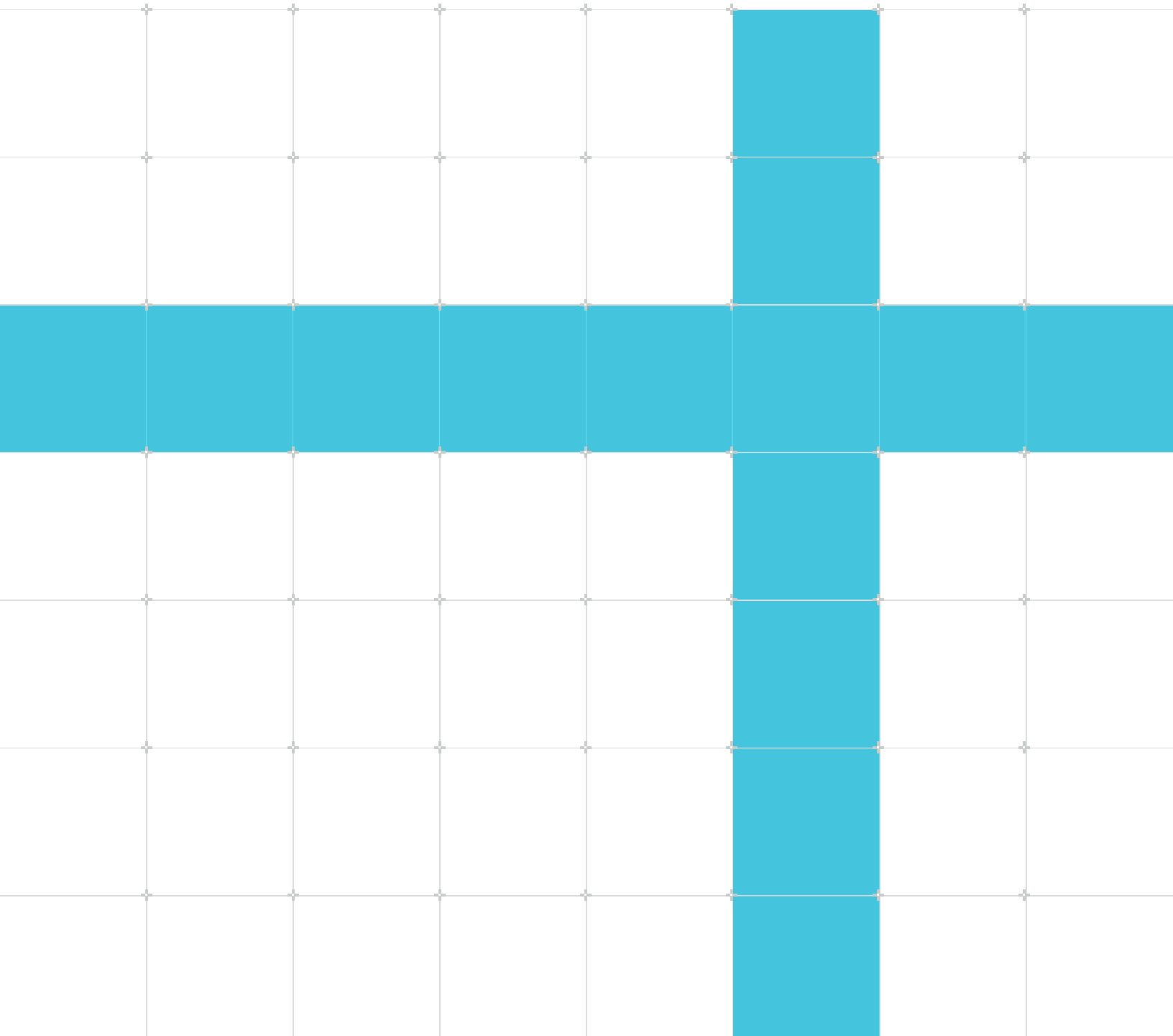
Learn the architecture - Realm Management Extension

Version 1.0

Issue 02
den0126_0100_02_en



Learn the architecture - Realm Management Extension

Release information

Document history

Issue	Date	Confidentiality	Change
0100-02	23 June 2021	Non-Confidential	First 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, has undertaken no analysis to identify or understand the scope and content of, third party 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)

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 introduces the Realm Management Extension (RME), an extension to the architecture.

RME is the hardware component of the Arm Confidential Compute Architecture (Arm CCA) which also includes software elements. RME dynamically transfers resources and memory to a new protected address space that higher privileged software or TrustZone firmware cannot access. Because of this address space, Arm CCA constructs protected execution environments called realms.

Realms allow a lower-privileged software, like an application or a Virtual Machine (VM), to protect its content. Realms also prevent execution from attacks using software that runs at higher privilege levels, like an OS or a hypervisor. Higher-privileged software is still responsible for allocating and managing the resources that a realm uses. However, this higher-privileged software cannot access the contents of the realm or affect its execution flow.

RME can also dynamically transfer memory to a protected address space for realms. With RME, the memory available to TrustZone Software entities can be varied dynamically.

This guide describes the key hardware features that RME introduces or changes and introduces you to the software architecture.

You will learn about the following concepts:

- Understand the new Security states and Physical Address (PA) spaces in systems with RME
- Describe how a region of memory can be dynamically assigned between PA spaces
- Understand the system requirements for an RME-enabled system

This guide explains the following changes that RME introduces to the processor architecture:

- Additional Security states
- Additional Physical addresses
- Support for Granule Protection Checks, which allow granules of memory to be dynamically assigned to a physical address space


Note

Diversity and inclusion are important values to Arm. Because of this, we are re-evaluating the terminology we use in our documentation. Older Arm documentation uses the terms master and slave.

This guide uses replacement terminology, as follows:

- The new term Requester is synonymous with master in older documentation
- The new term Subordinate is synonymous with slave in older documentation

Before you begin

We assume that you are familiar with the AArch64 Exception model, AArch64 memory management, and TrustZone. To learn more about these topics, see the following guides:

- AArch64 Exception model introduces the exception and privilege model in AArch64
- AArch64 memory management introduces the MMU, which is used to control virtual to physical address translation
- TrustZone for AArch64 introduces TrustZone, an efficient, system-side approach to security with hardware-enforced isolation built into the CPU

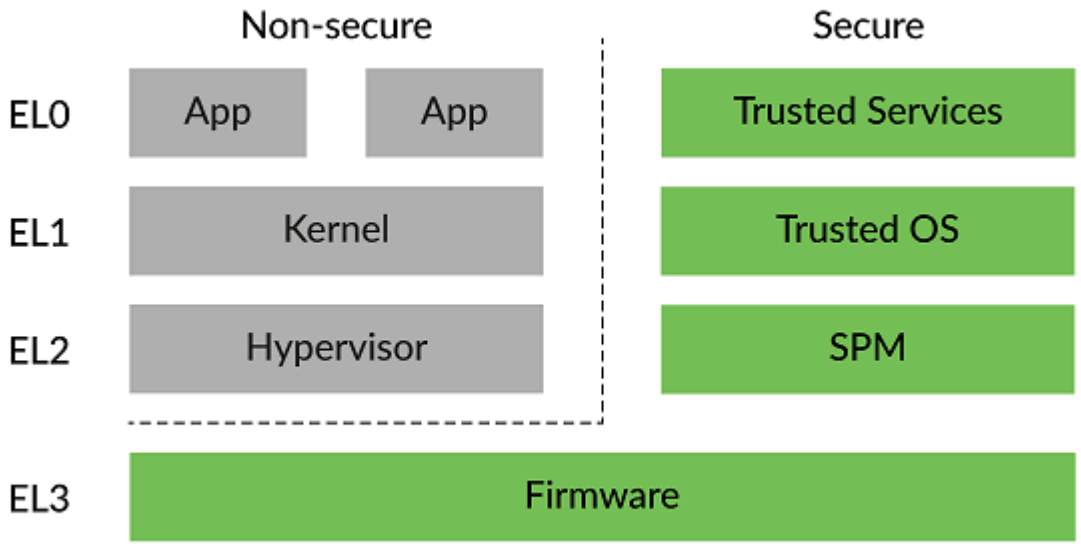
2. Security states

RME builds on the Arm TrustZone technology. TrustZone was introduced in Armv6 and provides the following two Security states:

- Secure state
- Non-secure state

The following diagram shows the two Security states in AArch64 with the software components that are typically found in each Security state:

Figure 2-1: Security states before RME



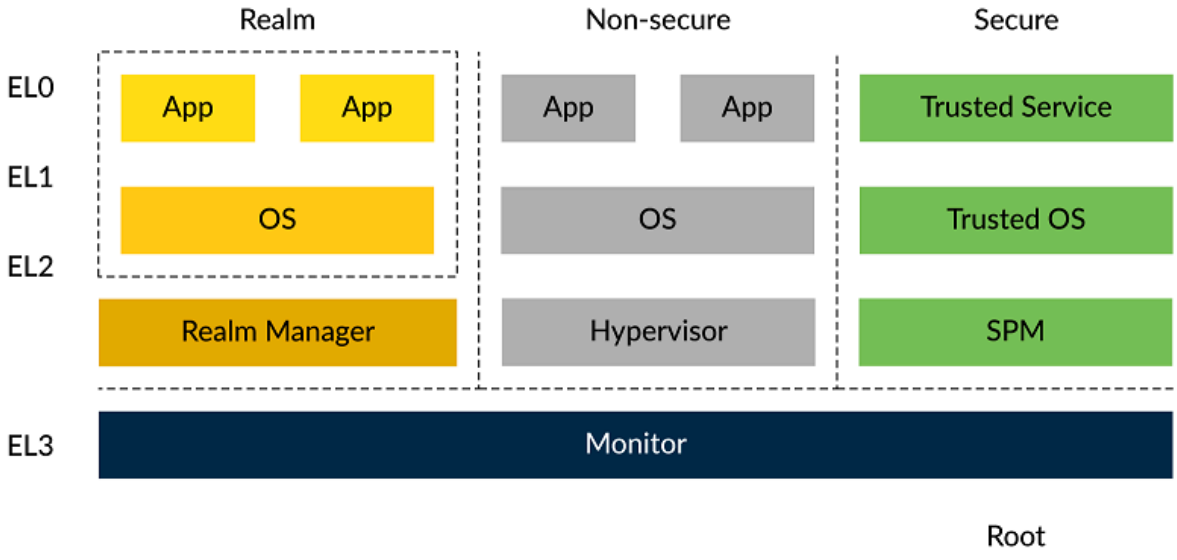
The architecture isolates software running in Secure state from software running in Non-secure state. This isolation enables a software architecture in which trusted code runs in Secure state and is protected from code in Non-secure state.

RME extends this model, and provides the following four Security states:

- Secure state
- Non-secure state
- Realm state
- Root state

The following diagram shows the Security states in an RME-enabled PE, and how these Security states map to Exception levels:

Figure 2-2: Security states with RME



Maintaining Secure state provides backwards compatibility with existing TrustZone use cases. These use cases can also be upgraded to take advantage of new features added by RME, like dynamic memory assignment.

Realm state constructs protected execution environments called realms. Importantly, RME extends the isolation model introduced in TrustZone.

The architecture provides isolation for the following states:

- Secure state from both Non-secure and realm states
- Realm state from both Non-secure and Secure states

This isolation model provides a software architecture in which the software in Secure and realm states are mutually distrusting.

With RME, Exception level 3 moves out of Secure state and into its own Security state called root. RME isolates Exception level 3 from all other Security states. Exception level 3 hosts the platform and initial boot code and therefore must be trusted by the software in Non-secure, Secure, and realm states. Because these Security states do not trust each other, Exception level 3 must be in a Security state of its own.

Controlling the current Security state

A combination of the Exception level and SCR_EL3 registers controls the current Security state.

Exception level 3 is now its own root Security state. While in Exception level 3, the Security state is always root, and no other Exception level can be in root state.

When in lower Exception levels such as Exception level 0, Exception level and Exception level 2, the NS and NSE fields in SCR_EL3 controls the Security state. The exception levels are shown in the following table:

SCR_EL3.{NSE,NS}	Security state
0b00	Secure
0b01	Non-Secure
0b10	-
0b11	Realm

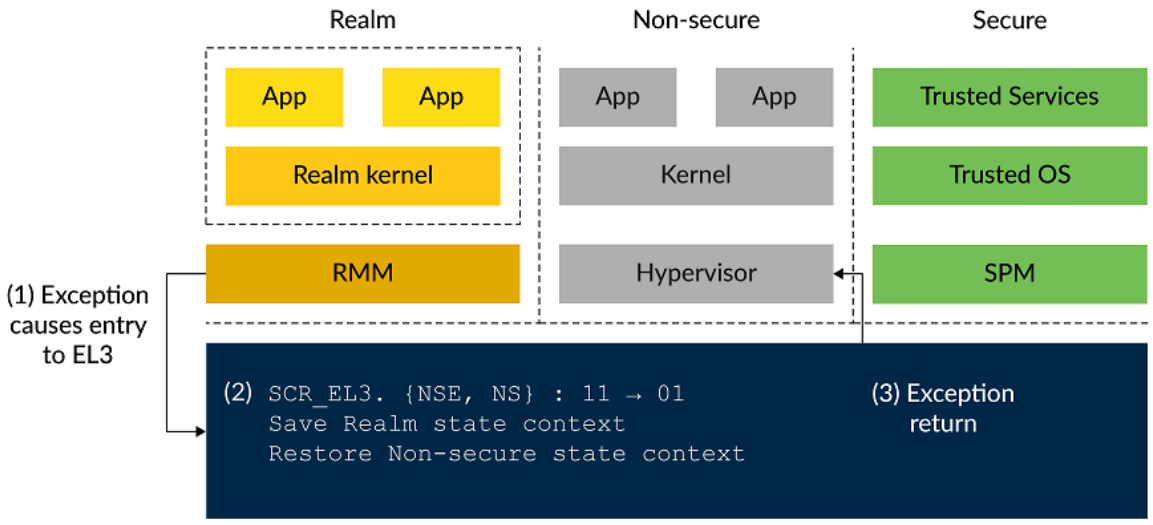
There is no encoding for root state. While in Exception level 3, the current Security state is always root, regardless of the SCR_EL3.{NSE,NS} value.

In Exception level 3, the current value of SCR_EL3.{NSE,NS} is used to control some operations. For example, when issuing a Translation Lookaside Buffer (TLB) invalidation instruction at Exception level 3 for a lower Exception level, SCR_EL3.{NSE,NS} controls which Security state the operation applies to.

Moving between Security states

The principles of moving between Security states is inherited from TrustZone. To change Security state, execution must pass through Exception level 3, as shown in the following diagram:

Figure 2-3: Changing security state example



The changing Security state process in the diagram follows these steps:

1. Execution starts in realm state and SCR_EL3.{NSE,NS} is set to `0b11`. The software executes a Secure Monitor Call (SMC) instruction, which causes an exception to be taken to Exception level 3.
2. The processor enters Exception level 3 and is now in root state because Exception level 3 is always in root state. However, SCR_EL3.{NSE,NS} still has the same Security state that the exception was taken from. Software in Exception level 3 changes SCR_EL3.{NSE,NS} to the corresponding value for the required Security state and executes an Exception Return (ERET).
3. The ERET causes exit from Exception level 3. When leaving Exception level 3, SCR_EL3.{NSE,NS} controls which Security state is entered. In this diagram, the Security state is Non-secure.

There is only one copy of the vector registers, the general-purpose registers, and most System registers. When moving between Security states it is the responsibility of the software, not hardware, to save and restore register context. The software that saves and restores this register context is called the Monitor.

3. Physical Addresses

In addition to two Security states, TrustZone provides the following two Physical Address (PA) spaces:

- Secure physical address space
- Non-secure physical address space

These separate PA spaces form part of the TrustZone isolation guarantee. Non-secure state cannot access an address in a Secure PA space. This isolation means that there are confidentiality and integrity guarantees for data belonging to Secure state.

RME extends this guarantee to support the following PA spaces:

- Secure physical address space
- Non-secure physical address space
- Realm physical address space
- Root physical address space

The architecture limits which PA spaces are visible in each Security state. The following table shows the PA spaces that you can access in each Security state:

Physical address space	Secure state	Non-secure state	Realm state	Root state
Secure PAS	Yes	No	No	Yes
Non-secure PAS	Yes	Yes	Yes	Yes
Realm PAS	No	No	Yes	Yes
Root PAS	No	No	No	Yes

In this table, Y means accessible and N means not accessible.

When documentation refers to a physical address, prefixes are used to identify which address space is being referred to, for example:

- SP:`0x8000` means address `0x8000` in the Secure PA space
- NSP:`0x8000` means address `0x8000` in the Non-secure PA space
- RLP:`0x8000` means address `0x8000` in the realm PA space
- RTP:`0x8000` means address `0x8000` in the root PA space

Architecturally, each example is an independent memory location. This means that SP:`0x8000` and RTP:`0x8000` are treated as different physical locations. All four locations can exist in an RME-enabled system although in practice, this is unlikely.

Virtual address spaces

To support the new Security states, RME introduces the following translation regimes for Realm state:

Realm EL1& 0 translation regime

This regime includes two virtual address (VA) regions, similar to the Non-secure EL1&0 translation regime. This translation regime is subject to stage 2 translation.

Realm Exception level 2 and 0 translation regime

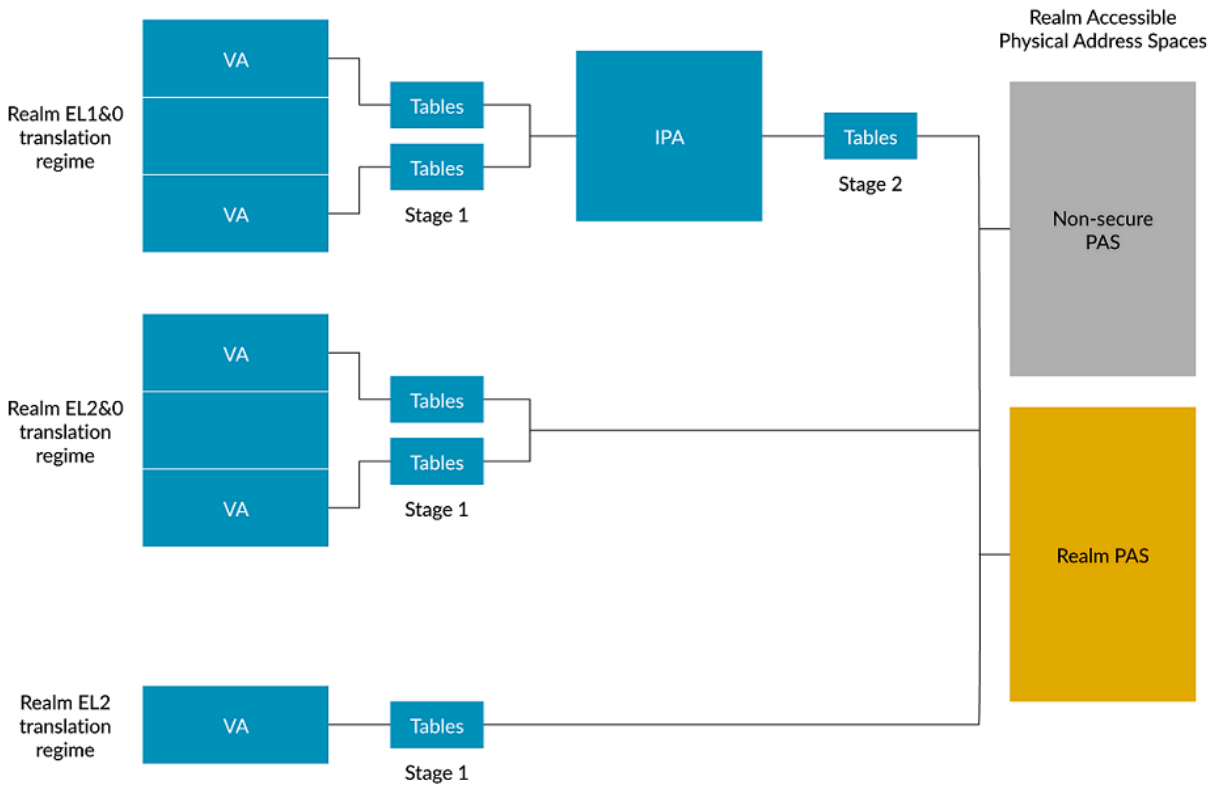
This regime includes two VA regions, similar to the Secure Exception level 2 and 0 translation regime.

Realm Exception level 2 and 2 translation regime

This regime includes a single VA region, similar to the Secure Exception level 2 translation regime.

The following diagram shows the realm state translation regimes:

Figure 3-1: Realm translation regimes



For all realm translation regimes, any address that translates to a Non-secure physical address is treated as execute-never.

Root state translation regime

Only Exception level 3 exists in root state. This means that there is a single translation regime for root state, which is the Exception level 3 translation regime. This translation regime existed before RME but was previously considered part of Secure state.

RME contains the following changes to the Exception level 3 translation regime:

- Virtual addresses can translate to physical addresses in any of the four physical address spaces
- Any address that translates to a Non-secure, Secure, or realm physical address is treated as execute-never
- MMU table walks can only access the root PA space
- When the MMU is disabled at Exception level 3, all output addresses are in the root PA space

Exception level 3 continues to use the Exception level 3 translation regime, which has a single VA range.

Controlling output PAS

When a virtual address is translated by the MMU, the output PA space is controlled by a combination of the following:

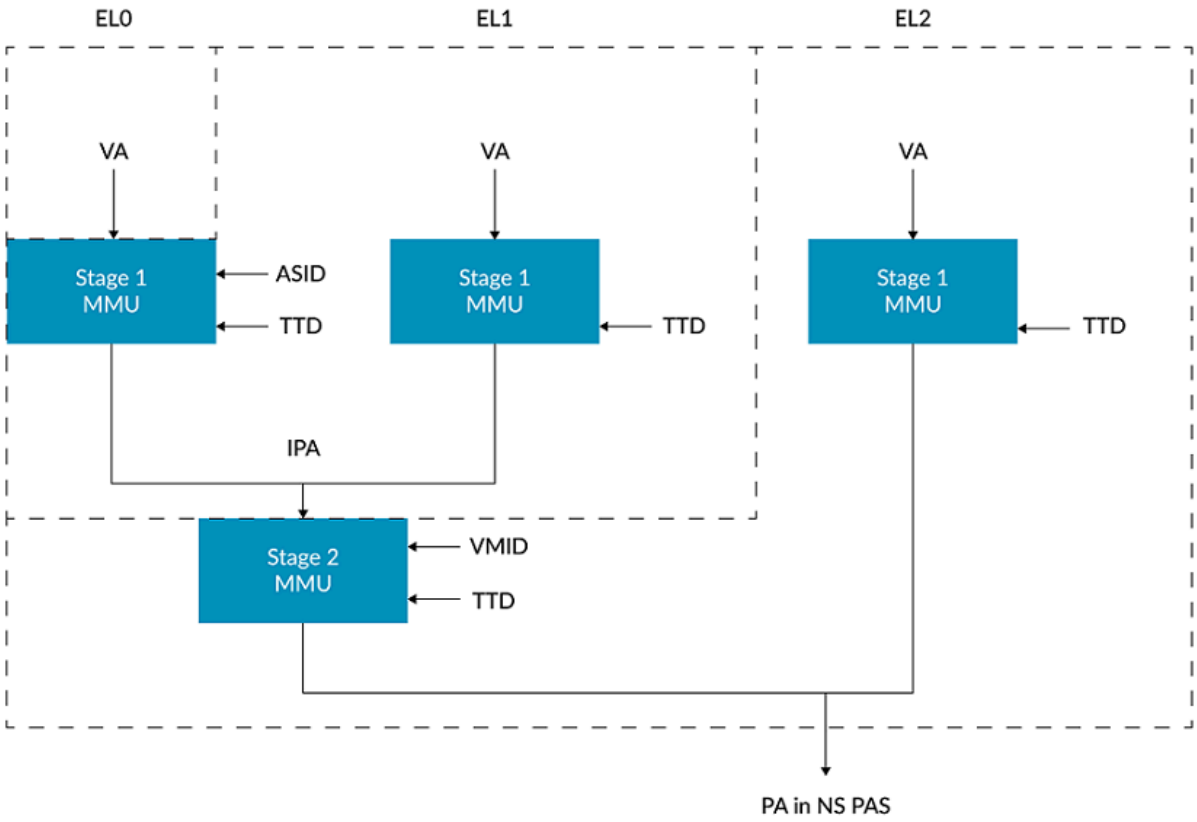
- Current Security state
- Current Exception level
- Translation tables
- System registers

The controls that are available to software vary depending on the translation regime. Now we will look at the controls in each Security state and translation regime.

Non-secure state translation regimes

The following diagram shows an overview of the Non-secure translation regimes:

Figure 3-2: Non-secure translation regimes



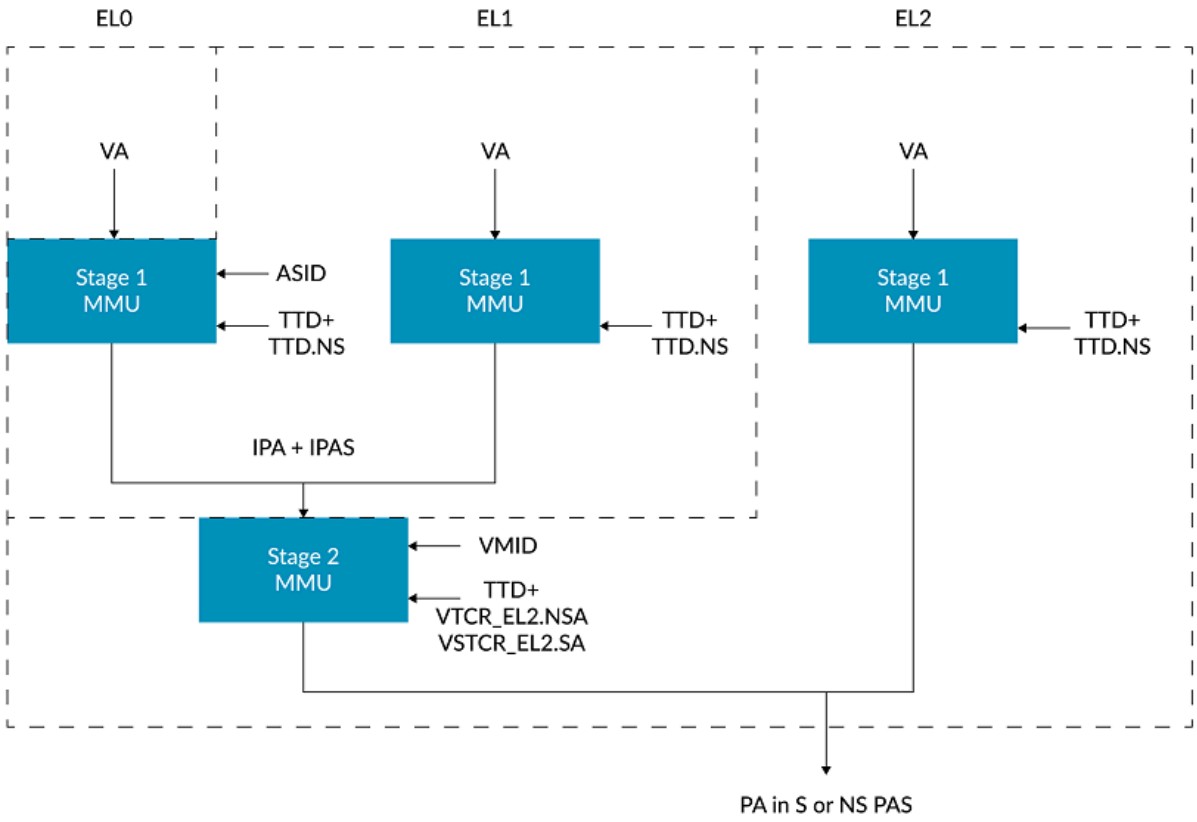
Non-secure state can only access the Non-secure PA space. Therefore, in Non-secure state there are no controls at stage 1 or stage 2 for controlling the output IPA space or PA space.

In this diagram, TTD means Translation Table Descriptor.

Secure state translation regimes

Secure state can access the Secure and Non-secure PA spaces, as shown in the following diagram:

Figure 3-3: Secure translation regimes



For the Secure translation regime, the NS bit in the stage 1 translation tables entries selects between two Intermedial Physical Address (IPA) spaces.

There are controls at Exception level 2, VTCR_EL2, and VSTCR_EL2, to map each of those IPA spaces to a PA space.

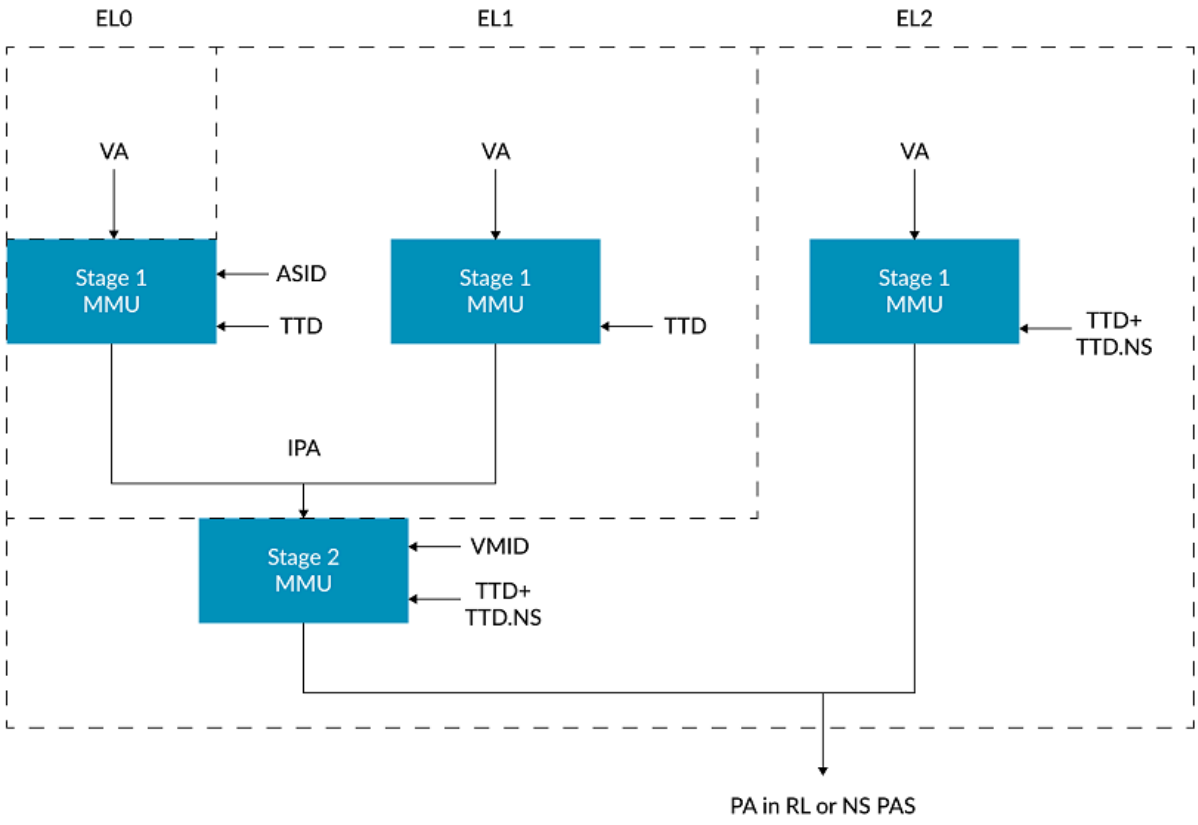
Note

In Secure state, the controls are unchanged by RME and are described in this guide for completeness.

Realm state translation regimes

Realm state can access the realm and Non-secure PA spaces, as shown in the following diagram:

Figure 3-4: Realm translation regimes



The realm EL0/1 translation regime has a single realm IPA space. Therefore, there is no NS bit in the EL0/1 stage one translation table entries.

The realm stage two TTDs include an NS bit, to map to either the realm or Non-secure PAS. This means that, unlike Secure state, realm state has per-page controls at stage two.

The realm EL2 translation regime and realm EL0/2 translation regime have stage one NS bits to control the output PA space.

The NS bit in the translation table entries was introduced by TrustZone to allow Secure state to select the output PA space. In Secure state, the NS bit is encoded as follows:

- NS=0: Secure
- NS=1: Non-secure

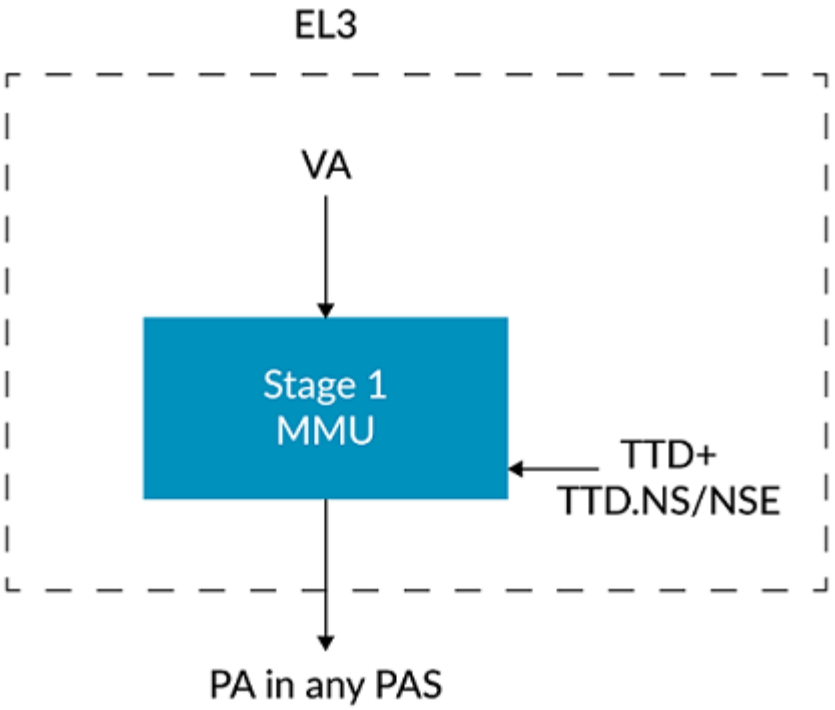
With RME, the NS field is also used in realm EL0/1 stage two and realm EL2/0 stage one translation tables, but is encoded as:

- NS=0: realm
- NS=1: Non-secure

Root state translation regimes

Root state can access all PA spaces, as shown in the following diagram:

Figure 3-5: EL3 translation regime



The Exception level 3 stage one translation regime has two bits, NS and NSE, in the translation table entries to control the output PA space. These encodings are similar to the encodings that are used for SCR_EL3.{NSE,NS}, except that there is an encoding for root, as shown in the following table:

TTD.{NSE.NS}	Security state
0b00	Secure
0b01	Non-Secure
0b10	Root
0b11	Realm

Note

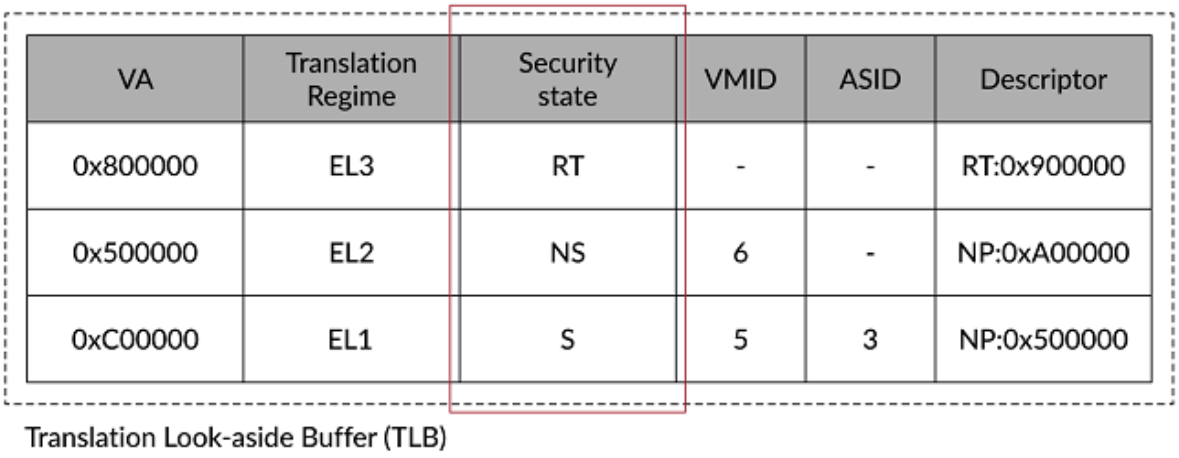
Only Exception level 1 exists in root state. Exception level 3 is only subject to stage one translation.

Impact on Translation Lookaside Buffers and caches

TLBs cache recently used translations. Translation Lookaside Buffer (TLB) entries need to record which translation regime an entry belongs to. During a TLB look-up, an entry can only be returned if the translation regime in the entry matches the requested translation regime. This prevents one Security state from using the TLB entries of another Security state.

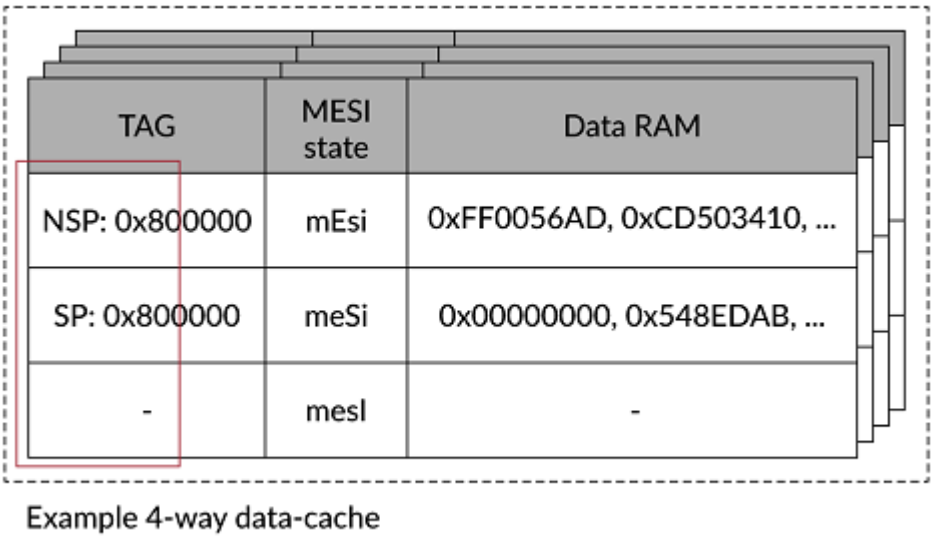
The following table shows an example simplified TLB, recording the translation regime:

Figure 3-6: Translation Lookaside Buffer



Similarly, caches lines need to record the associated PA space, as shown in the following diagram:

Figure 3-7: Example cache with PAS recorded



4. Granule Protection Checks

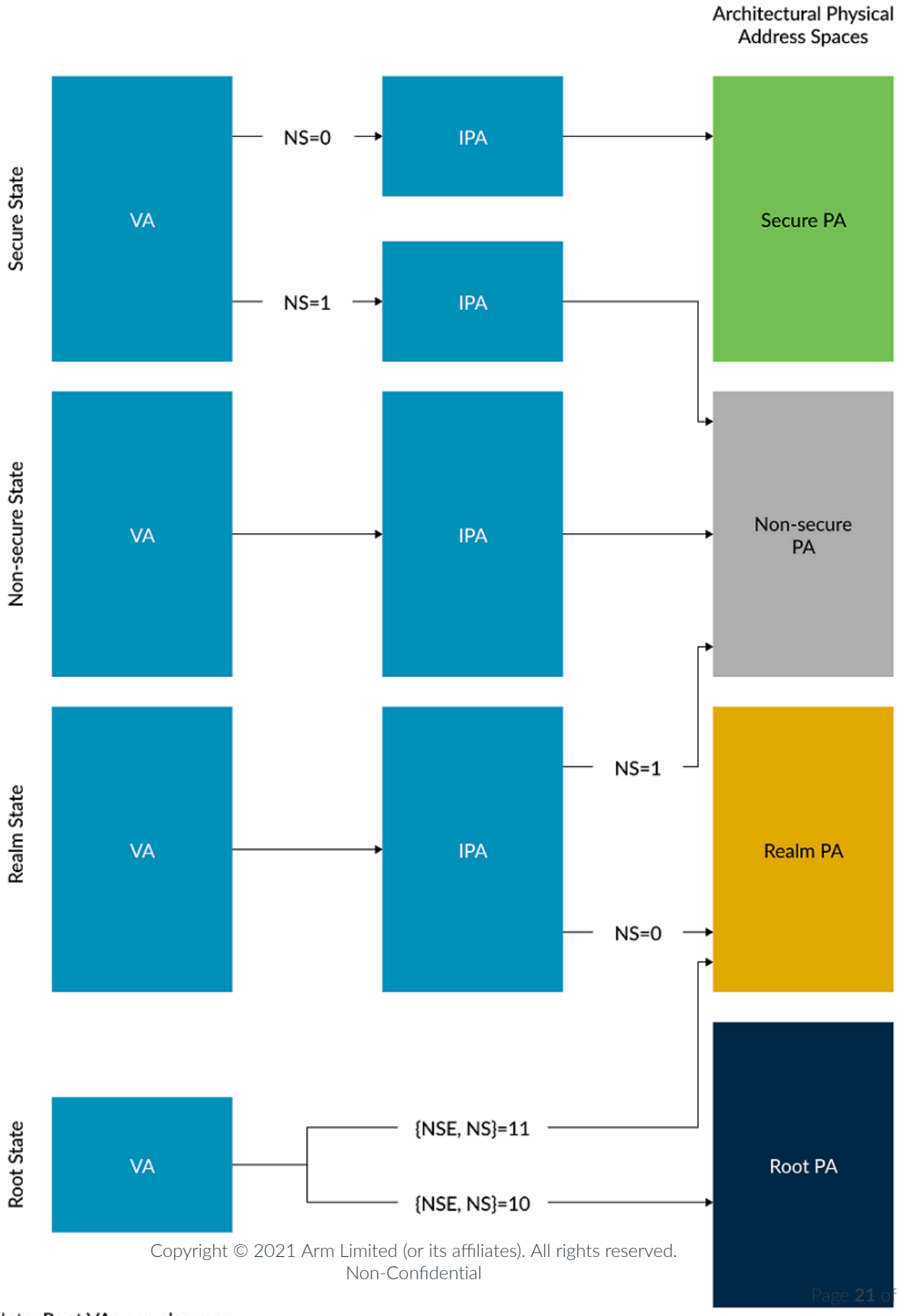
This section describes Granule Protection Checks introduced by RME. Granule Protection Checks enable the dynamic assigning of memory regions between different physical addresses spaces.

This section teaches you about the following features:

- The structure of Granule Protection tables
- Fault reporting for Granule Protection Checks
- How regions transition between PA spaces

As described in Physical addresses, RME provides four physical address spaces. The following diagram shows these physical address spaces:

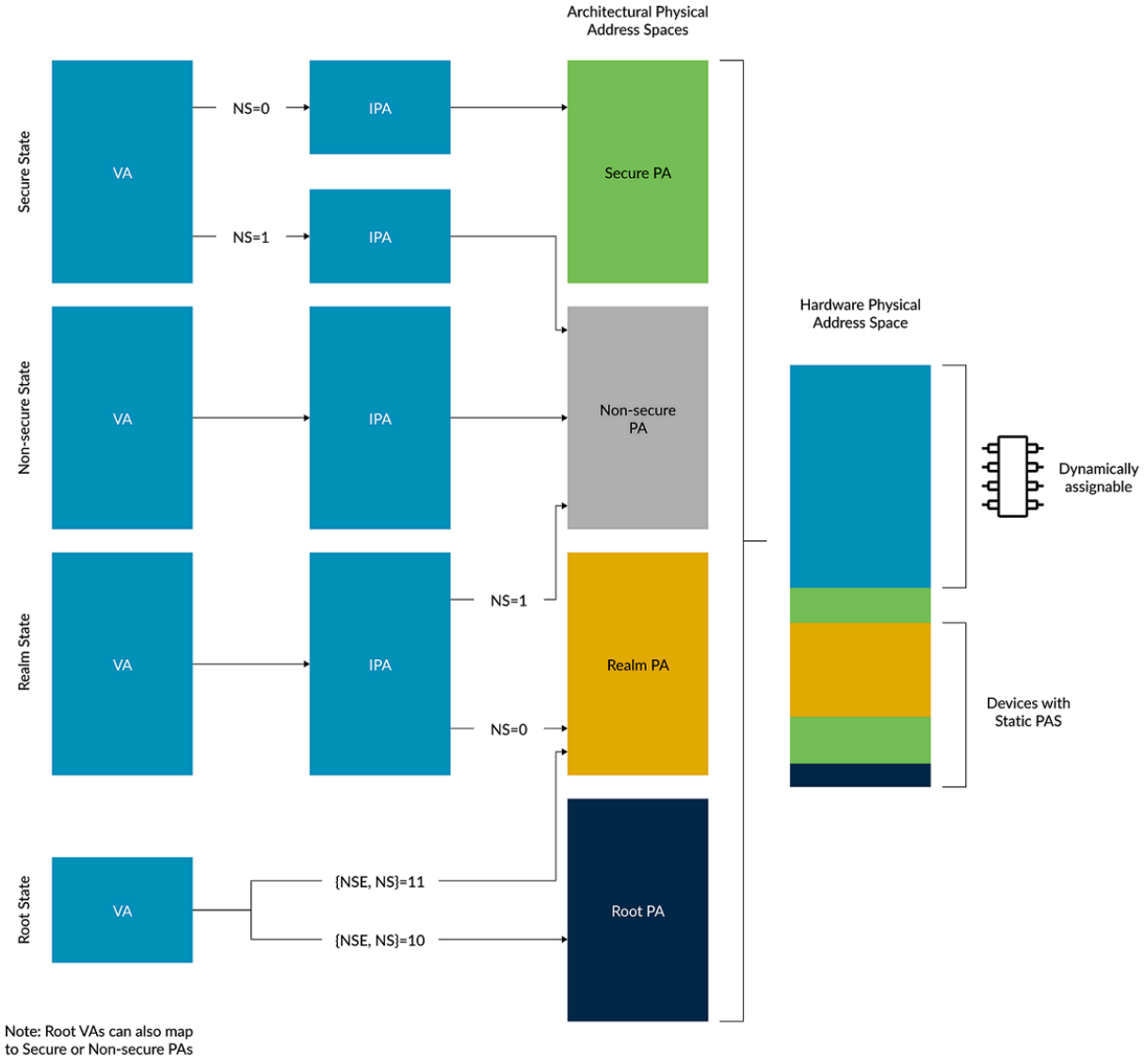
Figure 4-1: VA to PA mapping



Note: Root VAs can also map to Secure or Non-secure PAs

In theory, each PA space is separate and independent and could be fully populated. In practice, most designs have a single effective PA space for DRAM regions, using the PA spaces to partition the space into regions, as shown in the following diagram:

Figure 4-2: Multiple PA spaces



For on-chip devices and memories, the memory system is typically enforces isolation. This isolation is provided either in the end peripheral or in the interconnect. This configuration is referred to as completer side filtering, for example:

- On-chip ROM and SRAM, which is root-only and the interconnect enforces it. Example use cases include system boot.

- Generic Interrupt Controller (GIC). Transactions are routed to the GIC regardless of PA space. GIC internally uses the security of the access to control which state and configuration is accessible.
- For bulk memory, RME provides a mechanism to dynamically allocate pages to different PA spaces at runtime. For example, when starting a realm, ownership of some memory is transferred from Non-secure state to realm state. When that realm is terminated the memory is reclaimed, and ownership returns to a Non-secure state.


Note

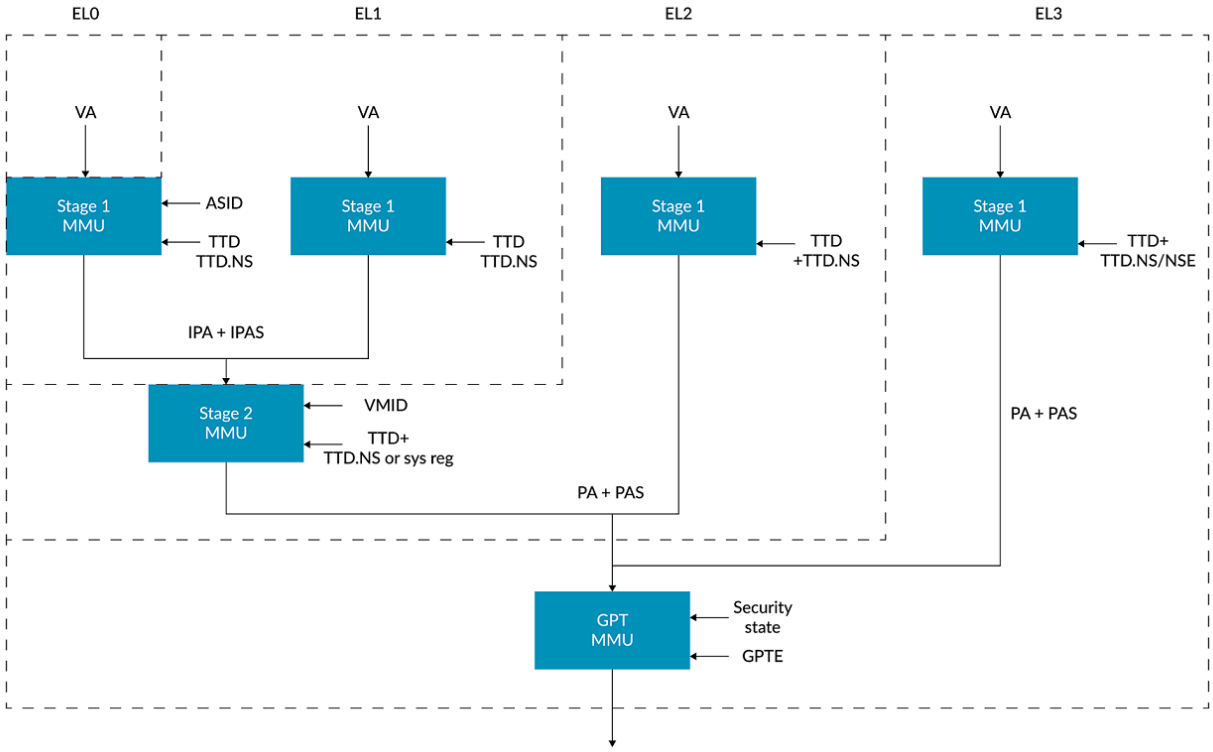
In the system architecture, the physical address space region that is assigned is called the Resource PA space.

The Granule Protection Checks in the MMU enables dynamic allocation of pages to PA spaces. A set of Granule Protection Tables (GPTs) records for every location that is either of the following:

- Completer side filtered. In this allocation, the MMU permits all accesses and relies on memory system checks. These checks can be carried out in either the interconnect or the peripheral.
- Allocated to a PAS:
  - In this allocation, the MMU only permits access where the output physical address space from VA to PA translation matches the PA space in the GPTs
  - Where the physical address space does not match, the MMU blocks the access and returns a Granule Protection Fault (GPF)

Conceptually, the MMU after stage one and stage two translations performs Granule Protections Checks, as shown the following diagram:

Figure 4-3: MMU translation stages



In the diagram, stages are shown as serial however, the process is more complicated. In the following table, we show an example of an LDR instruction that is executed in NS_EL1. For simplicity, we assume a single table level at stages 1 and 2:

Step	Action by PE	Notes
1	LDR instruction issued	
2	Read TTBR<n>_EL1 to get IPA of stage 1 table	Before the stage 1 descriptor can be fetched, the IPA must be translated into a PA
3	Read VTTBR_EL2 to get PA of the stage 2 table	Before the stage 2 descriptor can be fetched, the PA must have a Granule Protection Check
4	Perform Granule Protection Check on PA of stage 2 table descriptor	
5	Read stage 2 table descriptor and translate IPA of stage 1 descriptor to a PA	We now have the PA of the stage 1 descriptor. Before it can be fetched, the PA must have a Granule Protection Check.
6	Perform Granule Protection Check on physical address of stage 1 table entry	
7	Read stage 1 table entry and translate input VA to IPA	
8	Perform Granule Protection Check on physical address of stage 2 table entry	Before the stage 2 descriptor can be fetched, it must have a Granule Protection Check

For a running system, most accesses reuse cached translations in the TLBs. However, the example highlights the interaction between the different stages of translation and Granule Protection Checks.

In Elision, we show that some parts of the process can be optimized.

Granule Protection Tables

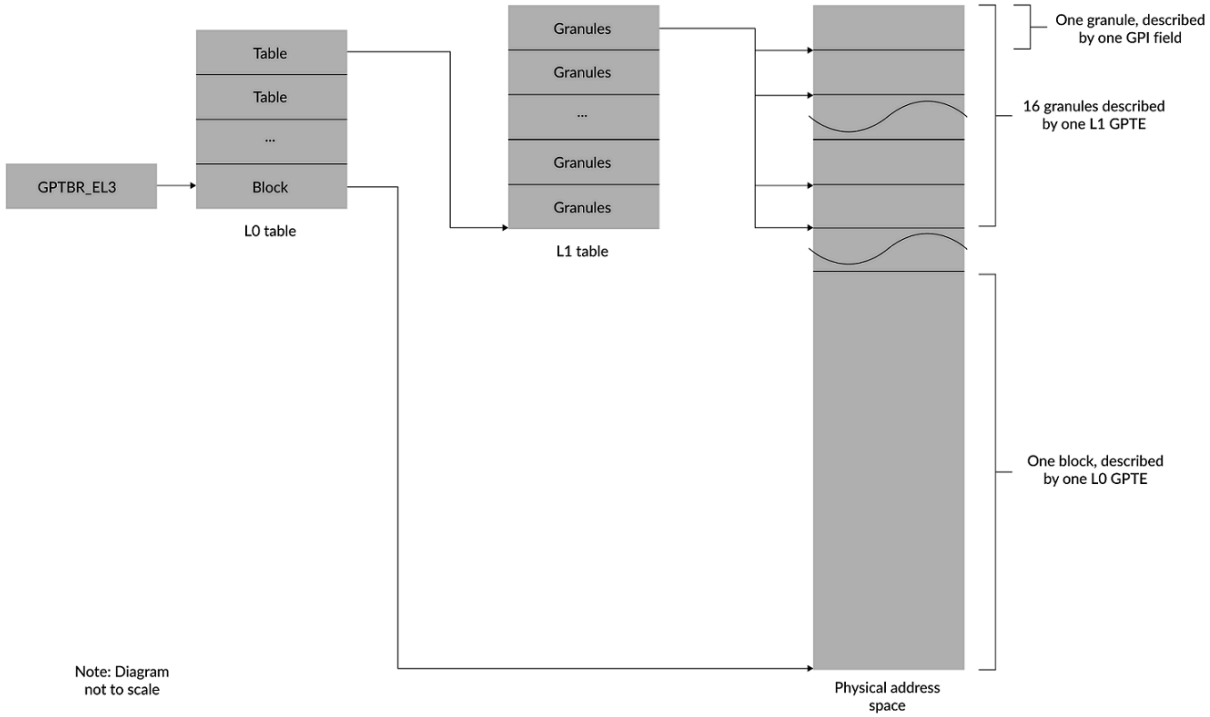
A set of tables, called Granule Protection Tables (GPT), configures which PA space each granule is associated with. When the processor performs an access, the MMU walks the GPTs to determine whether the access is permitted.

Granule Protection Checks (GPCs) are configured using the following System registers:

- GPCCR_EL3 to configure:
- Granule Protection Checks Enable
- Granule size: 4K, 16K, or 64K
- Size of protected region
- Elision enable
- GPTBR_EL3 to configure the PA of the GPT, which is in the root PA space

GPTs have a two-level table structure, as shown in the following diagram:

Figure 4-4: Example simple GPT structure



GPTBR_EL3 points to the base of the level 0 table. Each level 0 table entry covers a 1GB region, and can be one of the following formats:

- Block Descriptor. The block is assigned to a specific PAS or be configured to allow all PA spaces.
- Table Descriptor:
  - The level 0 table entry is subdivided into granules represents the region, with a level 1 GPT describing the mapping of those granules
  - The descriptor gives the PA of the level 1 table. The table must be in the root PA space.

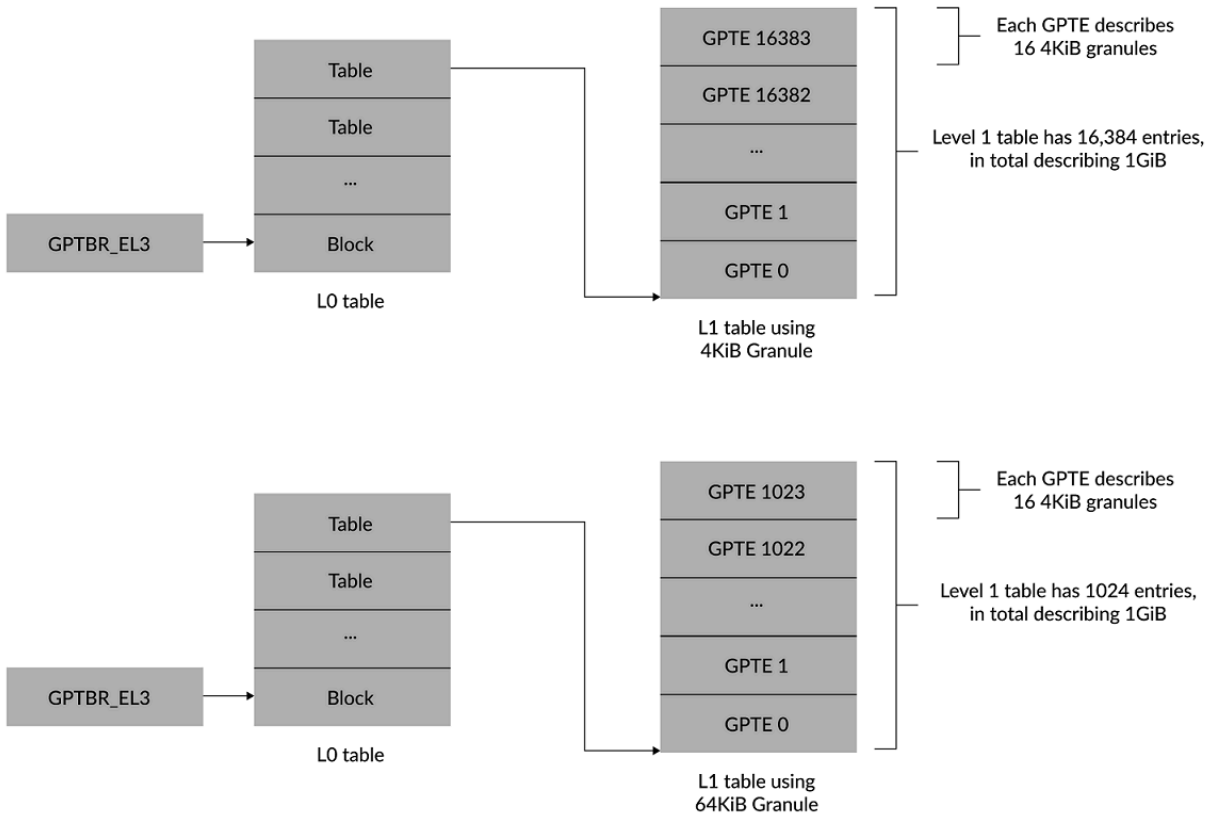
Each entry in a level 1 GPT is one of the following:

- Granules Descriptor
  - Contains 16 GPI fields, with each GPI field describing one granule of PA space
  - Each granule can be independently assigned to a single PA space or configured to allow all PA spaces. This configuration delegates responsibility to check the legality of the access to a completer-side filter.
- Contiguous Descriptor is like a Granule Descriptor, but describes larger regions. Using larger regions can enable more efficient caching in the TLBs.

Each entry in a L1 table describes 16 granules, with separate fields per granule. The granule size is configurable through GPCCR_EL3 and matches the granules sizes that are available for the translation tables.

Because a level 1 table is fixed at 1GB but the granule size is variable covers the address range, the number of entries in a level 1 table also varies. Selecting a smaller granule size results in larger level 1 tables. The following example shows the level 1 tables sizes using a 4KiB and 64KiB granule size:

Figure 4-5: Effect of granule size



A GPTE refers to an entry in either a level 0 or level 1 GPT. GPI refers to the fields in a GPTE that describe the assigned PA space for a region of memory.

Note

GPI entries for level 0 entries are only used in block descriptors.

The GPTBR_EL3.PPS defines the region that the GPT covers starts at address 0 and extends to an upper bound. Any address beyond the range that GPTBR_EL3.PPS defines, is treated as belonging to the Non-secure PA space.

Granule Protection Check faults

If an access fails its Granule Protection Checks, a fault is reported. Collectively, these faults are referred to as Granule Protection Check (GPC) faults.

The types of GPC fault are as follows:

- Granule Protection Fault (GPF): The GPT walk completed successfully, but the access was not permitted.

- GPT Walk Fault: The GPT walk failed to complete because of an invalid GPT entry Descript.
- GPT address size fault: The GPT walk failed because of attempted access to an address beyond the configured range.
- Synchronous External abort on GPT fetch: The GPT walk failed because a read of a GPT entry returned an External abort.

GPC faults are reported as one of the following exception types:

- Data Abort exception
- Instruction Abort exception
- GPC exception
- GPC exception is a new synchronous exception type introduced by RME.


Note

GPFs on accesses to the trace and Statistical Profiling Extension (SPE) buffers are handled differently. For more information, see Self-hosted trace and SPE.

Granule Protection Faults

A is generated when the PA space returned by the VMSA VA to PA translation does not match the PA space that the granule is assigned to in the GPTs.

For example, software attempts to access RLP:`0x8000`, but PA `0x8000` is allocated to the Non-secure PA space.

GPFs can be reported as GPC exceptions, Instruction Abort exceptions, or Data Abort exceptions as summarized in the following table:

Fault generated at	SCR_EL3.GPF	HCR_EL2.GPF	HCR_EL2.TGE	Resulting exception
EL0	0	0	0	Instruction or Data Abort, taken to Exception level 1
	0	0	1	Instruction or Data Abort, taken to Exception level 2
	0	1	x	Instruction or Data Abort taken to Exception level 2
	1	x	x	GPC exception, taken to Exception level 3
EL1	0	0	n/a	Instruction or Data Abort, taken to Exception level 1
	0	1	n/a	Instruction or Data Abort, taken to Exception level 2
	1	x	n/a	GPC exception, taken to Exception level 3
EL2	0	x	x	Instruction or Data Abort taken to Exception level 2
	1	x	x	GPC exception, taken to Exception level 3
EL3	x	x	x	Instruction or Data Abort, taken to Exception level 3

The exception syndrome that is provided for Instruction Aborts and Data Aborts has been extended to give information on GPFs.

A GPT walk can fail to complete, and one of the following GPC faults is reported:

- GPT address size fault
- GPT walk fault
- Synchronous External abort on GPTE fetch

Arm expects these faults to be rare in system set up correctly. These faults would typically represent an Exception level 3 software error or a loss of consistency, which is likely to be fatal.

These fault types are always reported as GPC exceptions, and taken to Exception level 3.

Transitioning a granule between physical address spaces

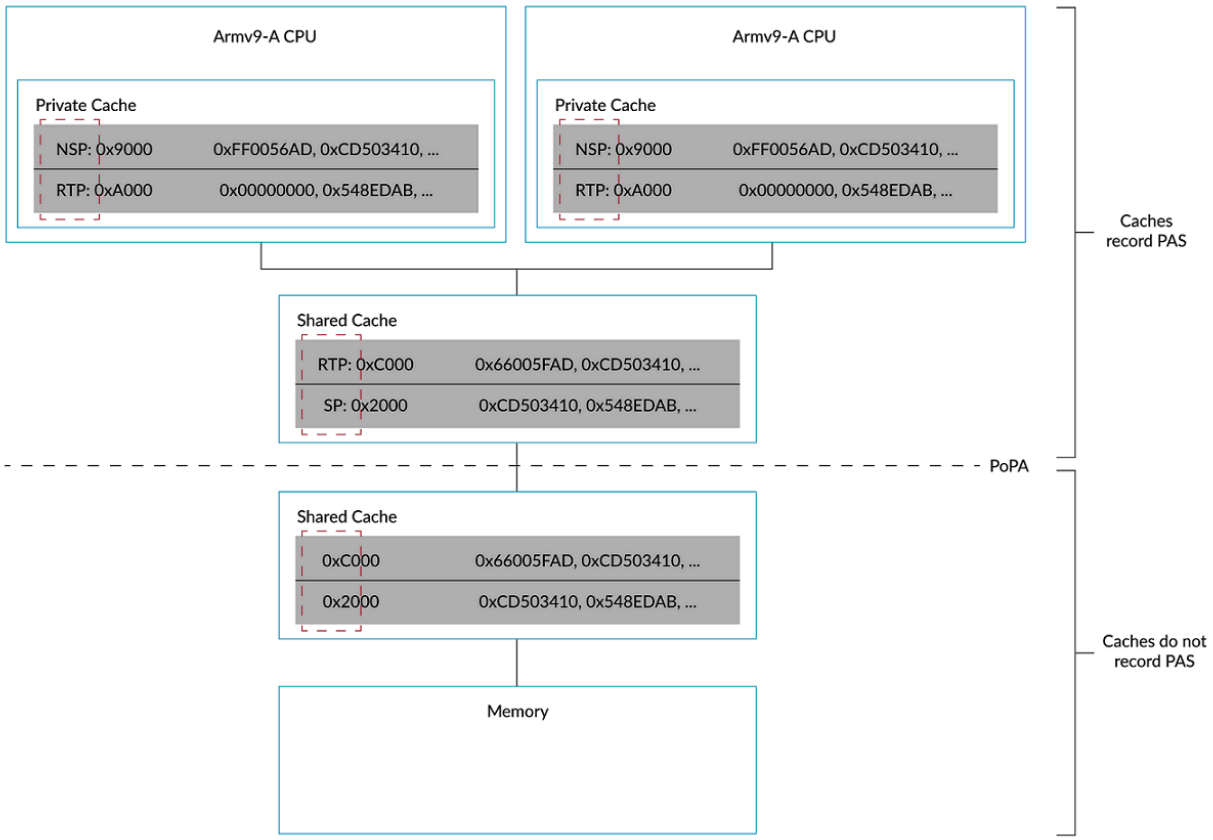
A granule is moved between PA spaces by updating the GPTs. The architecture specification includes the required sequences that software must follow.

Impact on caches

As part of transitioning a block or granule, Exception level 3 software ensures that copies of the location held in caches with the PA space are removed.

To remove the location copies, RME introduces Point of Physical Aliasing (PoPA), a new conceptual point in the cache hierarchy. The PoPA is the point beyond which an access using any PA space uses the same copy in a cache or memory. The following diagram shows an example of PoPA:

Figure 4-6: PoPA example



As part of the transition flow, software cleans and invalidates the caches to PoPA. This ensures that no copies of the granule are held in caches with the old PA space.

Note

Not all systems include caches beyond the PoPA.

Impact on TLBs

TLBs hold copies of recently used translations and can include the result of Granule Protection Checks. When a granule is moved between PA spaces, software must issue invalidate operations to remove any cached copies of the GPTE information. RME introduces new TLBI instructions for this purpose.

The Arm architecture does not specify how TLBs are structured, and the structure can vary between implementations. Possible approaches include one of or a combination the following:

- Caching the result of different stages separately
- Caching the result of multiple stages as a single entry

Software does not need to be aware of which approach is implemented.

Elision

In Granule Protection Checks, we saw an example sequence showing the Granule Protection Checks during a table walk. To minimize the performance penalty of the additional checks, RME supports a mode in which some checks are elided.

When elision is enabled (GPCCR_EL3.GPCP == 1), the MMU might not carry out Granule Protection Checks for reads of stage 2 Table Descriptors. All other accesses, including stage 1 descriptor fetches and fetches of stage 2 Block and Page Descriptors, occur as normal.

The analysis of whether the elision is acceptable to a security model includes a combination of:

- The use and style of memory encryption
- The low probability of ciphertext being a valid translation table descriptor
- The correct implementation of physical address space checks for read-sensitive locations

5. Debug, trace, and profiling

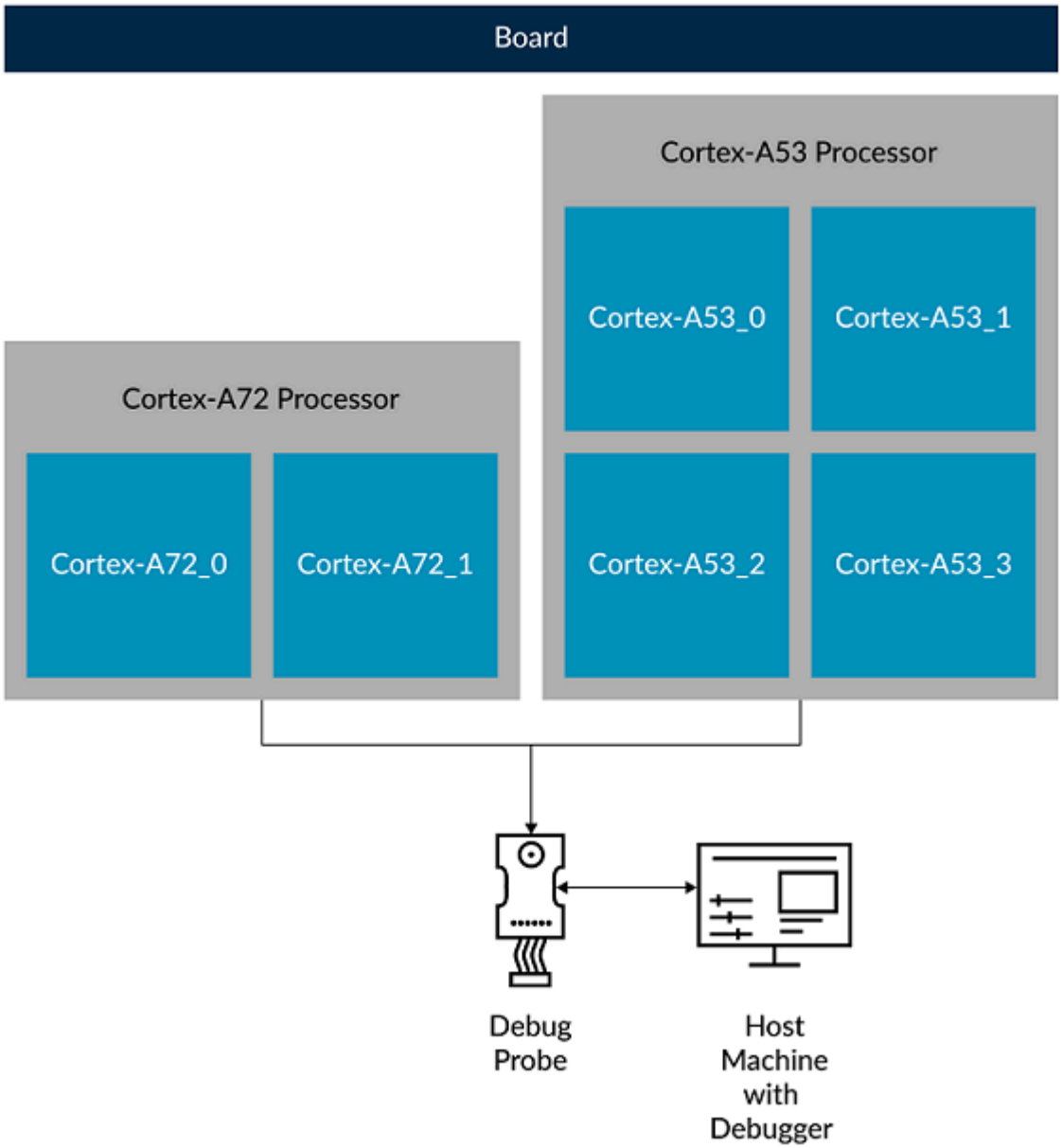
Modern Arm systems include extensive features to support debugging and profiling. We must ensure that these features cannot be used to compromise the security of the system. The Arm architecture, with RME, provides controls to limit which parts of a system can be debugged.

This section assumes familiarity with the base features in Armv9-A and summarizes the changes that RME introduces.

External debug

External debug refers to a situation in which software is debugged by an agent that is outside of the processor. The following diagram shows an example of a debug probe that is connected to a development board and a host machine running a debugger:

Figure 5-1: Example of external debug

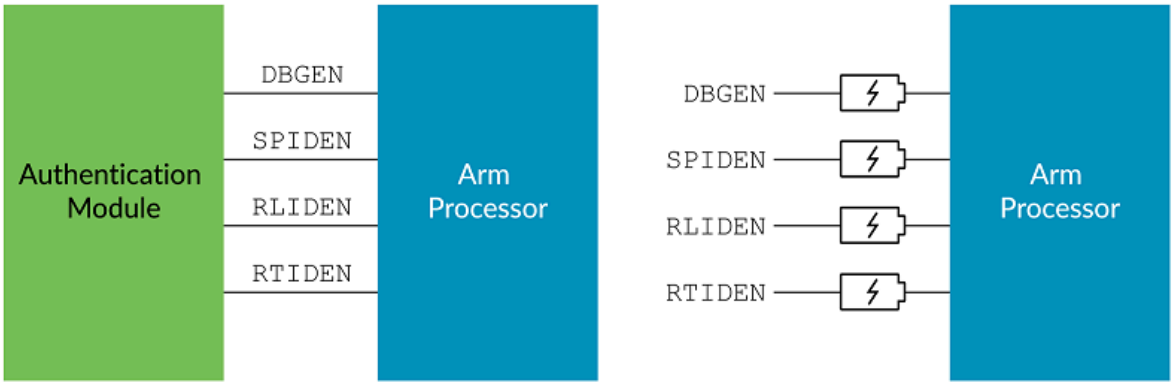


Signals to enable the different debug, trace, and profiling features help deal with the situation in this example. The following are separate signals that enable debug in different Security states:

- DBGEN: Top-level invasive debug enable
- SPIDEN: Secure Invasive Debug Enable, controls external ability to debug in Secure state
- RLPIDEN: Realm Invasive Debug Enable, controls external ability to debug in realm state
- RTPIDEN: Root Invasive Debug Enable, controls external ability to debug in root state

The debug authentications signals are typically connected to fuses or an authentication module, as shown in the following diagram:

Figure 5-2: Debug authentication examples



A silicon vendor uses early development silicon internally. This silicon has the fuses intact, allowing all aspects of the system to be debugged.

Later devices might have a combination of the fuses for RLIDEN, SPIDEN, and RTIDEN blown. This situation allows a development team to only debug specific areas of the system.

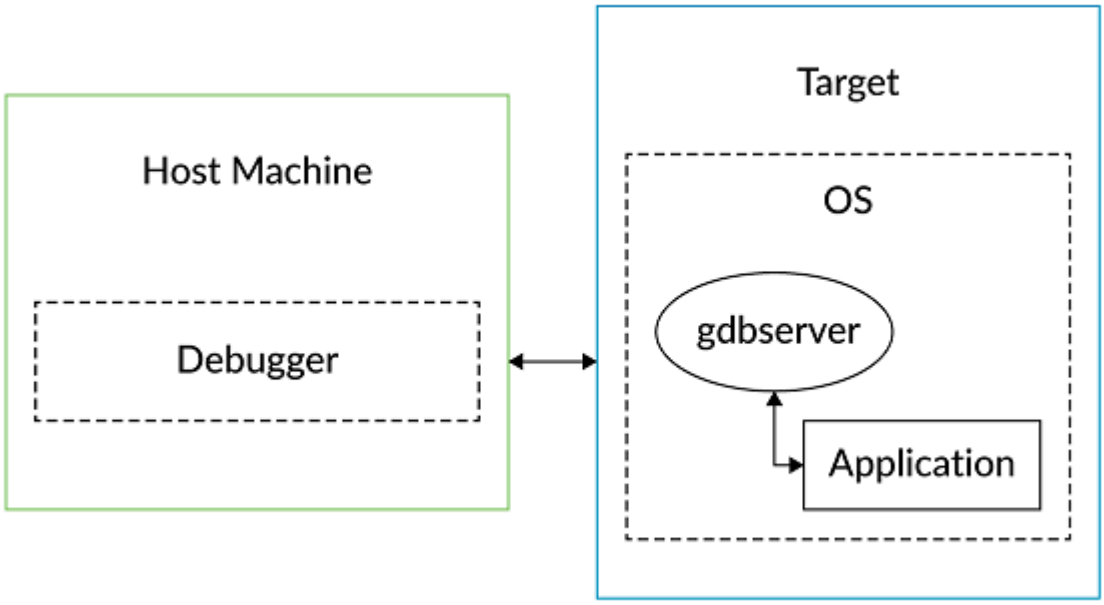
Final production silicon in shipping devices has all the fuses blown, to prevent external debug in any state.

When external debug is disabled for a Security state, for example by blowing a fuse, the processor cannot enter Debug state while in that Security state. An external debugger can still connect to the target. However, the debugger is unable to access the target state or perform run-control until the processor enters a Security state for which external debug is permitted.

Self-hosted debug

Self-hosted debug refers to a situation in which software is debugged by an agent that runs on the same target. The following diagram shows an example of running an application under GDB server:

Figure 5-3: Self-hosted debug



In the diagram, the GDB server is the debug agent running on the target machine. You can connect a graphical debugger to the server. The debugger can also be running on the target or on another machine, for example, over an

The architectural support for self-hosted debug is available in realm state. This means that a VM operating within a realm can run a debug server. In this case, the registers that control self-hosted debug are part of the realm context and are saved on realm exit and restored on realm entry.

Self-hosted trace and SPE

RME makes minimal changes to the self-hosted trace support and SPE in Armv9-A. Additional fields are added in MDCR_EL3 to control the Security states in which trace and profiling data can be collected.

When used by software in multiple Security states, the registers that control these extensions must be context switched by Exception level 3.

Both self-hosted trace and SPE use software-defined buffers in memory to hold profiling data. Accesses to the Trace Buffer and Profiling Buffer are subject to Granule Protection Checks. If the check fails, the fault is reported using a Buffer Management Event, instead of an exception. This reporting process is consistent with how VMSA stage 1 and stage 2 faults are handled.

Performance monitoring

The Performance Monitor Extension (PMU) and Activity Monitor Extension (AMU) are unchanged by RME.

When the PMU is used by software in multiple Security states, the registers that control these extensions must be context switched. This is to avoid the PMU being used to expose information related to one Security state to another.

The AMU provides more limited information and is intended for used for activity monitoring as part of power management. Arm recommends that CNT_CYCLES and CPU_CYCLES are not context switched or disabled when realms are scheduled. Where the AMU provides more counters, access to counters is controlled by Exception level 3.

Branch Record Buffer Extension

The Armv9.2-A Branch Record Buffer Extension (BRBE) is unchanged by RME. When used by software in multiple Security states, the registers that control BRBE must be context switched.

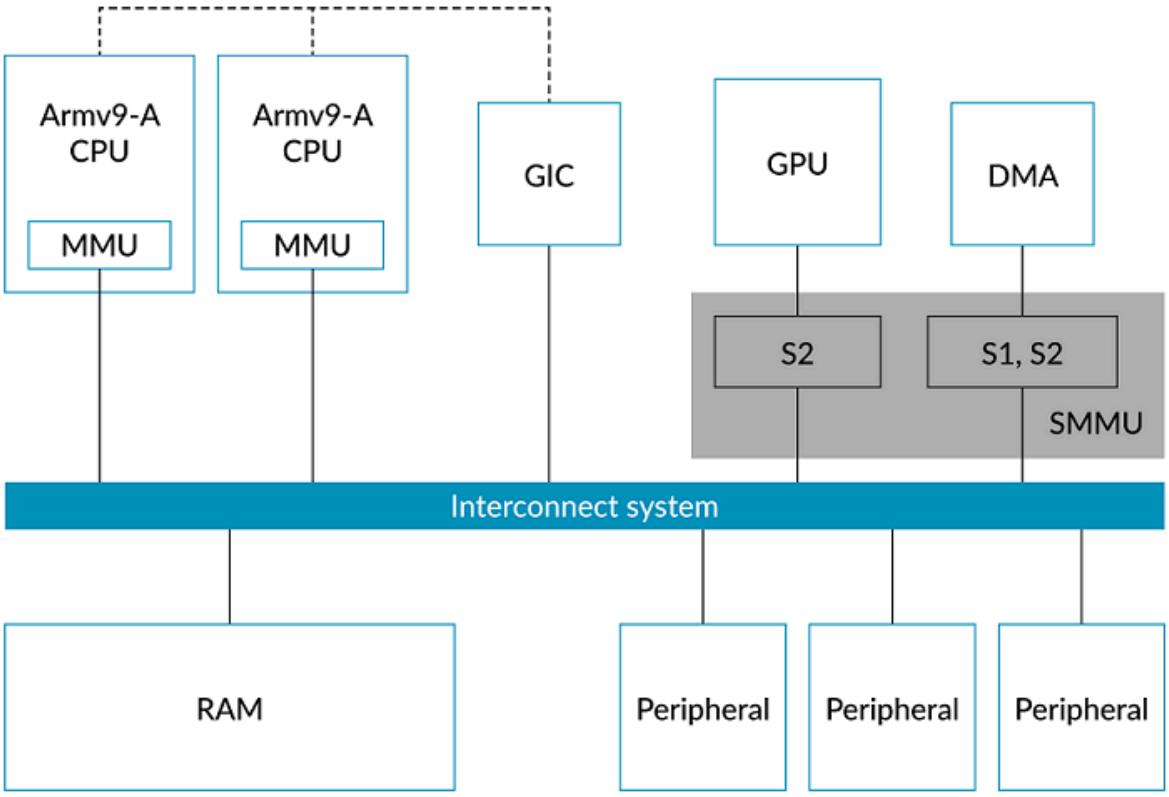
6. SMMU architecture

The SMMU architecture is extended to support Granule Protection Checks. In this section of the guide, we describe how SMMUs are used in an RME-enabled system and the key changes to the SMMU architecture.

SMMUs in an RME-enabled system

A system includes several devices that can independently access memory, like DMA controllers or GPUs. A simplified system is shown in the following diagram:

Figure 6-1: Example system before RME



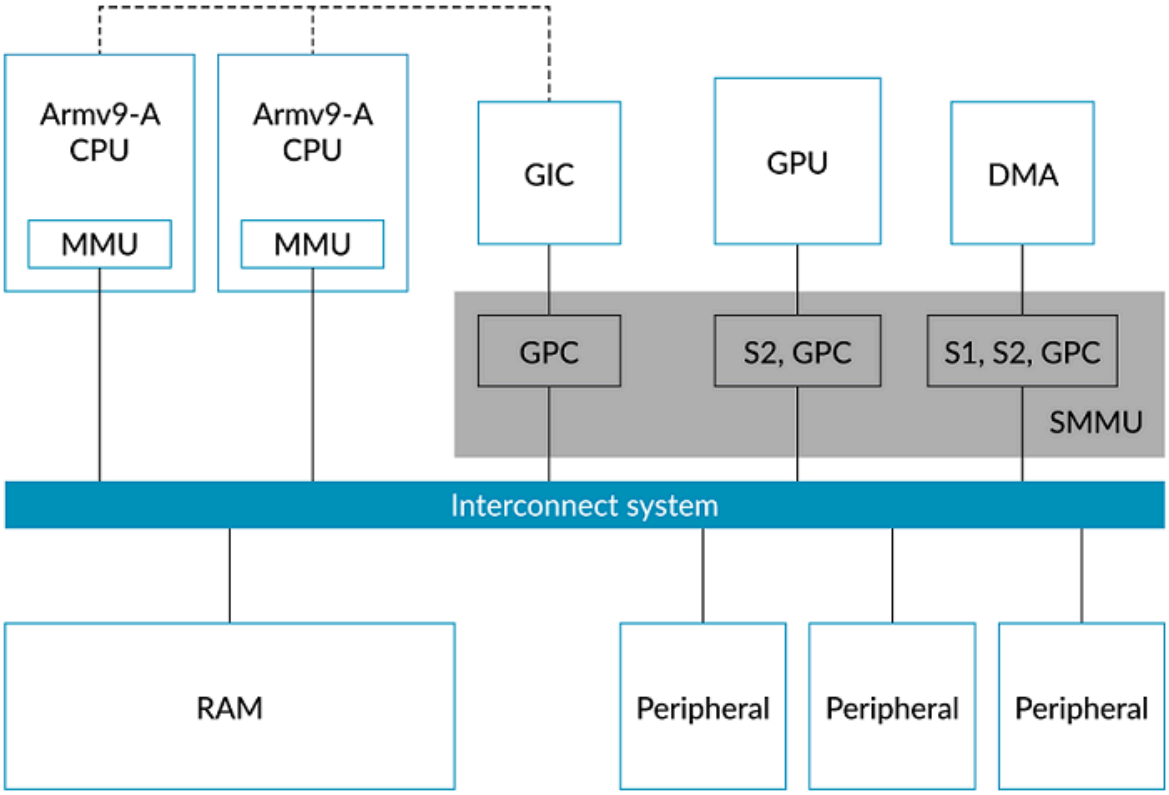
Any device that can access memory, and is therefore a Requester, must be subject to the physical address space isolation guarantees of TrustZone and RME.

For regions that rely on completer-side filtering, this isolation guarantee is achieved in the memory system. The Requester specifies the PA and PAS space that it wants to access. The memory system or target peripheral determines whether the access is permitted. This process is unchanged from TrustZone.

RME provides support to dynamically assign pages of memory to different PASs. Access by all devices to assignable locations must be checked against the GPT. For the CPU, this access is handled by the MMU. For other devices, access is handled by the SMMU.

The following diagram shows an example system using the SMMU to provide granule protection checks:

Figure 6-2: Example system with SMMU



In the diagram showing the simplified example system before RME, the GPU and DMA use SMMUs to provide translation. In an RME enabled system, the SMMU also provides Granule Protection Checks.

However, the GIC is an example of a device that would not previously have been connected through an SMMU. The GIC can access memory and needs Granule Protection Checks. In the RME enabled example system, the GIC is connected through the SMMU, but the SMMU only provides Granule Protection Checks.

Changes to the SMMU architecture for RME

- Client devices and SEC_SID

In the SMMU architecture, SEC_SID identifies the Security state of a client device. Only Secure and Non-secure client devices can be identified. Root or Realm client devices are not currently supported.

- Clients devices that do not have a StreamID

  In the SMMU architecture, a StreamID is used to identify the client device that sent a transaction, and to determine what translations to perform.

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

- SMMU-originated accesses

  Accesses originating from the SMMU, for example reads of the Stream Table, are subject to Granule Protection Checks. These checks are the same as accesses as part of a stage 1 or stage 2 walk by the PE MMU, which are subject to Granule Protection Checks.

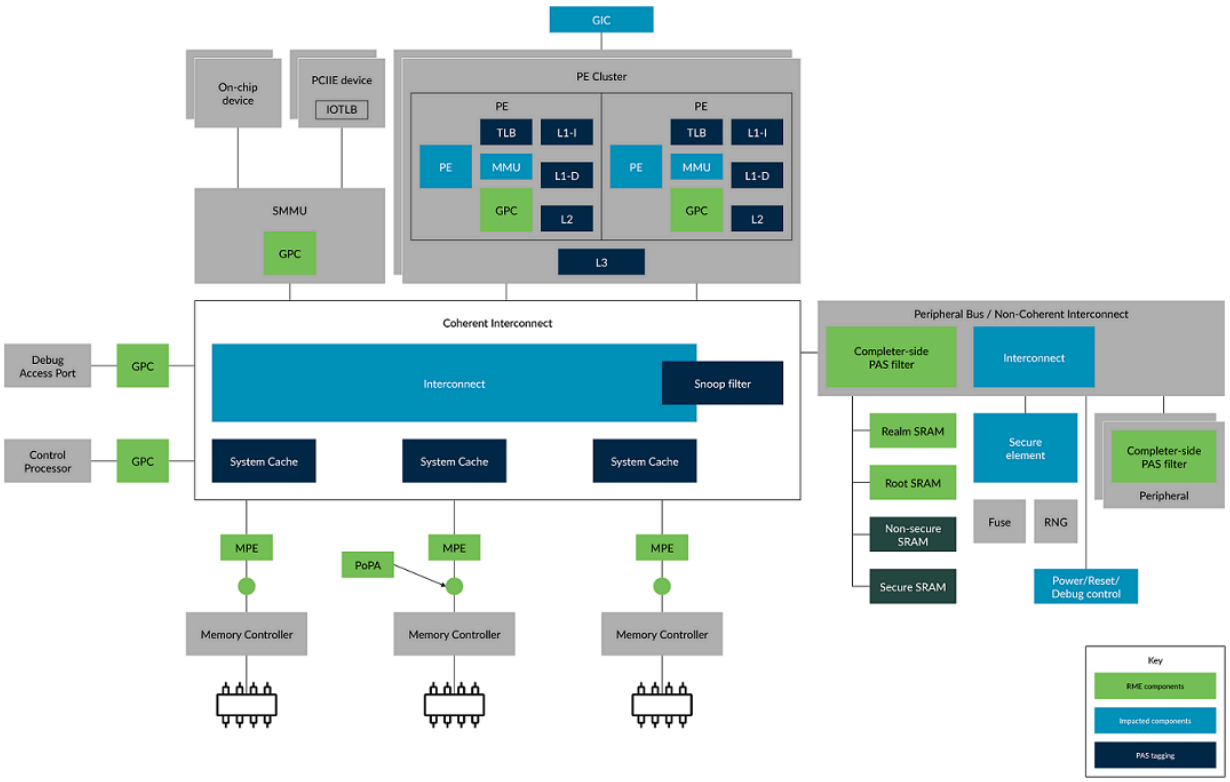
  Where an SMMU-originated access triggers a fault from the Granule Protection Checks, it is reported as though the SMMU experienced an External abort.

7. System architecture

RME is more than just a set of processor features. To take advantage of the features introduced in RME, we need support in the rest of the system.

The following diagram shows an example system and the components affected by the introduction of RME:

Figure 7-1: Example system with impacts of RME



Main memory protection

RME-enabled systems include memory encryption and, potentially, integrity. The baseline encryption requirement supports encryption for external memory, using a separate encryption key or tweak for each PA space. This encryption also provides spatial isolation using an address tweak.

This guide uses the term Memory Protection Engine (MPE) to describe the component that provides external memory encryption and integrity services.

MPAM

Pre-RME, the Memory Partitioning and Monitoring extension (MPAM) defined independent PARTID spaces for Non-secure and Secure states distinguished by the MPAM_NS value. MPAM_NS selects between Non-secure and Secure banks of resource usage monitors. Each

monitor can be sensitive to a PARTID or PARTID and PMG, or PMG alone. The PARTID space as conveyed by MPAM_NS is used to choose which monitor bank tracks the resource usage.

Sharing the PARTID space between Non-secure and realm Security states can be required for host resource management considerations. In some cases, sharing PARTID space can leak information on workload behavior. For example, monitoring realm accesses using a Non-secure monitor is a potential side channel. Allocating cache partitions to a realm is a mechanism that could be used in cache timing attacks.

With RME, MPAM_NS is replaced with a 2-bit MPAM_SP. This MPAM space field allows systems to implement four independent PARTID spaces, one for each Security state. MPAM_SP also defines how multiple Security states can share a single PARTID space. Security states allow the system integrator to decide how MPAM is implemented in an Arm CCA system. This implementation decision is based on the risks exposed by each MPAM Memory System Component (MSC).

RAS

A key requirement of RAS support for RME-enabled systems is to maintain the security isolation boundaries, of confidentiality and integrity, that RME provides. A challenge with this requirement is that you might want RAS to be triaged in the Non-secure state, in hypervisor or kernel code. This implies that PEs running in Non-secure state have direct access to sanitized Error Record registers.

RME introduces the idea of confidential information, which is information that is not accessible to the current Security state under normal operation. For example, Non-secure state does not normally have access to contents of a Secure or realm memory location.

RME places restrictions on what information is exposed by RAS:

- Root state can see information belonging to any Security state
- Secure state can see confidential information belonging to Secure or Non-secure state, but not root or realm
- Realm state can see confidential information belonging to realm or Non-secure state, but not root or Secure
- Non-secure state cannot see confidential information belonging to any other state

Not all information that is associated with an error is considered confidential. Usually, the address of the resource and the severity can be reported. For example, an RAS fault on a realm location could be reported to Non-secure state. The fault record could report the error type, and the address of the location that suffered the fault. It would not, however, be allowed to expose anything about the contents of that memory location.

8. Related information

Here are some resources that are related to the material in this guide:

- Arm community
- Confidential computing

9. Next steps

This guide introduces the Realm Management Extension (RME), an extension to the Armv9-A architecture.

To learn about the RME in more depth, read the Arm Architecture Reference Manual Supplement, The Realm Management Extension (RME), for Armv9-A.

To learn more about the Arm A-profile architecture, read Learn the Architecture guides.