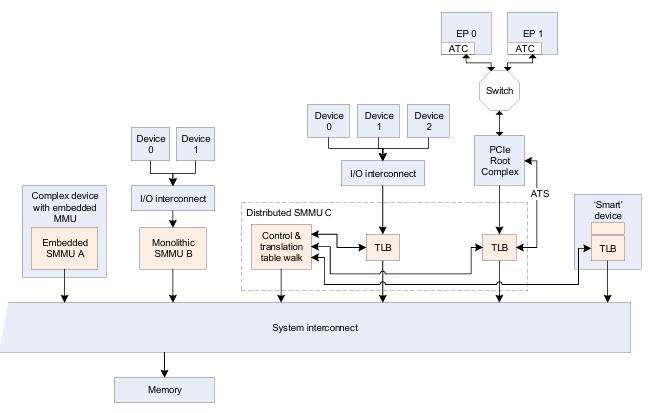


One SMMU interfaces incoming traffic from two client devices to the system interconnect. The devices can perform DMA using virtual, IPA or other bus address schemes and the SMMU translates these addresses to PAs. The second example SMMU interfaces one to one to a PCIe Root Complex (which itself hosts a network of endpoints). This illustrates an additional interface specified in this document, an ATS port to support PCIe ATS and PRI (or similar functionality for compatible non-PCIe devices).

In general, Requesters are behind an SMMU (or, in the case of PEs, have an inbuilt MMU), so outgoing accesses to the system interconnect and Completer devices are mediated by the MMU of the Requester. If a Requester has no MMU, it has full-system access. Therefore, its DMA must be mediated by software, and in this case only the most privileged system software can program it.

The SMMU interface to the system interconnect is intended to be IO-coherent, thereby providing IO-coherent access for the client devices of the SMMU. The SMMU interface for traffic incoming from client devices does not require any coherency support. In addition, because there is no address translation in the outgoing direction, snoop traffic cannot be forwarded from the system towards the client devices so fully-coherent device caches cannot be placed behind an SMMU. Note: It is feasible to implement an SMMU as part of a complex device containing fully-coherent caches in the same way that the MMU of a PE is paired to fully-coherent PE caches. Practically, this means the caches must be tagged with physical addresses.



The SMMU has three interfaces that software uses:

1. Memory-based data structures to map devices to translation tables which are used to translate client device addresses.

2. Memory-based circular buffer queues. These are a Command queue for commands to the SMMU, an Event queue for event/fault reports from the SMMU, and a PRI queue for receipt of PCIe page requests. Note: The PRI queue is only present on SMMUs supporting PRI services. This additional queue allows processing of PRI requests from devices separate from event or fault reports.

3. Aset of registers, for each supported Security state, for discovery and SMMU-global configuration. The registers indicate the base addresses of the structures and queues, provide feature detection and identification registers and a global control register to enable queue processing and translation of traffic. When Secure state is supported, an additional register set exists to allow Secure software to maintain Secure device structures, issue commands on a second Secure Command queue and read Secure events from a Secure Event queue.

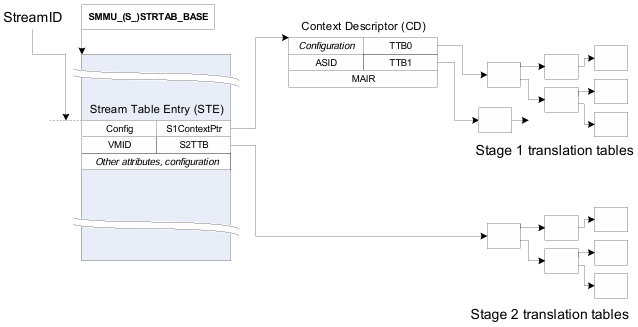
An incoming transaction has an address, size, and attributes such as read/write, Secure/Non-secure, Shareability, Cache ability. If more than one client device uses the SMMU traffic must also have a sideband StreamID so the sources can be differentiated. How a StreamID is constructed and carried through the system is IMPLEMENTATION DEFINED. Logically, a StreamID corresponds to a device that initiated a transaction.

Arm recommends that StreamID be a dense namespace starting at 0. The StreamID namespace is per-SMMU. Devices assigned the same StreamID but behind different SMMUs are seen to be different sources. A device might emit traffic with more than one StreamID, representing data streams differentiated by device-specific state.

Another property, SubstreamID, might optionally be provided to an SMMU implementing stage 1 translation. The SubstreamID is of IMPLEMENTATION DEFINED size, between 0 and 20 bits, and differentiates streams of traffic originating from the same logical block in order to associate different application address translations to each. Note: An example would be a compute accelerator with 8 contexts that might each map to a different user process, but where the single device has common configuration meaning it must be assigned to a VM whole. Note: The SubstreamID is equivalent to a PCIe PASID. Because the concept can be applied to non-PCIe systems, it has been given a more generic name in the SMMU. The maximum size of SubstreamID, 20 bits, matches the maximum size of a PCIe PASID. The incoming transaction flags whether or not a SubstreamID is supplied and this might differ on a per-transaction basis.

The SMMU optionally supports Secure state and, if supported, the StreamID input to the SMMU is qualified by a SEC\_SID flag that determines whether the input StreamID value refers to the Secure or Non-secure StreamID namespace. A Non-secure StreamID identifies an STE within the Non-secure Stream table and a Secure StreamID identifies an STE within the Secure Stream table.

Arm expects that, for PCI, StreamID is generated from the PCI RequesterID so that StreamID[15:0] == RequesterID[15:0]. When more than one Root Complex is hosted by one SMMU, Arm recommends that the 16-bit RequesterID namespaces are arranged into a larger StreamID namespace by using upper bits of StreamID to differentiate the contiguous RequesterID namespaces, so that StreamID[N:16] indicates which Root Complex (PCIe domain/segment) is the source of the stream source. In PCIe systems, the SubstreamID is intended to be directly provided from the PASID [1] in a one to one fashion. Therefore, for SMMU implementations intended for use with PCI clients, supported StreamID size must be at least 16 bits.



The SMMU uses a set of data structures in memory to locate translation data. Registers hold the base addresses of the initial root structure, the Stream table.

A Stream Table Entry (STE) contains stage 2 translation table base pointers, and also locates stage 1 configuration structures, which contain translation table base pointers.

A Context Descriptor (CD) represents stage 1 translation, and a Stream Table Entry represents stage 2 translation.

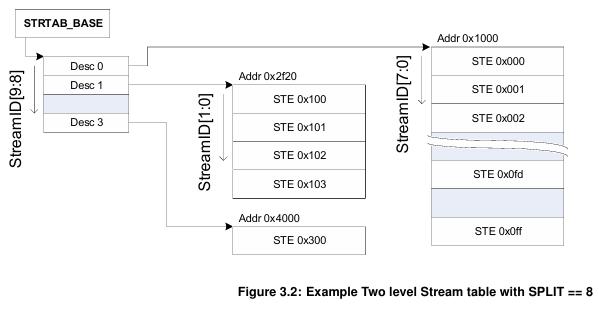
Therefore, there are two distinct groups of structures used by the SMMU:

* Configuration structures, which map from the StreamID of a transaction (a device originator identifier) to the translation table base pointers, configuration, and context under which the translation tables are accessed.
* Translation table structures that are used to perform the VA to IPA and IPA to PA translation of addresses for stage 1 and stage 2, respectively.

The procedure for translation of an incoming transaction is to first locate configuration appropriate for that transaction, identified by its StreamID and, optionally, SubstreamID, and then to use that configuration to locate translations for the address used.

The first step in dealing with an incoming transaction is to locate the STE, which tells the SMMU what other configuration it requires. Conceptually, an STE describes configuration for a client device in terms of whether it is subject to stage 1 or stage 2 translation or both.

Multiple devices can be associated with a single Virtual Machine, so multiple STEs can share common stage 2 translation tables. Similarly, multiple devices (strictly, streams) might share common stage 1 configuration, therefore multiple STEs could share common CDs.



A 2-level Stream table is a structure consisting of one top-level table that contains descriptors that point to multiple second-level tables that contain linear arrays of STEs. The span of StreamIDs covered by the entire structure is configurable up to the maximum number supported by the SMMU but 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.

The STE contains the configuration for each stream indicating:

* Whether traffic from the device is enabled.
* Whether it is subject to stage 1 translation.
* Whether it is subject to stage 2 translation, and the relevant translation tables.
* Which data structures locate translation tables for stage 1.

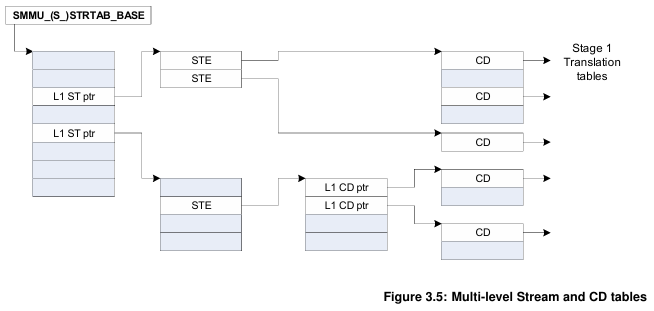
If stage 1 is used, the STE indicates the address of one or more CDs in memory using the STE.S1ContextPtr field. The CDassociates the StreamID with stage 1 translation table base pointers (to translate VA into IPA), per-stream configuration, and ASID.

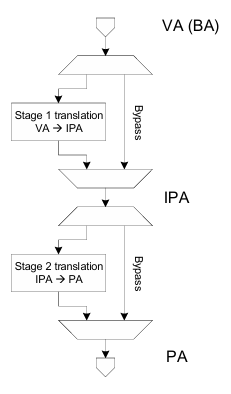
If substreams are in use, multiple CDs indicate multiple stage 1 translations, one for each substream. Transactions provided with a SubstreamID are terminated when stage 1 translation is not enabled.

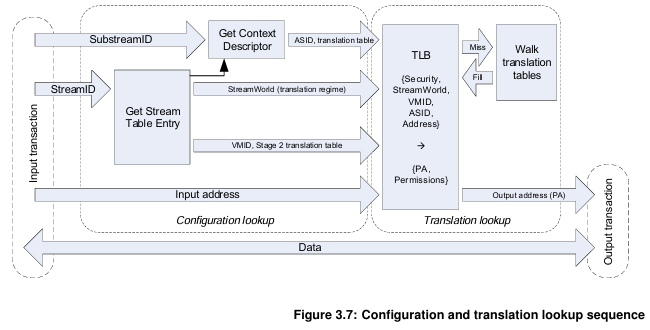
If stage 2 is used, the STE contains the stage 2 translation table base pointer (to translate IPA to PA) and VMID. If multiple devices are associated with a particular virtual machine, meaning they share stage 2 translation tables, then multiple STEs might map to one stage 2 translation table.

Note: Arm expects that, where hypervisor software is present, the Stream table and stage 2 translation table are managed by the hypervisor and the CDs and stage 1 translation tables associated with devices under guest control are managed by the guest OS. Additionally, the hypervisor can make use of separate hypervisor stage 1 translations for its own internal purposes. Where a hypervisor is not used, a bare-metal OS manages the Stream table and CDs.

The ASID and VMID values provided by the CD and STE structures tag TLB entries created from translation lookups performed through configuration from the CD and STEs. These tags are used on lookup to differentiate translation address spaces between different streams, or to match entries for invalidation on receipt of broadcast TLB maintenance operations. Implementations might also use these tags to efficiently allow sharing of identical translation tables between different streams







Above figure does not show error reporting paths or CD fetch through stage 2 translation (which would also access the TLB or translation table walk facilities). An implementation might choose to flatten or combine some of the steps shown, while maintaining the same behaviour.

An incoming transaction is first subject to a configuration lookup, and the SMMU determines how to begin to translate the transaction.

This involves locating the appropriate STE then, if required, a CD. The configuration lookup stage does not depend on the input address and is a function of the:

* SMMU global register configuration.
* Incoming transaction StreamID.
* Incoming transaction SubstreamID (if supplied).

The result of the configuration lookup is the stream or substream-specific configuration that locates the translation, including:

* Stage 1 translation table base pointers, ASID, and properties modifying the interpretation or walk of the translation tables (such as translation granule).
* Stage 2 translation table base pointer, VMID and properties modifying the interpretation or walk of the translation table.
* Stream-specific properties, such as the StreamWorld (the Exception Level, or translation regime, in PE terms) to which the stream is assigned.

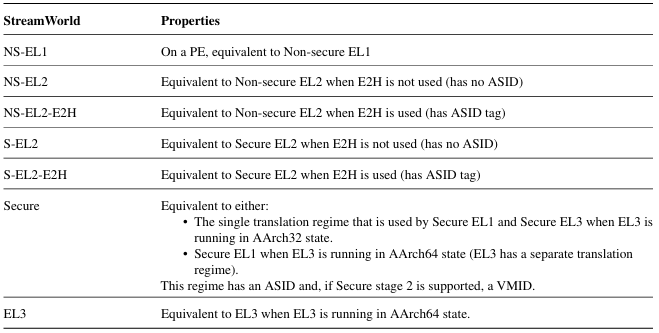
The translation lookup stage logically works the same way as a PE memory address translation system. The output is the final physical address provided to the system, which is a function of the:

* Input address
* StreamWorld (Stream Security state and Exception level), ASID and VMID (which are provided from the previous step).

For example, the following are unique and can all co-exist in a translation cache:

* Entries with the same address, but different ASIDs.
* Entries with the same address and ASID, but different VMIDs.
* Entries with the same address and ASID but a different StreamWorld

In addition to an address, size and read/write attributes, an incoming transaction might be presented to the SMMU with other attributes, such as an access type (for example Device, WB-cached Normal memory), Shareability (for example Outer Shareable), cache allocation hints, and permissions-related attributes, instruction/data, privileged/unprivileged, Secure/Non-secure. Some of these attributes are used to check the access against the page permissions that are determined from the translation tables. After passing through the SMMU, a transaction presented to the system might also have a set of attributes, which might have been affected by the SMMU.



Permission-related attributes (instruction/data, privileged/unprivileged) and read/write properties are used for checking against translation table permissions, which might deny the access. Other attributes (memory type, Shareability, cache hints) are intended to have an effect on the memory system rather than the SMMU, for example, control cache lookup for the transaction.

There are three address size concepts to consider in the SMMU, the input address size from the system, the Intermediate Address Size (IAS), and the Output Address Size (OAS):

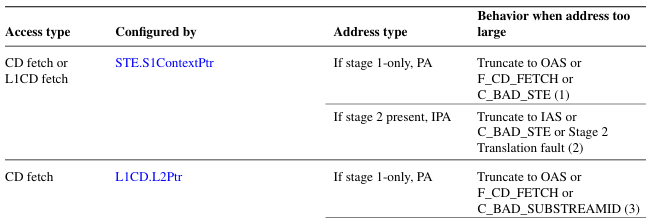
* The SMMU input address size is 64 bits.
* IAS reflects the maximum usable IPA of an implementation that is generated by stage 1 and input to stage 2.
* OAS reflects the maximum usable PA output from the last stage of AArch64 translations, and must match the system physical address size. The OAS is discoverable from SMMU\_IDR5.OAS. Final-stage AArch32 translations always output 40 bits which are zero-extended into a larger OAS, or truncated to a smaller OAS.

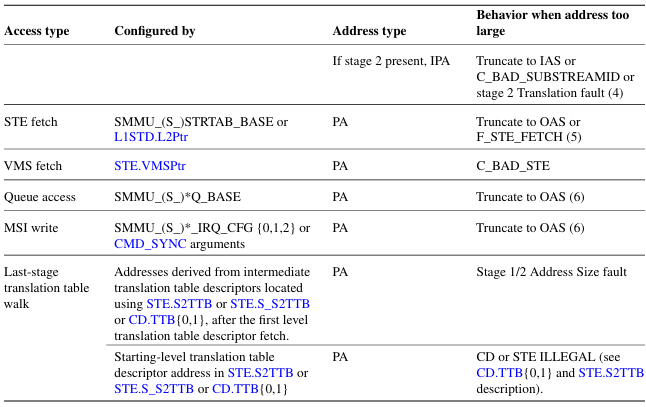
On input, a stage 1 Translation fault (F\_TRANSLATION) occurs if the VA is outside the range specified by the relevant CD.

The SMMU architecture does not check the alignment of incoming transaction addresses.

Distinct from client device accesses forwarded into the system, the SMMU originates accesses to the system for the purposes of:

* Configuration structure access (STE, CD).
* Queue access (Command, Event, PRI).
* MSI interrupt writes.
* Last-stage translation table walks:– Note: Addresses output from stage 1 walks in a nested configuration are input to stage 2 and translated in the expected manner (including causing stage 1 Address Size faults, or stage 2 Translation faults from IPAs outside the stage 2 translation range), rather than being output into the system directly.





All SMMU queues for both input to, and output from the SMMU are arranged as circular buffers in memory. A programming interface has one Command queue for input and one Event queue (and optionally one PRI queue) for output. Each queue is used in a producer-consumer fashion so that an output queue contains data produced by the SMMU and consumed by software. An input queue contains data produced by software, consumed by the SMMU.

A queue is arranged as a 2n-items sized circular FIFO with a base pointer and two index registers, PROD and CONS indicating the producer and consumer current positions in the queue. In each of the output and input roles, only one index is maintained by the SMMU, with the other is maintained by software.

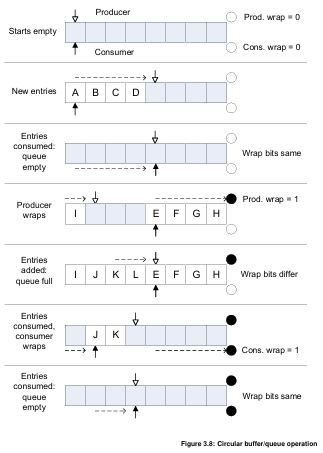
For an input queue (Command queue), the PROD index is updated by software after inserting an item into the queue, and is read by the SMMU to determine new items. The CONS index is updated by the SMMU as items are consumed, and is read by software to determine that items are consumed and space is free. An output queue is the exact opposite.

PROD indicates the index of the location that can be written next, if the queue is not full, by the producer. CONS indicates the index of the location that can be read next, if the queue is not empty. The indexes must always increment and wrap to the bottom when they pass the top entry of the queue.

Each index has a wrap flag, represented by the next higher bit adjacent to the index value contained in PROD and CONS. This bit must toggle each time the index wraps off the high end and back onto the low end of the buffer. It is the responsibility of the owner of each index, producer or consumer, to toggle this bit when the owner updates the index after wrapping. It is intended that software reads the register, increments or wraps the index (toggling wrap when required) and writes back both wrap and index fields at the same time. This single update prevents inconsistency between index and wrap state.

* If the two indexes are equal and their wrap bits are equal, the queue is empty and nothing can be consumed from it.
* If the two indexes are equal and their wrap bits are different, the queue is full and nothing can be produced to it.
* If the two indexes differ or the wrap bits differ, the consumer consumes entries, incrementing the CONS index until the queue is empty (both indices and wrap bits are equal).

Therefore, the wrap bits differentiate the cases of an empty buffer and a full buffer where otherwise both indexes would indicate the same location in both full and empty cases.



Any producer (whether the SMMU or software) must ensure that if an update to the PROD index value is observable by the consumer, all new queue entries are observable by the consumer. For output queues from the SMMU (Event and PRI queues), the SMMU writes queue data to memory and, when that data becomes visible with respect to the rest of the Shareability domain, the SMMU allows the updated PROD index value to be observed. This is the first point that a new queue entry is visible to the consumer.

A consumer must not assume presence of a new valid entry in a queue through any mechanism other than having first observed an updated PROD index that covers the entry position. If a consumer reads a queue entry beyond the point indicated by the last read of the PROD index, the entry contains UNKNOWN data.

The SMMU might support configurable behaviour on Translation-related faults, which enable a faulting transaction to be stalled, pending later resolution, or terminated which immediately aborts the transaction.

Events are recorded into the Event queue in response to a configuration error or translation-related fault associated with an incoming transaction. A sequence of faults or errors caused by incoming transactions could 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, but 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.

Implementations are permitted to merge some event records together. This might happen where multiple identical events occurred, and can be used to reduce the volume of events recorded into the Event queue where individual events do not supply additional useful information.

Events can be merged where all of the following conditions are upheld:

* The event types and all fields are identical, except fields explicitly indicated in section 7.3. Event records.
* If present, the Stall field is 0. Stall fault records are not merged

An SMMU can implement multiple Command queues for the Non-secure or Secure SMMU programming interfaces.

Arm expects that the Non-secure Stream table, Command queue, Event queue and PRI queue are controlled by the most privileged Non-secure system software.

If present, Arm expects that the Secure Stream table, Secure Command queue and Secure Event queue are controlled by Secure software.

For example, these would be controlled by software in EL3 if a separation in control between Secure EL1 and EL3 is required. Arm expects that the stage 2 translation tables indicated by all STEs are controlled by a hypervisor.

The ownership of stage 1 CDs and translation tables depends on the configuration in use. If pointed to by a Secure STE, they are controlled by Secure software (one of EL3, S-EL2 or S-EL1). If pointed to by a Non-secure STE, they are controlled by Non-secure software (either NS-EL2 or NS-EL1).

Note: For example, the context might be one of the following:

* Used by a bare-metal OS, which controls the descriptor and translation tables and is addressed by PA.
* Used internally by a hypervisor, which controls the descriptor and translation tables and is addressed by PA.
* Used by a guest, in which case Arm expects that the CD and translation tables are controlled by the guest, and addressed by IPA.

Note: When a hypervisor is used in a given Security state, Arm expects that the Event queue for that Security state is managed by the hypervisor, which forwards events into guest VMs as appropriate. StreamIDs might be mapped from physical to virtual equivalents during this process. In virtualized scenarios, Arm expects a hypervisor to:

* Convert guest STEs into physical SMMU STEs, controlling permissions and features as required. Note: The physical StreamIDs might be hidden from the guest, which would be given virtual StreamIDs, so a mapping between virtual and physical StreamIDs must be maintained.
* Read and interpret commands from the guest Command queue. These might result in commands being issued to the SMMU or invalidation of internal shadowed data structures.
* Consume new entries from the PRI and Event queues, mapping from host StreamIDs to guest, and deliver appropriate entries to guest Event and PRI queues.

The SMMU registers occupy a set of contiguous 64K pages of system address space that contain mechanisms for discovering capabilities and configuring pointers to in-memory structures and queues. After initialization, runtime access to the registers is generally limited to maintenance of the Command, Event and PRI queue pointers and interaction with the SMMU is performed using these in-memory queues.

Devices can be put under direct guest control with stage 2-only mappings, without requiring guest interaction with the SMMU. To the guest OS, they appear as though the device is directly connected and might request DMA to PAs (IPAs) directly.

The SMMU does not provide programming interfaces for use directly by virtual machines. Arm expects that, where stage 1 facilities are required for use by a guest in virtualization scenarios, this is supported using hypervisor emulation of a virtual SMMU, or a similar interface for use by a virtual machine.

SMMU versions:

Devices can be put under direct guest control with stage 2-only mappings, without requiring guest interaction with the SMMU. To the guest OS, they appear as though the device is directly connected and might request DMA to PAs (IPAs) directly. The SMMU does not provide programming interfaces for use directly by virtual machines.

Arm expects that, where stage 1 facilities are required for use by a guest in virtualization scenarios, this is supported using hypervisor emulation of a virtual SMMU, or a similar interface for use by a virtual machine.