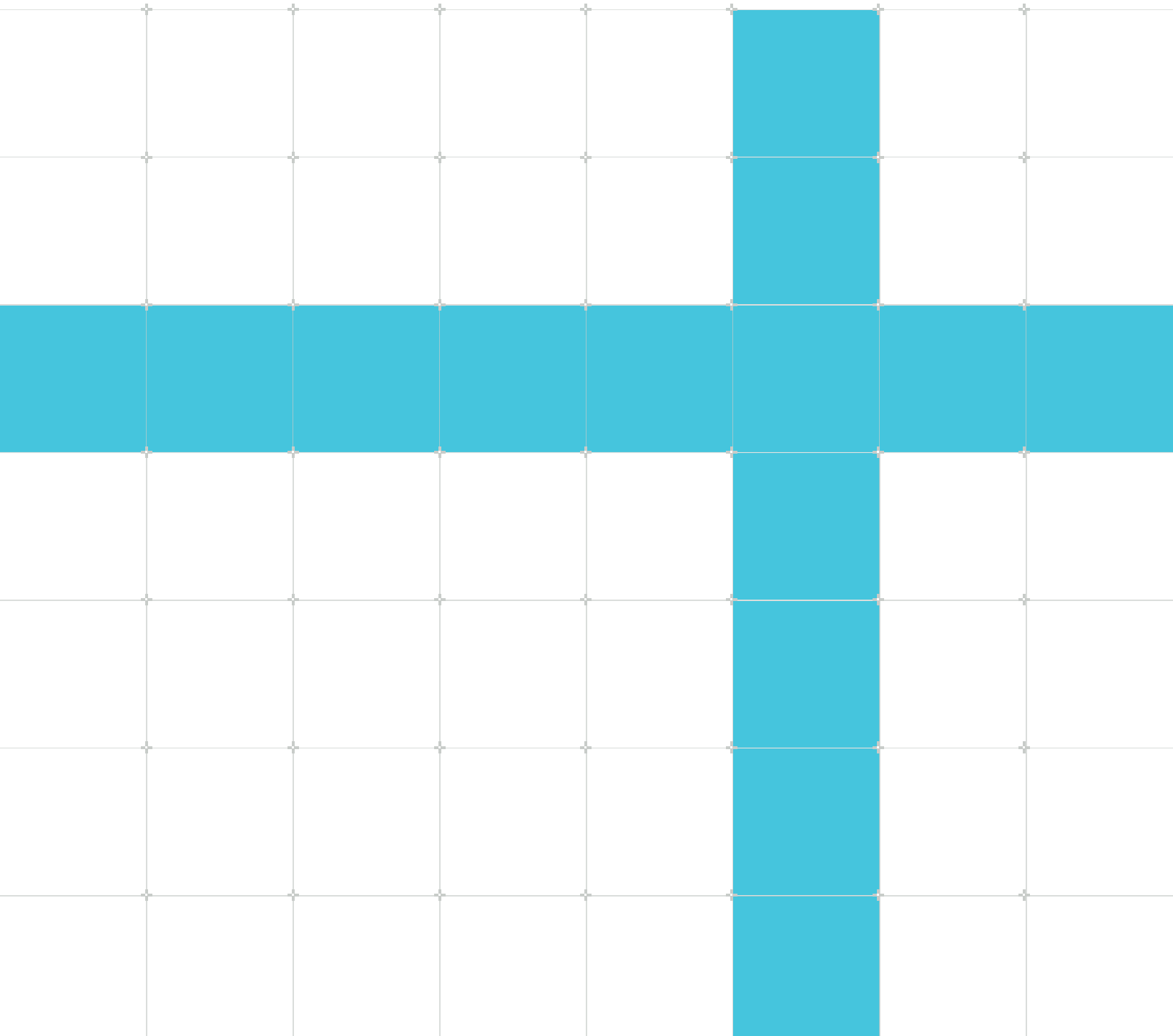
arm

Realm Management Extension (RME) and Memory Encryption Contexts (MEC) example software flows

Version 1.1

Non-Confidential

Issue 01
109839_0101_01_en



Realm Management Extension (RME) and Memory Encryption Contexts (MEC) example software flows

Release information

Document history

Issue	Date	Confidentiality	Change
0101-01	23 July 2024	Non-Confidential	Update
0100-01	11 June 2024	Non-Confidential	First release

Proprietary Notice

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

PRE-1121-V1.0

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 several programming examples of software flows. These flows help you understand the Realm Management Extension (RME) Arm Architecture.

This guide assumes that you are familiar with the Arm A-profile architecture and the Realm Management Extension.

This guide includes the following examples:

- Granule Transition Flow
- Procedures for changing the size of a GPT contiguous region
- Mismatched MECID avoidance
- Granule MECID association flows
- Primary and alternate MECID use models

2. RME examples

The following sections provide example software sequences for using the new features introduced within the The Realm Management Extension (RME), for Armv9-A.

2.1 Granule Transition Flow

The properties of the Granule Transition Flow (GTF) are:

1. The GTF changes the Physical Address Space (PAS) association of a physical granule from a previous physical address space to a new physical address space.
2. The GTF completes when the association of the physical granule with the new physical address space is observable.
3. When the GTF completes, the following outcomes are guaranteed:
   - Writes to the previous physical address space do not become observable.
   - If the previous physical address space is Realm or Secure, then no accesses, including Speculative read accesses, to the previous physical address space can observe unscrubbed values from before the GTF.
   - If the previous physical address space is Realm or Secure then no instructions, including execution under Speculation, can observe unscrubbed values from before the GTF.
   - If the previous physical address space is Realm or Secure then no accesses to the granule in the new physical address space observe unscrubbed values.
4. GTF outcomes are guaranteed by EL3 without relying on cooperative behavior of SW that has access to the previous physical address space (for example, software running at EL2) other than performing scrubbing operations where appropriate.

Delegate

This sequence transitions the physical granule at address `addr` from Non-secure to Secure or Realm physical address space.

On implementations with FEAT_MTE2, Root firmware must issue `DC_CIGDPAPA` instead of `DC_CIPAPA`, to additionally clean and invalidate Allocation Tags associated with the affected locations.

```
Delegate(phys_addr* addr, PAS target_pas) {

    // In order to maintain mutual distrust between Realm and Secure
    // states, remove any data speculatively fetched into the target
    // physical address space.
    for (i = 0; i<granule_size; i+=cache_line_size)
        DC_CIPPAPA((addr+i)), target_p);

    DSB(OSH);

    write_gpt(addr, target_pas)
    DSB(OSHST);
```

```
    TLBI_RPALOS(addr, granule_size);
    DSB(OSH)

    for (i = 0; i<granule_size; i+=cache_line_size)
        DC_CIPAPA((addr+i), PAS_Ns);

    DSB(OSH);

}
```

Undelegate

This sequence transitions the physical granule at address `addr` from Secure or Realm physical address space to Non-secure.

The sequence assumes that the EL2 software for Secure or Realm state has already scrubbed the appropriate locations.

On implementations with FEAT_MTE2, Root firmware must issue `DC_CIGDPAPA` instead of `DC_CIPAPA`, to additionally clean and invalidate Allocation Tags associated with the affected locations.

```
Undelegate(phys_addr* addr, PAS current_pas)

   // In order to maintain mutual distrust between Realm and Secure
   // states, remove access now, in order to guarantee that writes
   // to the currently-accessible physical address space will not
   // later become observable.
   write_gpt(addr, No_access);
   DSB(OSHST;)
   TLBI_RPALOS(addr, granule_size);
   DSB (OSH);

   // Ensure that the scrubbed data has made it past the PoPA
   for (i = 0; i<granule_size; i+=cache_line_size)
       DC_CIPAPA((addr+i), current_pas);

   DSB(OSH);

   // Ensure that GPC completes, non-coherent caches are properly flushed
   TLBI_RPALOS(addr, granule_size);
   DSB(OSH);


   // Remove any data loaded speculatively in NS space from before the scrubbing
   for (i = 0; i<granule_size; i+=cache_line_size)
       DC_CIPAPA((addr+i), PAS_NS);

   DSB(OSH);

   write_gpt (addr, PAS_NS);
   DSB(OSHST);

   // Ensure that all agents observe the new NS configuration
   TLBI_RPALOS(addr, granule_size);
   DSB(OSH);
```

2.2 Procedures for changing the size of a GPT contiguous region

The following example shows a procedure to increase contiguity from 4KB to 2MB, assuming PGS is 4KB. This example procedure does not consider mutual exclusion.

```
// Parameters:
// base = base address of desired contig region
// expected_gpi = value of all GPIs in the region

   assert IS_ALIGNED(base,2MB);
   assert IS_VALID_GPI (expected_gpi)

   // Required GPTE is 16 GPI values, all the same
   uint64_t required_gpte;
   required_gpte = expected_gpi;
   required_gpte |= required_gpte << 4;
   required_gpte |= required_gpte << 8;
   required_gpte |= required_gpte << 16;
   required_gpte |= required_gpte << 32;

   for(gpte_addr=base, gpte_addr < base+2MB, gpte_addr += 64KB) {
       actual_gpte = gpt_entry(gpte_addr);
       // All entries must be consistent before the change
       if (actual_gpte !=required_gpte)
           return false;
   }

   uint64_t new_gpte = 0x1; // Contiguous descriptor
   new_gpte |= expected_gpi<<4; // GPI field
   new_gpte |= 01<<8; // Contig field

   for(gpte_addr=base, gpte_addr < base+2MB, gpte_addr += 64KB) {
      set_gpt_entry(gpte_addr, new_gpte);
   }

   // No TLB maintenance required
   return true;
```

The following example shows a procedure to decrease contig from 2MB to 4KB, assuming PGS is 4KB.

This example procedure does not consider mutual exclusion.

```
// Parameters
// base = = base address of desired contig region
// expected_gpi = value of all GPIs in the region

    assert IS_ALIGNED(base, 2MB);
    assert IS_VALID_GPI(expected_gpi);

    // Required GPTE value
    uint64_t required_gpte = 0x1; // Contiguous descriptor
    required_gpte |= expected_gpi<<4; // GPI field
    required_gpte |= 01<<8; // Contig field

    for(gpte_addr=base, gpte_addr < base+2MB, gpte_addr += 64KB) {
        // All entries must be consistent before the change
        if (actual_gpte !=required_gpte)
            return false;
    }
```

```
// New GPTE is 16 GPI values, all the same
uint64_t new_gpte;
new_gpte = expected_gpi;
new_gpte |= new_gpte << 4;
new_gpte |= new_gpte << 8;
new_gpte |= new_gpte << 16;
new_gpte |= new_gpte << 32;

for(gpte_addr=base, gpte_addr < base+2MB, gpte_addr += 64KB) {
    set_gpt_entry(gpte_addr, new_gpte);
}

// It is assumed that the entries are being cracked so that they can
// be changed, for whatever reason. In which case it required to
// perform TLB maintenance now.
DSB();
TLBI_RPALOS(base, 2MB);
DSB();
return true;
```

3. MEC examples

The following sections provide example software sequences for using the new features introduced by Memory Encryption Contexts (MEC), for Armv9-A.

The sections refer:

- Generic “software” if the example is appropriate to more than one exception level.
- Conceptual “Realm Management software” when describing procedures for Realm EL2.

3.1 Mismatched MECID avoidance

Software must ensure that mismatched Memory Encryption Context ID (MECID) memory accesses do not occur, because they corrupt memory within the Realm PA space.

Software achieves this by ensuring that:

- All active translations to a physical granule use the same associated MECID value:
  - If multiple translations exist to a physical granule with different MECID values, then mismatched MECIDs can occur due to architected or speculative accesses.
- Granules are not accessible through an active translation when updating the associated MECID value in a system register.
- All cache lines are cleaned and invalidated to the PoE, before any change of associated MECID or MECID state.

3.2 Granule MECID association flows

The properties required for the granule MECID association flows are:

1. Granules are present in the Realm PA space, by following the Granule Transition Flow.
   - Granules removed from the Realm PA space, by the Granule Transition Flow, do not require DC CIPAE operations.
2. Granules that are not allocated to a Realm or used by Realm management software have the following properties:
   - The granule is not mapped by any translation table.
   - The granule’s cache lines are not present in any cache.
   - The granule may contain confidential data if the previously associated MECID is still valid.
3. Realm management software will track:
   - MECID association with every granule. For example, by tracking the Realm owner of each granule.
   - MECID validity and life cycle.

Remove a granule from Realm

The following is:

```
RemoveGranule(phys_addr* addr){

    ASSERT(IsAligned(addr, TG_size));
    // Remove stage 2 mapping for granule
    // including all TLBIs and DSBs
    unmap_granule_EL10_S2(addr)
   // Remove any R_EL2 stage 1 mappings if present
    unmap_granule_EL20_S1(addr)

   // Clean and invalidate granule to PoE
   for (i = 0; i<granule_size; i+=cache_line_size)
    DC_CIPAE((addr+i), REALM);

   DSB(OSH);
}
```

Note:Unmapping can happen in bulk before the CMOs.

Add a granule from Realm

The following is:

```
AddGranule(phys_addr* pa, ipa_addr+ ipa, old_MECID_state) {

    if( old_MECID_state == live ) {
        temp_va = map_granule_EL2_S1(pa, AMEC = 1 )

        // Scrub pa granule of correlate-able contents
        for (i = 0; i<granule_size; i+=cache_line_size)
            DC_ZVA(temp_va+i); // or writing random data instead of ZVA is
               →acceptable

        unmap_granule_EL2_S1(pa)

        // Assume a constant MECID value when operating on a Realm VM
        // where MECID_A_EL2 == VMECID_P_EL2
        // no need for additional CMOs after the scrub operation
    }
    // Add stage 2 mapping for pa granule
    map_granule_EL10_S2(ipa,pa)
}
```

Note: Scrubbing can happen in bulk before granule mapping.

Enable of SCTLR2_EL2.EMEC

SCTLR2_EL2.EMEC can be enabled from either:

- EL3 software, since EL2 is out-of-context.
- EL2 software if MECID_P0_EL2 == 0 and MECID_P1_EL2 == 0, to match the default MECID of zero.

Update AMEC value in a descriptor

Changing the AMEC bit is a destructive operation to the memory contents of that granule.

The RemoveGranule() flow should be applied to the granule before remapping with a new AMEC value, and in the general case, software is required to follow a break before make procedure when changing the AMEC value.

Update Primary MECID register

Primary MECID registers can be safely updated in the following situations:

- By a higher exception level.
- For Realm EL2 the MECID_P0_EL2 value of zero can be chosen, to align with the default MECID of zero, before enabling SCTLR2_EL2.EMEC.

Update Alternate MECID register

Alternate MECID registers can be safely updated when there are no valid translations associated with the MECID, and the TLBs with AMEC = 1 have been invalidated.

TCR2_EL2.AMEC0 and TCR2_EL2.AMEC1 are provided to allow Realm EL2 software to update MECID_A0_EL2 or MECID_A1_EL2 respectively, by preventing any subsequent TTD.AMEC == 1 translations.

MECID reuse and reset of MECID tweak value

Resetting the MECID tweak value is a **IMPLEMENTATION DEFINED** function provided by EL3 firmware. The MECID tweak will be assigned a new value, unique within the current boot session.

Resetting a MECID tweak is destructive to all memory contents associated with the previous tweak value.

To guarantee data destruction and prepare all granules previously associated with a MECID for reuse, the granules should be either:

- Transitioned to the Non-secure PA space using the EL3 Granule Transition Flow.
- Processed using the RemoveGranule() flow.

3.3 Primary and alternate MECID use models

The following are examples of Primary and alternate MECID use models.

Realm EL1&0 translation regimes

This regime includes two translation tables. This translation regime is subject to stage 2 translation.

Primary MECID

The VMECID_P_EL2 is the primary MECID, for the stage 2 Realm EL1&0 translation regime. VMECID_P_EL2 is used implicitly by the EL1&0 Realm VM to protect the Realm VM state and provide consistent security guarantees. The Realm state includes:

- Private code and data pages
- EL1&0 stage 1 translation tables

- EL1&0 stage 2 translation tables

Alternate MECID

The VMECID_A_EL2 is the alternate MECID, for the stage 2 Realm EL1&0 translation regime. VMECID_A_EL2 is designed to support the secondary use cases of:

- Shared data between two or more Realm VMs
- Shared code between two or more Realm VMs

A shared data model between Realm VMs has significant challenges with mutual authentication, that exceed the current scope of the CCA Security Model.

MEC provides only one alternate context per EL1 Realm. This greatly limits the creation of 1:1 encrypted shared memory channels, between multiple Realms.

Realm EL2&0 translation regimes

When HCR_EL2.E2H == 1 the Realm EL2&0 translation regime has two sets of translation tables, referenced by TTBR0_EL2 and TTBR1_EL2, each with their own primary and alternate MECIDs.

A common usage model is for supervisory software to use TTBR1_EL2 translation tables for its own code and data, and dynamically use the TTBR0_EL2 translation tables for the software it is currently supervising. The following explanations of EL2 MECIDs assumes this model.

Primary 0 MECID

The EL2&0 primary 0 MECID, when set to be equal VMECID_P_EL2, can enable the Realm management software to access the Realm VM state and stage 2 translation tables.

And optionally, Realm management software may protect additional Realm state with this MECID:

- Meta-data
- vCPU contexts

This MECID is configured in MECID_P0_EL2 for the TTBR0_EL2 translation table.

The primary Realm VM MECID value will be identical between VMECID_P_EL2 and MECID_P0_EL2

Alternate 0 MECID

The alternative 0 MECID can be used by the Realm management software to implement use cases such as code or data sharing.

Realm management software having simultaneous access to primary 0 MECID and alternate 0 MECID of a Realm VM, can allow for efficient implementation of data sharing schemes that require data copying between Realm contexts.

This MECID is configured in MECID_A0_EL2 for the TTBR0_EL2 translation tables. For Realm management software to accesses memory shared with a Realm VM, the alternate Realm VM MECID value, VMECID_A_EL2, would be identical to that in MECID_A0_EL2.

Primary 1 MECID

The primary 1 MECID can be used by the Realm management software for its own code, private data and the TTBR1_EL2 and TTBR0_EL2 Realm EL2 translation tables.

This MECID is configured in MECID_P1_EL2 for the TTBR1_EL2 translation tables.

Alternate 1 MECID

The alternate 1 MECID can be used by the Realm management software for private data it wishes to isolate from the primary 1 MECID context, if the Realm management software security model requires such isolation. This MECID is configured in MECID_A1_EL2 for the TTBR1_EL2 translation tables.

EL3 translation regime

All accesses to the Secure, Non-Secure and Root PA space from EL3, use the default MECID zero.

Memory accesses to the Realm PAS, when translation table descriptor {NSE,NS} = {1,1}, are associated with the MECID configured in MECID_RL_A_EL3.

MECID_RL_A_EL3 is most likely to contain the MECID associated with the Realm management software.

EL3 Monitor can use MECID_RL_A_EL3 when accessing the Realm EL2 memory space to:

- Populate Realm PA space memory during Realm EL2 boot
- Write attestation reports for Realm management software

If non-delegated attestation signing is in use, the Monitor could directly write into the requesting Realm's memory, provided it is supplied with the granule PA and Realm MECID

4. Related information

The following resources are related to material in this guide:

- Arm Architecture Reference Manual for A-profile architecture
- Arm System Memory Management Unit Architecture (SMMU) Specification
- Learn the Architecture - SMMU Software Guide
- Arm Realm Management Extension (RME) System Architecture
- Learn the architecture - Realm Management Extension
- Learn the architecture - Introducing Arm Confidential Compute Architecture