in this version, and may be implemented in later versions, or are under development. The related information is noted in each use case.

6.1 Stage 1 use cases

This section describes use of a single stage 1 translation.

6.1.1 Scatter-gather

Some advanced devices can request DMAs to addresses using scatter-gather lists, which contain several addresses of non-contiguous buffers that form a single request.

However, simple devices usually do not support scatter-gather. They can only request DMAs to a contiguous buffer. When the system runs, physical memory becomes fragmented, so a continuous buffer is not necessarily guaranteed. An OS might have to break up a large range of DMA requests into multiple smaller ranges of DMA requests. This affects performance.

An SMMU can solve this problem. The OS uses the SMMU to present the device with a contiguous buffer, even if the buffer is not physically contiguous. This is done by allocating pages with contiguous virtual addresses in a translation table which maps them to non-contiguous physical addresses. Then the OS programs the DMA address using the virtual address.

In Linux, if a device driver allocates scattered pages using DMA APIs for a device which does not support scatter-gather, the SMMU driver does the “gather” work. This work is transparent to the device driver.

Because this address space is created only for the device, it is unrelated to application process virtual memory address spaces. Broadcast TLB maintenance cannot be used to invalidate SMMU TLBs from the PE. Although the SMMU also uses ASIDs to differentiate different address spaces like the PE, the ASID namespace to the SMMU and the ASID namespace to the PE are independent. To make sure they are in different ASID namespaces, the OS needs to set `CD.ASET = 1` so that SMMU TLB entries created from this context are not expected to be invalidated by some broadcast invalidations.

6.1.2 32-bit devices in a 64-bit OS

Some legacy devices support DMA with 32-bit addresses only. Because an OS might allocate DMA buffers on physical addresses that are beyond the 32-bit range, a solution is necessary to ensure that these legacy devices still work.

The SMMU solves this problem by creating a separate virtual memory map for each 32-bit device, based on its StreamID. This address translation enables the device to issue DMA requests on addresses within the 32-bit range even when the physical address is beyond this range.

In Linux, If both of the following conditions are met.

- A device driver allocates a page using DMA APIs for a device.
- The physical address of the page is beyond 4GB.

The SMMU driver does the address mapping work which maps the IOVA below 4GB to the allocated page beyond 4GB. This work is transparent to the device driver.

Like the previous scatter-gather use case, this address space is created only for the device. It is unrelated to the application process virtual memory address spaces. You cannot use broadcast TLB maintenance to invalidate SMMU TLBs from the PE.

The alternative for supporting 32-bit legacy devices on 64-bit systems is undesirable: the OS maintains a pool of DMA “bounce buffers” within the low address limit, and copies data between a buffer and the original 64-bit location. For example, *SWIOTLB* does this work for systems not having the SMMU or when the SMMU is disabled in Linux. Using the SMMU is significantly faster for large transfers.

6.1.3 OS device isolation

A malicious device might initiate an improper DMA transfer over the interconnect. The DMA transfer could corrupt or retrieve data that does not belong to the malicious device. In systems with an SMMU, a device can only access pages mapped for DMA.

Many operating systems have common APIs that device drivers use when performing DMA transfers. For example, in Linux, these DMA APIs arrange allocations, ensure appropriate address restrictions, and perform cache maintenance operations when using non-cache coherent DMA. These kinds of API can track which pages are accessed by any particular device.

These DMA APIs configure an SMMU to map the correctly-referenced addresses but prevent access to any other addresses. Any programming errors are detected and contained. A malfunctioning device or driver error cannot affect data outside of the set of pages configured for DMA. This can increase system reliability compared to permitting a device to make unrestricted DMA. It is particularly important for security when untrusted or malicious devices might exist. For example they are attached via external interconnects.

In this use case, the SMMU translation table is not shared with PEs. A shared ASID is not necessary.

6.1.4 Userspace device driver

Some systems might require userspace device drivers. Userspace drivers are used for performance, flexibility, robustness, or security. For example, an application might need low-latency access to an accelerator without using a syscall, and avoiding an overhead of copying the data between user-space and kernel-space memory.

To do this, the OS maps device MMIO registers through to a user process virtual address space. If a userspace-controlled device can request DMA, the resulting transactions are at the same level of trust as the user processes. The device must be behind an SMMU. The SMMU ensures that DMA access from the device can only be performed correctly to permitted areas within the user process's address map. The SMMU protects the system from DMA controlled by a malicious or incorrect userspace program.

When a device is assigned to an application, the software allocates special memory regions that DMA targets. The OS then sets up a private translation table used by the SMMU for the device, mapping these memory regions for device access. We recommend for this memory to appear at the same address in process VA space and device address space so that a programmer can configure DMA pointers in an obvious manner, but this is not mandatory.

In Linux, the VFIO subsystem enables device usage from an application. VFIO allows the application to choose an IOVA that is mapped to a chosen memory region in the process, and the application uses the IOVA when programming DMA pointers into the device. The SMMU enables the memory backing the chosen regions of the process to be accessible by the device's DMA. The device and PE share data through that memory. The SMMU prevents the device from accessing any other parts of the process address space, or that of other processes.

Because userspace memory is usually demand-allocated and swapped, page faults on DMA must be considered. In many systems, a translation-related fault is fatal to DMA transfers. Many device drivers avoid translation-related faults by pinning pages that are the target of DMA. Because DMA occurs only to specific memory regions, and these regions are often small, pinning the memory is a reasonable requirement.

6.1.5 Userspace Shared Virtual Addressing (SVA)

The use case Userspace device driver presented a userspace device driver, with some limitations: the SMMU translation tables must map special DMA regions in the address space of the userspace device driver.

This use case considers the alternative of sharing the entire user process address space with a device, instead of specific regions. Sharing process address spaces with devices relies on CPU kernel memory management for DMA. It removes some complexity from application and device drivers. The device MMIO registers are virtually-mapped by PE in the same way as in Userspace device driver. However, now the SMMU associates the device with the whole process address space instead of a subset. The device is required to be physically capable of accessing any userspace virtual address. For example, SVA cannot be achieved with a device only capable of using 32-bit addresses and a process using a 48-bit virtual address.

To achieve a common address space, the SMMU can directly use the process translation table and ASID. Because a common ASID is used, and virtual addresses have the same meaning on both the PE and SMMU, broadcast TLB maintenance can be used to invalidate SMMU translations from the PE. We recommend broadcast invalidates as a potentially slow round-trip to the SMMU for manual TLB invalidation commands can be avoided. There might also be software simplifications from using a common invalidation approach.

It is impractical to pin every single page in case the process tries to program DMA to them. DMA might occur to any address, whether the page table has mapped it or not, so the device and system must support page faults on DMA access. This means that:

- The device DMA stream must not be fatally aborted when a page fault is detected.
- A mechanism is required to inform the OS of the following to resume the DMA transfer.
    - A fault has occurred.
    - Action is required to make the required page accessible.

Both the SMMU and device cooperate to support the requirement for page faults on DMA to be non-fatal. For example, a device DMA stream might be configured in the SMMU to stall on a fault. When a fault occurs, the SMMU reports a fault descriptor to the software and the faulting transaction is stalled until the page fault is resolved. The software issues commands to the SMMU to retry the transaction, or abort the transaction if the fault was caused by an incorrect access. A device experiences a small delay in the response to its DMA request.

Alternatively, a more intelligent device might be able to efficiently schedule traffic if it is informed of faults. For example, the SMMU is configured to abort the faulting transaction and the device is informed of the error. The device reports the event to its device driver, which resolves the cause of the page fault. This method has the advantage of avoiding stalling traffic in the SMMU, which might affect other transactions. However, using this method requires device-specific software code. For example, Linux does not support SVA for non-PCIe devices that only support aborting the faulting transaction.

A similar method is used by PCIe PRI, which reports faults back to the device in an ATS response. PRI provides a standard Page Request Interface which enables a device to request that software resolve a page fault, via the Root Port and SMMU. PRI does not require a device-specific page fault-resolution routine. This helps to reduce device-specific software requirements in standard software such as hypervisors.

In Linux, the VFIO subsystem provides an interface to enable SVA for an application. To support SVA, the device, buses and the SMMU must support the following features:

- Multiple address spaces per device, for example using the PCI PASID extension. The SMMU driver allocates a PASID and the device uses it in DMA transactions.
- I/O Page Faults.
    - For PCIe devices, PRI is required.
    - For non-PCIe devices, the Stall model is requred.
    - The core memory management handles translation faults from the SMMU.
- MMU and SMMU implement compatible page table formats.

6.2 Stage 2 use cases

This section describes use of stage 2 translation.

6.2.1 VM device assignment

A hypervisor can assign a device to an untrusted VM, and configure the SMMU to protect the rest of the system from the device controlled by the VM.

PE stage 2 translations map device MMIO registers into IPA space, as if the guest OS is running on the physical machine. Stage 2 translations in the SMMU map the guest physical memory of the VM through to the device for DMA. The guest OS seems to run on the physical machine with a directly-connected device. The guest programs the device with guest physical addresses (IPAs) and there is no stage 1 translation. Device assignment enables the device to DMA to any guest memory, just like a directly-connected device on a physical machine.

Either the PE stage 2 translation table is shared directly with the SMMU, or a separate translation table used by the SMMU is kept in sync with the PE tables. The same IPA address space is used by SMMUs and PEs.

A device might not be tolerant of DMA page faults, including when it is assigned a device to a VM. Many hypervisors avoid page faults on DMA by pinning all VM memory if there are any devices assigned to a guest.

In Linux, by leveraging the VFIO subsystem in the host kernel, a device can be exclusively managed by a userspace Virtual Machine Manager (VMM) like QEMU. In the VM with assigned device, the VM should be able to see exactly the same device just like in the host. By assigning the device to a guest, you can have almost the same performance in guest comparing to in the host.

For this use case, a separate translation table is used by the SMMU in Linux. However, the IPA-to-PA mappings are the same with the stage 2 translation table for the VM. The stage 1 translation is used for the SMMU by default, although logically it is stage 2 translation. A VMM like QEMU can opt to use the stage 2 translation but generally use the stage 1 translation. It makes no difference because the IPA-to-PA mappings are the same.

6.2.2 VM device assignment with guest OS SMMU usage

If a hypervisor assigns a device to a VM, it might also give the appearance of an SMMU for use by the guest, so that the VM and assigned device can also enjoy the benefits of Stage 1 use cases. To implement this, assign a software-emulated virtual SMMU, vSMMU for short, to the VM. Interactions with a vSMMU are translated by the hypervisor into actions performed with a real SMMU programming interface.

The guest OS can then use any of the Stage 1 use cases, programming configuration and translation tables into its VM-assigned SMMU interface. The hypervisor-maintained stage 2

translation tables nest underneath the VM-maintained stage 1 translation tables. For example, if the guest OS uses a userspace device driver, the stage 1 translation tables protect the IPA space from the guest's userspace process and the hypervisor's stage 2 translation tables protect the host's physical address space from any guest-controlled DMA.

This scenario can also be supported with only one actual stage of translation and software shadowing of translation tables. The hypervisor observes the guest configuration and translation table pointers programmed into the vSMMU interface. The hypervisor combines the guest stage 1 translation table contents with the hypervisor's stage 2 tables into a single hypervisor-owned shadow translation table for the single stage of translation. Guest invalidation commands sent to the vSMMU trigger the re-shadowing of guest translation table entries.

In Linux, nesting stage 1 translation and stage 2 translation is not supported at the moment. However, like VM device assignment, it still requires pinning all guest memory to populate the stage 2 tables before VM boots. A shadow translation table is currently supported, but it does not use a vSMMU in the VM. Instead it uses a paravirtual IOMMU or virtio-iommu. The virtio-iommu enables a guest to communicate IOVA-to-IPA mappings to the host using a map/unmap interface. The host installs IOVA-to-PA mappings in the SMMU on behalf of the guest. The virtio-iommu is implemented by Linux and several VMMs such as QEMU, cloud-hypervisor, and crosvm. The main use case for introducing it was userspace device drivers in the guest.

6.2.3 Media protection

Some protected media solutions use an SMMU to protect specific media buffers which can be carve-out memory or delivered by the untrusted OS. This is achieved by the devices, such as a crypto engine and a video decoder, sharing the IPA memory view of the untrusted OS except for the protected media buffers, which are made invisible or unaccessible to the untrusted OS. This is achieved by using more restrictive stage 2 permissions for the untrusted OS compared to those used with the devices. This use case does not actually require the translation. That is, the stage 2 can be identity-mappings. The configuration of the buffers is performed with cooperation between the firmware (Protected Media Manager) and untrusted OS.

The firmware is responsible for creating the stage 2 translation tables for the untrusted OS and devices and ensuring that the configuration of the system is at all times compliant with a pre-defined policy. The policy specifies a set of constraints, including whether a given device in the system should be allowed to access a particular memory location. There are different stage 2 translation tables for the untrusted OS and devices. The SMMU stage 2 translation table maps the pre-arranged protected media buffer pages, giving access to the allowed devices. However, the IPA space of the untrusted OS might not map pre-arranged protected media buffer pages, or might map them with reduced permissions, so that access to the buffers by the untrusted OS is restricted.

6.2.4 Host isolation

The pKVM hypervisor provides a separation of privileges between the host and hypervisor parts of KVM, where the hypervisor is trusted by guests but the host is not. The host is initially trusted

during boot, but its privileges are reduced after KVM is initialized, so that if an adversary later gains access to the large attack surface of the host, it cannot access guest data.

Currently with pKVM, the host can still instruct DMA-capable devices like the GPU to access guest and hypervisor memory, which undermines this isolation. Preventing DMA attacks requires an SMMU, owned by the hypervisor.

The simplest way to achieve host isolation is sharing the stage 2 page tables, which translate IPAs to PAs for the PE and SMMU. Whatever the host can access on the CPU, it can also access with DMA. Memory that is not accessible to the host because it was delivered to the hypervisor or guests, cannot be accessed by the DMA device either.

This mechanism has some restrictions. pKVM normally populates the host stage 2 page tables lazily. That is, the host stage 2 page tables are not populated until the host first accesses them. This relies on CPU page faults. However, DMA generally cannot fault because Linux does not support the stage 2 SVA currently. The entire set of stage 2 page tables must therefore be populated at boot. This is easy to do because the IPA-to-PA mapping is flat.

It gets more complicated when delivering some pages to guests, when this involves removing those pages from the host stage 2 page tables. To save memory and use TLBs efficiently, the stage 2 translation is mapped with block mappings, such as 1GB or 2MB blocks, rather than individual 4KB units. When donating a page from that range, the hypervisor must remove the block mapping, and replace it with a table that excludes the donated page. Because a device might be simultaneously performing DMA on other pages in the range, this replacement operation must be atomic. Otherwise the DMA might reach the SMMU during a small period of time when the mapping is invalid, and fatally abort. The CPU/SMMU architecture supports atomic replacement of block mappings only when FEAT_BBM is supported. See Arm Architecture Reference Manual for A-profile architecture for more information about FEAT_BBM.

A flexible alternative uses private page tables in the SMMU, entirely disconnected from the CPU page tables. With this, the SMMU can implement a reduced set of features, or even implement only one stage of translation. This also provides a virtual I/O address space to the host, which enables more efficient memory allocation for large buffers, and for devices with limited addressing abilities.

7. Related information

Here are some resources related to material in this guide:

- Arm System Memory Management Unit Architecture Specification version 3 provides a complete specification of the SMMUv3 architecture.
- Arm Architecture Reference Manual for A-profile architecture provides a complete specification of VMSA.
- AArch64 memory management provides information about the AArch64 memory management.
- AArch64 memory attributes and properties provides information about the AArch64 memory attributes and properties.
- Learn the architecture - Realm Management Extension provides information about RME.
- PCI Express Base Specification provides a complete specification of the PCIe protocol.
- Learn the architecture - Memory System Resource Partitioning and Monitoring (MPAM) Overview provides information about MPAM.
- Arm Base System Architecture describes the hardware requirements for the SMMU and client devices in a BSA-compliant system.
- Locality-Specific Peripheral Interrupts Arm Generic Interrupt Controller v3 and v4 provides information about ITS.