Realm Management Monitor specification

Document number	DEN0137
Document quality	BET
Document version	A-bet0
Document confidentiality	Non-confidential
Document build information	d96e12d7 doctool 0.53.1-af18927f

Quality Status: Beta (BET)

Beta quality status has a particular meaning to Arm of which the recipient must be aware. At this quality level the release is sufficiently stable and committed for initial product development. The recipient can expect some changes to the Beta quality released material.

Realm Management Monitor specification

Release information

Date	Version	Changes
2022/Jul/15	A-bet0	• Initial Beta release

Arm Non-Confidential Document Licence (“Licence”)

This Licence is a legal agreement between you and Arm Limited (“**Arm**”) for the use of Arm’s intellectual property (including, without limitation, any copyright) embodied in the document accompanying this Licence (“**Document**”). Arm licenses its intellectual property in the Document to you on condition that you agree to the terms of this Licence. By using or copying the Document you indicate that you agree to be bound by the terms of this Licence.

“**Subsidiary**” means any company the majority of whose voting shares is now or hereafter owner or controlled, directly or indirectly, by you. A company shall be a Subsidiary only for the period during which such control exists.

This Document is **NON-CONFIDENTIAL** and any use by you and your Subsidiaries (“Licensee”) is subject to the terms of this Licence between you and Arm.

Subject to the terms and conditions of this Licence, Arm hereby grants to Licensee under the intellectual property in the Document owned or controlled by Arm, a non-exclusive, non-transferable, non-sub-licensable, royalty-free, worldwide licence to:

(i) use and copy the Document for the purpose of designing and having designed products that comply with the Document;

(ii) manufacture and have manufactured products which have been created under the licence granted in (i) above; and

(iii) sell, supply and distribute products which have been created under the licence granted in (i) above.

Licensee hereby agrees that the licences granted above shall not extend to any portion or function of a product that is not itself compliant with part of the Document.

Except as expressly licensed above, Licensee acquires no right, title or interest in any Arm technology or any intellectual property embodied therein.

THE 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. Arm may make changes to the Document at any time and without notice. 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, third party patents, copyrights, trade secrets, or other rights.

NOTWITHSTANDING ANYTHING TO THE CONTRARY CONTAINED IN THIS LICENCE, TO THE FULLEST EXTENT PERMITTED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, IN CONTRACT, TORT OR OTHERWISE, IN CONNECTION WITH THE SUBJECT MATTER OF THIS LICENCE (INCLUDING WITHOUT LIMITATION) (I) LICENSEE’S USE OF THE DOCUMENT; AND (II) THE IMPLEMENTATION OF THE DOCUMENT IN ANY PRODUCT CREATED BY LICENSEE UNDER THIS LICENCE). THE EXISTENCE OF MORE THAN ONE CLAIM OR SUIT WILL NOT ENLARGE OR EXTEND THE LIMIT. LICENSEE RELEASES ARM FROM ALL OBLIGATIONS, LIABILITY, CLAIMS OR DEMANDS IN EXCESS OF THIS LIMITATION.

This Licence shall remain in force until terminated by Licensee or by Arm. Without prejudice to any of its other rights, if Licensee is in breach of any of the terms and conditions of this Licence then Arm may terminate this Licence immediately upon giving written notice to Licensee. Licensee may terminate this Licence at any time. Upon termination of this Licence by Licensee or by Arm, Licensee shall stop using the Document and destroy all copies of the Document in its possession. Upon termination of this Licence, all terms shall survive except for the licence grants.

Any breach of this Licence by a Subsidiary shall entitle Arm to terminate this Licence as if you were the party in breach. Any termination of this Licence shall be effective in respect of all Subsidiaries. Any rights granted to any Subsidiary hereunder shall automatically terminate upon such Subsidiary ceasing to be a Subsidiary.

The Document consists solely of commercial items. Licensee shall be responsible for ensuring that any use, duplication or disclosure of the Document complies fully with any relevant export laws and regulations to assure that the Document or any portion thereof is not exported, directly or indirectly, in violation of such export laws.

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

LES-PRE-21585 version 4.0

Contents

Realm Management Monitor specification

Preface

Conventions

Typographical conventions

The typographical conventions are:

italic

Introduces special terminology, and denotes citations.

`monospace`

Used for pseudocode and source code examples.

Also used in the main text for instruction mnemonics and for references to other items appearing in pseudocode and source code examples.

SMALL CAPITALS

Used for some common terms such as IMPLEMENTATION DEFINED.

Used for a few terms that have specific technical meanings, and are included in the Glossary.

Red text

Indicates an open issue.

Blue text

Indicates a link. This can be

- A cross-reference to another location within the document
- A URL, for example http://developer.arm.com

Numbers

Numbers are normally written in decimal. Binary numbers are preceded by `0b`, and hexadecimal numbers by `0x`. In both cases, the prefix and the associated value are written in a monospace font, for example `0xFFFF0000`. To improve readability, long numbers can be written with an underscore separator between every four characters, for example `0xFFFF_0000_0000_0000`. Ignore any underscores when interpreting the value of a number.

Pseudocode descriptions

This book uses a form of pseudocode to provide precise descriptions of the specified functionality. This pseudocode is written in a monospace font. The pseudocode language is described in the Arm Architecture Reference Manual.

Addresses

Unless otherwise stated, the term *address* in this specification refers to a physical address.

Rules-based writing

This specification consists of a set of individual *content items*. A content item is classified as one of the following:

- Declaration
- Rule
- Goal
- Information
- Rationale
- Implementation note
- Software usage

Declarations and Rules are normative statements. An implementation that is compliant with this specification must conform to all Declarations and Rules in this specification that apply to that implementation.

Declarations and Rules must not be read in isolation. Where a particular feature is specified by multiple Declarations and Rules, these are generally grouped into sections and subsections that provide context. Where appropriate, these sections begin with a short introduction.

Arm strongly recommends that implementers read *all* chapters and sections of this document to ensure that an implementation is compliant.

Content items other than Declarations and Rules are informative statements. These are provided as an aid to understanding this specification.

Content item identifiers

A content item may have an associated identifier which is unique among content items in this specification.

After this specification reaches beta status, a given content item has the same identifier across subsequent versions of the specification.

Content item rendering

In this document, a content item is rendered with a token of the following format in the left margin: L_{iiiii}

- *L* is a label that indicates the content class of the content item.
- *iiiii* is the identifier of the content item.

Content item classes

Declaration

A Declaration is a statement that does one or more of the following:

- Introduces a concept
- Introduces a term
- Describes the structure of data
- Describes the encoding of data

A Declaration does not describe behaviour.

A Declaration is rendered with the label *D*.

Rule

A Rule is a statement that describes the behaviour of a compliant implementation.

A Rule explains what happens in a particular situation.

A Rule does not define concepts or terminology.

A Rule is rendered with the label *R*.

Goal

A Goal is a statement about the purpose of a set of rules.

A Goal explains why a particular feature has been included in the specification.

A Goal is comparable to a “business requirement” or an “emergent property.”

A Goal is intended to be upheld by the logical conjunction of a set of rules.

A Goal is rendered with the label *G*.

Information

An Information statement provides information and guidance as an aid to understanding the specification.

An Information statement is rendered with the label *I*.

Rationale

A Rationale statement explains why the specification was specified in the way it was.

A Rationale statement is rendered with the label *X*.

Implementation note

An Implementation note provides guidance on implementation of the specification.

An Implementation note is rendered with the label *U*.

Software usage

A Software usage statement provides guidance on how software can make use of the features defined by the specification.

A Software usage statement is rendered with the label *S*.

Additional reading

This section lists publications by Arm and by third parties.

See Arm Developer (http://developer.arm.com) for access to Arm documentation.

[1] *Introducing Arm CCA*. (ARM DEN 0125) Arm Limited.

[2] *Arm Architecture Reference Manual Supplement, The Realm Management Extension (RME), for Armv9-A*. (ARM DDI 0615) Arm Ltd.

[3] *Arm Architecture Reference Manual for Armv8-A architecture profile*. (ARM DDI 0487 G.b) Arm Ltd.

[4] *Arm CCA Security model*. (ARM DEN 0096) Arm Limited.

[5] *Arm Generic Interrupt Controller (GIC) Architecture Specification version 3 and version 4*. (ARM IHI 0069 G) Arm Ltd.

[6] *Concise Binary Object Representation (CBOR)*.

[7] *CBOR Object Signing and Encryption (COSE)*.

[8] *Entity Attestation Token (EAT)*.

[9] *Concise Data Definition Language (CDDL)*.

[10] *IANA Hash Function Textual Names*.

[11] *SEC 1: Elliptic Curve Cryptography, version 2.0*.

[12] *Tormore system architecture spec*. (ARM AES 0015) Arm Ltd.

[13] *Arm SMC Calling Convention*. (ARM DEN 0028 D) Arm Ltd.

[14] *Arm Specification Language Reference Manual*. (ARM DDI 0612) Arm Ltd.

[15] *Secure Hash Standard (SHS)*.

[16] *Arm Power State Coordination Interface (PSCI)*. (ARM DEN 0022 D.b) Arm Ltd.

Feedback

Arm welcomes feedback on its documentation.

Feedback on this book

If you have comments on the content of this book, send an e-mail to errata@arm.com. Give:

- The title (Realm Management Monitor specification).
- The number (DEN0137 A-bet0).
- The page numbers to which your comments apply.
- The rule identifiers to which your comments apply, if applicable.
- A concise explanation of your comments.

Arm also welcomes general suggestions for additions and improvements.

Note

Arm tests PDFs only in Adobe Acrobat and Acrobat Reader, and cannot guarantee the appearance or behavior of any document when viewed with any other PDF reader.

Open issues

The following table lists known open issues in this version of the document.

Key	Description

Part A
Architecture

Chapter A1
Overview

The RMM is a software component which forms part of a system which implements the Arm Confidential Compute Architecture (Arm CCA). Arm CCA is an architecture which provides protected execution environments called *Realms*.

The threat model which Arm CCA is designed to address is described in *Introducing Arm CCA* [1].

The hardware architecture of Arm CCA is called the Realm Management Extension (RME), and is described in *Arm Architecture Reference Manual Supplement, The Realm Management Extension (RME), for Armv9-A* [2].

A1.1 Confidential computing

The Armv8-A architecture (*Arm Architecture Reference Manual for Armv8-A architecture profile* [3]) includes mechanisms that establish a privilege hierarchy. Software operating at higher privilege levels is responsible for managing the resources (principally memory and processor cycles) that are used by entities at lower privilege levels.

Prior to Arm CCA, resource management was coupled with a right of access. That is, a resource that is managed by a higher-privileged entity is also accessible by it. A *Realm* is a protected execution environment for which this coupling is broken, so that the right to manage resources is separated from the right to access those resources.

The purpose of a Realm is to provide to the Realm owner an environment for confidential computing, without requiring the Realm owner to trust the software components that manage the resources used by the Realm.

Construction of a Realm, and allocation of resources to a Realm at runtime, are the responsibility of the Virtual Machine Monitor (VMM). In this specification, the term *Host* is used to refer to the VMM.

See also:

- A2.1 *Realm*

A1.2 System software components

The system software architecture of Arm CCA is summarised in the following figure.

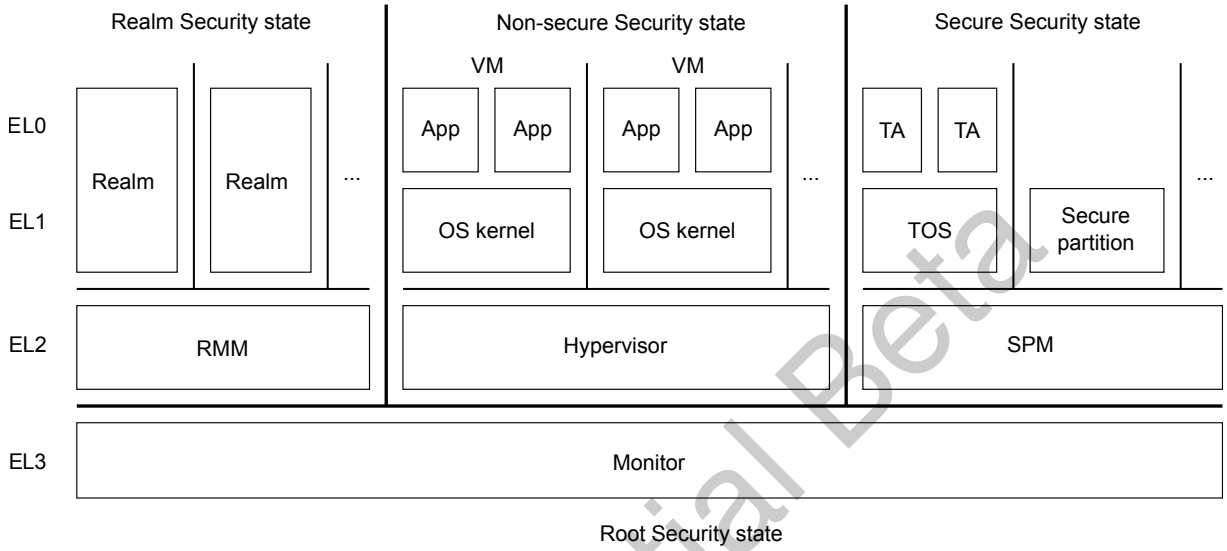


Figure A1.1: System software architecture

The components shown in the diagram are listed below.

Component	Description
Monitor	The most privileged software component, which is responsible for switching between the Security states used at EL2, EL1 and EL0.
Realm	A protected execution environment.
Realm Management Monitor (RMM)	The software component which is responsible for the management of Realms.
Virtual Machine (VM)	An execution environment within which an operating system can run. Note that a Realm is a VM which executes in the Realm security state.
Hypervisor	The software component which is responsible for the management of VMs.
Secure Partition Manager (SPM)	The software component which is responsible for the management of Secure Partitions.
Trusted OS (TOS)	An operating system which runs in a Secure Partition.
Trusted Application (TA)	An application hosted by a TOS.

A1.3 Realm Management Monitor

The Realm Management Monitor (RMM) is the system component that is responsible for the management of Realms.

The responsibilities of the RMM are to:

- Provide services that allow the Host to create, populate, execute and destroy Realms.
- Provide services that allow the initial configuration and contents of a Realm to be attested.
- Protect the confidentiality and integrity of Realm state during the lifetime of the Realm.
- Protect the confidentiality of Realm state during and following destruction of the Realm.

The RMM exposes the following interfaces, which are accessed via SMC instructions, to the Host:

- The *Realm Management Interface* (RMI), which provides services for the creation, population, execution and destruction of Realms.

The RMM exposes the following interfaces, which are accessed via SMC instructions, to Realms:

- The *Realm Services Interface* (RSI), which provides services used to manage resources allocated to the Realm, and to request an attestation report.
- The *Power State Coordination Interface* (PSCI), which provides services used to control power states of VPEs within a Realm. Note that the HVC conduit for PSCI is not supported for Realms.

The RMM operates by manipulating data structures which are stored in memory accessible only to the RMM.

See also:

- Chapter B3 *Realm Management Interface*
- Chapter B4 *Realm Services Interface*
- Chapter B5 *Power State Control Interface*

Chapter A2
Concepts

This chapter introduces the following concepts which are central to the RMM architecture:

A2.1 Realm

This section describes the concept of a Realm.

A2.1.1 Overview

D_{DLRSR} A *Realm* is an execution environment which is protected from agents in the Non-secure and Secure Security states, and from other Realms.

A2.1.2 Realm execution environment

I_{LQYLY} The execution environment of a Realm is an EL0 + EL1 environment, as described in *Arm Architecture Reference Manual for Armv8-A architecture profile* [3].

A2.1.2.1 Realm registers

R_{NJHQK} On first entry to a Realm VPE, PE state is initialized according to "PE state on reset to AArch64 state" in *Arm Architecture Reference Manual for Armv8-A architecture profile* [3], except for GPR and PC values which are specified by the Host during Realm creation.

G_{ZFCQX} Confidentiality is guaranteed for a Realm VPE's general purpose and SIMD / floating point registers.

G_{QHZCS} Confidentiality is guaranteed for other Realm VPE register state (including stack pointer, program counter and EL0 / EL1 system registers).

G_{XRMHP} Integrity is guaranteed for a Realm VPE's general purpose and SIMD / floating point registers.

G_{YKRWG} Integrity is guaranteed for other Realm VPE register state (including stack pointer, program counter and EL0 / EL1 system registers).

I_{GPGFB} A Realm can use a Host call to pass arguments to the Host and receive results from the Host.

See also:

- A2.3 *Realm Execution Context*
- A4.5 *Host call*
- B3.3.9 *RMI_REALM_CREATE command*

A2.1.2.2 Realm memory

I_{TQMMZ} A Realm is able to determine whether a given IPA is *protected* or *unprotected*.

G_{LQFQH} Confidentiality is guaranteed for memory contents accessed via a protected address. Informally, this means that a change to the contents of such a memory location is not observable by any agent outside the *CCA platform*.

G_{QMLCJ} Integrity is guaranteed for memory contents accessed via a protected address. Informally, this means that the Realm does not observe the contents of the location to change unless the Realm itself has either written a different value to the location, or provided consent to the RMM for integrity of the location to be violated.

See also:

- A5.2.1 *Realm IPA space*

A2.1.2.3 Realm processor features

R_{JGHYJ} The value returned to a Realm from reading a feature register is architecturally valid and describes the set of features which are present in the Realm's execution environment.

I_{KKBDP} The RMM may suppress a feature which is supported by the underlying hardware platform, if exposing that feature to a Realm could lead to a security vulnerability.

See also:

- A3.1 *Realm feature discovery and selection*

A2.1.2.4 IMPDEF system registers

R_{FQCKH} A Realm read from or write to an IMPLEMENTATION DEFINED system register causes an Unknown exception taken to the Realm.

A2.1.3 Realm attributes

This section describes the attributes of a Realm.

D_{JSGFY} A *Realm attribute* is a property of a Realm whose value can be observed or modified either by the Host or by the Realm.

I_{TTDVX} An example of a way in which a Realm attribute may be observable is the outcome of an RMM command.

D_{MHJCK} The attributes of a Realm are summarized in the following table.

Name	Type	Description
ipa_width	UInt8	IPA width in bits
measurements	RmmRealmMeasurement[5]	Realm measurements
hash_algo	RmmHashAlgorithm	Algorithm used to compute Realm measurements
rec_index	UInt64	Index of next REC to be created
rtt_base	Address	Realm Translation Table base address
rtt_level_start	Int64	RTT starting level
rtt_num_start	UInt64	Number of physically contiguous starting level RTTs
state	RmmRealmState	Lifecycle state
vmid	Bits16	Virtual Machine Identifier
rpv	Bits512	Realm Personalization Value

D_{MGGPT} A *Realm Initial Measurement* (RIM) is a measurement of the configuration and contents of a Realm at the time of activation.

D_{GRFCS} A *Realm Extensible Measurement* (REM) is a measurement value which can be extended during the lifetime of a Realm.

I_{FMPYL} Attributes of a Realm include an array of measurement values. The first entry in this array is a RIM. The remaining entries in this array are REMs.

X_{DNDKV} During Realm creation, the Host provides ipa_width, rtt_level_start and rtt_num_start values as Realm parameters. According to the VMSA, the rtt_num_start value is architecturally defined as a function of the ipa_width and rtt_level_start values. It would therefore have been possible to design the Realm creation interface such that the Host provided only the ipa_width and rtt_level_start values. However, this would potentially allow a Realm to be successfully created, but with a configuration which did not match the Host's intent. For this reason, it was decided that the Host should specify all three values explicitly, and that Realm creation should fail if the values are not consistent. See *Arm Architecture Reference Manual for Armv8-A architecture profile* [3] for further details.

I_{QRVTT} The VMID of a Realm is chosen by the Host. The VMID must be within the range supported by the hardware platform. The RMM ensures that every Realm on the system has a unique VMID.

D_{FTWBK} A *Realm Personalization Value* (RPV) is a provided by the Host, to distinguish between Realms which have the same Realm Initial Measurement, but different behavior.

S_{FCNBF} Possible uses of the RPV include:

- A GUID

- Hash of Realm Owner public key
- Hash of a "personalisation document" which is provided to the Realm via a side-band (for example, via NS memory) and contains configuration information used by Realm software.

I_{ZFSWC} The RMM treats the RPV as an opaque value.

I_{BFSRK} The RPV is included in the Realm attestation report as a separate claim.

See also:

- A2.1.5 *Realm lifecycle*
- A2.3 *Realm Execution Context*
- A3.1.2 *Realm LPA2 and IPA width*
- A5.2.1 *Realm IPA space*
- A5.5 *Realm Translation Table*
- A7.1 *Realm measurements*
- A7.2.3.1.2 *Realm Personalization Value claim*
- C1.10 *RmmRealm type*

A2.1.4 Realm liveness

D_{WTXTJ} *Realm liveness* is a property which means that there exists one or more Granules, other than the RD and the starting level RTTs, which are owned by the Realm.

I_{PVPQB} If a Realm is live, it cannot be destroyed.

D_{PCKRN} A Realm is *live* if any of the following is true:

- The number of RECs owned by the Realm is not zero
- A starting level RTT of the Realm is live

I_{VKKPJ} If a Realm owns a non-zero number of Data Granules, this implies that it has a starting level RTT which is live, and therefore that the Realm itself is live.

See also:

- A2.1.5 *Realm lifecycle*
- A2.2.2 *Granule ownership*
- A2.2.3 *Granule lifecycle*
- A2.3 *Realm Execution Context*
- A5.5.8 *RTT liveness*
- B2.19 *RealmIsLive function*
- B3.3.10 *RMI_REALM_DESTROY command*

A2.1.5 Realm lifecycle

See also:

- Chapter A3 *Realm creation*
- D1.2 *Realm lifecycle flows*

A2.1.5.1 States

D_{GDQPJ} The states of a Realm are listed below.

State	Description
NEW	Under construction. Not eligible for execution.
ACTIVE	Eligible for execution.
SYSTEM_OFF	System has been turned off. Not eligible for execution.

A2.1.5.2 State transitions

IRRHFG Permitted Realm state transitions are shown in the following table. The rightmost column lists the events which can cause the corresponding state transition.

A transition from *NULL* represents creation of a Realm object. A transition to *NULL* represents destruction of a Realm object.

From state	To state	Events
NULL	NEW	RMI_REALM_CREATE
NEW	*NULL*	RMI_REALM_DESTROY
ACTIVE	*NULL*	RMI_REALM_DESTROY
NEW	ACTIVE	RMI_REALM_ACTIVATE
ACTIVE	SYSTEM_OFF	PSCI_SYSTEM_OFF PSCI_SYSTEM_RESET

IYCPWW Permitted Realm state transitions are shown in the following figure. Each arc is labeled with the events which can cause the corresponding state transition.

A transition from *NULL* represents creation of an RD. A transition to *NULL* represents destruction of an RD.

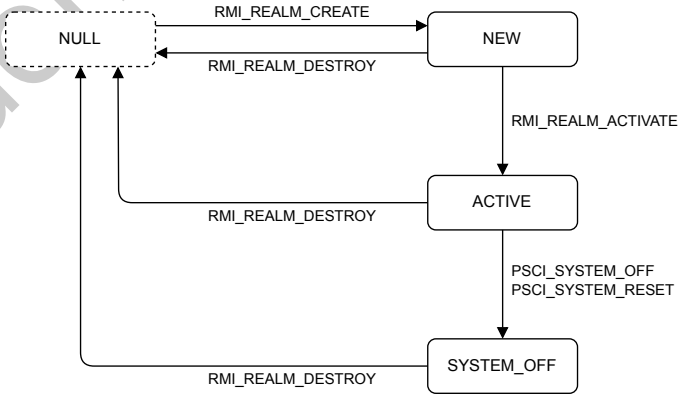


Figure A2.1: Realm state transitions

See also:

- B3.3.8 *RMI_REALM_ACTIVATE command*
- B3.3.9 *RMI_REALM_CREATE command*
- B3.3.10 *RMI_REALM_DESTROY command*
- B5.3.6 *PSCI_SYSTEM_OFF command*
- B5.3.7 *PSCI_SYSTEM_RESET command*

A2.1.6 Realm parameters

DTGMVZ A *Realm parameter* is a value which is provided by the Host during Realm creation.

See also:

- A2.1.3 *Realm attributes*
- A3.1 *Realm feature discovery and selection*
- B2.20 *RealmParams function*

- B3.3.9 *RMI_REALM_CREATE command*
- B3.4.10 *RmiRealmParams type*

A2.1.7 Realm Descriptor

D_{TNSBY} A *Realm Descriptor* (RD) is an RMM data structure which stores attributes of a Realm.

D_{GGKWX} The size of an RD is one Granule.

See also:

- A2.1.3 *Realm attributes*
- A2.2.3 *Granule lifecycle*

A2.2 Granule

This section describes the concept of a Granule.

D_{NBXXX} A *Granule* is a unit of physical memory whose size is 4KB.

I_{DJGZW} A Granule may be used to store one of the following:

- Code or data used by the Host
- Code or data used by software in the Secure Security state
- Code or data used by a Realm
- Data used by the RMM to manage a Realm

The use of a Granule is reflected in its lifecycle state.

D_{ZVRXC} A Granule is *delegable* if it can be delegated by the Host for use by the RMM or by a Realm.

U_{KHKLP} In a typical implementation, all memory which is presented to the Host as RAM is delegable. Examples of non-delegable memory may include the following:

- Memory which is carved out for use by the Root world, the RMM or the Secure world
- Device memory

See also:

- A2.2.1 *Granule attributes*
- A2.2.3 *Granule lifecycle*

A2.2.1 Granule attributes

This section describes the attributes of a Granule.

D_{JPBBC} A *Granule attribute* is a property of a Granule whose value can be observed or modified either by the Host or by a Realm.

I_{WVXGK} Examples of ways in which a Granule attribute may be observable include the outcome of an RMM command, and whether a memory access generates a fault.

D_{DVMRF} The attributes of a Granule are summarized in the following table.

Name	Type	Description
pas	RmmPhysicalAddressSpace	Physical Address Space
state	RmmGranuleState	Lifecycle state

D_{QZNGW} The set of Physical Address Spaces is:

- NS
- REALM
- OTHER

X_{LQZFB} The RMM cannot distinguish whether a Granule is in the Secure or Root PAS, so these two values are combined as OTHER.

I_{YYVSN} If the state of a Granule is not UNDELEGATED then the PAS of the Granule is REALM.

I_{BQDWY} If the state of a Granule is UNDELEGATED then the PAS of the Granule is not REALM.

I_{MPGJV} If the state of a Granule is UNDELEGATED then the RMM does not prevent the PAS of the Granule from being changed by another agent to any value except REALM.

D_{VRSKZ} An *NS Granule* is a Granule whose PAS is NS.

See also:

- A2.1 *Realm*
- A2.1.7 *Realm Descriptor*
- A2.2.3 *Granule lifecycle*
- C1.1 *RmmGranule type*

A2.2.2 Granule ownership

I_{DMVQM} A Granule whose state is neither UNDELEGATED nor DELEGATED is owned by a Realm.

I_{PRNTM} The owner of a Granule is identified by the address of a Realm Descriptor (RD).

I_{ZXBZM} For a Granule whose state is RD, the ownership relation is recursive: the owning Realm is identified by the address of the RD itself.

I_{TYHTD} A Granule whose state is RTT is one of the following:

- A starting level RTT. The address of this RTT is stored in the RD of the owning Realm.
- A non-starting level RTT. The address of this RTT is stored in its parent RTT, in an RTT entry whose state is TABLE. Recursively following the parent relationship leads to the RD of the owning Realm.

I_{QCNRM} A Granule whose state is DATA is mapped at a Protected IPA, in an RTT entry whose state is ASSIGNED. The Realm which owns the RTT is the owner of the DATA Granule.

I_{HHPVB} A REC has an “owner” attribute which points to the RD of the owning Realm.

X_{NDNHG} A REC is not mapped at a Protected IPA. Its ownership therefore needs to be recorded explicitly.

See also:

- A2.1 *Realm*
- A2.1.7 *Realm Descriptor*
- A2.3 *Realm Execution Context*
- A5.2.1 *Realm IPA space*
- A5.5 *Realm Translation Table*
- B3.3.1 *RMI_DATA_CREATE command*
- B3.3.2 *RMI_DATA_CREATE_UNKNOWN command*
- B3.3.12 *RMI_REC_CREATE command*
- B3.3.15 *RMI_RTT_CREATE command*

A2.2.3 Granule lifecycle

A2.2.3.1 States

D_{MPLGT} The states of a Granule are listed below.

State	Description
UNDELEGATED	Not delegated for use by the RMM.
DELEGATED	Delegated for use by the RMM.
RD	Realm Descriptor.
REC	Realm Execution Context.
REC_AUX	Realm Execution Context auxiliary Granule.
DATA	Realm code or data.

State	Description
RTT	Realm Translation Table.

A2.2.3.2 State transitions

IZJBTT Permitted Granule state transitions are shown in the following table. The rightmost column lists the events which can cause the corresponding state transition.

From state	To state	Events
UNDELEGATED	DELEGATED	RMI_GRANULE_DELEGATE
DELEGATED	UNDELEGATED	RMI_GRANULE_UNDELEGATE
DELEGATED	RD	RMI_REALM_CREATE
RD	DELEGATED	RMI_REALM_DESTROY
DELEGATED	DATA	RMI_DATA_CREATE RMI_DATA_CREATE_UNKNOWN
DATA	DELEGATED	RMI_DATA_DESTROY
DELEGATED	REC	RMI_REC_CREATE
REC	DELEGATED	RMI_REC_DESTROY
DELEGATED	REC_AUX	RMI_REC_CREATE
REC_AUX	DELEGATED	RMI_REC_DESTROY
DELEGATED	RTT	RMI_REALM_CREATE RMI_RTT_CREATE
RTT	DELEGATED	RMI_REALM_DESTROY RMI_RTT_DESTROY

IVVGVM Permitted Granule state transitions are shown in the following figure. Each arc is labeled with the events which can cause the corresponding state transition.

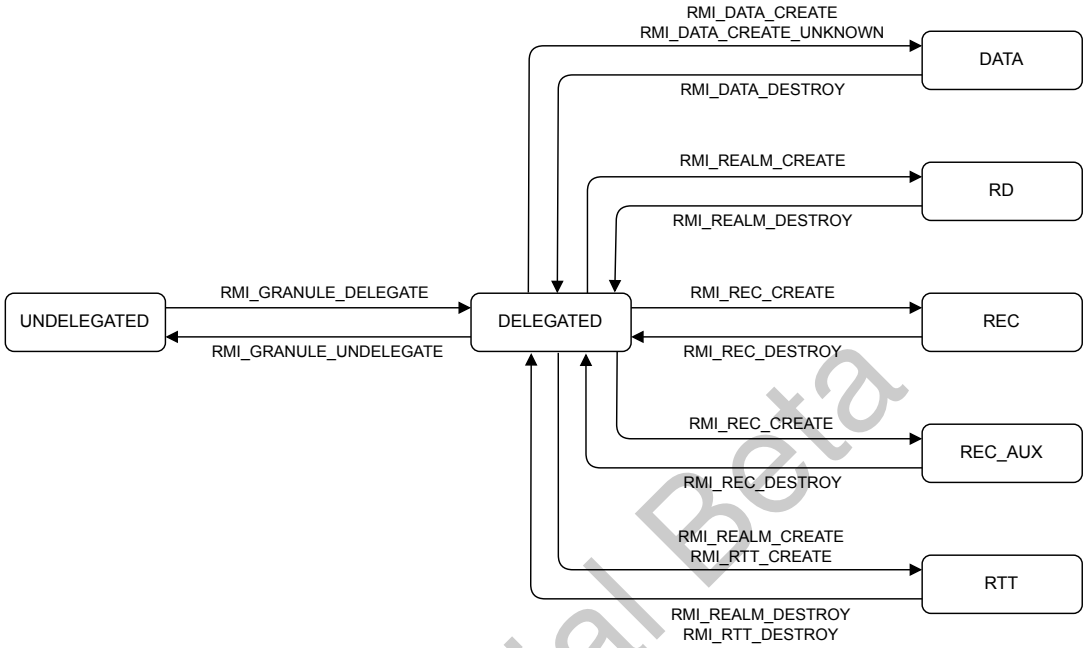


Figure A2.2: Granule state transitions

See also:

- B3.3.1 *RMI_DATA_CREATE command*
- B3.3.2 *RMI_DATA_CREATE_UNKNOWN command*
- B3.3.3 *RMI_DATA_DESTROY command*
- B3.3.5 *RMI_GRANULE_DELEGATE command*
- B3.3.6 *RMI_GRANULE_UNDELEGATE command*
- B3.3.9 *RMI_REALM_CREATE command*
- B3.3.10 *RMI_REALM_DESTROY command*
- B3.3.12 *RMI_REC_CREATE command*
- B3.3.13 *RMI_REC_DESTROY command*
- B3.3.15 *RMI_RTT_CREATE command*
- B3.3.16 *RMI_RTT_DESTROY command*

A2.2.4 Granule wiping

R_{TMGSL} When the state of a Granule has transitioned from *P* to DELEGATED and then to any other state, any content associated with *P* has been *wiped*.

X_{CTGQZ} Any sequence of Granule state transitions which passes through the DELEGATED state causes the Granule contents to be wiped. This is necessary to ensure that information does not leak from one Realm to another, or from a Realm to the Host. Note that no agent can observe the contents of a Granule while its state is DELEGATED.

D_{WTWJR} *Wiping* is an operation which changes the observable value of a memory location from *X* to *Y*, such that the value *X* cannot be determined from the value *Y*.

R_{BSXXV} Wiping of a memory location does not reveal, directly or indirectly, any confidential Realm data.

I_{MRPCQ} Wiping is not guaranteed to be implemented as zero filling.

S_{VJWYH} Realm software should not assume that the initial contents of uninitialized memory (that is, Realm IPA space which is backed by DATA Granules created using RMI_DATA_CREATE_UNKNOWN) are zero.

See also:

- *Arm CCA Security model* [4]
- B3.3.2 *RMI_DATA_CREATE_UNKNOWN command*
- B3.3.6 *RMI_GRANULE_UNDELEGATE command*

A2.3 Realm Execution Context

This section describes the concept of a Realm Execution Context (REC).

A2.3.1 Overview

D_{LRFCP} A *Realm Execution Context* (REC) is an RMM data structure which stores the saved context of a Realm VPE.

See also:

- A2.1.2 *Realm execution environment*
- Chapter A4 *Realm exception model*

A2.3.2 REC attributes

This section describes the attributes of a REC.

D_{ZLGLT} A *REC attribute* is a property of a REC whose value can be observed or modified either by the Host or by the Realm which owns the REC.

I_{CSGGT} Examples of ways in which a REC attribute may be observable include the outcome of an RMM command, and the PE state following Realm entry.

D_{LQSFT} The attributes of a REC are summarized in the following table.

Name	Type	Description
attest_state	RmmRecAttestState	Attestation token generation state
attest_addr	Address	Address of under-construction attestation token
attest_challenge	Bits512	Challenge for under-construction attestation token
aux	Address[16]	Addresses of auxiliary Granules
emulatable_abort	RmmRecEmulatableAbort	Whether the most recent exit from this REC was due to an Emulatable Data Abort
flags	RmmRecFlags	Flags which control REC behavior
gprs	Bits64[32]	General-purpose register values
mpidr	Bits64	MPIDR value
owner	Address	PA of RD of Realm which owns this REC
pc	Address	Program counter value
psci_pending	RmmPsciPending	Whether a PSCI request is pending
state	RmmRecState	Lifecycle state
sysregs	RmmSystemRegisters	EL1 and EL0 system register values
ripas_addr	Address	Next address to be processed in RIPAS change
ripas_top	Address	Top address of pending RIPAS change
ripas_value	RmmRipas	RIPAS value of pending RIPAS change
host_call_pending	RmmHostCallPending	Whether a Host call is pending

I_{PVMTY} The *aux* attribute of a REC is a list of *auxiliary Granules*.

I_{RWFZF} The number of auxiliary Granules required for a REC is returned by the RMI_REC_AUX_COUNT command.

X_{LRWHB} Depending on the configuration of the CCA platform and of the Realm, the amount of storage space required for a REC may exceed a single Granule.

I_{TGLBK} The number of auxiliary Granules required for a REC can vary between Realms on a CCA platform.

R_{MMBNR} The number of auxiliary Granules required for a REC is a constant for the lifetime of a given Realm.

I_{BGVRT} The *gprs* attribute of a REC is the set of general-purpose register values which are saved by the RMM on exit from the REC and restored by the RMM on entry to the REC.

I_{FPJDL} The *mpidr* attribute of a REC is a value which can be used to identify the VPE associated with the REC.

I_{BLVKZ} The *pc* attribute of a REC is the program counter which is saved by the RMM on exit from the REC and restored by the RMM on entry to the REC.

I_{GHFNQ} The *runnable* flag of a REC determines whether the REC is eligible for execution. The RMI_REC_ENTER command results in a REC entry only if the value of the flag is RUNNABLE.

I_{SCCMH} The runnable flag of a REC is controlled by the Realm. Its initial value is reflected in the Realm Initial Measurement, and during Realm execution its value can be changed by execution of the PSCI_CPU_ON and PSCI_CPU_OFF commands.

I_{PMYBG} The *state* attribute of a REC is controlled by the Host, by execution of the RMI_REC_ENTER command.

D_{CDXDZ} The *sysregs* attribute of a REC is the set of system register values which are saved by the RMM on exit from the REC and restored by the RMM on entry to the REC.

See also:

- A2.3.3 *REC index and MPIDR value*
- A2.3.4 *REC lifecycle*
- A4.3.4.3 *REC exit due to Data Abort*
- B3.3.14 *RMI_REC_ENTER command*
- B5.3.2 *PSCI_CPU_OFF command*
- B5.3.3 *PSCI_CPU_ON command*
- C1.13 *RmmRec type*

A2.3.3 REC index and MPIDR value

D_{KQVHN} The *REC index* is the unsigned integer value generated by by concatenation of MPIDR fields:

```
index = Aff3:Aff2:Aff1:Aff0[3:0]
```

This is illustrated by the following table.

REC index	Aff3	Aff2	Aff1	Aff0[3:0]
0	0	0	0	0
1	0	0	0	1
…	…	…	…	…
16	0	0	1	0
…	…	…	…	…
4096	0	1	0	0
…	…	…	…	…
1048576	1	0	0	0

REC index	Aff3	Aff2	Aff1	Aff0[3:0]
...	...	...	...	...

I_{PVLZY} The `Aff0[7:4]` field of a REC MPIDR value is RES0 for compatibility with GICv3.

I_{TTWVM} When creating the *n*th REC in a Realm, the Host is required to use the MPIDR corresponding to REC index *n*.

See also:

- B2.30 *RecIndex function*
- B3.3.12 *RMI_REC_CREATE command*
- B3.4.16 *RmiRecMpidr type*

A2.3.4 REC lifecycle

A2.3.4.1 States

D_{HTXQY} The states of a REC are listed below.

State	Description
READY	REC is not currently running.
RUNNING	REC is currently running.

A2.3.4.2 State transitions

I_{PHMWT} Permitted REC state transitions are shown in the following table. The rightmost column lists the events which can cause the corresponding state transition.

A transition from *NULL* represents creation of a REC object. A transition to *NULL* represents destruction of a REC object.

From state	To state	Events
NULL	READY	RMI_REC_CREATE
READY	*NULL*	RMI_REC_DESTROY
READY	RUNNING	RMI_REC_ENTER
RUNNING	READY	Return from RMI_REC_ENTER

I_{FNSTJ} Permitted REC state transitions are shown in the following figure. Each arc is labeled with the events which can cause the corresponding state transition.

A transition from *NULL* represents creation of a REC. A transition to *NULL* represents destruction of a REC.

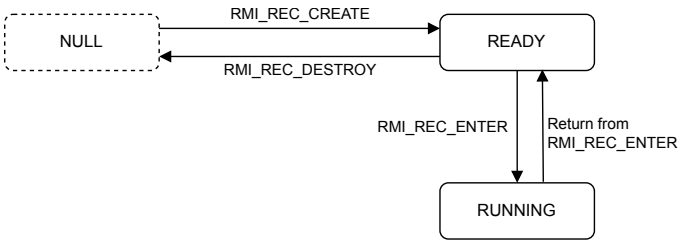


Figure A2.3: REC state transitions

See also:

- B3.3.12 *RMI_REC_CREATE command*
- B3.3.13 *RMI_REC_DESTROY command*
- B3.3.14 *RMI_REC_ENTER command*

Chapter A3
Realm creation

This section describes the process of creating a Realm.

See also:

- A2.1 *Realm*
- D1.2 *Realm lifecycle flows*

A3.1 Realm feature discovery and selection

I_{GJSMC} RMM implementations across different CCA platforms may support disparate features and may offer disparate configuration options for Realms.

I_{YRSDX} The features supported by an RMM implementation are discovered by reading feature pseudo-register values using the RMI_FEATURES command.

X_{WPHWG} The term *pseudo-register* is used because, although these values are stored in memory, their usage model is similar to feature registers specified in the Arm A-profile architecture.

I_{QNJTQ} On Realm creation, the Host specifies a set of desired features by providing feature pseudo-register values in a Realm parameters structure to the RMI_REALM_CREATE command. The RMM checks that the features specified by the Host are supported by the implementation.

I_{RRHJJ} The features specified at Realm creation time are included in the Realm Initial Measurement.

See also:

- A2.1.6 *Realm parameters*
- A7.1.1 *Realm Initial Measurement*
- B3.3.4 *RMI_FEATURES command*
- B3.3.9 *RMI_REALM_CREATE command*

A3.1.1 Realm hash algorithm

I_{WMKGX} The set of hash algorithms supported by the implementation is reported by the RMI_FEATURES command in RmiFeatureRegister0.

R_{KPBQM} Requesting an unsupported hash algorithm causes execution of RMI_REALM_CREATE to fail.

See also:

- A7.1 *Realm measurements*
- B3.3.9 *RMI_REALM_CREATE command*
- B3.4.6 *RmiFeatureRegister0 type*

A3.1.2 Realm LPA2 and IPA width

I_{NKLXQ} Usage of LPA2 for Realm Translation Tables is an attribute which is set by the Host during Realm creation, using RmiFeatureRegister0.LPA2.

I_{LKJGN} Realm IPA width is an attribute which is set by the Host during Realm creation, using RmiFeatureRegister0.S2SZ.

R_{SZVDK} Requesting a larger-than-supported IPA width causes execution of RMI_REALM_CREATE to fail.

I_{GKCCS} The Host can choose a smaller IPA width than the maximum supported IPA width reported by RMI_FEATURES. This is true regardless of whether LPA2 is enabled for the Realm.

X_{FTVXQ} The Host may want to enable LPA2 for a Realm due to either or both of the following reasons:

- to allow the Realm to be configured with a larger IPA width
- to allow access from mappings in the Realm's stage 2 translation to a larger PA space

I_{XDBQB} A Realm can query its IPA width using the RSI_REALM_CONFIG command.

See also:

- A5.2.1 *Realm IPA space*
- B3.3.9 *RMI_REALM_CREATE command*
- B3.4.6 *RmiFeatureRegister0 type*
- B4.3.8 *RSI_REALM_CONFIG command*

A3.1.3 Realm support for Scalable Vector Extension

I_{ZJSMJ} Availability of the Scalable Vector Extension (FEAT_SVE) to a Realm is determined by RmiFeatureRegister0.SVE_EN, which is set by the Host during Realm creation.

I_{VNLNH} SVE vector length for a Realm is determined by RmiFeatureRegister0.SVE_VL, which is set by the Host during Realm creation.

R_{FZZDS} Requesting a larger-than-supported SVE vector length causes execution of RMI_REALM_CREATE to fail. This is different from the behaviour of the hardware architecture, in which a larger-than-supported SVE vector length value is silently truncated.

X_{YGWTK} The RMI ABI provides a natural mechanism to signal an invalid feature selection, via the return code of RMI_REALM_CREATE. The analog in the hardware architecture would be to generate an illegal exception return, which would cause undesirable coupling between two disparate parts of the architecture, namely the exception model and the SVE feature.

R_{NBYKC} If SVE is supported by the platform but is disabled for the Realm via the RMI_REALM_CREATE command then a read of `ID_AA64PFR0_EL1.SVE` indicates that SVE is not supported.

U_{ZRJXL} The RMM should trap and emulate reads of `ID_AA64PFR0_EL1.SVE`.

S_{VXRNN} A Realm should discover SVE support by reading `ID_AA64PFR0_EL1.SVE` rather than based on the platform identity read from `MIDR_EL1`.

See also:

- B3.3.9 *RMI_REALM_CREATE command*
- B3.4.6 *RmiFeatureRegister0 type*

A3.1.4 Realm support for self-hosted debug

I_{SSTJD} Self-hosted debug is always available in Armv8-A.

I_{LVMFG} The number of breakpoints and watchpoints are attributes which are set by the Host during Realm creation, using RmiFeatureRegister0.NUM_BP and RmiFeatureRegister0.NUM_WP respectively.

R_{CJQTB} Requesting a number of breakpoints which is different from the number of breakpoints available causes execution of RMI_REALM_CREATE to fail.

R_{PLMDH} Requesting a number of watchpoints which is different from the number of watchpoints available causes execution of RMI_REALM_CREATE to fail.

See also:

- B3.3.9 *RMI_REALM_CREATE command*
- B3.4.6 *RmiFeatureRegister0 type*

A3.1.5 Realm support for Performance Monitors Extension

I_{NHCFC} Availability of the Performance Monitors Extension (FEAT_PMU) to a Realm is determined by RmiFeatureRegister0.PMU_EN, which is set by the Host during Realm creation.

I_{XZMKC} The number of PMU counters available to a Realm is determined by RmiFeatureRegister0.PMU_NUM_CTRS, which is set by the Host during Realm creation.

R_{XVRGD} Requesting a number of PMU counters which is different from the number of PMU counters available causes RMI_REALM_CREATE to fail.

See also:

- A8.1 *Realm PMU*
- B3.3.9 *RMI_REALM_CREATE command*
- B3.4.6 *RmiFeatureRegister0 type*

A3.1.6 Realm support for Activity Monitors Extension

R_{JJVZS} The Activity Monitors Extension (FEAT_AMUv1) is not available to a Realm.

A3.1.7 Realm support for Statistical Profiling Extension

R_{DCBNL} The Statistical Profiling Extension (FEAT_SPE) is not available to a Realm.

A3.1.8 Realm support for Trace Buffer Extension

R_{NXDXG} The Trace Buffer Extension (FEAT_TRBE) is not available to a Realm.

Chapter A4
Realm exception model

This section describes how Realms are executed, and how exceptions which cause exit from a Realm are handled.

See also:

- A2.1.2 *Realm execution environment*

A4.1 Exception model overview

D_{HCGWL} A *Realm entry* is a transfer of control to a Realm.

D_{RMGWJ} A *Realm exit* is a transition of control from a Realm.

I_{SMPWB} When executing in a Realm, an exception taken to R-EL2 or EL3 results in a Realm exit.

D_{XSNZP} A *REC entry* is a Realm entry due to execution of RMI_REC_ENTER.

I_{FQZJG} The Host provides the address of a REC as an input to the RMI_REC_ENTER command.

I_{MDQWG} In this chapter, both `rec` and "the target REC" refer to the REC which is provided to the RMI_REC_ENTER command.

D_{BLJGY} A *RecRun object* is a data structure used to pass values between the RMM and the Host on REC entry and on REC exit.

I_{VCCFV} A RecRun object is stored in Non-secure memory.

I_{WBHYZ} The Host provides the address of a RecRun object as an input to the RMI_REC_ENTER command.

I_{HMSQM} An implementation is permitted to return RMI_SUCCESS from RMI_REC_ENTER without performing a REC entry. For example, on observing a pending interrupt, the implementation can generate a REC exit due to IRQ without entering the target REC.

D_{TJCGH} A *REC exit* is return from an execution of RMI_REC_ENTER which caused a REC entry.

I_{HPWVY} The following diagram summarises the possible control flows that result from a Realm exit.

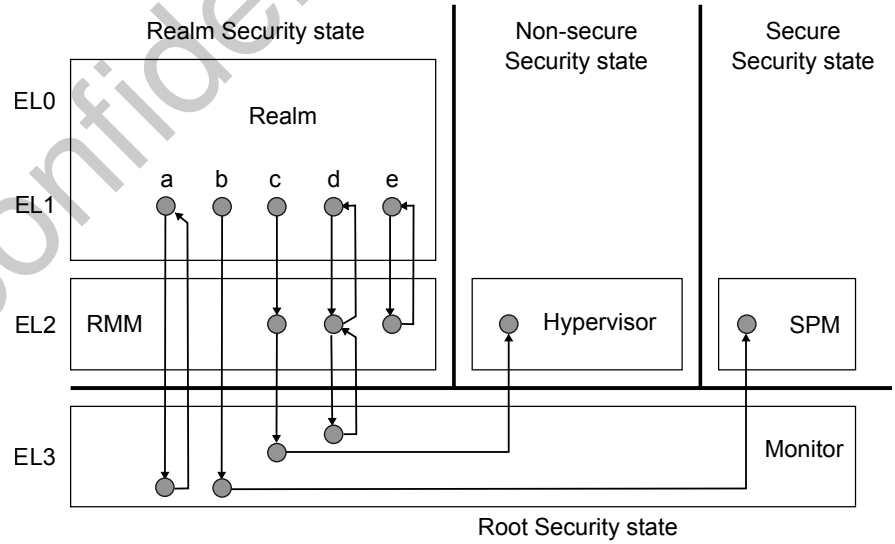


Figure A4.1: Realm exit paths

a. The exception is taken to EL3. The Monitor handles the exception and returns control to the Realm.

b. The exception is taken to EL3. The Monitor pre-empts Realm Security state and passes control to the Secure Security state. This may be for example due to an FIQ.

c. The exception is taken to EL2. The RMM decides to perform a REC exit. The RMM executes an SMC instruction, requesting the Monitor to pass control to the Non-secure Security state.

d. The exception is taken to EL2. The RMM executes an SMC instruction, requesting the Monitor to perform an operation, then returns control to the Realm.

e. The exception is taken to EL2. The RMM handles the exception and returns control to the Realm.

See also:

- A4.2 *REC entry*
- A4.3 *REC exit*
- B3.3.14 *RMI_REC_ENTER command*
- B3.4.18 *RmiRecRun type*

A4.2 REC entry

This section describes REC entry.

See also:

- A4.3 *REC exit*
- B3.3.14 *RMI_REC_ENTER command*

A4.2.1 RecEntry object

D_{SVSJM} A *RecEntry object* is a data structure used to pass values from the Host to the RMM on REC entry.

I_{YSKDN} A RecEntry object is stored in the RecRun object which is passed by the Host as an input to the RMI_REC_ENTER command.

I_{TRKKX} On REC entry, execution state is restored from the REC and from the RecEntry object to the PE.

I_{GHDLM} A RecEntry object contains attributes which are used to manage Realm virtual interrupts.

D_{CLNLW} The attributes of a RecEntry object are summarized in the following table.

Name	Byte offset	Type	Description
flags	0x0	RmiRecEntryFlags	Flags
gprs[0]	0x200	Bits64	Registers
gprs[1]	0x208	Bits64	Registers
gprs[2]	0x210	Bits64	Registers
gprs[3]	0x218	Bits64	Registers
gprs[4]	0x220	Bits64	Registers
gprs[5]	0x228	Bits64	Registers
gprs[6]	0x230	Bits64	Registers
gprs[7]	0x238	Bits64	Registers
gprs[8]	0x240	Bits64	Registers
gprs[9]	0x248	Bits64	Registers
gprs[10]	0x250	Bits64	Registers
gprs[11]	0x258	Bits64	Registers
gprs[12]	0x260	Bits64	Registers
gprs[13]	0x268	Bits64	Registers
gprs[14]	0x270	Bits64	Registers
gprs[15]	0x278	Bits64	Registers
gprs[16]	0x280	Bits64	Registers
gprs[17]	0x288	Bits64	Registers
gprs[18]	0x290	Bits64	Registers
gprs[19]	0x298	Bits64	Registers
gprs[20]	0x2a0	Bits64	Registers

Name	Byte offset	Type	Description
gprs[21]	0x2a8	Bits64	Registers
gprs[22]	0x2b0	Bits64	Registers
gprs[23]	0x2b8	Bits64	Registers
gprs[24]	0x2c0	Bits64	Registers
gprs[25]	0x2c8	Bits64	Registers
gprs[26]	0x2d0	Bits64	Registers
gprs[27]	0x2d8	Bits64	Registers
gprs[28]	0x2e0	Bits64	Registers
gprs[29]	0x2e8	Bits64	Registers
gprs[30]	0x2f0	Bits64	Registers
gicv3_hcr	0x300	Bits64	GICv3 Hypervisor Control Register value
gicv3_lrs[0]	0x308	Bits64	GICv3 List Register values
gicv3_lrs[1]	0x310	Bits64	GICv3 List Register values
gicv3_lrs[2]	0x318	Bits64	GICv3 List Register values
gicv3_lrs[3]	0x320	Bits64	GICv3 List Register values
gicv3_lrs[4]	0x328	Bits64	GICv3 List Register values
gicv3_lrs[5]	0x330	Bits64	GICv3 List Register values
gicv3_lrs[6]	0x338	Bits64	GICv3 List Register values
gicv3_lrs[7]	0x340	Bits64	GICv3 List Register values
gicv3_lrs[8]	0x348	Bits64	GICv3 List Register values
gicv3_lrs[9]	0x350	Bits64	GICv3 List Register values
gicv3_lrs[10]	0x358	Bits64	GICv3 List Register values
gicv3_lrs[11]	0x360	Bits64	GICv3 List Register values
gicv3_lrs[12]	0x368	Bits64	GICv3 List Register values
gicv3_lrs[13]	0x370	Bits64	GICv3 List Register values
gicv3_lrs[14]	0x378	Bits64	GICv3 List Register values
gicv3_lrs[15]	0x380	Bits64	GICv3 List Register values

I_{ZWRQP} In this chapter, both `entry` and "the RecEntry object" refer to the RecEntry object which is provided to the RMI_REC_ENTER command.

I_{LFYDV} On REC exit, all `entry` fields are ignored unless specified otherwise.

See also:

- A2.3 *Realm Execution Context*
- A4.3.1 *RecExit object*
- Chapter A6 *Realm interrupts and timers*
- B3.4.12 *RmiRecEntry type*

A4.2.2 General purpose registers restored on REC entry

R_{NMSFT} On REC entry, if the most recent exit from the target REC was a REC exit due to PSCI, then all of the following occur:

- X0 to X6 contain the PSCI return code and PSCI output values.
- GPR values X7 to X30 are restored from the REC to the PE.

R_{RZRRM} On REC entry, if the most recent exit from the target REC was not a REC exit due to PSCI, then GPR values X0 to X30 are restored from the REC to the PE.

R_{YSNYQ} On REC entry, if `rec.host_call_pending` is HOST_CALL_PENDING, then GPR values X0 to X6 are copied from `entry.gprs[0..6]` to the RsiHostCall data structure.

R_{YWHKC} On REC entry, if writing to the RsiHostCall data structure fails due to the target IPA not being mapped then a REC exit to Data Abort results.

R_{TZVNK} On REC entry, if writing to the RsiHostCall data structure succeeds then `rec.host_call_pending` is NO_HOST_CALL_PENDING.

R_{NLVXB} On REC entry, if RMM access to `entry` causes a GPF then the RMI_REC_ENTER command fails with RMI_ERROR_INPUT.

See also:

- A4.3.3 *General purpose registers saved on REC exit*
- A4.3.4.3 *REC exit due to Data Abort*
- A4.3.7 *REC exit due to PSCI*
- A4.3.9 *REC exit due to Host call*
- A4.5 *Host call*

A4.2.3 REC entry following REC exit due to Data Abort

R_{BWZKH} On REC entry, if the most recent exit from the target REC was a REC exit due to Emulatable Data Abort and `entry.flags.emul_mmio == RMI_EMULATED_MMIO`, then the return address is the next instruction following the faulting instruction.

R_{SCJWG} On REC entry, if the most recent exit from the target REC was a REC exit due to Emulatable Data Abort and the Realm memory access was a read and `entry.flags.emul_mmio == RMI_EMULATED_MMIO`, then the register indicated by `ESR_EL2.ISS.SRT` is set to `entry.gprs[0]`.

R_{LJWRK} On REC entry, if the most recent exit from the target REC was a REC exit due to Data Abort at an Unprotected IPA and `entry.flags.emul_mmio == RMI_NOT_EMULATED_MMIO` and `entry.flags.inject_sea == RMI_INJECT_SEA`, then a Synchronous External Abort is taken to the Realm..

See also:

- A4.3.4.3 *REC exit due to Data Abort*
- A4.4 *Emulated Data Aborts*
- A5.2.4 *Realm access to an Unprotected IPA*
- A5.2.5 *Synchronous External Aborts*

A4.3 REC exit

This section describes REC exit.

See also:

- A4.2 *REC entry*
- B3.3.14 *RMI_REC_ENTER command*

A4.3.1 RecExit object

D_{PBDCB} A *RecExit object* is a data structure used to pass values from the RMM to the Host on REC exit.

I_{VHJTL} A RecExit object is stored in the RecRun object which is passed by the Host as an input to the RMI_REC_ENTER command.

I_{JKWPB} On REC exit, execution state is saved from the PE to the REC and to the RecExit object.

I_{ZSCNM} A RecExit object contains attributes which are used to manage Realm virtual interrupts and Realm timers.

D_{FFCMN} The attributes of a RecExit object are summarized in the following table.

Name	Byte offset	Type	Description
exit_reason	0x0	RmiRecExitReason	Exit reason
esr	0x100	Bits64	Exception Syndrome Register
far	0x108	Bits64	Fault Address Register
hpfar	0x110	Bits64	Hypervisor IPA Fault Address register
gprs[0]	0x200	Bits64	Registers
gprs[1]	0x208	Bits64	Registers
gprs[2]	0x210	Bits64	Registers
gprs[3]	0x218	Bits64	Registers
gprs[4]	0x220	Bits64	Registers
gprs[5]	0x228	Bits64	Registers
gprs[6]	0x230	Bits64	Registers
gprs[7]	0x238	Bits64	Registers
gprs[8]	0x240	Bits64	Registers
gprs[9]	0x248	Bits64	Registers
gprs[10]	0x250	Bits64	Registers
gprs[11]	0x258	Bits64	Registers
gprs[12]	0x260	Bits64	Registers
gprs[13]	0x268	Bits64	Registers
gprs[14]	0x270	Bits64	Registers
gprs[15]	0x278	Bits64	Registers
gprs[16]	0x280	Bits64	Registers
gprs[17]	0x288	Bits64	Registers

Name	Byte offset	Type	Description
gprs[18]	0x290	Bits64	Registers
gprs[19]	0x298	Bits64	Registers
gprs[20]	0x2a0	Bits64	Registers
gprs[21]	0x2a8	Bits64	Registers
gprs[22]	0x2b0	Bits64	Registers
gprs[23]	0x2b8	Bits64	Registers
gprs[24]	0x2c0	Bits64	Registers
gprs[25]	0x2c8	Bits64	Registers
gprs[26]	0x2d0	Bits64	Registers
gprs[27]	0x2d8	Bits64	Registers
gprs[28]	0x2e0	Bits64	Registers
gprs[29]	0x2e8	Bits64	Registers
gprs[30]	0x2f0	Bits64	Registers
gicv3_hcr	0x300	Bits64	GICv3 Hypervisor Control Register value
gicv3_lrs[0]	0x308	Bits64	GICv3 List Register values
gicv3_lrs[1]	0x310	Bits64	GICv3 List Register values
gicv3_lrs[2]	0x318	Bits64	GICv3 List Register values
gicv3_lrs[3]	0x320	Bits64	GICv3 List Register values
gicv3_lrs[4]	0x328	Bits64	GICv3 List Register values
gicv3_lrs[5]	0x330	Bits64	GICv3 List Register values
gicv3_lrs[6]	0x338	Bits64	GICv3 List Register values
gicv3_lrs[7]	0x340	Bits64	GICv3 List Register values
gicv3_lrs[8]	0x348	Bits64	GICv3 List Register values
gicv3_lrs[9]	0x350	Bits64	GICv3 List Register values
gicv3_lrs[10]	0x358	Bits64	GICv3 List Register values
gicv3_lrs[11]	0x360	Bits64	GICv3 List Register values
gicv3_lrs[12]	0x368	Bits64	GICv3 List Register values
gicv3_lrs[13]	0x370	Bits64	GICv3 List Register values
gicv3_lrs[14]	0x378	Bits64	GICv3 List Register values
gicv3_lrs[15]	0x380	Bits64	GICv3 List Register values
gicv3_misr	0x388	Bits64	GICv3 Maintenance Interrupt State Register value
gicv3_vmcr	0x390	Bits64	GICv3 Virtual Machine Control Register value
cntp_ctl	0x400	Bits64	Counter-timer Physical Timer Control Register value

Name	Byte offset	Type	Description
cntp_cval	0x408	Bits64	Counter-timer Physical Timer CompareValue Register value
cntv_ctl	0x410	Bits64	Counter-timer Virtual Timer Control Register value
cntv_cval	0x418	Bits64	Counter-timer Virtual Timer CompareValue Register value
ripas_base	0x500	Bits64	Base address of pending RIPAS change
ripas_size	0x508	UInt64	Size of pending RIPAS change
ripas_value	0x510	RmiRipas	RIPAS value of pending RIPAS change
imm	0x600	Bits16	Host call immediate value
pmu_ovf	0x700	Bits64	PMU overflow
pmu_intr_en	0x708	Bits64	PMU interrupt enable
pmu_cntr_en	0x710	Bits64	PMU counter enable

I_{FQZXZ} In this chapter, both `exit` and "the RecExit object" refer to the RecExit object which is provided to the RMI_REC_ENTER command.

R_{PNWZV} On REC exit, all `exit` fields are zero unless specified otherwise.

See also:

- A2.3 *Realm Execution Context*
- A4.2.1 *RecEntry object*
- A4.5 *Host call*
- Chapter A6 *Realm interrupts and timers*
- Chapter A8 *Realm debug and performance monitoring*
- B3.4.14 *RmiRecExit type*

A4.3.2 Realm exit reason

I_{DYWHJ} On return from the RMI_REC_ENTER command, the reason for the REC exit is indicated by `exit.exit_reason` and `exit.esr`.

See also:

- B3.4.15 *RmiRecExitReason type*

A4.3.3 General purpose registers saved on REC exit

R_{PBKVB} On REC exit due to PSCI, all of the following are true:

- `exit.gprs[0]` contains the PSCI FID.
- `exit.gprs[1..3]` contain the corresponding PSCI arguments. If the PSCI command has fewer than 3 arguments, the remaining values contain zero.
- GPR values X7 to X30 are saved from the PE to the REC.

R_{FNZKM} On REC exit for any reason which is not REC exit due to PSCI, GPR values X0 to X30 are saved from the PE to the REC.

R_{MZGPT} On REC exit for any reason which is neither REC exit due to Host call nor REC exit due to PSCI, `exit.gprs` is zero.

R_{FRGVT} On REC exit, if RMM access to `exit` causes a GPF then the RMI_REC_ENTER command fails with RMI_ERROR_INPUT.

See also:

- A4.2.2 *General purpose registers restored on REC entry*
- A4.3.7 *REC exit due to PSCI*
- A4.3.9 *REC exit due to Host call*

A4.3.4 REC exit due to synchronous exception

I_{SNDHF} A synchronous exception taken to R-EL2 can cause a REC exit.

I_{RPSNC} The following table summarises the behavior of synchronous exceptions taken to R-EL2.

Exception class	Behavior
Trapped WFI or WFE instruction execution	REC exit due to WFI or WFE
HVC instruction execution in AArch64 state	Unknown exception taken to Realm
SMC instruction execution in AArch64 state	One of: • REC exit due to PSCI • RSI command handled by RMM, followed by return to Realm
Trapped MSR, MRS or System instruction execution in AArch64 state	Emulated by RMM, followed by return to Realm
Instruction Abort from a lower Exception level	REC exit due to Instruction Abort
Data Abort from a lower Exception level	REC exit due to Data Abort

R_{YLFMD} Realm execution of an SMC which is not part of one of the following ABIs results in a return value of SMCCC_NOT_SUPPORTED:

- PSCI
- RSI

See also:

- A4.5 *Host call*
- Chapter B4 *Realm Services Interface*
- Chapter B5 *Power State Control Interface*

A4.3.4.1 REC exit due to WFI or WFE

D_{GLHPX} A *REC exit due to WFI or WFE* is a REC exit due to WFI, WFIT, WFE or WFET instruction execution in a Realm.

R_{VTJQF} On WFI or WFIT instruction execution in a Realm, a REC exit due to WFI or WFE is caused if `entry.trap_wfi` is RMI_TRAP.

R_{GBNGW} On WFE or WFET instruction execution in a Realm, a REC exit due to WFI or WFE is caused if `entry.trap_wfe` is RMI_TRAP.

R_{YQWST} On REC exit due to WFI or WFE, all of the following are true:

- `exit.exit_reason` is RMI_EXIT_SYNC.
- `exit.esr.EC` contains the value of `ESR_EL2.EC` at the time of the Realm exit.
- `exit.esr.ISS.TI` contains the value of `ESR_EL2.ISS.TI` at the time of the Realm exit.
- All other `exit` fields are zero.

R_{BPYBC} On REC exit due to WFI or WFE, if the exit was caused by WFET or WFIT instruction execution then `exit.gprs[0]` contains the timeout value.

A4.3.4.2 REC exit due to Instruction Abort

D_{GYQXK} A *REC exit due to Instruction Abort* is a REC exit due to a Realm instruction fetch from a Protected IPA whose HIPAS is UNASSIGNED or DESTROYED and whose RIPAS is RAM.

R_{MGWRC} On REC exit due to Instruction Abort, all of the following are true:

- `exit.exit_reason` is RMI_EXIT_SYNC.
- `exit.esr.EC` contains the value of `ESR_EL2.EC` at the time of the Realm exit.
- `exit.esr.ISS.SET` contains the value of `ESR_EL2.ISS.SET` at the time of the Realm exit.
- `exit.esr.ISS.EA` contains the value of `ESR_EL2.ISS.EA` at the time of the Realm exit.
- `exit.esr.ISS.IFSC` contains the value of `ESR_EL2.ISS.IFSC` at the time of the Realm exit.
- `exit.hpfar` contains the value of `HPFAR_EL2` at the time of the Realm exit.
- All other `exit` fields are zero.

See also:

- A5.2.2 *Realm IPA state*
- A5.2.3 *Realm access to a Protected IPA*

A4.3.4.3 REC exit due to Data Abort

D_{CYRMT} A *REC exit due to Emulatable Data Abort* is a REC exit due to a Realm data access to an Unprotected IPA whose HIPAS is UNASSIGNED, where the access caused `ESR_EL2.ISS.ISV` to be set to `'1'`.

D_{MTZMC} A *REC exit due to Non-emulatable Data Abort* is a REC exit due to a Realm data access to one of the following:

- an Unprotected IPA whose HIPAS is UNASSIGNED, where the access caused `ESR_EL2.ISS.ISV` to be set to `'0'`
- an Unprotected IPA whose HIPAS is VALID_NS, where caused a stage 2 permission fault
- a Protected IPA whose HIPAS is UNASSIGNED or DESTROYED and whose RIPAS is RAM.

R_{RYVFL} On REC exit due to Data Abort, all of the following are true:

- `exit.exit_reason` is RMI_EXIT_SYNC.
- `exit.esr.EC` contains the value of `ESR_EL2.EC` at the time of the Realm exit.
- `exit.esr.ISS.SET` contains the value of `ESR_EL2.ISS.SET` at the time of the Realm exit.
- `exit.esr.ISS.EA` contains the value of `ESR_EL2.ISS.EA` at the time of the Realm exit.
- `exit.esr.ISS.DFSC` contains the value of `ESR_EL2.ISS.DFSC` at the time of the Realm exit.
- `exit.hpfar` contains the value of `HPFAR_EL2` at the time of the Realm exit.

On REC exit due to Emulatable Data Abort, all of the following are true:

- `rec.emulatable_abort` is EMULATABLE_ABORT.
- `exit.esr.ISS.ISV` contains the value of `ESR_EL2.ISS.ISV` at the time of the Realm exit.
- `exit.esr.ISS.SAS` contains the value of `ESR_EL2.ISS.SAS` at the time of the Realm exit.
- `exit.esr.ISS.FnV` contains the value of `ESR_EL2.ISS.FnV` at the time of the Realm exit.
- `exit.esr.ISS.WnR` contains the value of `ESR_EL2.ISS.WnR` at the time of the Realm exit.
- All other `exit.esr` fields are zero.
- `exit.far` contains the value of `FAR_EL2` at the time of the Realm exit, with bits more significant than the size of a Granule masked to zero.

On REC exit due to Non-emulatable Data Abort, all of the following are true:

- All other `exit.esr` fields are zero.
- `exit.far` is zero.

On REC exit due to Data Abort, all of the other `exit` fields are zero.

X_{XHXJC} On REC exit due to Emulatable Data Abort, `ESR_EL2.ISS.{SSE,SF}` are not propagated to the Host. This is because these fields are used to emulate sign extension on loads, which must be performed by the RMM so that the Realm can rely on architecturally correct behavior of the virtual execution environment.

X_{HSWFR} On REC exit due to Emulatable Data Abort, the Host can calculate the faulting IPA from the `exit.hpfar` and `exit.far` values.

R_{FFNHW} On REC exit due to Emulatable Data Abort, if the Realm memory access was a write, `exit.gprs[0]` contains the value of the register indicated by `ESR_EL2.ISS.SRT` at the time of the Realm exit.

R_{QBTPR} On REC exit not due to Emulatable Data Abort, `rec.emulatable_abort` is NOT_EMULATABLE_ABORT.

See also:

- A4.2.3 *REC entry following REC exit due to Data Abort*
- A4.4 *Emulated Data Aborts*
- A5.2.1 *Realm IPA space*
- A5.2.3 *Realm access to a Protected IPA*
- A5.2.4 *Realm access to an Unprotected IPA*

A4.3.5 REC exit due to IRQ

D_{YLWXK} A *REC exit due to IRQ* is a REC exit due to an IRQ exception which should be handled by the Host.

R_{TYJSX} On REC exit due to IRQ, `exit.exit_reason` is RMI_EXIT_IRQ.

R_{CSQXV} On REC exit due to IRQ, `exit.esr` is zero.

See also:

- Chapter A6 *Realm interrupts and timers*

A4.3.6 REC exit due to FIQ

D_{ZTYMM} A *REC exit due to FIQ* is a REC exit due to an FIQ exception which should be handled by the Host.

R_{PDSBD} On REC exit due to FIQ, `exit.exit_reason` is RMI_EXIT_FIQ.

R_{GXZRF} On REC exit due to FIQ, `exit.esr` is zero.

See also:

- Chapter A6 *Realm interrupts and timers*

A4.3.7 REC exit due to PSCI

I_{ZSGFP} A PSCI function executed by a Realm is either:

- handled by the RMM, returning to the Realm, or
- forwarded by the RMM to the Host via a *REC exit due to PSCI*.

D_{RFTQD} A *REC exit due to PSCI* is a REC exit due to Realm PSCI function execution by SMC instruction which was forwarded by the RMM to the Host.

I_{VBJXY} The following table summarises the behavior of PSCI function execution by a Realm.

PSCI functions not listed in this table are not supported. Calling a non-supported PSCI function results in a return value of PSCI_NOT_SUPPORTED.

PSCI function	Can result in REC exit due to PSCI	Requires Host to call RMI_PSCI_COMPLETE
PSCI_VERSION	No	-

PSCI function	Can result in REC exit due to PSCI	Requires Host to call RMI_PSCI_COMPLETE
PSCI_FEATURES	No	-
PSCI_CPU_SUSPEND	Yes	No
PSCI_CPU_OFF	Yes	No
PSCI_CPU_ON	Yes	Yes
PSCI_AFFINITY_INFO	Yes	Yes
PSCI_SYSTEM_OFF	Yes	No
PSCI_SYSTEM_RESET	Yes	No

R_{NTZNJ} On REC exit due to PSCI, `exit.exit_reason` is RMI_EXIT_PSCI.

R_{SXGJK} On REC exit due to PSCI, `exit.gprs` contains sanitised parameters from the PSCI call.

I_{KKFMQ} Following REC exit due to PSCI, if the command arguments include an MPIDR value, the Host must provide the corresponding REC by calling the RMI_PSCI_COMPLETE command. This is because the RMM does not maintain a mapping from MPIDR values to REC addresses. On execution of RMI_PSCI_COMPLETE, the RMM validates that REC provided by the Host matches the MPIDR value, and then completes the PSCI operation.

See also:

- A4.3.3 *General purpose registers saved on REC exit*
- B3.3.7 *RMI_PSCI_COMPLETE command*
- Chapter B5 *Power State Control Interface*
- D1.4 *PSCI flows*

A4.3.8 REC exit due to RIPAS change pending

D_{JGCVY} A *REC exit due to RIPAS change pending* is a REC exit due to the Realm issuing a *RIPAS change request*.

R_{QSSKK} On REC exit due to RIPAS change pending, all of the following are true:

- `exit.exit_reason` is RMI_EXIT_RIPAS_CHANGE.
- `exit.ripas_base` is the base address of the region on which a RIPAS change is pending.
- `exit.ripas_size` is the size of the region on which a RIPAS change is pending.
- `exit.ripas_value` is the requested RIPAS value.
- `rec.ripas_addr` is the base address of the region on which a RIPAS change is pending.
- `rec.ripas_top` is the top address of the region on which a RIPAS change is pending.
- `rec.ripas_value` is the requested RIPAS value.

I_{MCKKH} On REC exit due to RIPAS change pending:

- `exit` holds the base address and the size of the region on which a RIPAS change is pending. These values inform the Host of the bounds of the RIPAS change request.
- `rec` holds the next address to be processed in a RIPAS change, and the top of the requested RIPAS change region. These values are used by the RMM to enforce that the RMI_RTT_SET_RIPAS command can only apply RIPAS change within the bounds of the RIPAS change request, and to report the progress of the RIPAS change to the Realm on the next REC entry.

R_{QRMMN} On REC exit not due to RIPAS change pending, all of the following are true:

- `rec.ripas_addr` is 0
- `rec.ripas_top` is 0

See also:

- A2.3.2 *REC attributes*
- A5.4 *RIPAS change*

A4.3.9 REC exit due to Host call

D_{WFZXK} A *REC exit due to Host call* is a REC exit due to RSI_HOST_CALL execution in a Realm.

R_{GTJRP} On REC exit due to Host call, all of the following are true:

- `rec.host_call_pending` is HOST_CALL_PENDING.
- `exit.exit_reason` is RMI_EXIT_HOST_CALL.
- `exit.imm` contains the immediate value passed to the RSI_HOST_CALL command.
- `exit.gprs[0..6]` contain the register values passed to the RSI_HOST_CALL command.
- All other `exit` fields are zero.

See also:

- A4.5 *Host call*
- B4.3.3 *RSI_HOST_CALL command*

A4.3.10 REC exit due to SError

D_{PGMHP} A *REC exit due to SError* is a REC exit due to an SError interrupt during Realm execution.

R_{LRCFP} On REC exit due to SError, all of the following occur:

- `exit.exit_reason` is RMI_EXIT_SERROR.
- `exit.esr.EC` contains the value of `ESR_EL2.EC` at the time of the Realm exit.
- `exit.esr.ISS.IDS` contains the value of `ESR_EL2.ISS.IDS` at the time of the Realm exit.
- `exit.esr.ISS.AET` contains the value of `ESR_EL2.ISS.AET` at the time of the Realm exit.
- `exit.esr.ISS.EA` contains the value of `ESR_EL2.ISS.EA` at the time of the Realm exit.
- `exit.esr.ISS.DFSC` contains the value of `ESR_EL2.ISS.DFSC` at the time of the Realm exit.
- All other `exit` fields are zero.

A4.4 Emulated Data Aborts

I_{SVYDC} On REC exit due to Emulatable Data Abort, sufficient information is provided to the Host to enable it to emulate the access, for example to emulate a virtual peripheral.

On taking the REC exit, the Host can either

- Establish a mapping in the RTT, in which case it would want the Realm to re-attempt the access. In this case, on the next REC entry the Host sets `entry.flags.emul_mmio = RMI_NOT_EMULATED_MMIO`, which indicates that instruction emulation was not performed. This causes the return address to be the faulting instruction.
- Emulate the access. For an emulated write, the data is provided in `exit.gprs[0]`. For an emulated read, the data is provided in `entry.gprs[0]`. In this case, on the next REC entry the Host sets `entry.flags.emul_mmio = RMI_EMULATED_MMIO`, which indicates that the instruction was emulated. This causes the return address to be the address of the instruction which generated the Data Abort plus 4 bytes.

See also:

- A4.2.3 *REC entry following REC exit due to Data Abort*
- A4.3.4.3 *REC exit due to Data Abort*
- A5.2.1 *Realm IPA space*

A4.5 Host call

This section describes the programming model for Realm communication with the Host.

D_{YDJWT} A *Host call* is a call made by the Realm to the Host, by execution of the RSI_HOST_CALL command.

I_{XNFKZ} A Host call can be used by a Realm to make a hypercall.

R_{DNBQF} On Realm execution of HVC, an Unknown exception is taken to the Realm.

See also:

- A4.2.2 *General purpose registers restored on REC entry*
- A4.3.9 *REC exit due to Host call*
- B4.3.3 *RSI_HOST_CALL command*
- D1.3.2 *Host call flow*

Chapter A5
Realm memory management

This section describes how Realm memory is managed. This includes:

- How the translation tables which describe the Realm's address space are managed by the Host.
- Properties of the Realm's address space, and of the memory which can be mapped into it.
- How faults caused by Realm memory accesses are handled.

See also:

- A2.1.2 *Realm execution environment*
- D1.5 *Realm memory management flows*
- Chapter D2 *Realm shared memory protocol*

A5.1 Realm memory management overview

Realm memory management can be viewed from one of two standpoints: the Realm and the Host.

From the Realm's point of view, the RMM provides security guarantees regarding the IPA space of the Realm and the memory which is mapped into it. These security guarantees are upheld via RSI commands which the Realm can execute in order to query the initial configuration and contents of its address space, and to modify properties of the address space at runtime.

From the Host's point of view, Realm memory management involves manipulating the stage 2 translation tables which describe the Realm's address space, and handling faults which are caused by Realm memory accesses. These operations are similar to those involved in managing the memory of a normal VM, but in the case of a Realm they are performed via execution of RMI commands.

See also:

- A5.2 *Realm view of memory management*
- A5.3 *Host view of memory management*

A5.2 Realm view of memory management

This section describes memory management from the Realm's point of view.

A5.2.1 Realm IPA space

I_{DLRZF} The IPA space of a Realm is divided into two halves: Protected IPA space and Unprotected IPA space.

S_{LZHXC} Software in a Realm should treat the most significant bit of an IPA as a protection attribute.

D_{KXGDV} A *Protected IPA* is an address in the lower half of a Realm's IPA space. The most significant bit of a Protected IPA is `0`.

D_{MRWGM} An *Unprotected IPA* is an address in the upper half of a Realm's IPA space. The most significant bit of an Unprotected IPA is `1`.

See also:

- A2.1.3 *Realm attributes*
- A3.1.2 *Realm LPA2 and IPA width*

A5.2.2 Realm IPA state

D_{WWCBD} A Protected IPA has an associated *Realm IPA state* (RIPAS).

The RIPAS values are shown in the following table.

RIPAS	Description
EMPTY	Unused address
RAM	Private code or data owned by the Realm

I_{BSGSW} Changing the RIPAS of a Protected IPA for a Realm in the NEW state causes the Realm Initial Measurement to be updated.

I_{NZXPG} A Realm in the ACTIVE state can request changes to the RIPAS of a region of Protected IPA space.

See also:

- A5.4 *RIPAS change*

- A7.1.1 *Realm Initial Measurement*

A5.2.3 Realm access to a Protected IPA

R_{JVQQR} Realm data access to a Protected IPA whose RIPAS is EMPTY causes a Synchronous External Abort taken to the Realm.

I_{PGHBT} Realm data access to a Protected IPA can cause an REC exit due to Data Abort.

X_{HQLVY} The Host can, by executing RMI_RTT_DESTROY, transition the HIPAS of a range of Protected IPAs from UNASSIGNED to DESTROYED. Within this range, individual pages may have been configured with different RIPAS values. The architecture has to choose whether a Realm access to any IPA in this range causes a Synchronous External Data Abort taken to the Realm, or a REC exit due to Data Abort. The former would effectively allow the Host to inject an SEA at any Protected IPA which had been configured with RIPAS=RAM, and therefore potentially trigger unexpected behavior in the Realm. The latter does not have any negative impacts on Realm security, and is therefore the option which has been chosen by the architecture.

R_{MKLSD} Realm instruction fetch from a Protected IPA whose RIPAS is EMPTY causes a Synchronous External Abort taken to the Realm.

I_{XHKQY} Realm instruction fetch from a Protected IPA whose RIPAS is RAM can cause a REC exit due to Instruction Abort.

See also:

- A4.3.4.2 *REC exit due to Instruction Abort*
- A4.3.4.3 *REC exit due to Data Abort*
- A5.2.5 *Synchronous External Aborts*

A5.2.4 Realm access to an Unprotected IPA

I_{KQJML} An access by a Realm to an Unprotected IPA can result in a *Granule Protection Fault* (GPF).

The RMM does not ensure that the PAS of a Granule mapped at an Unprotected IPA is NS.

S_{ZZBQF} Realm software must be able to handle taking a GPF during access to an Unprotected IPA.

I_{WCVBZ} Realm data access to an Unprotected IPA can cause a REC exit due to Data Abort.

I_{RNDTJ} On taking a REC exit due to Data Abort at an Unprotected IPA, the Host can inject a Synchronous External Abort to the Realm.

X_{MGBDH} The Host can inject an SEA in response to an unexpected Realm data access to an Unprotected IPA.

I_{FVYCM} Realm data access to an Unprotected IPA which caused `ESR_EL2.ISS.ISV` to be set to `'1'` can be emulated by the Host.

R_{XLSKP} Realm instruction fetch from an Unprotected IPA causes a Synchronous External Abort taken to the Realm.

See also:

- A4.2.3 *REC entry following REC exit due to Data Abort*
- A4.3.4.3 *REC exit due to Data Abort*
- A4.4 *Emulated Data Aborts*
- A5.2.5 *Synchronous External Aborts*

A5.2.5 Synchronous External Aborts

R_{VKNJW} When a Synchronous External Abort is taken to a Realm, `ESR_EL1.EA == '1'`.

A5.2.6 Realm access outside IPA space

R_{GYVZQ} If stage 1 translation is enabled, Realm access to an IPA which is greater than the IPA space of the Realm causes a stage 1 Address Size Fault taken to the Realm, with the fault status code indicating the level at which the fault occurred.

R_{LSJJR} If stage 1 translation is disabled, Realm access to an IPA which is greater than the IPA space of the Realm causes a stage 1 level 0 Address Size Fault taken to the Realm.

A5.2.7 Summary of Realm IPA space properties

I_{TPGKW} The following table summarises the properties of Realm IPA space.

Realm IPA	Data access causes abort to Realm?	Data access causes REC exit due to Data Abort?	Data access can be emulated by Host?	Instruction fetch causes abort to Realm?	Instruction fetch causes REC exit due to Instruction Abort?
Protected, RIPAS=EMPTY	Always (SEA)	Permitted, under control of Host	No	Always (SEA)	Never
Protected, RIPAS=RAM	Never	Permitted, under control of Host	No	Never	Permitted, under control of Host
Unprotected	Permitted (SEA), under control of Host	Permitted, under control of Host	Yes	Always (SEA)	Never
Outside Realm IPA space	Always (Address Size Fault)	Never	No	Always (Address Size Fault)	Never

A5.3 Host view of memory management

This section describes memory management from the Host's point of view.

A5.3.1 Host IPA state

D_{YZTZJ} A Realm IPA has an associated *Host IPA state* (HIPAS).

The HIPAS values for a Protected IPA are shown in the following table.

HIPAS	Description
UNASSIGNED	Address is not associated with any Granule.
ASSIGNED	Address is associated with a DATA Granule.
DESTROYED	Address is not associated with any Granule. This address cannot be used for the rest of the lifetime of the Realm.

The HIPAS values for an Unprotected IPA are shown in the following table.

HIPAS	Description
UNASSIGNED	Address is not associated with any Granule.
VALID_NS	Host-owned memory is mapped at this address.

A5.3.2 Host control of RIPAS and HIPAS

I_{TRSKJ} HIPAS values are stored in a Realm Translation Table (RTT).

I_{GZMKQ} HIPAS transitions are caused by execution of RMI commands.

I_{VZCZV} RIPAS values are stored in an RTT.

I_{BSBHN} RIPAS transitions for a NEW Realm are caused by execution of RMI_RTT_INIT_RIPAS.

I_{FRJJH} RIPAS transitions for an ACTIVE Realm are caused by a *RIPAS change* process, which consists of RSI commands executed by the Realm, followed by RMI commands executed by the Host.

I_{NQCGS} A mapping at a Protected IPA is valid if the HIPAS is ASSIGNED and the RIPAS is RAM.

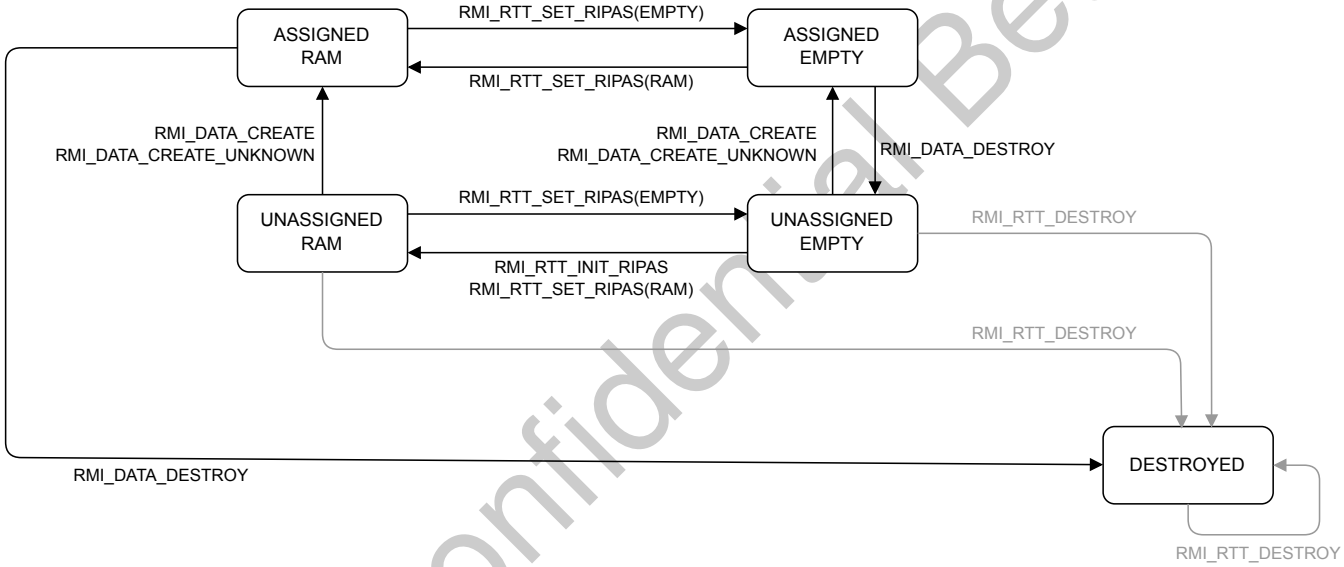
I_{YMNSR} The following table summarises, for each combination of RIPAS and HIPAS for a Protected IPA:

- the translation table entry attributes, and
- the behavior which results from Realm access to that IPA.

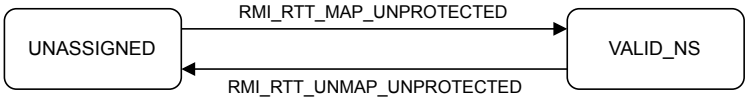
RIPAS	HIPAS	TTE.ADDR	TTE.NS	TTE.VALID	Data access	Instruction fetch
EMPTY	UNASSIGNED			0	SEA to Realm	SEA to Realm
EMPTY	ASSIGNED	DATA		0	SEA to Realm	SEA to Realm
EMPTY	DESTROYED			0	REC exit due to Data Abort	REC exit due to Instruction Abort

RIPAS	HIPAS	TTE.ADDR	TTE.NS	TTE.VALID	Data access	Instruction fetch
RAM	UNASSIGNED			0	REC exit due to Data Abort	REC exit due to Instruction Abort
RAM	ASSIGNED	DATA	0	1	Data access	Instruction fetch
RAM	DESTROYED			0	REC exit due to Data Abort	REC exit due to Instruction Abort

I_{FDCST} The following diagram summarises RIPAS and HIPAS transitions for a Protected IPA.



I_{YNYBY} The following diagram summarises HIPAS transitions for an Unprotected IPA.



See also:

- A5.4 *RIPAS change*
- A5.5 *Realm Translation Table*
- B3.3.1 *RMI_DATA_CREATE command*
- B3.3.2 *RMI_DATA_CREATE_UNKNOWN command*
- B3.3.3 *RMI_DATA_DESTROY command*
- B3.3.16 *RMI_RTT_DESTROY command*
- B3.3.18 *RMI_RTT_INIT_RIPAS command*
- B3.3.21 *RMI_RTT_SET_RIPAS command*
- B4.3.5 *RSI_IPA_STATE_SET command*

A5.4 RIPAS change

D_{BTSQY} A *RIPAS change* is a process via which the RIPAS of a region of Protected IPA space is changed, for a Realm whose state is ACTIVE.

I_{KXXBV} A RIPAS change consists of actions taken by first the Realm, and then the Host:

- The Realm issues a *RIPAS change request* by executing RSI_IPA_STATE_SET.
  - The input values to this command include the requested IPA range, and the requested RIPAS value.
  - The RMM records these values in the REC, and then performs a REC exit due to RIPAS change pending.
- In response, the Host executes zero or more RMI_RTT_SET_RIPAS commands.

The return value from RSI_IPA_STATE_SET indicates the top of the IPA range which has been modified by the command.

I_{RFVTG} The RIPAS change process, together with the Realm Initial Measurement ensures that a Realm can always reliably determine the RIPAS of any Protected IPA.

I_{LPZWK} A RIPAS change is applied by one or more calls to the RMI_RTT_SET_RIPAS command.

I_{MMHMZ} Successful execution of RMI_RTT_SET_RIPAS targets an RTTE at address `rec.ripas_addr`.

I_{JHJGZ} Successful execution of RMI_RTT_SET_RIPAS increments `rec.ripas_addr` by the size of the address space described by the target RTTE.

I_{HXKPB} On REC entry following a REC exit due to RIPAS change, GPR values are updated to indicate for how much of the target IPA range the RIPAS change has been applied.

S_{TZYZV} To complete a RIPAS change for a given target IPA range, a Realm should execute RSI_IPA_STATE_SET in a loop, until the value of X1 reaches the top of the target IPA range.

See also:

- A2.3.2 *REC attributes*
- A4.2 *REC entry*
- A4.3.8 *REC exit due to RIPAS change pending*
- A5.2.2 *Realm IPA state*
- A7.1.1 *Realm Initial Measurement*
- B3.3.14 *RMI_REC_ENTER command*
- B3.3.21 *RMI_RTT_SET_RIPAS command*
- B4.3.5 *RSI_IPA_STATE_SET command*
- D1.5.3 *RIPAS change flow*

A5.5 Realm Translation Table

This section introduces the stage 2 translation table used by a Realm.

A5.5.1 RTT overview

D_{FRNCX} A *Realm Translation Table* (RTT) is an abstraction over an Armv8-A stage 2 translation table used by a Realm.

I_{MBCVZ} The attributes and format of an Armv8-A stage 2 translation table are defined by the Armv8-A Virtual Memory System Architecture (VMSA) *Arm Architecture Reference Manual for Armv8-A architecture profile* [3].

R_{PXNHQ} The translation granule size of an RTT is 4KB.

I_{TQVTP} The RMM architecture can only be deployed on a hardware platform which implements a translation granule size of 4KB.

I_{PHGQQ} The contents of an RTT are not directly accessible to the Host.

I_{FPLRL} The contents of an RTT are manipulated using RMM commands. These commands allow the Host to manipulate the contents of the RTT used by a Realm, subject to constraints imposed by the RMM.

D_{QTZDW} An *RTT entry* (RTTE) is an abstraction over an Armv8-A stage 2 translation table descriptor.

I_{VYLTT} An RTTE contains an output address which can point to one of the following:

- Another RTT
- A DATA Granule which is owned by the Realm
- Non-secure memory which is accessible to both the Realm and the Host

A5.5.2 RTT structure and configuration

D_{VHLWF} An *RTT tree* is a hierarchical data structure composed of RTTs, connected via Table Descriptors.

I_{KNPNX} An RTT contains an array of RTTEs.

D_{HYTCJ} An *RTT level* is the depth of an RTT within an RTT tree.

I_{KKMSX} An RTT does not have an intrinsic “level” attribute. The level of an RTT is determined by its position within an RTT tree.

D_{QSYBS} The RTT level of the root of an RTT tree is called the *starting level*.

I_{SSDBT} The maximum depth of an RTT tree depends on all of the following:

- whether LPA2 is selected when the Realm is created
- the rtt_level_start attribute of the Realm
- the ipa_width attribute of the Realm.

See also:

- A2.1.3 *Realm attributes*
- A3.1.2 *Realm LPA2 and IPA width*

A5.5.3 RTT starting level

I_{FDWZF} The RTT starting level is set when a Realm is created.

I_{YCPMF} The number of starting level RTTs is architecturally defined as a function of the Realm IPA width and the RTT starting level. See *Arm Architecture Reference Manual for Armv8-A architecture profile* [3] for further details.

I_{RYNXB} The address of the first starting level RTT is stored in the RTT base attribute of the owning Realm.

I_{XXWQW} The RTT base attribute is set when a Realm is created.

See also:

- A2.1.3 *Realm attributes*

A5.5.4 RTT entry

I_{ZBGGZ} An RTT entry (RTTE) is an abstraction over an Armv8-A stage 2 translation table descriptor. The attributes and format of an Armv8-A stage 2 translation table descriptor are defined by the Armv8-A Virtual Memory System Architecture (VMSA) *Arm Architecture Reference Manual for Armv8-A architecture profile* [3].

D_{BNHQQ} An RTTE has a *state*.

The values of *RTTE state* are:

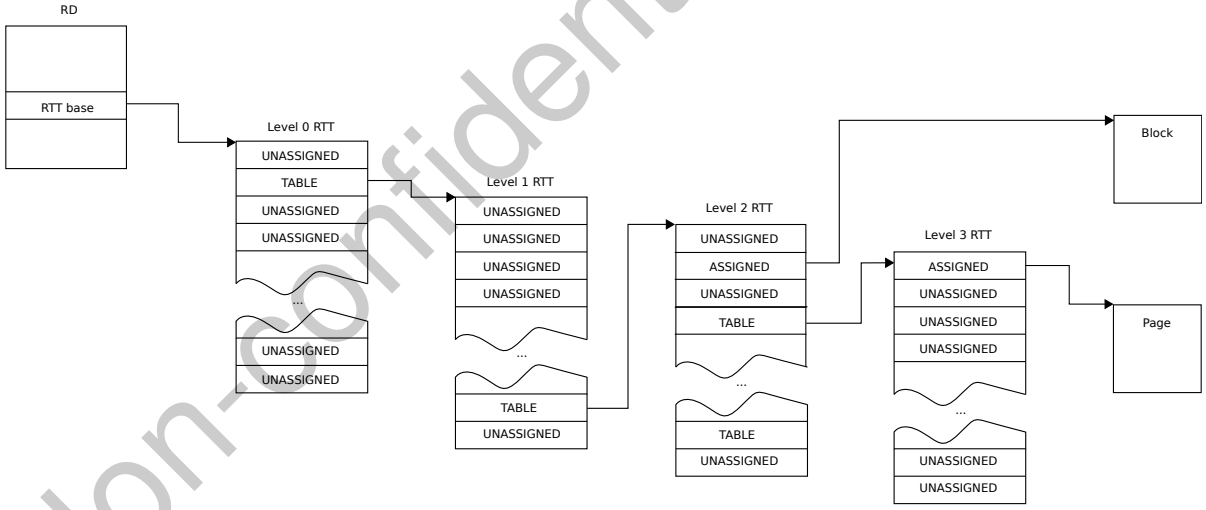
- TABLE: the output address of the RTTE points to another RTT
- A HIPAS value

I_{QWQSB} The state of an RTTE in a RTT which is not level 2 or level 3 is UNASSIGNED, DESTROYED or TABLE.

D_{NSHSL} The output address of an RTTE whose state is TABLE and which is in a level *n* RTT is the physical address of a level *n+1* RTT.

I_{DJZTM} An RTT whose level *n* is not the starting RTT level is pointed-to by exactly one TABLE RTTE in a level *n-1* RTT.

I_{DXQWZ} The following diagram shows an example RTT tree, annotated with RTTE states.



I_{FGWQS} The function `AddrIsRttLevelAligned()` is used to evaluate whether an address is aligned to the address range described by an RTTE at a specified RTT level.

See also:

- A5.3.1 *Host IPA state*
- B1.4 *Command condition expressions*

A5.5.5 RTT reading

I_{KJWKQ} Attributes of an RTTE, including the RTTE state, can be read by calling the RMI_RTT_READ_ENTRY command. The set of RTTE attributes which are returned depends on the state of the RTTE.

See also:

- B3.3.20 *RMI_RTT_READ_ENTRY command*

A5.5.6 RTT folding

D_{RMCLC} An RTT is *homogeneous* if its entries satisfy one of the conditions in the following table. If an RTT is homogeneous, the following table specifies the state to which the parent RTTE is set.

Conditions on child RTT contents	Parent RTTE state
All of the following are true: • State of all entries is UNASSIGNED • RIPAS of all entries is the same	UNASSIGNED
State of all entries is DESTROYED	DESTROYED
All of the following are true: • Level is 3 • State of all entries is ASSIGNED • Output address of first entry is aligned to size of level 2 entry • Output addresses of all entries are contiguous • RIPAS of all entries is the same	ASSIGNED
All of the following are true: • Level is 3 • State of all entries is VALID_NS • Output address of first entry is aligned to size of level 2 entry • Output addresses of all entries are contiguous • Attributes of all entries are identical	VALID_NS

I_{KDXLT} The function `RttIsHomogeneous()` is used to evaluate whether an RTT is homogeneous.

D_{QPXCP} *RTT folding* is the operation of destroying a homogeneous child RTT, and updating the state of the parent RTTE.

I_{QMGWK} On RTT folding, the state of the parent RTTE is determined from the contents of the child RTTEs.

I_{LLWGH} The function `RttFold()` is used to evaluate the parent RTTE state which results from an RTT folding operation.

I_{TPMGT} On RTT folding, if the state of the parent RTTE is VALID_NS then the attributes of the parent RTTE are copied from the child RTTEs.

See also:

- B2.55 *RttFold function*
- B2.56 *RttIsHomogeneous function*
- B3.3.17 *RMI_RTT_FOLD command*

A5.5.7 RTT unfolding

D_{HQQMG} *RTT unfolding* is the operation of creating a child RTT, and populating it based on the contents of the parent RTTE.

I_{KWZXN} On RTT unfolding, the state of all RTTEs in the child RTT are set to the state of the parent RTTE.

I_{HMYSW} On RTT unfolding, if the state of the parent RTTE is ASSIGNED or VALID_NS, then the output addresses of RTTEs in the child RTT are set to a contiguous range which starts from the address of the parent RTTE.

See also:

- B3.3.15 *RMI_RTT_CREATE command*

A5.5.8 RTT liveness

D_{MPWLR} *RTT liveness* is a property which means that there exists another RMM data structure which is referenced by the RTT.

D_{YPSLW} An RTT is *live* if the state of any entry in the RTT is either ASSIGNED or TABLE.

I_{YPLKM} The function `RttIsLive()` is used to evaluate whether an RTT is live.

See also:

- A5.5.9 *RTT destruction*
- B2.57 *RttIsLive function*

A5.5.9 RTT destruction

D_{VXRZW} *RTT destruction* is the operation of destroying a child RTT, and updating the state of the parent RTTE to DESTROYED.

I_{PRMFR} An RTT cannot be destroyed if it is live.

See also:

- A5.5.8 *RTT liveness*
- B3.3.16 *RMI_RTT_DESTROY command*

A5.5.10 RTT walk

I_{CBWSX} An IPA is translated to a PA by walking an RTT tree, starting at the RTT base.

I_{FDWYV} The behaviour of an RTT walk is defined by the Armv8-A Virtual Memory System Architecture (VMSA) *Arm Architecture Reference Manual for Armv8-A architecture profile* [3].

I_{TVGQD} The inputs to an RTT walk are:

- a Realm Descriptor, which contains the address of the initial RTT
- a target IPA
- a target RTT level.

The RTT walk terminates when either:

- it reaches the target RTT level, or
- it reaches an RTTE whose state is not TABLE.

D_{RBHVQ} The result of an RTT walk performed by the RMM is a data structure of type RmmRttWalkResult.

The attributes of an `RmmRttWalkResult` are summarized in the following table.

Name	Type	Description
level	Int8	RTT level reached by the walk
rtt_addr	Address	Address of RTT reached by the walk
entry	RmmRttEntry	RTTE reached by the walk

I_{ZSRCD} The function `RmmRttWalkResult RttWalk(rd, addr, level)` is used to represent an RTT walk.

I_{FBZPQ} The input address to an RTT walk is always less than `2^w`, where `w` is the IPA width of the target Realm.

See also:

- A2.1.3 *Realm attributes*

- B1.4 *Command condition expressions*
- B2.66 *RttWalk function*
- B3.3.1 *RMI_DATA_CREATE command*
- B3.3.2 *RMI_DATA_CREATE_UNKNOWN command*
- B3.3.3 *RMI_DATA_DESTROY command*
- B3.3.15 *RMI_RTT_CREATE command*
- B3.3.16 *RMI_RTT_DESTROY command*
- B3.3.19 *RMI_RTT_MAP_UNPROTECTED command*
- B3.3.22 *RMI_RTT_UNMAP_UNPROTECTED command*
- C1.23 *RmmRttWalkResult type*

A5.5.11 RTT entry attributes

R_{KCFCT} The cacheability attributes of an RTT entry whose state is ASSIGNED are independent of any stage 1 descriptors and of the state of the stage 1 MMU.

U_{NPVGN} The RMM uses FEAT_S2FWB to ensure that the cacheability attributes of an RTT entry whose state is ASSIGNED are independent of stage 1 translation.

R_{JZKMH} The attributes of an RTT entry whose state is ASSIGNED include the following:

- Normal memory
- Inner Write-Back Cacheable
- Inner Shareable

D_{FJTMF} The following attributes of an RTT entry whose state is VALID_NS are *Host-controlled RTT attributes*:

- `ADDR`
- `MemAttr[2:0]`
- `S2AP`
- `SH`

X_{QHLKB} In an RTT entry whose state is VALID_NS, `MemAttr[3]` is RES0 because the RMM uses FEAT_S2FWB.

R_{JRZTL} Hardware access flag and dirty bit management is disabled for the stage 2 translation used by a Realm.

I_{QFGJC} Hardware access flag and dirty bit management may be enabled by software executing within the Realm, for its own stage 1 translation.

See also:

- A5.2.1 *Realm IPA space*
- B2.51 *RttDescriptorIsValidForUnprotected function*
- B3.3.19 *RMI_RTT_MAP_UNPROTECTED command*
- B3.3.20 *RMI_RTT_READ_ENTRY command*

Chapter A6
Realm interrupts and timers

This specification requires that a virtual Generic Interrupt Controller (vGIC) is presented to a Realm. This vGIC should be architecturally compliant with respect to GICv3 with no legacy operation.

The Host is able to inject virtual interrupts using the GIC virtual CPU interface.

The vGIC presented to a Realm is expected to be implemented via a combination of Host emulation and RMM mediation, as follows:

- Management of Non-secure physical interrupts is performed by the Host, via the GIC Interrupt Routing Infrastructure (IRI).
- The Host is responsible for emulating a GICv3 distributor MMIO interface.
- The Host is responsible for emulating a GICv3 redistributor MMIO interface for each REC.
- The GIC MMIO interfaces emulated by the Host must be presented to the Realm via its Unprotected IPA space.
- The Host may optionally provide a virtual Interrupt Translation Service (ITS). The Realm must allocate ITS tables within its Unprotected IPA space.
- The RMM allows the Host to control some of the GIC virtual CPU interface state which is observed by the Realm. This state is designed to be the minimum required to allow the Host to correctly manage interrupts for the Realm, with integrity guaranteed by the RMM for the remainder of the GIC CPU interface state.
- On REC exit, the RMM exposes some of the GIC virtual CPU interface state to the Host. This state is designed to be the minimum required to allow the Host to correctly manage interrupts for the Realm, with confidentiality guaranteed by the RMM for the remainder of the GIC virtual CPU interface state.

On every REC exit, the EL1 timer state is exposed to the Host. The RMM guarantees that a Realm exit occurs whenever a Realm EL1 timer asserts or de-asserts its output.

See also:

- *Arm Generic Interrupt Controller (GIC) Architecture Specification version 3 and version 4* [5]
- A5.2.1 *Realm IPA space*
- D1.6 *Realm interrupts and timers flows*

A6.1 Realm interrupts

This section describes the programming model for a REC's GIC CPU interface.

D_{XZVGB} The value of `entry.gicv3_lrs[n]` is valid if all of the following are true:

- The value is an architecturally valid encoding of `ICH_LR<n>_EL2` according to *Arm Generic Interrupt Controller (GIC) Architecture Specification version 3 and version 4* [5].
- `HW == '0'`.

X_{DMSDZ} The GICv3 architecture states that, if `HW == '1'` then the virtual interrupt must be linked to a physical interrupt whose state is Active, otherwise behavior is undefined. The RMM is unable to validate that invariant, so it imposes the constraint that `HW == '0'`.

D_{CPLDX} The value of `entry.gicv3_hcr` is valid if the value is an architecturally valid encoding of `ICH_HCR_EL2` according to *Arm Generic Interrupt Controller (GIC) Architecture Specification version 3 and version 4* [5].

R_{HLFRY} REC entry fails if the value of any `entry.gicv3_*` attribute is invalid.

R_{WNFRW} On REC entry, `ICH_LR<n>_EL2` is set to `entry.gicv3_lrs[n]`, for all values of `n` supported by the PE.

R_{WVGFJ} On REC entry, the following fields in `ICH_HCR_EL2` are set to the corresponding values in `entry.gicv3_hcr`:

- `UIE`
- `LRENPIE`
- `NPIE`
- `VGrp0EIE`
- `VGrp0DIE`
- `VGrp1EIE`
- `VGrp1DIE`
- `TDIR`

I_{SMHXB} On REC entry, fields in `entry.gicv3_hcr` are RES0 except for the following:

- `UIE`
- `LRENPIE`
- `NPIE`
- `VGrp0EIE`
- `VGrp0DIE`
- `VGrp1EIE`
- `VGrp1DIE`
- `TDIR`

X_{LMXCX} The RMM provides access to the GIC virtual CPU interface to the Realm and therefore controls the enable bit and most trap bits in `ICH_HCR_EL2`. The maintenance interrupt control bits are controlled by the Host, because the maintenance interrupts are provided as hints to the hypervisor to allocate List Registers optimally and to correctly emulate GICv3 behavior. The `TDIR` bit is also controlled by the Host because it is used when supporting `EOImode == '1'` in the Realm. This mode is used to allow deactivation of virtual interrupts across RECs. This deactivation must be handled by the Host because the RMM can only operate on a single REC during execution of RMI_REC_ENTER.

R_{LNQRL} A REC exit due to IRQ is not generated for an interrupt which is masked by the value of `ICC_PMR_EL1` at the time of REC entry.

U_{GXCHC} The RMM should preserve the value of `ICC_PMR_EL1` during REC entry.

R_{NKPNC} On REC exit, `exit.gicv3_vmcr` contains the value of `ICH_VMCR_EL2` at the time of the Realm exit.

R_{SKQNF} On REC exit, `exit.gicv3_misr` contains the value of `ICH_MISR_EL2` at the time of the Realm exit.

X_{DBGXB} The Host could in principle infer the value of `ICH_MISR_EL2` at the time of the Realm exit from the combination of `exit.gicv3_lrs[n]` and `exit.gicv3_hcr`. However, this would be cumbersome, error-prone, and diverge from the design of existing hypervisor software.

R_{QKZXD} On REC exit, `exit.gicv3_lrs[n]` contains the value of `ICH_LR<n>_EL2` at the time of the Realm exit, for all values of `n` supported by the PE.

R_{SNVZH} On REC exit, the following fields in `exit.gicv3_hcr` contains the value of the corresponding field in `ICH_HCR_EL2` at the time of the Realm exit:

- `EOIcount`
- `UIE`
- `LRENPIE`
- `NPIE`
- `VGrp0EIE`
- `VGrp0DIE`
- `VGrp1EIE`
- `VGrp1DIE`
- `TDIR`

All other fields contain zero.

R_{FGQXT} On REC exit, the values of the following registers may have changed:

- `ICH_AP0R<n>_EL2`
- `ICH_AP1R<n>_EL2`
- `ICH_LR<n>_EL2`
- `ICH_VMCR_EL2`
- `ICH_HCR_EL2`

S_{QMJVJ} It is the responsibility of the caller to save and restore GIC virtualization system control registers if their value needs to be preserved following execution of RMI_REC_ENTER.

X_{KDGHF} On REC entry, the values of the GIC virtualization control system registers are overwritten. The Non-secure hypervisor runs at EL2 and therefore does not make direct use of the virtual GIC CPU interface for its own execution. This means that saving / restoring the caller's GIC virtualization control system registers would typically not be required and would add additional runtime overhead for each execution of RMI_REC_ENTER.

R_{VSBBS} On REC exit, `ICH_HCR_EL2.En == '0'`.

X_{WLTBX} Disabling the virtual GIC CPU interface ensures that the caller does not receive unexpected GIC maintenance interrupts. A stronger constraint, for example stating that all GIC virtualization control system registers are zero on REC exit, was considered. However, this was rejected on the basis that it may preclude future optimisations, such as returning early from execution of RMI_REC_ENTER, without needing to first write zero to all GIC virtualization control system registers, if an interrupt is pending.

See also:

- *Arm Generic Interrupt Controller (GIC) Architecture Specification version 3 and version 4* [5]
- A4.2 *REC entry*
- A4.3 *REC exit*
- B3.3.14 *RMI_REC_ENTER command*
- B3.4.12 *RmiRecEntry type*
- B3.4.14 *RmiRecExit type*
- D1.6.1 *Interrupt flow*

A6.2 Realm timers

This section describes the programming model for Realm EL1 timers.

R_{LKNDV} Architectural timers are available to a Realm and behave according to their architectural specification.

R_{YWXTJ} During Realm execution, if a Realm EL1 timer asserts its output, a Realm exit occurs.

I_{VFYJV} If the Host has programmed an EL1 timer to assert its output during Realm execution, that timer output is not guaranteed to assert.

R_{FKCHX} If the Host has programmed an EL2 timer to assert its output during Realm execution, that timer output is guaranteed to assert.

R_{RJZRP} Both the virtual and physical counter values are guaranteed to be monotonically increasing when read by a Realm, in accordance with the architectural counter behavior.

R_{JSMQP} When read by a Realm, either the virtual or physical counter returns the same value at a given point in time on a given PE.

X_{YCDMW} In order to ensure that the Realm has a consistent view of time, the virtual timer offset must be fixed for the lifetime of the Realm. The absolute value of the virtual timer offset is not important, so the value zero has been chosen for simplicity of both the specification and the implementation.

Provisional

I_{FKMGZ} The rule that virtual and physical counter values are identical may need to be amended if a future version of the specification supports migration and / or virtualization of time based on the virtual counter differing from the physical counter.

R_{VWQDH} On REC exit, Realm EL1 timer state is exposed via the RecExit object:

- `exit.cntv_ctl` contains the value of `CNTV_CTL_EL0` at the time of the Realm exit.
- `exit.cntv_cval` contains the value of `CNTV_CVAL_EL0` at the time of the Realm exit, expressed as if the virtual counter offset was zero.
- `exit.cntp_ctl` contains the value of `CNTP_CTL_EL0` at the time of the Realm exit.
- `exit.cntp_cval` contains the value of `CNTP_CVAL_EL0` at the time of the Realm exit, expressed as if the physical counter offset was zero.

S_{PYWWF} The Host should check the Realm EL1 timer state on every return from RMI_REC_ENTER, and if a timer condition is met, the Host should inject a virtual interrupt. This is true regardless of the value of `exit.exit_reason`: even if the return occurred for a reason unrelated to timer state (for example, a REC exit due to Data Abort), the timer condition should be checked.

This is to ensure that the Realm does not miss a timer interrupt if, for example, there is no other event causing a return from RMI_REC_ENTER. In other words, the RMM only guarantees that the Host can observe a change in timer output state during return from RMI_REC_ENTER, but does not guarantee a REC exit specifically indicating an asserted timer output change.

See also:

- A4.3 *REC exit*
- B3.4.14 *RmiRecExit type*
- D1.6.2 *Timer interrupt delivery flow*

Chapter A7
Realm measurement and attestation

This section describes how the initial state of a Realm is measured and can be attested.

A7.1 Realm measurements

This section describes how Realm measurement values are calculated.

D_{SJWWS} A Realm measurement value is a rolling hash.

D_{YKDBY} A *Realm Hash Algorithm* (RHA) is an algorithm which is used to extend a Realm measurement value.

I_{NRKWB} The RHA used by a Realm is selected via the `hash_algo` attribute.

See also:

- A2.1.3 *Realm attributes*
- A3.1.1 *Realm hash algorithm*
- A7.2.3.1.3 *Realm Initial Measurement claim*
- A7.2.3.1.4 *Realm Extensible Measurements claim*

A7.1.1 Realm Initial Measurement

This section describes how the Realm Initial Measurement (RIM) is calculated.

I_{XKSBZ} The initial RIM value for a Realm is calculated from a subset of the Realm parameters.

I_{NCNDK} A RIM is extended by applying the RHA to the inputs of RMM operations which are executed during Realm construction.

I_{NQQTF} The following operations cause a RIM to be extended:

- Creation of a DATA Granule during Realm construction
- Creation of a REC
- Changes to RIPAS of Protected IPA during Realm construction

R_{VMPZG} On execution of an operation which requires extension of a RIM, the RMM first constructs a *measurement descriptor* structure. The measurement descriptor contents include the current RIM value. The new RIM value is computed by applying the RHA to the measurement descriptor.

$$desc = MeasurementDescriptor(M_{i-1}, ...)$$
$$M_i = RHA(desc)$$

I_{FQHFC} A RIM is immutable while the state of the Realm is ACTIVE. This implies that a RIM reflects the configuration and contents of the Realm, at the moment when it transitioned from the NEW to the ACTIVE state.

I_{DQGPT} A RIM depends upon the order of the RMM operations which are executed during Realm construction.

S_{VZNCW} The order in which RMM operations are executed during Realm construction must be agreed between the Realm owner (or a delegate of the Realm owner which will receive and validate the RIM) and the Host which executes the RMM commands. This ensures that a correctly-constructed Realm will have the expected measurement.

I_{LTWBL} The value of a RIM can be read using the RSI_MEASUREMENT_READ command.

See also:

- B3.3.1.4 *RMI_DATA_CREATE extension of RIM*
- B3.3.9.4 *RMI_REALM_CREATE initialization of RIM*
- B3.3.12.4 *RMI_REC_CREATE extension of RIM*
- B3.3.18.4 *RMI_RTT_INIT_RIPAS extension of RIM*
- B4.3.7 *RSI_MEASUREMENT_READ command*

A7.1.2 Realm Extensible Measurement

This section describes the behavior of a Realm Extensible Measurement (REM).

I_{QJDWM} A REM is extended using the RSI_MEASUREMENT_EXTEND command.

I_{CTMBT} The value of a REM can be read using the RSI_MEASUREMENT_READ command.

I_{MDQRP} The initial value of a REM is zero.

See also:

- B4.3.6 *RSI_MEASUREMENT_EXTEND command*
- B4.3.7 *RSI_MEASUREMENT_READ command*

A7.2 Realm attestation

This section describes the primitives which are used to support remote Realm attestation.

A7.2.1 Attestation token

D_{VRRLN} A *CCA attestation token* is a collection of claims about the state of a Realm and of the CCA platform on which the Realm is running.

I_{BXBSD} A CCA attestation token consists of two parts:

- Realm token

  Contains attributes of the Realm, including:

  – Realm Initial Measurement
  – Realm Extensible Measurements

- CCA platform token

  Contains attributes of the CCA platform on which the Realm is running, including:

  – CCA platform identity
  – CCA platform lifecycle state
  – CCA platform software component measurements

See also:

- A7.1.1 *Realm Initial Measurement*
- A7.1.2 *Realm Extensible Measurement*

A7.2.2 Attestation token generation

I_{KRMRH} The process for a Realm to obtain an attestation token is:

- Call RSI_ATTESTATION_TOKEN_INIT once
- Call RSI_ATTESTATION_TOKEN_CONTINUE in a loop, until the result is not RSI_INCOMPLETE

S_{XMLMF} The following pseudocode illustrates the process of a Realm obtaining an attestation token.

```
int get_attestation_token(...)
{
    int ret;

    ret = RSI_ATTESTATION_TOKEN_INIT(...);
    if (ret) {
        return ret;
    }

    do {
        ret = RSI_ATTESTATION_TOKEN_CONTINUE(...);
    } while (ret == RSI_INCOMPLETE)

    return ret;
}
```

I_{ZWQCB} Up to one attestation token generation operation may be ongoing on a REC.

I_{TMJVG} On execution of RSI_ATTESTATION_TOKEN_INIT, if an attestation token generation operation is ongoing on the calling REC, it is terminated.

R_{BXKKY} The size of an attestation token is no larger than 4KB.

I_{WTKDD} The challenge value provided to RSI_ATTESTATION_TOKEN_INIT is included in the generated attestation token. This allows the relying party to establish freshness of the attestation token.

If the size of the challenge provided by the relying party is less than 64 bytes, it should be zero-padded prior to calling RSI_ATTESTATION_TOKEN_INIT. Arm recommends that the challenge should contain at least 32 bytes of unique data.

I_{LTDJM} The token address passed to RSI_ATTESTATION_TOKEN_CONTINUE must match the token address passed to the preceding call to RSI_ATTESTATION_TOKEN_INIT.

I_{GKDJW} Generation of an attestation token can be a long-running operation, during which interrupts may need to be handled.

I_{CXSJP} If a physical interrupt becomes pending during execution of RSI_ATTESTATION_TOKEN_CONTINUE, a REC exit due to IRQ can occur.

On the next entry to the REC:

- If a virtual interrupt is pending on that REC, it is taken to the REC's exception handler
- RSI_ATTESTATION_TOKEN_CONTINUE returns RSI_INCOMPLETE
- The REC should call RSI_ATTESTATION_TOKEN_CONTINUE again

See also:

- A4.3.5 *REC exit due to IRQ*
- A6.1 *Realm interrupts*
- A7.2.3.1.1 *Realm challenge claim*
- B4.3.1 *RSI_ATTESTATION_TOKEN_CONTINUE command*
- B4.3.2 *RSI_ATTESTATION_TOKEN_INIT command*
- D1.7.1 *Attestation token generation flow*
- D1.7.2 *Handling interrupts during attestation token generation flow*

A7.2.3 Attestation token format

I_{TFHGX} The CCA attestation token is a profiled IETF Entity Attestation Token (EAT).

I_{LPTVH} The CCA attestation token is a Concise Binary Object Representation (CBOR) map, in which the map values are the Realm token and the CCA platform token.

I_{YZPHG} The Realm token contains structured data in CBOR, wrapped with a COSE_Sign1 envelope according to the CBOR Object Signing and Encryption (COSE) standard.

I_{MMQZG} The Realm token is signed by the Realm Attestation Key (RAK).

I_{WBGNP} The CCA platform token contains structured data in CBOR, wrapped with a COSE_Sign1 envelope according to the COSE standard.

I_{CGYKX} The CCA platform token is signed by the Initial Attestation Key (IAK).

I_{CCGQH} The CCA platform token contains a hash of RAK_pub. This establishes a cryptographic binding between the Realm token and the CCA platform token.

I_{PTKYD} The CCA attestation token is defined as follows:

```
cca-token = #6.399(cca-token-collection) ; EAT token-collection extension

cca-platform-token = COSE_Sign1_Tagged
cca-realm-delegated-token = COSE_Sign1_Tagged

cca-token-collection = {
    44234 => cca-platform-token          ; 44234 = 0xACCA
    44241 => cca-realm-delegated-token
}
```

```
; EAT standard definitions
COSE_Sign1_Tagged = #6.18(COSE_Sign1)

; Deliberately shortcut these definitions until EAT is finalised and able to
; pull in the full set of definitions
COSE_Sign1 = "COSE-Sign1 placeholder"
```

IHZNNH The composition of the CCA attestation token is summarised in the following figure.

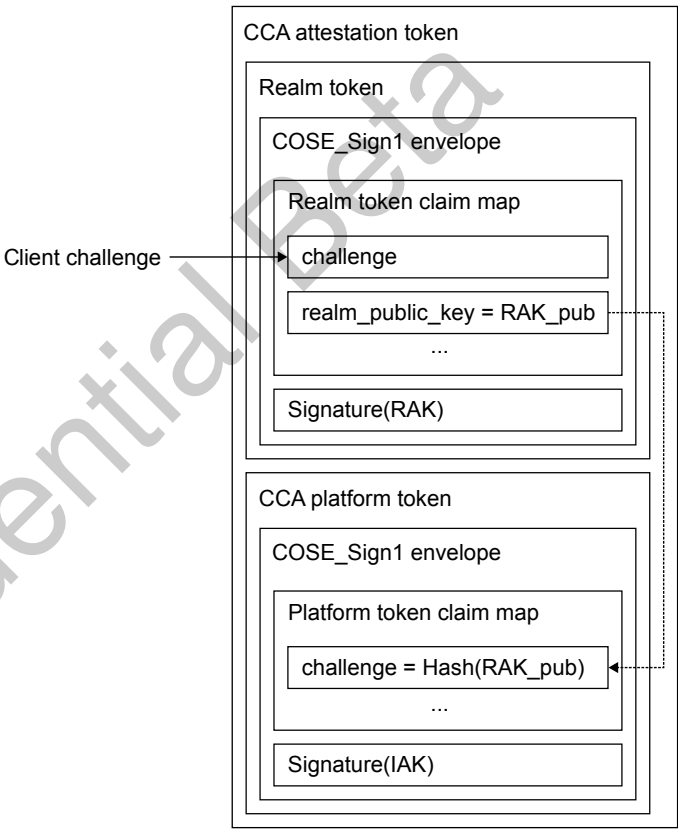


Figure A7.1: Attestation token format

See also:

- *Arm CCA Security model* [4]
- *Concise Binary Object Representation (CBOR)* [6]
- *CBOR Object Signing and Encryption (COSE)* [7]
- *Entity Attestation Token (EAT)* [8]
- A7.2.3.1 *Realm claims*
- A7.2.3.2 *CCA platform claims*

A7.2.3.1 Realm claims

This section defines the format of the Realm token claim map. The format is described using a combination of Concise Data Definition Language (CDDL) and text description.

I_{HKBHC} The Realm token claim map is defined as follows:

```
cca-realm-claims = (cca-realm-claim-map)

cca-realm-claim-map = {
    cca-realm-challenge
    cca-realm-personalization-value
    cca-realm-initial-measurement
    cca-realm-extensible-measurements
    cca-realm-hash-algo-id
    cca-realm-public-key
    cca-realm-public-key-hash-algo-id
}
```

See also:

- *Concise Data Definition Language (CDDL)* [9]
- A7.2.3.1.1 *Realm challenge claim*
- A7.2.3.1.2 *Realm Personalization Value claim*
- A7.2.3.1.3 *Realm Initial Measurement claim*
- A7.2.3.1.4 *Realm Extensible Measurements claim*
- A7.2.3.1.5 *Realm hash algorithm ID claim*
- A7.2.3.1.6 *Realm public key claim*
- A7.2.3.1.7 *Realm public key hash algorithm identifier claim*
- A7.2.3.1.8 *Collated CDDL for Realm claims*
- A7.2.3.1.9 *Example Realm claims*

A7.2.3.1.1 Realm challenge claim

I_{TFWXQ} The Realm challenge claim is used to carry the challenge provided by the caller to demonstrate freshness of the generated token.

I_{RVLZK} The Realm challenge claim is identified using the EAT `nonce` label (10).

I_{MNVNP} The length of the Realm challenge is 64 bytes.

I_{PXMXF} The Realm challenge claim must be present in a Realm token.

I_{BXGFN} The format of the Realm challenge claim is defined as follows:

```
cca-realm-challenge-label = 10
cca-realm-challenge-type = bytes .size 64

cca-realm-challenge = (
    cca-realm-challenge-label => cca-realm-challenge-type
)
```

See also:

- A7.2.2 *Attestation token generation*
- B4.3.2 *RSI_ATTESTATION_TOKEN_INIT command*

A7.2.3.1.2 Realm Personalization Value claim

I_{SCNXB} The Realm Personalization Value claim contains the RPV which was provided at Realm creation.

I_{BKZPD} The Realm Personalization Value claim must be present in a Realm token.

I_{QKNDV} The format of the Realm Personalization Value claim is defined as follows:

```
cca-realm-personalization-value-label = 44235
cca-realm-personalization-value-type = bytes .size 64

cca-realm-personalization-value = (
    cca-realm-personalization-value-label => cca-realm-personalization-value-type
)
```

See also:

- A2.1.3 *Realm attributes*

A7.2.3.1.3 Realm Initial Measurement claim

I_{BXKGD} The Realm Initial Measurement claim contains the values of the Realm Initial Measurement.

I_{FZQSM} The Realm Initial Measurement claim must be present in a Realm token.

I_{GGTNH} The format of the Realm Initial Measurement claim is defined as follows:

```
cca-realm-measurement-type = bytes .size 32 / bytes .size 48 / bytes .size 64

cca-realm-initial-measurement-label = 44238

cca-realm-initial-measurement = (
    cca-realm-initial-measurement-label => cca-realm-measurement-type
)
```

See also:

- A7.1 *Realm measurements*
- A7.2.3.1.4 *Realm Extensible Measurements claim*

A7.2.3.1.4 Realm Extensible Measurements claim

I_{KFNMV} The Realm Extensible Measurements claim contains the values of the Realm Extensible Measurements.

I_{DSNFB} The Realm Extensible Measurements claim must be present in a Realm token.

I_{ZKVMN} The format of the Realm measurements claim is defined as follows:

```
cca-realm-measurement-type = bytes .size 32 / bytes .size 48 / bytes .size 64

cca-realm-extensible-measurements-label = 44239

cca-realm-extensible-measurements = (
    cca-realm-extensible-measurements-label => [ 4*4 cca-realm-measurement-type ]
)
```

See also:

- A7.1 *Realm measurements*
- A7.2.3.1.3 *Realm Initial Measurement claim*

A7.2.3.1.5 Realm hash algorithm ID claim

I_{DGCGG} The Realm hash algorithm ID claim identifies the algorithm used to calculate all hash values which are present in the Realm token.

I_{PVLCJ} Arm recommends that the value of the Realm hash algorithm ID claim is an IANA Hash Function name *IANA Hash Function Textual Names* [10].

I_{WKVCQ} The Realm hash algorithm ID claim must be present in a Realm token.

I_{PWPLJ} The format of the Realm hash algorithm ID claim is defined as follows:

```
cca-realm-hash-algo-id-label = 44236

cca-realm-hash-algo-id = (
    cca-realm-hash-algo-id-label => text
)
```

A7.2.3.1.6 Realm public key claim

I_{ZCFMQ} The Realm public key claim identifies the key which is used to sign the Realm token.

I_{WBNHC} The value of the Realm public key claim is RAK_pub, encoded according to *SEC 1: Elliptic Curve Cryptography, version 2.0* [11].

I_{LSNPQ} The Realm public key claim must be present in a Realm token.

I_{NNNDS} The format of the Realm public key claim is defined as follows:

```
cca-realm-public-key-label = 44237

; TODO: support public key sizes other than ECC-P384
cca-realm-public-key-type = bytes .size 97

cca-realm-public-key = (
    cca-realm-public-key-label => cca-realm-public-key-type
)
```

See also:

- *SEC 1: Elliptic Curve Cryptography, version 2.0* [11]
- A7.2.3.1.7 *Realm public key hash algorithm identifier claim*
- A7.2.3.2.2 *CCA platform challenge claim*

A7.2.3.1.7 Realm public key hash algorithm identifier claim

I_{WWSLP} The Realm public key hash algorithm identifier claim identifies the algorithm used to calculate H(RAK_pub).

I_{TNRBN} The Realm public key hash algorithm identifier claim must be present in a Realm token.

I_{NNPVX} The format of the Realm public key hash algorithm identifier claim is defined as follows:

```
cca-realm-public-key-hash-algo-id-label = 44240

cca-realm-public-key-hash-algo-id = (
    cca-realm-public-key-hash-algo-id-label => text
)
```

See also:

- *SEC 1: Elliptic Curve Cryptography, version 2.0* [11]
- A7.2.3.1.6 *Realm public key claim*
- A7.2.3.2.2 *CCA platform challenge claim*

A7.2.3.1.8 Collated CDDL for Realm claims

D_{DCYXZ} The format of the Realm token claim map is defined as follows:

```
cca-realm-claims = (cca-realm-claim-map)

cca-realm-claim-map = {
    cca-realm-challenge
    cca-realm-personalization-value
    cca-realm-initial-measurement
    cca-realm-extensible-measurements
    cca-realm-hash-algo-id
    cca-realm-public-key
    cca-realm-public-key-hash-algo-id
}

cca-realm-challenge-label = 10
cca-realm-challenge-type = bytes .size 64

cca-realm-challenge = (
    cca-realm-challenge-label => cca-realm-challenge-type
)

cca-realm-personalization-value-label = 44235
cca-realm-personalization-value-type = bytes .size 64

cca-realm-personalization-value = (
    cca-realm-personalization-value-label => cca-realm-personalization-value-type
)

cca-realm-measurement-type = bytes .size 32 / bytes .size 48 / bytes .size 64

cca-realm-initial-measurement-label = 44238

cca-realm-initial-measurement = (
    cca-realm-initial-measurement-label => cca-realm-measurement-type
)

cca-realm-extensible-measurements-label = 44239

cca-realm-extensible-measurements = (
    cca-realm-extensible-measurements-label => [ 4*4 cca-realm-measurement-type ]
)

cca-realm-hash-algo-id-label = 44236

cca-realm-hash-algo-id = (
    cca-realm-hash-algo-id-label => text
)

cca-realm-public-key-label = 44237

; TODO: support public key sizes other than ECC-P384
cca-realm-public-key-type = bytes .size 97

cca-realm-public-key = (
    cca-realm-public-key-label => cca-realm-public-key-type
)

cca-realm-public-key-hash-algo-id-label = 44240
```

```
cca-realm-public-key-hash-algo-id = (
    cca-realm-public-key-hash-algo-id-label => text
)
```

A7.2.3.1.9 Example Realm claims

I_{CPTFR} An example Realm claim map is shown below in COSE-DIAG format:

```
/ Realm claim map /
{
    / cca-realm-challenge /
    10: h'ABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABAB
          ABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABAB',

    / cca-realm-personalization-value /
    44235: h'ABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABAB
             ABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABABAB',

    / cca-realm-initial-measurement /
    44238: h'000000000000000000000000000000000000000000000000000000000000000000000000',

    / cca-realm-extensible-measurements /
    44239: [
        h'000000000000000000000000000000000000000000000000000000000000000000000000',
        h'000000000000000000000000000000000000000000000000000000000000000000000000',
        h'000000000000000000000000000000000000000000000000000000000000000000000000',
        h'000000000000000000000000000000000000000000000000000000000000000000000000'
    ],

    / cca-realm-hash-algo-id /
    44236: "sha-256",

    / cca-realm-public-key /
    44237: h'0476F988091BE585ED41801AECFAB858548C63057E16B0E676120BBD0D2F9C29
             E056C5D41A0130EB9C21517899DC23146B28E1B062BD3EA4B315FD219F1CBB52
             8CB6E74CA49BE16773734F61A1CA61031B2BBF3D918F2F94FFC4228E50919544
             AE',

     / cca-realm-public-key-hash-algo-id /
     44240: "sha-256"
}
```

A7.2.3.2 CCA platform claims

This section defines the format of the CCA platform token claim map. The format is described using a combination of Concise Data Definition Language (CDDL) and text description.

I_{FJKFY} The Realm token claim map is defined as follows:

```
cca-platform-claims = (cca-platform-claim-map)

cca-platform-claim-map = {
    cca-platform-profile
    cca-platform-challenge
    cca-platform-implementation-id
    cca-platform-instance-id
    cca-platform-config
    cca-platform-lifecycle
    cca-platform-sw-components
    ? cca-platform-verification-service
    cca-platform-hash-algo-id
}
```

See also:

- *Concise Data Definition Language (CDDL)* [9]
- A7.2.3.2.1 *CCA platform profile claim*
- A7.2.3.2.2 *CCA platform challenge claim*
- A7.2.3.2.3 *CCA platform Implementation ID claim*
- A7.2.3.2.4 *CCA platform Instance ID claim*
- A7.2.3.2.5 *CCA platform config claim*
- A7.2.3.2.6 *CCA platform lifecycle claim*
- A7.2.3.2.7 *CCA platform software components claim*
- A7.2.3.2.8 *CCA platform verification service claim*
- A7.2.3.2.9 *CCA platform hash algorithm ID claim*
- A7.2.3.2.10 *Collated CDDL for CCA platform claims*
- A7.2.3.2.11 *Example CCA platform claims*

A7.2.3.2.1 CCA platform profile claim

I_{FQYTP} The CCA platform profile claim identifies the EAT profile to which the CCA platform token conforms. Note that because the platform token is expected to be issued when bound to a Realm token, the profile document should include a description of the Realm claims.

I_{XMVFR} The CCA platform profile claim is identified using the EAT `profile` label (265).

I_{GMKNR} The CCA platform profile claim must be present in a CCA platform token.

I_{MHRTD} The format of the CCA platform profile claim is defined as follows:

```
cca-platform-profile-label = 265 ; EAT profile

cca-profile-type = "http://arm.com/CCA-SSD/1.0.0"

cca-platform-profile = (
    cca-platform-profile-label => cca-profile-type
)
```

A7.2.3.2.2 CCA platform challenge claim

I_{TKTWZ} The CCA platform challenge claim contains a hash of the public key used to sign the Realm token.

I_{CLJKK} The CCA platform challenge claim is identified using the EAT `nonce` label (10).

I_{XHLYJ} The length of the CCA platform challenge is either 32, 48 or 64 bytes.

I_{GVHNX} The CCA platform challenge claim must be present in a CCA platform token.

I_{LRWHR} The format of the CCA platform challenge claim is defined as follows:

```
cca-hash-type = bytes .size 32 / bytes .size 48 / bytes .size 64

cca-platform-challenge-label = 10

cca-platform-challenge = (
    cca-platform-challenge-label => cca-hash-type
)
```

See also:

- A7.2.3.1.6 *Realm public key claim*

A7.2.3.2.3 CCA platform Implementation ID claim

I_{SMWND} The CCA platform Implementation ID claim uniquely identifies the implementation of the CCA platform.

I_{NDVFB} The value of the CCA platform Implementation ID claim can be used by a verification service to locate the details of the CCA platform implementation from an endorser or manufacturer. Such details are used by a verification service to determine the security properties or certification status of the CCA platform implementation.

I_{RXPVW} The semantics of the CCA platform Implementation ID value are defined by the manufacturer or a particular certification scheme. For example, the ID could take the form of a product serial number, database ID, or other appropriate identifier.

I_{SRPZY} The CCA platform Implementation ID claim does not identify a particular instance of the CCA implementation.

I_{NTCFY} The CCA platform Implementation ID claim must be present in a CCA platform token.

I_{DHYDG} The format of the CCA platform Implementation ID claim is defined as follows:

```
cca-platform-implementation-id-label = 2396 ; PSA implementation ID
cca-platform-implementation-id-type = bytes .size 32

cca-platform-implementation-id = (
    cca-platform-implementation-id-label => cca-platform-implementation-id-type
)
```

See also:

- *Arm CCA Security model* [4]
- A7.2.3.2.4 *CCA platform Instance ID claim*

A7.2.3.2.4 CCA platform Instance ID claim

I_{ZYRZB} The CCA platform Instance ID claim represents the unique identifier of the Initial Attestation Key (IAK) for the CCA platform.

I_{XVLLN} The CCA platform Instance ID claim is identified using the EAT `ueid` label (256).

R_{HVTNC} The first byte of the CCA platform Instance ID value must be `0x01`.

I_{ZNGDF} The CCA platform Instance ID claim must be present in a CCA platform token.

I_{VPKJN} The format of the CCA platform Instance ID claim is defined as follows:

```
cca-platform-instance-id-label = 256 ; EAT ueid

; TODO: require that the first byte of cca-platform-instance-id-type is 0x01
; EAT UEIDs need to be 7 - 33 bytes
cca-platform-instance-id-type = bytes .size 33

cca-platform-instance-id = (
    cca-platform-instance-id-label => cca-platform-instance-id-type
```

```
)
```

See also:

- *Arm CCA Security model* [4]
- A7.2.3.2.3 *CCA platform Implementation ID claim*

A7.2.3.2.5 CCA platform config claim

I_{WVQJT} The CCA platform config claim describes the set of chosen implementation options of the CCA platform. As an example, these may include a description of the level of physical memory protection which is provided.

U_{GPXWX} The CCA platform config claim is expected to contain the System Properties field which is present in the Root Non-volatile Storage (RNVS) public parameters.

I_{MJHQJ} The CCA platform config claim must be present in a CCA platform token.

```
cca-platform-config-label = 2401 ; PSA platform range
                                 ; TBD: add to IANA registration
cca-platform-config-type = bytes

cca-platform-config = (
    cca-platform-config-label => cca-platform-config-type
)
```

See also:

- *Tormore system architecture spec* [12]

A7.2.3.2.6 CCA platform lifecycle claim

I_{SYKFY} The CCA platform lifecycle claim identifies the lifecycle state of the CCA platform.

R_{NBFVV} The value of the CCA platform lifecycle claim is an integer which is divided as follows:

- value[15:8]: CCA platform lifecycle state
- value[7:0]: IMPLEMENTATION DEFINED

I_{WFZHV} The CCA platform lifecycle claim must be present in a CCA platform token.

I_{QFYLF} A non debugged CCA platform will be in psa-lifecycle-secured state. Realm Management Security Domain debug is always recoverable, and would therefore be represented by psa-lifecycle-non-psa-rot-debug state. Root world debug is recoverable on a HES system and would be represented by psa-lifecycle-recoverable-psa-rot state. On a non-HES system Root world debug is usually non-recoverable, and would be represented by psa-lifecycle-lifecycle-decommissioned state.

I_{HMZLL} The format of the CCA platform lifecycle claim is defined as follows:

```
cca-platform-lifecycle-label = 2395 ; PSA lifecycle

cca-platform-lifecycle-unknown-type = 0x0000..0x00ff
cca-platform-lifecycle-assembly-and-test-type = 0x1000..0x10ff
cca-platform-lifecycle-cca-platform-rot-provisioning-type = 0x2000..0x20ff
cca-platform-lifecycle-secured-type = 0x3000..0x30ff
cca-platform-lifecycle-non-cca-platform-rot-debug-type = 0x4000..0x40ff
cca-platform-lifecycle-recoverable-cca-platform-rot-debug-type = 0x5000..0x50ff
cca-platform-lifecycle-decommissioned-type = 0x6000..0x60ff

cca-platform-lifecycle-type =
    cca-platform-lifecycle-unknown-type /
    cca-platform-lifecycle-assembly-and-test-type /
    cca-platform-lifecycle-cca-platform-rot-provisioning-type /
    cca-platform-lifecycle-secured-type /
    cca-platform-lifecycle-non-cca-platform-rot-debug-type /
    cca-platform-lifecycle-recoverable-cca-platform-rot-debug-type /
```

```
    cca-platform-lifecycle-decommissioned-type

cca-platform-lifecycle = (
    cca-platform-lifecycle-label => cca-platform-lifecycle-type
)
```

See also:

- *Arm CCA Security model* [4]

A7.2.3.2.7 CCA platform software components claim

I_{PJCSC} The CCA platform software components claim is a list of software components which can affect the behavior of the CCA platform. It is expected that an implementation will describe the expected software component values within the profile.

I_{TJTXG} The CCA platform software components claim must be present in a CCA platform token.

I_{DPSKT} The format of the CCA platform software components claim is defined as follows:

```
cca-platform-sw-components-label = 2399 ; PSA software components

cca-platform-sw-component = {
  ? 1 => text,          ; component type
    2 => cca-hash-type, ; measurement value
  ? 4 => text,          ; version
    5 => cca-hash-type, ; signer id
  ? 6 => text,          ; hash algorithm identifier
}

cca-platform-sw-components = (
    cca-platform-sw-components-label => [ + cca-platform-sw-component ]
)
```

CCA platform software component type

I_{PDNCF} The CCA platform software component type is a string which represents the role of the software component.

I_{TPSYF} The CCA platform software component type is intended for use as a hint to help the relying party understand how to evaluate the CCA platform software component measurement value.

R_{RSNBH} The CCA platform software component type is optional in a CCA platform token.

CCA platform software component measurement value

I_{RWDKD} The CCA platform software component measurement value represents a hash of the state of the software component in memory at the time it was initialized.

R_{TVXRZ} The CCA platform software component measurement value must be a hash of 256 bits or stronger.

R_{LGBCM} The CCA platform software component measurement value must be present in a CCA platform token.

CCA platform software component version

I_{JVJFW} The CCA platform software component version is a text string whose meaning is defined by the software component vendor.

R_{CZRXB} The CCA platform software component version is optional in a CCA platform token.

CCA platform software component signer ID

I_{DCDMR} The CCA platform software component signer ID is the hash of a signing authority public key for the software component. It can be used by a verifier to ensure that the software component was signed by an expected trusted source.

R_{PXRMC} The CCA platform software component signer ID value must be a hash of 256 bits or stronger.

R_{XPHQC} The CCA platform software signer ID must be present in a CCA platform token.

CCA platform software hash algorithm ID

I_{TQWZX} The CCA platform software hash algorithm ID identifies the way in which the hash algorithm used to measure the CCA platform software component.

I_{HHBHG} Arm recommends that the value of the CCA platform software hash algorithm ID is an IANA Hash Function name *IANA Hash Function Textual Names* [10].

I_{NJYCM} Arm recommends that the hash algorithm used to measure the CCA platform software component is one of the algorithms listed in the *Arm CCA Security model* [4].

I_{HPHCD} The CCA platform software hash algorithm ID is optional in a CCA platform token.

A7.2.3.2.8 CCA platform verification service claim

I_{NSTDP} The CCA platform verification service claim is a hint which can be used by a relying party to locate a verifier for the token.

I_{RZJSQ} The value of the CCA platform verification service claim is a text string which can be used to locate the service or a URL specifying the address of the service.

I_{MFYCX} The CCA platform verification service claim may be ignored by a relying party in favor of other information.

I_{MRSXY} The CCA platform verification service claim is optional in a CCA platform token.

I_{WRJSX} The format of the CCA platform verification service claim is defined as follows:

```
cca-platform-verification-service-label = 2400 ; PSA verification service
cca-platform-verification-service-type = text

cca-platform-verification-service = (
    cca-platform-verification-service-label =>
        cca-platform-verification-service-type
)
```

A7.2.3.2.9 CCA platform hash algorithm ID claim

I_{VDZMF} The CCA platform hash algorithm ID claim identifies the algorithm used to calculate the extended measurements in the CCA platform token.

I_{YRPYY} Arm recommends that the value of the CCA platform hash algorithm ID claim is an IANA Hash Function name *IANA Hash Function Textual Names* [10].

I_{TQSTK} The CCA platform hash algorithm ID claim must be present in a CCA platform token.

I_{RKZJT} The format of the CCA platform hash algorithm ID claim is defined as follows:

```
cca-platform-hash-algo-id-label = 2402 ; PSA platform range
                                       ; TBD: add to IANA registration

cca-platform-hash-algo-id = (
    cca-platform-hash-algo-id-label => text
)
```

A7.2.3.2.10 Collated CDDL for CCA platform claims

DDVMJZ The format of the CCA platform token claim map is defined as follows:

```
cca-platform-claims = (cca-platform-claim-map)

cca-platform-claim-map = {
    cca-platform-profile
    cca-platform-challenge
    cca-platform-implementation-id
    cca-platform-instance-id
    cca-platform-config
    cca-platform-lifecycle
    cca-platform-sw-components
    ? cca-platform-verification-service
    cca-platform-hash-algo-id
}

cca-platform-profile-label = 265 ; EAT profile

cca-profile-type = "http://arm.com/CCA-SSD/1.0.0"

cca-platform-profile = (
    cca-platform-profile-label => cca-profile-type
)
cca-hash-type = bytes .size 32 / bytes .size 48 / bytes .size 64

cca-platform-challenge-label = 10

cca-platform-challenge = (
    cca-platform-challenge-label => cca-hash-type
)

cca-platform-implementation-id-label = 2396 ; PSA implementation ID
cca-platform-implementation-id-type = bytes .size 32

cca-platform-implementation-id = (
    cca-platform-implementation-id-label => cca-platform-implementation-id-type
)

cca-platform-instance-id-label = 256 ; EAT ueid

; TODO: require that the first byte of cca-platform-instance-id-type is 0x01
; EAT UEIDs need to be 7 - 33 bytes
cca-platform-instance-id-type = bytes .size 33

cca-platform-instance-id = (
    cca-platform-instance-id-label => cca-platform-instance-id-type
)

cca-platform-config-label = 2401 ; PSA platform range
                                 ; TBD: add to IANA registration
cca-platform-config-type = bytes

cca-platform-config = (
    cca-platform-config-label => cca-platform-config-type
)

cca-platform-lifecycle-label = 2395 ; PSA lifecycle

cca-platform-lifecycle-unknown-type = 0x0000..0x00ff
```

```
cca-platform-lifecycle-assembly-and-test-type = 0x1000..0x10ff
cca-platform-lifecycle-cca-platform-rot-provisioning-type = 0x2000..0x20ff
cca-platform-lifecycle-secured-type = 0x3000..0x30ff
cca-platform-lifecycle-non-cca-platform-rot-debug-type = 0x4000..0x40ff
cca-platform-lifecycle-recoverable-cca-platform-rot-debug-type = 0x5000..0x50ff
cca-platform-lifecycle-decommissioned-type = 0x6000..0x60ff

cca-platform-lifecycle-type =
    cca-platform-lifecycle-unknown-type /
    cca-platform-lifecycle-assembly-and-test-type /
    cca-platform-lifecycle-cca-platform-rot-provisioning-type /
    cca-platform-lifecycle-secured-type /
    cca-platform-lifecycle-non-cca-platform-rot-debug-type /
    cca-platform-lifecycle-recoverable-cca-platform-rot-debug-type /
    cca-platform-lifecycle-decommissioned-type

cca-platform-lifecycle = (
    cca-platform-lifecycle-label => cca-platform-lifecycle-type
)

cca-platform-sw-components-label = 2399 ; PSA software components

cca-platform-sw-component = {
  ? 1 => text,          ; component type
    2 => cca-hash-type, ; measurement value
  ? 4 => text,          ; version
    5 => cca-hash-type, ; signer id
  ? 6 => text,          ; hash algorithm identifier
}

cca-platform-sw-components = (
    cca-platform-sw-components-label => [ + cca-platform-sw-component ]
)

cca-platform-verification-service-label = 2400 ; PSA verification service
cca-platform-verification-service-type = text

cca-platform-verification-service = (
    cca-platform-verification-service-label =>
        cca-platform-verification-service-type
)

cca-platform-hash-algo-id-label = 2402 ; PSA platform range
                                       ; TBD: add to IANA registration

cca-platform-hash-algo-id = (
    cca-platform-hash-algo-id-label => text
)
```

A7.2.3.2.11 Example CCA platform claims

I_{TVHKL} An example CCA platform claim map is shown below in COSE-DIAG format:

```
/ CCA platform claim map /
{
    / cca-platform-profile /
    265: "http://arm.com/CCA-SSD/1.0.0",

    / cca-platform-challenge /
    10: h'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
          AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA',

    / cca-platform-implementation-id /
    2396: h'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA',

    / cca-platform-instance-id /
    256: h'010BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
           BB',

    / cca-platform-config /
    2401: h'CFCFCFCF',

    / cca-platform-lifecycle /
    2395: 12288,

    / cca-platform-sw-components /
    2399: [
        {
            / measurement value /
            2: h'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
                 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA',

            / signer id /
            5: h'BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
                 BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB',

            / version /
            4: "1.0.0",

            / hash algorithm identifier /
            6: "sha-256"
        },
        {
            / measurement value /
            2: h'CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
                 CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC',

            / signer id /
            5: h'DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
                 DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD',

            / version /
            4: "1.0.0",

            / hash algorithm identifier /
            6: "sha-256"
        }
    ],

    / cca-platform-verification-service /
```

```
    2400: "https://cca_verifier.org",

    / cca-platform-hash-algo-id /
    2402: "sha-256"
}
```

Chapter A8
Realm debug and performance monitoring

This section describes the debug and performance monitoring features which are available to a Realm.

A8.1 Realm PMU

This section describes the programming model for usage of PMU by a Realm.

R_{DNNQQ} On REC entry, Realm PMU state is restored from the REC.

R_{LHRYJ} On REC exit, the following Realm PMU state is exposed via the RecExit object:

- `exit.pmu_ovf` contains the value of `PMOVSSET_EL0` at the time of the Realm exit.
- `exit.pmu_intr_en` contains the value of `PMINTENSET_EL1` at the time of the Realm exit.
- `exit.pmu_cntr_en` contains the value of `PMCNTENSET_EL0` at the time of the Realm exit.

On REC exit, all other Realm PMU state is saved to the REC.

See also:

- A3.1.5 *Realm support for Performance Monitors Extension*
- A4.3 *REC exit*
- B3.4.14 *RmiRecExit type*

Part B
Interface

Chapter B1
Commands

This chapter describes how RMM commands are defined in this specification.

B1.1 Overview

I_{VZRKZ} The RMM exposes the following interfaces:

- The *Realm Management Interface* (RMI)
- The *Realm Services Interface* (RSI)
- The *Power State Coordination Interface* (PSCI)

I_{TKQXF} An RMM interface consists of a set of RMM commands.

I_{RTRYT} An RMM interface is compliant with the SMC Calling Convention (SMCCC).

R_{NNFPH} SMCCC version >= 1.2 is required.

X_{FDXJG} SMCCC version 1.2 increases the number of SMC64 arguments and return values from 4 to 17. Some RMM commands use more than 4 input or output values.

R_{VXJJQ} On a CCA platform which implements FEAT_SVE, SMCCC version >= 1.3 is required.

X_{KCMSY} SMCCC version 1.3 introduces a bit in the FID which a caller can use to indicate that SVE state does not need to be preserved across the SMC call.

R_{JNVJQ} On a CCA platform which implements FEAT_SME, SMCCC version >= 1.4 is required.

X_{QXMZL} SMCCC version 1.4 adds support for preservation of SME state across an SMC call.

R_{KWMVX} An RMM command uses the SMC64 calling convention.

S_{DFNMZ} To determine whether an RMM interface is implemented, software should use the following flow:

1. Determine whether the SMCCC_VERSION command is implemented, following the procedure described in *Arm SMC Calling Convention* [13].
2. Check that the SMCCC version is >= 1.1.
3. Execute the <Interface>.Version command, which returns:
   - SMCCC_NOT_SUPPORTED (-1) if <Interface> is not implemented.
   - A version number (>0) if <Interface> is implemented.

R_{YBXKR} All data types defined in this specification are little-endian.

See also:

- Chapter B3 *Realm Management Interface*
- Chapter B4 *Realm Services Interface*
- Chapter B5 *Power State Control Interface*

B1.2 Command definition

I_{WBMVP} The definition of an RMM command consists of:

- A *function identifier* (FID)
- A set of *input values* (referred to as "arguments" in SMCCC)
- A set of *output values* (referred to as "results" in SMCCC)
- A set of *context values*
- A partially-ordered set of *failure conditions*
- A set of *success conditions*
- A set of *footprint items*

I_{GCVWC} Each failure condition, success condition and footprint item has an associated identifier. Identifiers are unique within each of the above groups, within each command.

An identifier has no meaning. It is only a label by which a given condition or footprint item can be referred to.

See also:

- SMCCC *Arm SMC Calling Convention* [13]

B1.2.1 Example command

I_{NFVGF} The following command, EXAMPLE_ADD, is an example of how the components of an RMM command definition are presented in this document.

This command takes as an input value the address `params_ptr` of an NS Granule which contains two integer values `x` and `y`. On successful execution of the command:

- The output value `sum` contains the sum of `x` and `y`
- The output value `zero` indicates whether either of `x` or `y` is zero

EXAMPLE_ADD is defined as follows:

Interface

FID

`0x042`

Input values

Name	Register	Field	Type	Description
`fid`	`X0`	[63:0]	*UInt64*	Command FID
`params_ptr`	`X1`	[63:0]	*Address*	PA of parameters

Context

The EXAMPLE_ADD command operates on the following context.

Name	Type	Value	Before	Description
`params`	ExampleParams	`Params(params_ptr)`	false	Parameters

Output values

Name	Register	Field	Type	Description
result	X0	[15:0]	CommandReturnCode	Command return status
sum	X1	[63:0]	UInt64	Sum of x and y
zero	X2	[63:0]	UInt64	Whether either x or y was zero

Failure conditions

ID	Condition
params_align	pre: !AddrIsGranuleAligned(params_ptr) post: ResultEqual(result, ERROR_INPUT)
params_state	pre: if Granule(params_ptr).state != NS post: ResultEqual(result, ERROR_MEMORY)

Success conditions

ID	Post-condition
sum	sum == params.x + params.y
zero	zero == (params.x == 0) || (params.y == 0)

B1.3 Command registers

D_{ZDGNM} An *FID* is a value which identifies a particular RMM command.

I_{MJQGK} The FID of an RMM command is unique among the RMM commands in an RMM interface.

I_{RVPGY} An FID is read from general-purpose register X0.

D_{XLSFS} An *input value* is a value read by an RMM command from general-purpose registers.

D_{VCDCW} An *output value* is a value written by an RMM command to general-purpose registers.

D_{CZLVJ} A *command return code* is a value which specifies whether an RMM command succeeded or failed.

I_{FRZFT} A command return code is written to general-purpose register X0.

B1.4 Command condition expressions

D_{CHRYB} A *condition expression* is an expression which evaluates to a boolean value.

I_{ZTNXL} A condition expression can contain the following macros:

- `__OFFSETOF(struct_name, member_name)`

  Expands to the offset in bytes of the struct member from the start of the struct.

- `__SIZEOF(struct_name, member_name)`

  Expands to the size in bytes of the struct member.

I$_{BNPKQ}$ Following expansion of macros, a *condition expression* is a valid expression in Arm Specification Language (ASL).

See also:

- *Arm Specification Language Reference Manual* [14]
- Chapter B2 *Command condition functions*

B1.5 Command context values

D$_{DLBYC}$ A *context value* is a value which is derived from the value of a command input register and which is used by a command condition expression.

I$_{VKKKY}$ A context value can be thought of as a local variable for use by command condition expressions.

For example, consider the following example command condition expressions:

```
!AddrIsGranuleAligned(RealmParams(params_ptr).rtt_base)

!RmiFeatureRegister0IsValid(RealmParams(params_ptr).features_0)
```

By introducing a context value `params` with the value `RealmParams(params_ptr)`, these two command condition expressions can be re-written as:

```
!AddrIsGranuleAligned(params.rtt_base)

!RmiFeatureRegister0IsValid(params.features_0)
```

D$_{QDFNW}$ The `before` property of a context value indicates whether its expression is re-evaluated after the command has executed.

- `before = true`: the expression is not re-evaluated after the command has executed
- `before = false`: the expression is re-evaluated after the command has executed

I$_{LTLQN}$ Specifying `before = true` for a context value allows system state to be sampled before command execution, and then used after command execution in a command success condition.

For example, the RMI_REALM_DESTROY command takes as an input value the address `rd` of a Realm Descriptor. Successful execution of the command results observable effects including the following:

- The state of the RD Granule changes from RD to DELEGATED
- The state of the RTT base Granule, whose address was previously held in the RD, changes from RTT to DELEGATED

The address of the RTT base Granule is not included in the input values of the command.

A context value is defined as follows:

Name	Type	Value	Before	Description
`rtt_base`	Address	`Realm(rd).rtt_base`	true	RTT base address

The state change of the RTT Granule can then be expressed as:

```
Granule(rtt_base).state == DELEGATED
```

I_{YNDGD} The *before* property of a context value has no effect if the value is only used in command failure conditions.

D_{XBHPB} An *in-memory value* is a value passed to a command via an in-memory data structure, the address of which is passed in an input register.

I_{ZTYSS} An in-memory value is a context value.

See also:

- B3.3.9 *RMI_REALM_CREATE command*

B1.6 Command failure conditions

D_{DNQQC} An RMM command *failure condition* defines a way in which the command can fail.

I_{GVBBZ} A failure condition consists of a *pre-condition* and a *post-condition*.

I_{WTSZH} A failure pre-condition can be thought of as the "trigger" of the failure: if the pre-condition is true then the command fails.

I_{KJHNX} A failure post-condition can be thought of as the "effect" of the failure: if the command failed due to a particular trigger, then the post-condition defines the error code which is returned.

I_{CVTGY} A failure pre-condition is a condition expression whose terms can include input values and context values.

I_{HNDNN} A failure post-condition is a condition expression whose terms can include input values and context values.

I_{KHJDY} Observability of the checking of command failure conditions is subject to a partial order.

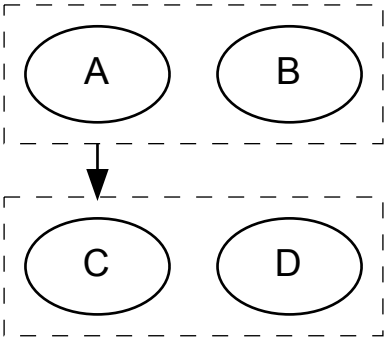
An ordering relation "*A* precedes *B*" means either of the following:

- The pre-condition of B is well-formed only if the pre-condition of *A* is false. This is referred to as a *well-formedness ordering*.
- If the pre-conditions of *A* and *B* are both true, then the post-condition of *A* is observed. This is referred to as a *behavioral ordering*.

The absence of an ordering relation "*A* precedes *B*" means that, if the pre-conditions of *A* and *B* are both true then either the post-condition of *A* is observed or the post-condition of *B* is observed.

Orderings are specified between groups of failure conditions. For example, the expression `[A, B] < [C, D]` means that both conditions A and B precede both conditions C and D.

The same information is also presented graphically, with failure conditions represented as nodes and ordering relations represented as edges.



The specification does not state whether an individual ordering relation is a well-formedness ordering or a behavioral ordering.

I_{JMTTY} A given implementation of the RMM is expected to have deterministic behavior. That is, for a runtime instance of the RMM in a particular state, two executions of a command without an interleaving of other commands, with the same input values, results in the same outcome (either success, or the same failure condition.)

R_{WXZJJ} If a failure pre-condition evaluates to true then the corresponding failure post-condition evaluates to true.

R_{DDGDW} If a failure pre-condition evaluates to true then the command is aborted.

R_{VHFHD} If no failure pre-condition evaluates to true then the command succeeds.

B1.7 Command success conditions

D_{SZGNZ} An RMM command *success condition* define an observable effect of a successful execution of the command.

I_{LZXHB} A success condition is a condition expression whose terms can include input values, context values and output values.

I_{NMCSF} The order in which success conditions are listed has no architectural significance.

I_{NJQFG} If an RMM command succeeds then the return code is <Interface>_SUCCESS.

R_{MKRVV} If an RMM command succeeds then all of its success conditions evaluate to true.

B1.8 Command footprint

D_{ZDJDB} The *footprint* of an RMM command defines the set of state items which successful execution of the command can modify.

I_{XMZYS} The footprint of an RMM command may include state items which are not modified by successful execution of the command.

I_{RWQMJ} If an RMM command changes the state of a Granule then the footprint typically does not include all attributes of the object which is created or destroyed.

For example, the footprint of RMI_REALM_CREATE includes the state of the RD Granule, but does not include attributes of the newly-created Realm.

R_{WZYBV} Except for items in the footprint of an RMM command and registers in the output values of the RMM command, execution of the command does not have any observable effects.

Chapter B2
Command condition functions

This chapter describes functions which are used in command condition expressions.

See also:

- B1.4 *Command condition expressions*

B2.1 AddrInRange function

Returns TRUE if `addr` is within `[base, base+size]`.

```
func AddrInRange(
    addr :: Address,
    base :: Address,
    size :: integer) => boolean
    return ((UInt(addr) >= UInt(base))
        && (UInt(addr) <= UInt(base) + size));
end
```

B2.2 AddrIsAligned function

Returns TRUE if address `addr` is aligned to an `n` byte boundary.

```
func AddrIsAligned(
    addr :: Address,
    n :: integer) => boolean
```

B2.3 AddrIsGranuleAligned function

Returns TRUE if address `addr` is aligned to the size of a Granule.

```
func AddrIsGranuleAligned(
    addr :: Address) => boolean
```

```
func AddrIsGranuleAligned(
    addr :: integer) => boolean
```

See also:

- A2.2 *Granule*

B2.4 AddrIsProtected function

Returns TRUE if address `addr` is a Protected IPA for `realm`.

```
func AddrIsProtected(
    addr :: Address,
    realm :: RmmRealm) => boolean
    return UInt(addr) < 2^(realm.ipa_width - 1);
end
```

B2.5 AddrIsRttLevelAligned function

Returns TRUE if Address `addr` is aligned to the size of the address range described by an RTTE in a level `level` RTT.

Returns FALSE if `level` is invalid.

```
func AddrIsRttLevelAligned(
    addr :: Address,
    level :: integer) => boolean
```

B2.6 AddrRangeIsProtected function

Returns TRUE if all addresses in range `[base, base+size)` are Protected IPAs for `realm`.

```
func AddrRangeIsProtected(
    base :: Address,
    size :: integer,
    realm :: RmmRealm) => boolean
    return (AddrIsProtected(base, realm)
        && size > 0
        && size < 2^realm.ipa_width
        && AddrIsProtected(ToAddress(UInt(base) + size - 1), realm));
end
```

B2.7 CurrentRealm function

Returns the current Realm.

```
func CurrentRealm() => RmmRealm
```

B2.8 CurrentRec function

Returns the current REC.

```
func CurrentRec() => RmmRec
```

B2.9 Gicv3ConfigIsValid function

Returns TRUE if the values of all `entry.gicv3_*` attribute are valid.

```
func Gicv3ConfigIsValid(
    gicv3_hcr :: bits(64),
    gicv3_lrs :: array [16] of bits(64)) => boolean
```

See also:

- A6.1 *Realm interrupts*
- B3.4.12 *RmiRecEntry type*

B2.10 Granule function

Returns the Granule located at physical address `addr`.

```
func Granule(
    addr :: Address) => RmmGranule
```

See also:

- A2.2 *Granule*

B2.11 MpidrEqual function

Returns TRUE if the specified MPIDR values are logically equivalent.

```
func MpidrEqual(
    rmm_mpidr :: bits(64),
    rmi_mpidr :: RmiRecMpidr) => boolean
    return (rmm_mpidr[ 3: 0] == rmi_mpidr.aff0
        &&  rmm_mpidr[15: 8] == rmi_mpidr.aff1
        &&  rmm_mpidr[23:16] == rmi_mpidr.aff2
        &&  rmm_mpidr[31:24] == rmi_mpidr.aff3);
end
```

B2.12 MpidrIsUsed function

Returns TRUE if the specified MPIDR value identifies a REC in the current Realm.

```
func MpidrIsUsed(
    mpidr :: bits(64)) => boolean
```

B2.13 PaIsDelegable function

Returns TRUE if the Granule located at physical address `addr` is delegable.

```
func PaIsDelegable(
    addr :: Address) => boolean
```

B2.14 PsciReturnCodeEncode function

Return encoding for a PsciReturnCode value.

```
func PsciReturnCodeEncode(
    value :: PsciReturnCode) => bits(64)
```

B2.15 ReadMemory function

Read contents of memory at address range [addr + offset, addr + offset + size)

offset and size are both numbers of bytes.

```
func ReadMemory(
    addr :: bits(64),
    offset :: integer,
    size :: integer) => bits(size * 8)
```

B2.16 Realm function

Returns the Realm whose RD is located at physical address `addr`.

```
func Realm(
    addr :: Address) => RmmRealm
```

See also:

- A2.1 *Realm*

B2.17 RealmConfig function

Returns Realm configuration stored at IPA `addr`, mapped in the current Realm.

```
func RealmConfig(
    addr :: Address) => RsiRealmConfig
```

B2.18 RealmHostCall function

Returns Host call data stored at IPA `addr`, mapped in the current Realm.

```
func RealmHostCall(
    addr :: Address) => RsiHostCall
```

B2.19 RealmIsLive function

Returns TRUE if the Realm whose RD is located at physical address `addr` is live.

```
func RealmIsLive(
    addr :: Address) => boolean
```

See also:

- A2.1.4 *Realm liveness*

B2.20 RealmParams function

Returns Realm parameters stored at physical address `addr`.

If the PAS of `addr` is not NS, the return value is UNKNOWN.

```
func RealmParams(
    addr :: Address) => RmiRealmParams
```

See also:

- A2.1.6 *Realm parameters*

B2.21 Rec function

Returns the REC located at physical address `addr`.

```
func Rec(
    addr :: Address) => RmmRec
```

See also:

- A2.3 *Realm Execution Context*

B2.22 RecAuxAlias function

Returns TRUE if any of the first `count` entries in a list of REC auxiliary Granule addresses are aliased - either among themselves, or with the REC address itself.

```
func RecAuxAlias(
    rec :: Address,
    aux :: array [16] of Address,
    count :: integer) => boolean
    assert 0 <= count && count <= 16;
    var sorted = RecAuxSort(aux, count);
    for i = 0 to count - 1 do
        if sorted[i] == rec then
            return TRUE;
        end
        if i >= 1 && sorted[i] == sorted[i - 1] then
            return TRUE;
        end
    end
    return FALSE;
end
```

B2.23 RecAuxAligned function

Returns TRUE if the first `count` entries in a list of REC auxiliary Granule addresses are aligned to the size of a Granule.

```
func RecAuxAligned(
    aux :: array [16] of Address,
    count :: integer) => boolean
    assert 0 <= count && count <= 16;
    for i = 0 to count - 1 do
        if !AddrIsGranuleAligned(aux[i]) then
            return FALSE;
        end
    end
    return TRUE;
end
```

B2.24 RecAuxCount function

Returns the number of auxiliary Granules required for a REC in the Realm described by `rd`.

```
func RecAuxCount(
    rd :: Address) => integer
```

B2.25 RecAuxEqual function

Returns TRUE if the first `count` entries in two lists of REC auxiliary Granule addresses are equal.

```
func RecAuxEqual(
    aux1 :: array [16] of Address,
    aux2 :: array [16] of Address,
    count :: integer) => boolean
    assert 0 <= count && count <= 16;
    for i = 0 to count - 1 do
        if aux1[i] != aux2[i] then
            return FALSE;
        end
    end
    return TRUE;
end
```

B2.26 RecAuxSort function

Sort first `count` entries in array of auxiliary Granule addresses.

```
func RecAuxSort(
    addrs :: array [16] of Address,
    count :: integer) => array [16] of Address
```

B2.27 RecAuxStateEqual function

Returns TRUE if the state of the first `count` entries in a list of REC auxiliary Granule addresses is equal to `state`.

```
func RecAuxStateEqual(
    aux :: array [16] of Address,
    count :: integer,
    state :: RmmGranuleState) => boolean
    assert 0 <= count && count <= 16;
    for i = 0 to count - 1 do
        if (!PaIsDelegable(aux[i])
                || Granule(aux[i]).state != state) then
            return FALSE;
        end
    end
    return TRUE;
end
```

B2.28 RecAuxStates function

Inductive function which identifies the states of the first `count` entries in a list of REC auxiliary Granules.

This function is used in the definition of command footprint.

```
func RecAuxStates(
    aux :: array [16] of Address,
    count :: integer)
```

B2.29 RecFromMpidr function

Returns the REC identified by the specified MPIDR value, in the current Realm.

```
func RecFromMpidr(
    mpidr :: bits(64)) => RmmRec
```

B2.30 RecIndex function

Returns the REC index which corresponds to `mpidr`.

```
func RecIndex(
    mpidr :: RmiRecMpidr) => integer
    return (UInt(mpidr.aff0)
        + 16 * UInt(mpidr.aff1)
        + 16 * 256 * UInt(mpidr.aff2)
        + 16 * 256 * 256 * UInt(mpidr.aff3));
end
```

See also:

- A2.3.3 *REC index and MPIDR value*

B2.31 RecParams function

Returns REC parameters stored at physical address `addr`.

If the PAS of `addr` is not NS, the return value is UNKNOWN.

```
func RecParams(
    addr :: Address) => RmiRecParams
```

B2.32 RecRun function

Returns the RecRun object stored at physical address `addr`.

```
func RecRun(
    addr :: Address) => RmiRecRun
```

See also:

- A4.2 *REC entry*
- A4.3 *REC exit*

B2.33 RemExtend function

Extend REM, using `size` LSBs from `new_value`, with the remaining bits zero-padded to form a 512-bit value.

```
func RemExtend(
    hash_algo :: RmmHashAlgorithm,
    old_value :: RmmRealmMeasurement,
    new_value :: RmmRealmMeasurement,
    size :: integer) => RmmRealmMeasurement
```

See also:

- A7.1.2 *Realm Extensible Measurement*

B2.34 ResultEqual function

Returns TRUE if command result matches the stated value.

```
func ResultEqual(
    result :: RmiCommandReturnCode,
    status :: RmiStatusCode) => boolean
```

```
func ResultEqual(
    result :: RmiCommandReturnCode,
    status :: RmiStatusCode,
    index :: integer) => boolean
```

B2.35 RimExtendData function

Extend RIM with contribution from DATA creation.

```
func RimExtendData(
    realm :: RmmRealm,
    ipa :: Address,
    data :: Address,
    flags :: RmiDataFlags) => RmmRealmMeasurement
```

See also:

- B3.3.1.4 *RMI_DATA_CREATE extension of RIM*

B2.36 RimExtendRec function

Extend RIM with contribution from REC creation.

```
func RimExtendRec(
    realm :: RmmRealm,
    params :: RmiRecParams) => RmmRealmMeasurement
```

See also:

- B3.3.12.4 *RMI_REC_CREATE extension of RIM*

B2.37 RimExtendRipas function

Extend RIM with contribution from RIPAS change.

```
func RimExtendRipas(
    realm :: RmmRealm,
    ipa :: Address,
    level :: integer) => RmmRealmMeasurement
```

See also:

- B3.3.18.4 *RMI_RTT_INIT_RIPAS extension of RIM*

B2.38 RimInit function

Initialize RIM.

```
func RimInit(
    hash_algo :: RmmHashAlgorithm,
    params :: RmiRealmParams) => RmmRealmMeasurement
```

See also:

- B3.3.9.4 *RMI_REALM_CREATE initialization of RIM*

B2.39 RmiFeatureRegister0IsValid function

Returns TRUE if `value` is a valid encoding of RmiFeatureRegister0IsValid.

```
func RmiFeatureRegister0IsValid(
    value :: bits(64)) => boolean
```

See also:

- A3.1 *Realm feature discovery and selection*

B2.40 RmiHashAlgorithmIsSupported function

Returns TRUE if the hash algorithm is supported by the implementation.

```
func RmiHashAlgorithmIsSupported(
    value :: RmiHashAlgorithm) => boolean
```

B2.41 RmiHashAlgorithmIsValid function

Returns TRUE if the value is a valid encoding of RmiHashAlgorithm.

```
func RmiHashAlgorithmIsValid(
    value :: bits(8)) => boolean
```

B2.42 RmiRecCreateFlagsIsValid function

Returns TRUE if `value` is a valid encoding of RmiRecCreateFlags.

```
func RmiRecCreateFlagsIsValid(
    value :: bits(64)) => boolean
```

B2.43 RmiRecMpidrIsValid function

Returns TRUE if the value is a valid encoding of RmiRecMpidr.

```
func RmiRecMpidrIsValid(
    value :: bits(64)) => boolean
    return (value[7:4] == Zeros()
        && value[63:32] == Zeros());
end
```

B2.44 RmiRipasIsValid function

Returns TRUE if the value is a valid encoding of RmiRipas.

```
func RmiRipasIsValid(
    value :: bits(32)) => boolean
```

B2.45 RsiRipasIsValid function

Returns TRUE if the value is a valid encoding of RsiRipas.

```
func RsiRipasIsValid(
```

```
    value :: bits(8)) => boolean
```

B2.46 Rtt function

Returns the RTT at address `rtt`.

```
func Rtt(
    addr :: Address) => RmmRtt
```

B2.47 RttAllEntriesContiguous function

Returns TRUE if all entries in the RTT at address `rtt` at level `level` have contiguous output addresses, starting with `addr`.

```
func RttAllEntriesContiguous(
    rtt :: RmmRtt,
    addr :: Address,
    level :: integer) => boolean
```

See also:

- A5.5 *Realm Translation Table*

B2.48 RttAllEntriesRipas function

Returns TRUE if all entries in the RTT at address `rtt` have RIPAS `ripas`.

```
func RttAllEntriesRipas(
    rtt :: RmmRtt,
    ripas :: RmmRipas) => boolean
```

B2.49 RttAllEntriesState function

Returns TRUE if all entries in the RTT at address `rtt` have state `state`.

```
func RttAllEntriesState(
    rtt :: RmmRtt,
    state :: RmmRttEntryState) => boolean
```

See also:

- A5.5 *Realm Translation Table*

B2.50 RttConfigIsValid function

Returns TRUE if the RTT configuration values provided are self-consistent and are supported by the platform.

```
func RttConfigIsValid(
    ipa_width :: integer,
    rtt_level_start :: integer,
    rtt_num_start :: integer) => boolean
```

See also:

- A5.5 *Realm Translation Table*

B2.51 RttDescriptorIsValidForUnprotected function

Returns TRUE if, within the descriptor `desc`, all of the following are true:

- All fields which are *Host-controlled RTT attributes* are set to architecturally valid values.
- All fields which are not *Host-controlled RTT attributes* are set to zero.

```
func RttDescriptorIsValidForUnprotected(
    desc :: bits(64)) => boolean
```

See also:

- A5.5.11 *RTT entry attributes*

B2.52 RttEntry function

Returns the `i`th entry in the RTT at address `rtt`.

```
func RttEntry(
    addr :: Address,
    i :: integer) => RmmRttEntry
```

See also:

- A5.5 *Realm Translation Table*

B2.53 RttEntryFromDescriptor function

Converts a descriptor to an RmmRttEntry object.

```
func RttEntryFromDescriptor(
    desc :: bits(64)) => RmmRttEntry
```

B2.54 RttEntryIndex function

Returns the index of the entry in a level `level` RTT which is identified by `addr`.

```
func RttEntryIndex(
    addr :: Address,
    level :: integer) => integer
```

See also:

- A5.5 *Realm Translation Table*

B2.55 RttFold function

Returns the RTTE which results from folding the homogeneous RTT at address `rtt`.

```
func RttFold(
    rtt :: RmmRtt) => RmmRttEntry
```

See also:

- A5.5.6 *RTT folding*

B2.56 RttIsHomogeneous function

Returns TRUE if the RTT at address `rtt` is homogeneous.

```
func RttIsHomogeneous(
    rtt :: RmmRtt) => boolean
```

See also:

- A5.5.6 *RTT folding*

B2.57 RttIsLive function

Returns TRUE if the RTT at address `rtt` is live.

```
func RttIsLive(
    rtt :: RmmRtt) => boolean
```

See also:

- A5.5.8 *RTT liveness*
- A5.5.9 *RTT destruction*

B2.58 RttLevelIsBlockOrPage function

Returns TRUE if `level` is either a block or page RTT level for the Realm described by `rd`.

```
func RttLevelIsBlockOrPage(
    rd :: Address,
    level :: integer) => boolean
```

See also:

- A5.5 *Realm Translation Table*

B2.59 RttLevelIsStarting function

Returns TRUE if `level` is the starting level of the RTT for the Realm described by `rd`.

```
func RttLevelIsStarting(
    rd :: Address,
    level :: integer) => boolean
```

See also:

- A5.5 *Realm Translation Table*

B2.60 RttLevelIsValid function

Returns TRUE if `level` is a valid RTT level for the Realm described by `rd`.

```
func RttLevelIsValid(
    rd :: Address,
    level :: integer) => boolean
```

See also:

- A5.5 *Realm Translation Table*

B2.61 RttLevelSize function

Returns the size of the address space described by each entry in an RTT at level.

If `level` is invalid, the return value is UNKNOWN.

```
func RttLevelSize(
    level :: integer) => integer
```

See also:

- A5.5 *Realm Translation Table*

B2.62 RttsAllEntriesRipas function

Returns TRUE if the RIPAS of all entries in all of the starting-level RTT Granules is equal to `ripas`.

```
func RttsAllEntriesRipas(
    rtt_base :: Address,
    rtt_num_start :: integer,
    ripas :: RmmRipas) => boolean
    for i = 0 to rtt_num_start - 1 do
        var addr = (UInt(rtt_base) + i * RMM_GRANULE_SIZE)[63:0];
        var rtt = Rtt(addr);
        if !RttAllEntriesRipas(rtt, ripas) then
            return FALSE;
        end
    end
    return TRUE;
end
```

B2.63 RttsAllEntriesState function

Returns TRUE if the state of all entries in all of the starting-level RTT Granules is equal to `state`.

```
func RttsAllEntriesState(
    rtt_base :: Address,
    rtt_num_start :: integer,
    state :: RmmRttEntryState) => boolean
    for i = 0 to rtt_num_start - 1 do
        var addr = (UInt(rtt_base) + i * RMM_GRANULE_SIZE)[63:0];
        var rtt = Rtt(addr);
        if !RttAllEntriesState(rtt, state) then
            return FALSE;
        end
    end
    return TRUE;
end
```

B2.64 RttsGranuleState function

Inductive function which identifies the states of the starting-level RTT Granules.

This function is used in the definition of command footprint.

```
func RttsGranuleState(
    rtt_base :: Address,
    rtt_num_start :: integer)
```

B2.65 RttsStateEqual function

Returns TRUE if the state of all of the starting-level RTT Granules is equal to `state`.

```
func RttsStateEqual(
    rtt_base :: Address,
    rtt_num_start :: integer,
    state :: RmmGranuleState) => boolean
    for i = 0 to rtt_num_start - 1 do
        var addr = (UInt(rtt_base) + i * RMM_GRANULE_SIZE)[63:0];
        if (!PaIsDelegable(addr)
                || Granule(addr).state != state) then
            return FALSE;
        end
    end
    return TRUE;
end
```

B2.66 RttWalk function

Returns the result of an RTT walk from the RTT base of `rd` to address `addr`.

If `level` is provided, the walk terminates at `level`.

```
func RttWalk(
    rd :: Address,
    addr :: Address) => RmmRttWalkResult
```

```
func RttWalk(
    rd :: Address,
    addr :: Address,
    level :: integer) => RmmRttWalkResult
```

See also:

- A5.5.10 *RTT walk*

B2.67 ToAddress function

Convert integer to Address.

```
func ToAddress(value :: integer) => Address
    return value[63:0];
end
```

B2.68 VmidIsFree function

Returns TRUE if `vmid` is unused.

```
func VmidIsFree(
    vmid :: bits(16)) => boolean
```

B2.69 VmidIsValid function

Returns TRUE if `vmid` is valid on the platform.

```
func VmidIsValid(
    vmid :: bits(16)) => boolean
```

Chapter B3
Realm Management Interface

This chapter defines the interface used by the Host to manage Realms.

B3.1 RMI version

R_{NCFDX} This specification defines version 1.0 of the Realm Management Interface.

See also:

- B3.3.23 *RMI_VERSION command*

B3.2 RMI command return codes

I_{JQMBN} The return code of an RMI command is a tuple which contains *status* and *index* fields.

I_{YCHQV} The *status* field of an RMI command return code indicates whether the command

- succeeded, or
- failed, and the reason for the failure.

I_{PPNST} If an RMI command succeeds then the status of its return code is RMI_SUCCESS.

I_{MBVPG} The *index* field of an RMI command return code can provide additional information about the reason for a command failure. The meaning of the index field depends on the status, and is described by the following table.

Status	Description	Meaning of index
RMI_SUCCESS	Command completed successfully	None: index is zero.
RMI_ERROR_INPUT	The value of a command input value caused the command to fail	None: index is zero.
RMI_ERROR_IN_USE	An operation cannot be completed because a resource is in use	None: index is zero.
RMI_ERROR_REALM	An attribute of a Realm does not match the expected value	Varies between usages. See individual commands for details.
RMI_ERROR_REC	An attribute of a REC does not match the expected value	None: index is zero.
RMI_ERROR_RTT	An RTT walk terminated before reaching the target RTT level, or reached an RTTE with an unexpected value	RTT level at which the walk terminated.

I_{QQQNB} Multiple failure conditions in an RMI command may return the same error code - that is, the same status and index values.

R_{XRDYQ} If an input to an RMI command uses an invalid encoding then the command fails and returns RMI_ERROR_INPUT.

Command inputs include registers and in-memory data structures.

Invalid encodings include:

- setting a “must be zero” bit to `'1'`
- using a reserved encoding in an enumeration

See also:

- B3.4.1 *RmiCommandReturnCode type*

B3.3 RMI commands

The following table summarizes the FIDs of commands in the RMI interface.

FID	Command
0xC4000153	RMI_DATA_CREATE
0xC4000154	RMI_DATA_CREATE_UNKNOWN
0xC4000155	RMI_DATA_DESTROY
0xC4000165	RMI_FEATURES
0xC4000151	RMI_GRANULE_DELEGATE
0xC4000152	RMI_GRANULE_UNDELEGATE
0xC4000164	RMI_PSCI_COMPLETE
0xC4000157	RMI_REALM_ACTIVATE
0xC4000158	RMI_REALM_CREATE
0xC4000159	RMI_REALM_DESTROY
0xC4000167	RMI_REC_AUX_COUNT
0xC400015A	RMI_REC_CREATE
0xC400015B	RMI_REC_DESTROY
0xC400015C	RMI_REC_ENTER
0xC400015D	RMI_RTT_CREATE
0xC400015E	RMI_RTT_DESTROY
0xC4000166	RMI_RTT_FOLD
0xC4000168	RMI_RTT_INIT_RIPAS
0xC400015F	RMI_RTT_MAP_UNPROTECTED
0xC4000161	RMI_RTT_READ_ENTRY
0xC4000169	RMI_RTT_SET_RIPAS
0xC4000162	RMI_RTT_UNMAP_UNPROTECTED
0xC4000150	RMI_VERSION

B3.3.1 RMI_DATA_CREATE command

Creates a Data Granule, copying contents from a Non-secure Granule provided by the caller.

See also:

- Chapter A5 *Realm memory management*
- B3.3.3 *RMI_DATA_DESTROY command*
- D1.2.3 *Initialize memory of New Realm flow*

B3.3.1.1 Interface

B3.3.1.1.1 Input values

Name	Register	Bits	Type	Description
fid	X0	63:0	UInt64	FID, value `0xC4000153`
data	X1	63:0	Address	PA of the target Data
rd	X2	63:0	Address	PA of the RD for the target Realm
ipa	X3	63:0	Address	IPA at which the Granule will be mapped in the target Realm
src	X4	63:0	Address	PA of the source Granule
flags	X5	63:0	RmiDataFlags	Flags

B3.3.1.1.2 Context

The RMI_DATA_CREATE command operates on the following context.

Name	Type	Value	Before	Description
realm	RmmRealm	`Realm(rd)`	true	Realm

walk	RmmRttWalkResult	`RttWalk(`
		`  rd, ipa,`
		`  RMM_RTT_PAGE_LEVEL)`
			false	RTT walk result

entry_idx	UInt64	`RttEntryIndex(` `  ipa, walk.level)`	false	RTTE index

B3.3.1.1.3 Output values

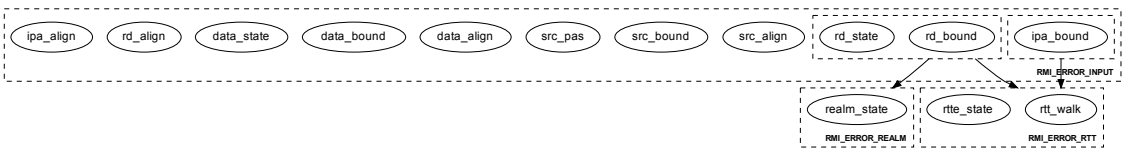
Name	Register	Bits	Type	Description
result	X0	63:0	RmiCommandReturnCode	Command return status

B3.3.1.2 Failure conditions

ID	Condition
src_align	pre: !AddrIsGranuleAligned(src) post: ResultEqual(result, RMI_ERROR_INPUT)
src_bound	pre: !PaIsDelegable(src) post: ResultEqual(result, RMI_ERROR_INPUT)
src_pas	pre: Granule(src).pas != NS post: ResultEqual(result, RMI_ERROR_INPUT)
data_align	pre: !AddrIsGranuleAligned(data) post: ResultEqual(result, RMI_ERROR_INPUT)
data_bound	pre: !PaIsDelegable(data) post: ResultEqual(result, RMI_ERROR_INPUT)
data_state	pre: Granule(data).state != DELEGATED post: ResultEqual(result, RMI_ERROR_INPUT)
rd_align	pre: !AddrIsGranuleAligned(rd) post: ResultEqual(result, RMI_ERROR_INPUT)
rd_bound	pre: !PaIsDelegable(rd) post: ResultEqual(result, RMI_ERROR_INPUT)
rd_state	pre: Granule(rd).state != RD post: ResultEqual(result, RMI_ERROR_INPUT)
ipa_align	pre: !AddrIsGranuleAligned(ipa) post: ResultEqual(result, RMI_ERROR_INPUT)
ipa_bound	pre: !AddrIsProtected(ipa, realm) post: ResultEqual(result, RMI_ERROR_INPUT)
realm_state	pre: realm.state != NEW post: ResultEqual(result, RMI_ERROR_REALM)
rtt_walk	pre: walk.level < RMM_RTT_PAGE_LEVEL post: ResultEqual(result, RMI_ERROR_RTT, walk.level)
rtte_state	pre: walk.entry.state != UNASSIGNED post: ResultEqual(result, RMI_ERROR_RTT, walk.level)

B3.3.1.2.1 Failure condition ordering

```
[rd_bound, rd_state] < [realm_state]
[rd_bound, rd_state] < [rtt_walk, rtte_state]
[ipa_bound] < [rtt_walk, rtte_state]
```



B3.3.1.3 Success conditions

ID	Condition
data_state	`Granule(data).state == DATA`
data_content	`Contents of `data` Granule are equal to the contents of `src` Granule.`
rtte_state	`walk.entry.state == ASSIGNED`
rtte_addr	`walk.entry.addr == data`
rim	`Realm(rd).measurements[0] == RimExtendData( realm, ipa, data, flags)`

B3.3.1.4 RMI_DATA_CREATE extension of RIM

On successful execution of RMI_DATA_CREATE, the new RIM value of the target Realm is calculated by the RMM as follows:

1. If flags.measure == RMI_MEASURE_CONTENT then using the RHA of the target Realm, compute the hash of the contents of the DATA Granule.
2. Allocate an RmmMeasurementDescriptorData data structure.
3. Populate the measurement descriptor:

- Set the desc_type field to the descriptor type.
- Set the len field to the descriptor length.
- Set the rim field to the current RIM value of the target Realm.
- Set the ipa field to the IPA at which the DATA Granule is mapped in the target Realm.
- Set the flags field to the flags provided by the Host.
- If flags.measure == RMI_MEASURE_CONTENT then set the content field to the hash of the contents of the DATA Granule. Otherwise, set the content field to zero.

4. Using the RHA of the target Realm, compute the hash of the measurement descriptor. Set the RIM of the target Realm to this value, zero filling upper bytes if the RHA output is smaller than the size of the RIM.

See also:

- A7.1.1 *Realm Initial Measurement*
- B2.35 *RimExtendData function*
- C1.5 *RmmMeasurementDescriptorData type*

B3.3.1.5 Footprint

ID	Value
data_state	`Granule(data).state`
rim	`Realm(rd).measurements[0]`
rtte	`RttEntry(walk.rtt_addr, entry_idx)`

B3.3.2 RMI_DATA_CREATE_UNKNOWN command

Creates a Data Granule with unknown contents.

See also:

- A2.2.4 *Granule wiping*
- Chapter A5 *Realm memory management*
- B3.3.3 *RMI_DATA_DESTROY command*
- D1.5.1 *Add memory to Active Realm flow*

B3.3.2.1 Interface

B3.3.2.1.1 Input values

Name	Register	Bits	Type	Description
fid	X0	63:0	UInt64	FID, value `0xC4000154`
data	X1	63:0	Address	PA of the target Data
rd	X2	63:0	Address	PA of the RD for the target Realm
ipa	X3	63:0	Address	IPA at which the Granule will be mapped in the target Realm

B3.3.2.1.2 Context

The RMI_DATA_CREATE_UNKNOWN command operates on the following context.

Name	Type	Value	Before	Description

walk	RmmRttWalkResult	`RttWalk(`
		`  rd, ipa,`
		`  RMM_RTT_PAGE_LEVEL)`
			false	RTT walk result

entry_idx	UInt64	`RttEntryIndex(` `  ipa, walk.level)`	false	RTTE index

B3.3.2.1.3 Output values

Name	Register	Bits	Type	Description
result	X0	63:0	RmiCommandReturnCode	Command return status

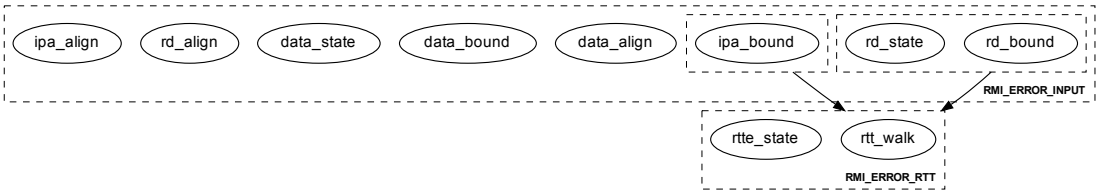
B3.3.2.2 Failure conditions

ID	Condition
data_align	`pre:  !AddrIsGranuleAligned(data)` `post: ResultEqual(result, RMI_ERROR_INPUT)`
data_bound	`pre:  !PaIsDelegable(data)` `post: ResultEqual(result, RMI_ERROR_INPUT)`

ID	Condition
data_state	pre: Granule(data).state != DELEGATED post: ResultEqual(result, RMI_ERROR_INPUT)
rd_align	pre: !AddrIsGranuleAligned(rd) post: ResultEqual(result, RMI_ERROR_INPUT)
rd_bound	pre: !PaIsDelegable(rd) post: ResultEqual(result, RMI_ERROR_INPUT)
rd_state	pre: Granule(rd).state != RD post: ResultEqual(result, RMI_ERROR_INPUT)
ipa_align	pre: !AddrIsGranuleAligned(ipa) post: ResultEqual(result, RMI_ERROR_INPUT)
ipa_bound	pre: !AddrIsProtected(ipa, Realm(rd)) post: ResultEqual(result, RMI_ERROR_INPUT)
rtt_walk	pre: walk.level < RMM_RTT_PAGE_LEVEL post: ResultEqual(result, RMI_ERROR_RTT, walk.level)
rtte_state	pre: walk.entry.state != UNASSIGNED post: ResultEqual(result, RMI_ERROR_RTT, walk.level)

B3.3.2.2.1 Failure condition ordering

```
[rd_bound, rd_state] < [rtt_walk, rtte_state]
[ipa_bound] < [rtt_walk, rtte_state]
```



B3.3.2.3 Success conditions

ID	Condition
data_state	Granule(data).state == DATA
data_content	Contents of target Granule are wiped.
rtte_state	walk.entry.state == ASSIGNED
rtte_addr	walk.entry.addr == data

B3.3.2.4 Footprint

ID	Value
data_state	`Granule(data).state`
rtte	`RttEntry(walk.rtt_addr, entry_idx)`

B3.3.3 RMI_DATA_DESTROY command

Destroys a Data Granule.

See also:

- Chapter A5 *Realm memory management*
- B3.3.1 *RMI_DATA_CREATE command*
- B3.3.2 *RMI_DATA_CREATE_UNKNOWN command*
- D1.2.5 *Realm destruction flow*

B3.3.3.1 Interface

B3.3.3.1.1 Input values

Name	Register	Bits	Type	Description
fid	X0	63:0	UInt64	FID, value `0xC4000155`
rd	X1	63:0	Address	PA of the RD which owns the target Data
ipa	X2	63:0	Address	IPA at which the Granule is mapped in the target Realm

B3.3.3.1.2 Context

The RMI_DATA_DESTROY command operates on the following context.

Name	Type	Value	Before	Description

walk	RmmRttWalkResult	`RttWalk(`
		`  rd, ipa,`
		`  RMM_RTT_PAGE_LEVEL)`
			false	RTT walk result

entry_idx	UInt64	`RttEntryIndex(` `  ipa, walk.level)`	false	RTTE index

data	Address	`RttWalk(rd, ipa,`
		`  RMM_RTT_PAGE_LEVEL`
		`).entry.addr`
			true	PA of the target Data

B3.3.3.1.3 Output values

Name	Register	Bits	Type	Description
result	X0	63:0	RmiCommandReturnCode	Command return status

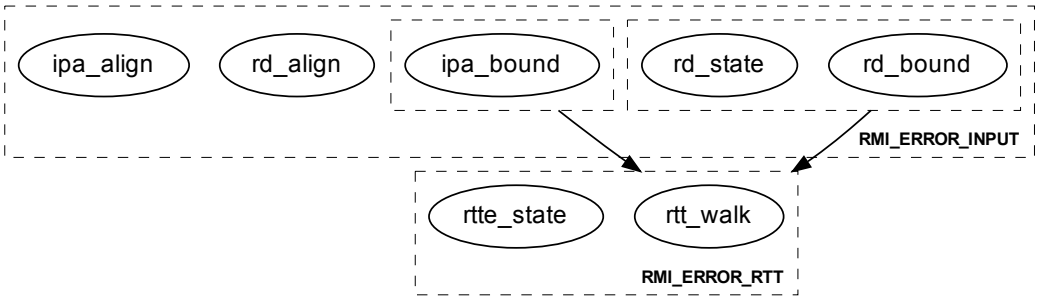
B3.3.3.2 Failure conditions

ID	Condition
rd_align	`pre:  !AddrIsGranuleAligned(rd)` `post: ResultEqual(result, RMI_ERROR_INPUT)`

ID	Condition
rd_bound	pre: !PaIsDelegable(rd) post: ResultEqual(result, RMI_ERROR_INPUT)
rd_state	pre: Granule(rd).state != RD post: ResultEqual(result, RMI_ERROR_INPUT)
ipa_align	pre: !AddrIsGranuleAligned(ipa) post: ResultEqual(result, RMI_ERROR_INPUT)
ipa_bound	pre: !AddrIsProtected(ipa, Realm(rd)) post: ResultEqual(result, RMI_ERROR_INPUT)
rtt_walk	pre: walk.level < RMM_RTT_PAGE_LEVEL post: ResultEqual(result, RMI_ERROR_RTT, walk.level)
rtte_state	pre: walk.entry.state != ASSIGNED post: ResultEqual(result, RMI_ERROR_RTT, walk.level)

B3.3.3.2.1 Failure condition ordering

```
[rd_bound, rd_state] < [rtt_walk, rtte_state]
[ipa_bound] < [rtt_walk, rtte_state]
```



B3.3.3.3 Success conditions

ID	Condition
data_state	Granule(data).state == DELEGATED
ripas_empty	pre: walk.entry.ripas == EMPTY post: walk.entry.state == UNASSIGNED
ripas_ram	pre: walk.entry.ripas == RAM post: walk.entry.state == DESTROYED

B3.3.3.4 Footprint

ID	Value
data_state	`Granule(data).state`
rtte	`RttEntry(walk.rtt_addr, entry_idx)`

B3.3.4 RMI_FEATURES command

Read feature register.

The following table indicates which feature register is returned depending on the index provided.

Index	Feature register
0	Feature register 0

See also:

- A3.1 *Realm feature discovery and selection*

B3.3.4.1 Interface

B3.3.4.1.1 Input values

Name	Register	Bits	Type	Description
fid	X0	63:0	UInt64	FID, value `0xC4000165`
index	X1	63:0	UInt64	Feature register index

B3.3.4.1.2 Output values

Name	Register	Bits	Type	Description
result	X0	63:0	RmiCommandReturnCode	Command return status
value	X1	63:0	Bits64	Feature register value

B3.3.4.2 Failure conditions

The RMI_FEATURES command does not have any failure conditions.

B3.3.4.3 Success conditions

ID	Condition
index	`pre:  index != 0` `post: X1 == Zeros()`

B3.3.4.4 Footprint

The RMI_FEATURES command does not have any footprint.

B3.3.5 RMI_GRANULE_DELEGATE command

Delegates a Granule.

See also:

- A2.2 *Granule*
- B3.3.6 *RMI_GRANULE_UNDELEGATE command*
- D1.2.1 *Realm creation flow*

B3.3.5.1 Interface

B3.3.5.1.1 Input values

Name	Register	Bits	Type	Description
fid	X0	63:0	UInt64	FID, value `0xC4000151`
addr	X1	63:0	Address	PA of the target Granule

B3.3.5.1.2 Output values

Name	Register	Bits	Type	Description
result	X0	63:0	RmiCommandReturnCode	Command return status

B3.3.5.2 Failure conditions

ID	Condition
gran_align	`pre: !AddrIsGranuleAligned(addr)` `post: ResultEqual(result, RMI_ERROR_INPUT)`
gran_bound	`pre: !PaIsDelegable(addr)` `post: ResultEqual(result, RMI_ERROR_INPUT)`
gran_state	`pre: Granule(addr).state != UNDELEGATED` `post: ResultEqual(result, RMI_ERROR_INPUT)`
gran_pas	`pre: Granule(addr).pas != NS` `post: ResultEqual(result, RMI_ERROR_INPUT)`

B3.3.5.2.1 Failure condition ordering

The RMI_GRANULE_DELEGATE command does not have any failure condition orderings.

B3.3.5.3 Success conditions

ID	Condition
gran_state	`Granule(addr).state == DELEGATED`
gran_pas	`Granule(addr).pas == REALM`

B3.3.5.4 Footprint

ID	Value
gran_pas	`Granule(addr).pas`
gran_state	`Granule(addr).state`

B3.3.6 RMI_GRANULE_UNDELEGATE command

Undelegates a Granule.

See also:

- A2.2 *Granule*
- B3.3.5 *RMI_GRANULE_DELEGATE command*
- D1.2.5 *Realm destruction flow*

B3.3.6.1 Interface

B3.3.6.1.1 Input values

Name	Register	Bits	Type	Description
fid	X0	63:0	UInt64	FID, value `0xC4000152`
addr	X1	63:0	Address	PA of the target Granule

B3.3.6.1.2 Output values

Name	Register	Bits	Type	Description
result	X0	63:0	RmiCommandReturnCode	Command return status

B3.3.6.2 Failure conditions

ID	Condition
gran_align	`pre: !AddrIsGranuleAligned(addr)` `post: ResultEqual(result, RMI_ERROR_INPUT)`
gran_bound	`pre: !PaIsDelegable(addr)` `post: ResultEqual(result, RMI_ERROR_INPUT)`
gran_state	`pre: Granule(addr).state != DELEGATED` `post: ResultEqual(result, RMI_ERROR_INPUT)`

B3.3.6.2.1 Failure condition ordering

The RMI_GRANULE_UNDELEGATE command does not have any failure condition orderings.

B3.3.6.3 Success conditions

ID	Condition
gran_pas	`Granule(addr).pas == NS`
gran_state	`Granule(addr).state == UNDELEGATED`
gran_content	`Contents of target Granule are wiped.`

See also:

- A2.2.4 *Granule wiping*

B3.3.6.4 Footprint

ID	Value
gran_pas	`Granule(addr).pas`
gran_state	`Granule(addr).state`

B3.3.7 RMI_PSCI_COMPLETE command

Completes a pending PSCI command which was called with an MPIDR argument, by providing the corresponding REC.

See also:

- A4.3.7 *REC exit due to PSCI*
- B5.3.1 *PSCI_AFFINITY_INFO command*
- B5.3.3 *PSCI_CPU_ON command*
- D1.4 *PSCI flows*

B3.3.7.1 Interface

B3.3.7.1.1 Input values

Name	Register	Bits	Type	Description
fid	X0	63:0	UInt64	FID, value `0xC4000164`
calling_rec	X1	63:0	Address	PA of the calling REC
target_rec	X2	63:0	Address	PA of the target REC

B3.3.7.1.2 Output values

Name	Register	Bits	Type	Description
result	X0	63:0	RmiCommandReturnCode	Command return status

B3.3.7.2 Failure conditions

ID	Condition
alias	`pre: calling_rec == target_rec` `post: ResultEqual(result, RMI_ERROR_INPUT)`
calling_align	`pre: !AddrIsGranuleAligned(calling_rec)` `post: ResultEqual(result, RMI_ERROR_INPUT)`
calling_bound	`pre: !PaIsDelegable(calling_rec)` `post: ResultEqual(result, RMI_ERROR_INPUT)`
calling_state	`pre: Granule(calling_rec).state != REC` `post: ResultEqual(result, RMI_ERROR_INPUT)`
target_align	`pre: !AddrIsGranuleAligned(target_rec)` `post: ResultEqual(result, RMI_ERROR_INPUT)`
target_bound	`pre: !PaIsDelegable(target_rec)` `post: ResultEqual(result, RMI_ERROR_INPUT)`
target_state	`pre: Granule(target_rec).state != REC` `post: ResultEqual(result, RMI_ERROR_INPUT)`
pending	`pre: Rec(calling_rec).psci_pending != PSCI_REQUEST_PENDING` `post: ResultEqual(result, RMI_ERROR_INPUT)`

ID	Condition
owner	`pre: Rec(target_rec).owner != Rec(calling_rec).owner` `post: ResultEqual(result, RMI_ERROR_INPUT)`
target	`pre: Rec(target_rec).owner != Rec(calling_rec).gprs[1]` `post: ResultEqual(result, RMI_ERROR_INPUT)`

B3.3.7.2.1 Failure condition ordering

The RMI_PSCI_COMPLETE command does not have any failure condition orderings.

B3.3.7.3 Success conditions

ID	Condition
pending	`Rec(calling_rec).psci_pending == NO_PSCI_REQUEST_PENDING`

on_already	`pre: (Rec(calling_rec).gprs[0] == FID_PSCI_CPU_ON`
	`&& Rec(target_rec).flags.runnable == RUNNABLE)`
	`post: (Rec(calling_rec).gprs[0] ==`
	`PsciReturnCodeEncode(PSCI_ALREADY_ON))`

ID	Condition
on_success	pre: (Rec(calling_rec).gprs[0] == FID_PSCI_CPU_ON && Rec(target_rec).flags.runnable != RUNNABLE) post: (Rec(target_rec).gprs[0] == Rec(calling_rec).gprs[2] && Rec(target_rec).gprs[1] == Zeros() && Rec(target_rec).gprs[2] == Zeros() && Rec(target_rec).gprs[3] == Zeros() && Rec(target_rec).gprs[4] == Zeros() && Rec(target_rec).gprs[5] == Zeros() && Rec(target_rec).gprs[6] == Zeros() && Rec(target_rec).gprs[7] == Zeros() && Rec(target_rec).gprs[8] == Zeros() && Rec(target_rec).gprs[9] == Zeros() && Rec(target_rec).gprs[10] == Zeros() && Rec(target_rec).gprs[11] == Zeros() && Rec(target_rec).gprs[12] == Zeros() && Rec(target_rec).gprs[13] == Zeros() && Rec(target_rec).gprs[14] == Zeros() && Rec(target_rec).gprs[15] == Zeros() && Rec(target_rec).gprs[16] == Zeros() && Rec(target_rec).gprs[17] == Zeros() && Rec(target_rec).gprs[18] == Zeros() && Rec(target_rec).gprs[19] == Zeros() && Rec(target_rec).gprs[20] == Zeros() && Rec(target_rec).gprs[21] == Zeros() && Rec(target_rec).gprs[22] == Zeros() && Rec(target_rec).gprs[23] == Zeros() && Rec(target_rec).gprs[24] == Zeros() && Rec(target_rec).gprs[25] == Zeros() && Rec(target_rec).gprs[26] == Zeros() && Rec(target_rec).gprs[27] == Zeros() && Rec(target_rec).gprs[28] == Zeros() && Rec(target_rec).gprs[29] == Zeros() && Rec(target_rec).gprs[30] == Zeros() && Rec(target_rec).gprs[31] == Zeros() && Rec(target_rec).pc == Rec(calling_rec).gprs[2] && Rec(target_rec).flags.runnable == RUNNABLE && Rec(calling_rec).gprs[0] == PsciReturnCodeEncode(PSCI_SUCCESS))
affinity_on	pre: (Rec(calling_rec).gprs[0] == FID_PSCI_AFFINITY_INFO && Rec(target_rec).flags.runnable == RUNNABLE) post: (Rec(calling_rec).gprs[0] == PsciReturnCodeEncode(PSCI_SUCCESS))
affinity_off	pre: (Rec(calling_rec).gprs[0] == FID_PSCI_AFFINITY_INFO && Rec(target_rec).flags.runnable != RUNNABLE) post: (Rec(calling_rec).gprs[0] == PsciReturnCodeEncode(PSCI_OFF))
gprs	(Rec(calling_rec).gprs[1] == Zeros() && Rec(calling_rec).gprs[2] == Zeros() && Rec(calling_rec).gprs[3] == Zeros())

B3.3.7.4 Footprint

ID	Value
target_flags	`Rec(target_rec).flags`
target_gprs	`Rec(target_rec).gprs`
target_pc	`Rec(target_rec).pc`
calling_pend	`Rec(calling_rec).psci_pending`
calling_gprs	`Rec(calling_rec).gprs`

B3.3.8 RMI_REALM_ACTIVATE command

Activates a Realm.

See also:

- A2.1 *Realm*

B3.3.8.1 Interface

B3.3.8.1.1 Input values

Name	Register	Bits	Type	Description
fid	X0	63:0	UInt64	FID, value `0xC4000157`
rd	X1	63:0	Address	PA of the RD

B3.3.8.1.2 Output values

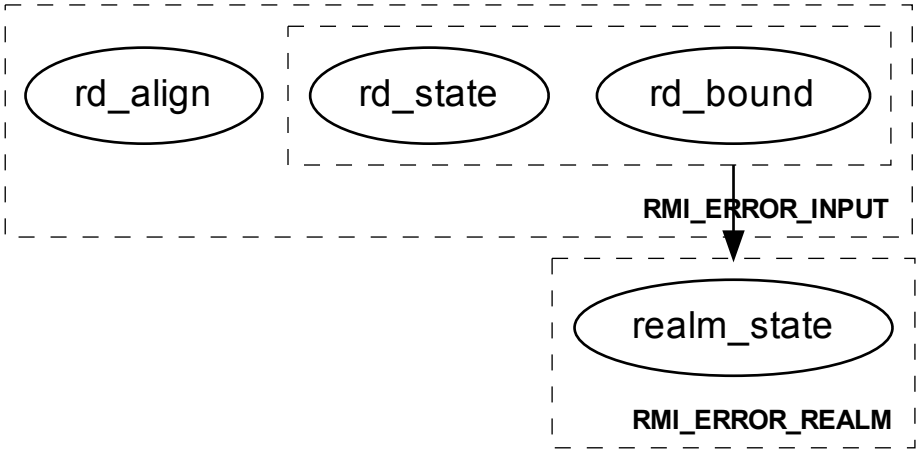
Name	Register	Bits	Type	Description
result	X0	63:0	RmiCommandReturnCode	Command return status

B3.3.8.2 Failure conditions

ID	Condition
rd_align	`pre: !AddrIsGranuleAligned(rd)` `post: ResultEqual(result, RMI_ERROR_INPUT)`
rd_bound	`pre: !PaIsDelegable(rd)` `post: ResultEqual(result, RMI_ERROR_INPUT)`
rd_state	`pre: Granule(rd).state != RD` `post: ResultEqual(result, RMI_ERROR_INPUT)`
realm_state	`pre: Realm(rd).state != NEW` `post: ResultEqual(result, RMI_ERROR_REALM)`

B3.3.8.2.1 Failure condition ordering

```
[rd_bound, rd_state] < [realm_state]
```



B3.3.8.3 Success conditions

ID	Condition
realm_state	`Realm(rd).state == ACTIVE`

B3.3.8.4 Footprint

ID	Value
realm_state	`Realm(rd).state`

B3.3.9 RMI_REALM_CREATE command

Creates a Realm.

See also:

- A2.1 *Realm*
- A2.1.6 *Realm parameters*
- B3.3.10 *RMI_REALM_DESTROY command*
- D1.2.1 *Realm creation flow*

B3.3.9.1 Interface

B3.3.9.1.1 Input values

Name	Register	Bits	Type	Description
fid	X0	63:0	UInt64	FID, value `0xC4000158`
rd	X1	63:0	Address	PA of the RD
params_ptr	X2	63:0	Address	PA of Realm parameters

B3.3.9.1.2 Context

The RMI_REALM_CREATE command operates on the following context.

Name	Type	Value	Before	Description
params	RmiRealmParams	`RealmParams(params_ptr)`	false	Realm parameters

B3.3.9.1.3 Output values

Name	Register	Bits	Type	Description
result	X0	63:0	RmiCommandReturnCode	Command return status

B3.3.9.2 Failure conditions

ID	Condition
params_align	`pre: !AddrIsGranuleAligned(params_ptr)` `post: ResultEqual(result, RMI_ERROR_INPUT)`
params_bound	`pre: !PaIsDelegable(params_ptr)` `post: ResultEqual(result, RMI_ERROR_INPUT)`
params_pas	`pre: Granule(params_ptr).pas != NS` `post: ResultEqual(result, RMI_ERROR_INPUT)`

ID	Condition
hash_valid	pre: !RmiHashAlgorithmIsValid(ReadMemory(params_ptr, __OFFSETOF(RmiRealmParams, hash_algo), __SIZEOF(RmiRealmParams, hash_algo))) post: ResultEqual(result, RMI_ERROR_INPUT)
hash_supp	pre: !RmiHashAlgorithmIsSupported(params.hash_algo) post: ResultEqual(result, RMI_ERROR_INPUT)
alias	pre: AddrInRange(rd, params.rtt_base, (params.rtt_num_start - 1) * RMM_GRANULE_SIZE) post: ResultEqual(result, RMI_ERROR_INPUT)
rd_align	pre: !AddrIsGranuleAligned(rd) post: ResultEqual(result, RMI_ERROR_INPUT)
rd_bound	pre: !PaIsDelegable(rd) post: ResultEqual(result, RMI_ERROR_INPUT)
rd_state	pre: Granule(rd).state != DELEGATED post: ResultEqual(result, RMI_ERROR_INPUT)
rtt_align	pre: !AddrIsAligned(params.rtt_base, params.rtt_num_start * RMM_GRANULE_SIZE) post: ResultEqual(result, RMI_ERROR_INPUT)
feat_valid	pre: !RmiFeatureRegister0IsValid(ReadMemory(params_ptr, __OFFSETOF(RmiRealmParams, features_0), __SIZEOF(RmiRealmParams, features_0))) post: ResultEqual(result, RMI_ERROR_INPUT)
rtt_num_level	pre: !RttConfigIsValid(params.features_0.S2SZ, params.rtt_level_start, params.rtt_num_start) post: ResultEqual(result, RMI_ERROR_INPUT)
rtt_state	pre: !RttsStateEqual(params.rtt_base, params.rtt_num_start, DELEGATED) post: ResultEqual(result, RMI_ERROR_INPUT)
vmid_valid	pre: !VmidIsValid(params.vmid) || !VmidIsFree(params.vmid) post: ResultEqual(result, RMI_ERROR_INPUT)

B3.3.9.2.1 Failure condition ordering

The RMI_REALM_CREATE command does not have any failure condition orderings.

B3.3.9.3 Success conditions

ID	Condition
rd_state	Granule(rd).state == RD
realm_state	Realm(rd).state == NEW

ID	Condition
rec_index	`Realm(rd).rec_index == 0`
rtt_base	`Realm(rd).rtt_base == params.rtt_base`
rtt_state	`RttsStateEqual(` `  Realm(rd).rtt_base, Realm(rd).rtt_num_start, RTT)`

rtte_states	`RttsAllEntriesState(`
	`  Realm(rd).rtt_base, Realm(rd).rtt_num_start,`
	`  UNASSIGNED)`

rtte_ripas	`RttsAllEntriesRipas(`
	`  Realm(rd).rtt_base, Realm(rd).rtt_num_start,`
	`  EMPTY)`

ipa_width	`Realm(rd).ipa_width == params.features_0.S2SZ`
hash_algo	`Equal(Realm(rd).hash_algo, params.hash_algo)`
rim	`Realm(rd).measurements[0] == RimInit(` `  Realm(rd).hash_algo, params)`

rem	`(Realm(rd).measurements[1] == Zeros()`
	`  && Realm(rd).measurements[2] == Zeros()`
	`  && Realm(rd).measurements[3] == Zeros()`
	`  && Realm(rd).measurements[4] == Zeros())`

rtt_level	`Realm(rd).rtt_level_start == params.rtt_level_start`
rtt_num	`Realm(rd).rtt_num_start == params.rtt_num_start`
vmid	`Realm(rd).vmid == params.vmid`
rpv	`Realm(rd).rpv == params.rpv`

B3.3.9.4 RMI_REALM_CREATE initialization of RIM

On successful execution of RMI_REALM_CREATE, the initial RIM value of the target Realm is calculated by the RMM as follows:

1. Allocate a zero-filled RmiRealmParams data structure to hold the measured Realm parameters.
2. Copy the following attributes from the Host-provided RmiRealmParams data structure into the measured Realm parameters data structure:
   - hash_algo
   - features_0
3. Using the RHA of the target Realm, compute the hash of the measured Realm parameters data structure. Set the RIM of the target Realm to this value, zero filling upper bytes if the RHA output is smaller than the size of the RIM.

See also:

- A7.1.1 *Realm Initial Measurement*
- B2.38 *RimInit function*
- B3.4.10 *RmiRealmParams type*

B3.3.9.5 Footprint

ID	Value
rd_state	`Granule(rd).state`
rtt_state	`RttsGranuleState( Realm(rd).rtt_base, Realm(rd).rtt_num_start)`

B3.3.10 RMI_REALM_DESTROY command

Destroys a Realm.

See also:

- A2.1 *Realm*
- B3.3.9 *RMI_REALM_CREATE command*
- D1.2.5 *Realm destruction flow*

B3.3.10.1 Interface

B3.3.10.1.1 Input values

Name	Register	Bits	Type	Description
fid	X0	63:0	UInt64	FID, value `0xC4000159`
rd	X1	63:0	Address	PA of the RD

B3.3.10.1.2 Context

The RMI_REALM_DESTROY command operates on the following context.

Name	Type	Value	Before	Description
realm	RmmRealm	`Realm(rd)`	true	Realm

B3.3.10.1.3 Output values

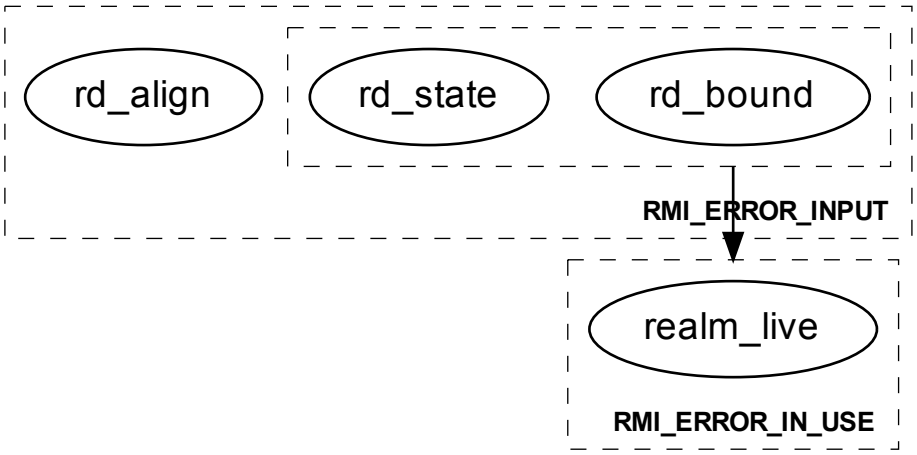
Name	Register	Bits	Type	Description
result	X0	63:0	RmiCommandReturnCode	Command return status

B3.3.10.2 Failure conditions

ID	Condition
rd_align	`pre: !AddrIsGranuleAligned(rd)` `post: ResultEqual(result, RMI_ERROR_INPUT)`
rd_bound	`pre: !PaIsDelegable(rd)` `post: ResultEqual(result, RMI_ERROR_INPUT)`
rd_state	`pre: Granule(rd).state != RD` `post: ResultEqual(result, RMI_ERROR_INPUT)`
realm_live	`pre: RealmIsLive(rd)` `post: ResultEqual(result, RMI_ERROR_IN_USE)`

B3.3.10.2.1 Failure condition ordering

```
[rd_bound, rd_state] < [realm_live]
```



B3.3.10.3 Success conditions

ID	Condition
rtt_state	`RttsStateEqual(realm.rtt_base, realm.rtt_num_start, DELEGATED)`
rd_state	`Granule(rd).state == DELEGATED`
vmid	`VmidIsFree(realm.vmid)`

B3.3.10.4 Footprint

ID	Value
rd_state	`Granule(rd).state`
rtt_state	`RttsGranuleState(realm.rtt_base, realm.rtt_num_start)`

B3.3.11 RMI_REC_AUX_COUNT command

Get number of auxiliary Granules required for a REC.

See also:

- A2.3 *Realm Execution Context*
- B3.3.12 *RMI_REC_CREATE command*
- B3.4.17 *RmiRecParams type*
- D1.2.4 *REC creation flow*

B3.3.11.1 Interface

B3.3.11.1.1 Input values

Name	Register	Bits	Type	Description
fid	X0	63:0	UInt64	FID, value `0xC4000167`
rd	X1	63:0	Address	PA of the RD for the target Realm

B3.3.11.1.2 Output values

Name	Register	Bits	Type	Description
result	X0	63:0	RmiCommandReturnCode	Command return status
aux_count	X1	63:0	UInt64	Number of auxiliary Granules required for a REC

B3.3.11.2 Failure conditions

ID	Condition
rd_align	`pre: !AddrIsGranuleAligned(rd)` `post: ResultEqual(result, RMI_ERROR_INPUT)`
rd_bound	`pre: !PaIsDelegable(rd)` `post: ResultEqual(result, RMI_ERROR_INPUT)`
rd_state	`pre: Granule(rd).state != RD` `post: ResultEqual(result, RMI_ERROR_INPUT)`

B3.3.11.2.1 Failure condition ordering

The RMI_REC_AUX_COUNT command does not have any failure condition orderings.

B3.3.11.3 Success conditions

ID	Condition
aux_count	`aux_count == RecAuxCount(rd)`

B3.3.11.4 Footprint

The RMI_REC_AUX_COUNT command does not have any footprint.

B3.3.12 RMI_REC_CREATE command

Creates a REC.

See also:

- A2.3 *Realm Execution Context*
- A2.3.3 *REC index and MPIDR value*
- B3.3.11 *RMI_REC_AUX_COUNT command*
- B3.3.13 *RMI_REC_DESTROY command*
- D1.2.4 *REC creation flow*

B3.3.12.1 Interface

B3.3.12.1.1 Input values

Name	Register	Bits	Type	Description
fid	X0	63:0	UInt64	FID, value `0xC400015A`
rec	X1	63:0	Address	PA of the target REC
rd	X2	63:0	Address	PA of the RD for the target Realm
params_ptr	X3	63:0	Address	PA of REC parameters

B3.3.12.1.2 Context

The RMI_REC_CREATE command operates on the following context.

Name	Type	Value	Before	Description
realm	RmmRealm	`Realm(rd)`	true	Realm
params	RmiRecParams	`RecParams(params_ptr)`	false	REC parameters
rec_index	UInt64	`Realm(rd).rec_index`	true	REC index

B3.3.12.1.3 Output values

Name	Register	Bits	Type	Description
result	X0	63:0	RmiCommandReturnCode	Command return status

B3.3.12.2 Failure conditions

ID	Condition
params_align	`pre: !AddrIsGranuleAligned(params_ptr)` `post: ResultEqual(result, RMI_ERROR_INPUT)`
params_bound	`pre: !PaIsDelegable(params_ptr)` `post: ResultEqual(result, RMI_ERROR_INPUT)`

ID	Condition
params_pas	pre: Granule(params_ptr).pas != NS post: ResultEqual(result, RMI_ERROR_INPUT)
rec_align	pre: !AddrIsGranuleAligned(rec) post: ResultEqual(result, RMI_ERROR_INPUT)
rec_bound	pre: !PaIsDelegable(rec) post: ResultEqual(result, RMI_ERROR_INPUT)
rec_state	pre: Granule(rec).state != DELEGATED post: ResultEqual(result, RMI_ERROR_INPUT)
rd_align	pre: !AddrIsGranuleAligned(rd) post: ResultEqual(result, RMI_ERROR_INPUT)
rd_bound	pre: !PaIsDelegable(rd) post: ResultEqual(result, RMI_ERROR_INPUT)
rd_state	pre: Granule(rd).state != RD post: ResultEqual(result, RMI_ERROR_INPUT)
realm_state	pre: realm.state != NEW post: ResultEqual(result, RMI_ERROR_REALM)
mpidr_valid	pre: !RmiRecMpidrIsValid(ReadMemory(params_ptr, __OFFSETOF(RmiRecParams, mpidr), __SIZEOF(RmiRecParams, mpidr))) post: ResultEqual(result, RMI_ERROR_INPUT)
mpidr_index	pre: RecIndex(params.mpidr) != realm.rec_index post: ResultEqual(result, RMI_ERROR_INPUT)
num_aux	pre: params.num_aux != RecAuxCount(rd) post: ResultEqual(result, RMI_ERROR_INPUT)
aux_align	pre: !RecAuxAligned(params.aux, params.num_aux) post: ResultEqual(result, RMI_ERROR_INPUT)
aux_alias	pre: RecAuxAlias(rec, params.aux, params.num_aux) post: ResultEqual(result, RMI_ERROR_INPUT)
aux_state	pre: !RecAuxStateEqual(params.aux, params.num_aux, DELEGATED) post: ResultEqual(result, RMI_ERROR_INPUT)
flags_valid	pre: !RmiRecCreateFlagsIsValid(ReadMemory(params_ptr, __OFFSETOF(RmiRecParams, flags), __SIZEOF(RmiRecParams, flags))) post: ResultEqual(result, RMI_ERROR_INPUT)

B3.3.12.2.1 Failure condition ordering

```
[rd_bound, rd_state] < [realm_state]
```

B3.3.12.3 Success conditions

ID	Condition
rec_index	`Realm(rd).rec_index == rec_index + 1`
rec_gran_state	`Granule(rec).state == REC`
rec_owner	`Rec(rec).owner == rd`
rec_attest	`Rec(rec).attest_state == NO_ATTEST_IN_PROGRESS`
rec_mpidr	`MpidrEqual(Rec(rec).mpidr, params.mpidr)`
rec_state	`Rec(rec).state == READY`
runnable	`pre: params.flags.runnable == RMI_RUNNABLE` `post: Rec(rec).flags.runnable == RUNNABLE`
not_runnable	`pre: params.flags.runnable == RMI_NOT_RUNNABLE` `post: Rec(rec).flags.runnable == NOT_RUNNABLE`

rec_gprs	`(Rec(rec).gprs[0] == params.gprs[0]`
	`&& Rec(rec).gprs[1] == params.gprs[1]`
	`&& Rec(rec).gprs[2] == params.gprs[2]`
	`&& Rec(rec).gprs[3] == params.gprs[3]`
	`&& Rec(rec).gprs[4] == params.gprs[4]`
	`&& Rec(rec).gprs[5] == params.gprs[5]`
	`&& Rec(rec).gprs[6] == params.gprs[6]`
	`&& Rec(rec).gprs[7] == params.gprs[7]`
	`&& Rec(rec).gprs[8] == Zeros()`
	`&& Rec(rec).gprs[9] == Zeros()`
	`&& Rec(rec).gprs[10] == Zeros()`
	`&& Rec(rec).gprs[11] == Zeros()`
	`&& Rec(rec).gprs[12] == Zeros()`
	`&& Rec(rec).gprs[13] == Zeros()`
	`&& Rec(rec).gprs[14] == Zeros()`
	`&& Rec(rec).gprs[15] == Zeros()`
	`&& Rec(rec).gprs[16] == Zeros()`
	`&& Rec(rec).gprs[17] == Zeros()`
	`&& Rec(rec).gprs[18] == Zeros()`
	`&& Rec(rec).gprs[19] == Zeros()`
	`&& Rec(rec).gprs[20] == Zeros()`
	`&& Rec(rec).gprs[21] == Zeros()`
	`&& Rec(rec).gprs[22] == Zeros()`
	`&& Rec(rec).gprs[23] == Zeros()`
	`&& Rec(rec).gprs[24] == Zeros()`
	`&& Rec(rec).gprs[25] == Zeros()`
	`&& Rec(rec).gprs[26] == Zeros()`
	`&& Rec(rec).gprs[27] == Zeros()`
	`&& Rec(rec).gprs[28] == Zeros()`
	`&& Rec(rec).gprs[29] == Zeros()`
	`&& Rec(rec).gprs[30] == Zeros()`
	`&& Rec(rec).gprs[31] == Zeros())`

ID	Condition
rec_pc	`Rec(rec).pc == params.pc`
rim	`Realm(rd).measurements[0] == RimExtendRec(` `  realm, params)`

rec_aux	`RecAuxEqual(`
	`  Rec(rec).aux, params.aux,`
	`  RecAuxCount(rd))`

rec_aux_state	`RecAuxStateEqual(` `  Rec(rec).aux, RecAuxCount(rd), REC_AUX)`
ripas_addr	`Rec(rec).ripas_addr == Zeros()`
ripas_top	`Rec(rec).ripas_top == Zeros()`
host_call	`Rec(rec).host_call_pending == NO_HOST_CALL_PENDING`

B3.3.12.4 RMI_REC_CREATE extension of RIM

On successful execution of RMI_REC_CREATE, the new RIM value of the target Realm is calculated by the RMM as follows:

1. Allocate a zero-filled RmiRecParams data structure to hold the measured REC parameters.
2. Copy the following attributes from the Host-provided RmiRecParams data structure into the measured REC parameters data structure:
   - gprs
   - pc
   - flags
3. Using the RHA of the target Realm, compute the hash of the measured REC parameters data structure.
4. Allocate an RmmMeasurementDescriptorRec data structure.
5. Populate the measurement descriptor:
   - Set the desc_type field to the descriptor type.
   - Set the len field to the descriptor length.
   - Set the rim field to the current RIM value of the target Realm.
   - Set the content field to the hash of the measured REC parameters.
6. Using the RHA of the target Realm, compute the hash of the measurement descriptor. Set the RIM of the target Realm to this value, zero filling upper bytes if the RHA output is smaller than the size of the RIM.

See also:

- A7.1.1 *Realm Initial Measurement*
- B2.36 *RimExtendRec function*
- B3.4.17 *RmiRecParams type*
- C1.6 *RmmMeasurementDescriptorRec type*

B3.3.12.5 Footprint

ID	Value
rec_index	`Realm(rd).rec_index`

ID	Value
rec_state	`Granule(rec).state`
rec_aux_state	`RecAuxStates(Rec(rec).aux, RecAuxCount(rd))`
rim	`Realm(rd).measurements[0]`

B3.3.13 RMI_REC_DESTROY command

Destroys a REC.

See also:

- A2.3 *Realm Execution Context*
- B3.3.12 *RMI_REC_CREATE command*
- D1.2.5 *Realm destruction flow*

B3.3.13.1 Interface

B3.3.13.1.1 Input values

Name	Register	Bits	Type	Description
fid	X0	63:0	UInt64	FID, value `0xC400015B`
rec	X1	63:0	Address	PA of the target REC

B3.3.13.1.2 Context

The RMI_REC_DESTROY command operates on the following context.

Name	Type	Value	Before	Description
rd	Address	`Rec(rec).owner`	true	RD address
rec_obj	RmmRec	`Rec(rec)`	true	REC

B3.3.13.1.3 Output values

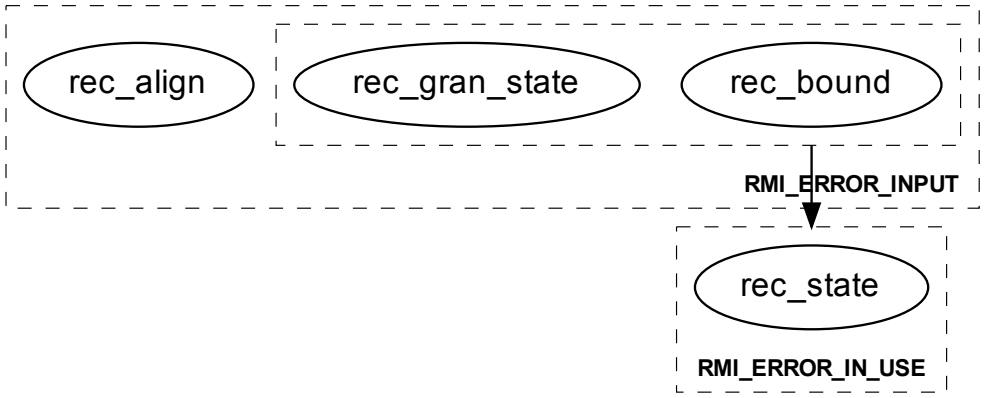
Name	Register	Bits	Type	Description
result	X0	63:0	RmiCommandReturnCode	Command return status

B3.3.13.2 Failure conditions

ID	Condition
rec_align	`pre: !AddrIsGranuleAligned(rec)` `post: ResultEqual(result, RMI_ERROR_INPUT)`
rec_bound	`pre: !PaIsDelegable(rec)` `post: ResultEqual(result, RMI_ERROR_INPUT)`
rec_gran_state	`pre: Granule(rec).state != REC` `post: ResultEqual(result, RMI_ERROR_INPUT)`
rec_state	`pre: Rec(rec).state == RUNNING` `post: ResultEqual(result, RMI_ERROR_IN_USE)`

B3.3.13.2.1 Failure condition ordering

```
[rec_bound, rec_gran_state] < [rec_state]
```



B3.3.13.3 Success conditions

ID	Condition
rec_gran_state	`Granule(rec).state == DELEGATED`
rec_aux_state	`RecAuxStateEqual(` `  rec_obj.aux, RecAuxCount(rd), DELEGATED)`

B3.3.13.4 Footprint

ID	Value
rec_state	`Granule(rec).state`
rec_aux_state	`RecAuxStates(rec_obj.aux, RecAuxCount(rd))`

B3.3.14 RMI_REC_ENTER command

Enter a REC.

See also:

- A2.3 *Realm Execution Context*
- Chapter A4 *Realm exception model*
- D1.3.1 *Realm entry and exit flow*

B3.3.14.1 Interface

B3.3.14.1.1 Input values

Name	Register	Bits	Type	Description
fid	X0	63:0	UInt64	FID, value `0xC400015C`
rec	X1	63:0	Address	PA of the target REC
run_ptr	X2	63:0	Address	PA of RecRun object

B3.3.14.1.2 Context

The RMI_REC_ENTER command operates on the following context.

Name	Type	Value	Before	Description
run	RmiRecRun	`RecRun(run_ptr)`	false	RecRun object

B3.3.14.1.3 Output values

Name	Register	Bits	Type	Description
result	X0	63:0	RmiCommandReturnCode	Command return status

B3.3.14.2 Failure conditions

ID	Condition
run_align	`pre: !AddrIsGranuleAligned(run_ptr)` `post: ResultEqual(result, RMI_ERROR_INPUT)`
run_bound	`pre: !PaIsDelegable(run_ptr)` `post: ResultEqual(result, RMI_ERROR_INPUT)`
run_pas	`pre: Granule(run_ptr).pas != NS` `post: ResultEqual(result, RMI_ERROR_INPUT)`
rec_align	`pre: !AddrIsGranuleAligned(rec)` `post: ResultEqual(result, RMI_ERROR_INPUT)`
rec_bound	`pre: !PaIsDelegable(rec)` `post: ResultEqual(result, RMI_ERROR_INPUT)`

ID	Condition
rec_gran_state	`pre: Granule(rec).state != REC` `post: ResultEqual(result, RMI_ERROR_INPUT)`
realm_new	`pre: Realm(Rec(rec).owner).state == NEW` `post: ResultEqual(result, RMI_ERROR_REALM, 0)`
system_off	`pre: Realm(Rec(rec).owner).state == SYSTEM_OFF` `post: ResultEqual(result, RMI_ERROR_REALM, 1)`
rec_state	`pre: Rec(rec).state == RUNNING` `post: ResultEqual(result, RMI_ERROR_IN_USE)`
rec_runnable	`pre: Rec(rec).flags.runnable == NOT_RUNNABLE` `post: ResultEqual(result, RMI_ERROR_REC)`

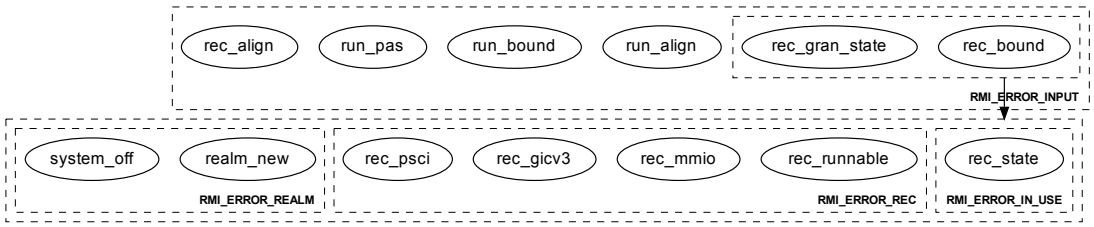
rec_mmio	`pre: (run.entry.flags.emul_mmio == RMI_EMULATED_MMIO`
	`&& Rec(rec).emulatable_abort != EMULATABLE_ABORT)`
	`post: ResultEqual(result, RMI_ERROR_REC)`

rec_gicv3	`pre: !Gicv3ConfigIsValid(`
	`run.entry.gicv3_hcr, run.entry.gicv3_lrs)`
	`post: ResultEqual(result, RMI_ERROR_REC)`

rec_psci	`pre: Rec(rec).psci_pending == PSCI_REQUEST_PENDING` `post: ResultEqual(result, RMI_ERROR_REC)`

B3.3.14.2.1 Failure condition ordering

```
[rec_bound, rec_gran_state] < [rec_state, rec_runnable, rec_mmio, realm_new,
    ↪system_off, rec_gicv3, rec_psci]
```



B3.3.14.3 Success conditions

ID	Condition
rec_exit	`run.exit contains Realm exit syndrome information.`
rec_emul_abt	`rec.emulatable_abort is updated.`

B3.3.14.4 Footprint

ID	Value
emul_abt	`Rec(rd).emulatable_abort`

B3.3.15 RMI_RTT_CREATE command

Creates an RTT.

See also:

- A5.5 *Realm Translation Table*
- A5.5.7 *RTT unfolding*
- B3.3.16 *RMI_RTT_DESTROY command*
- B3.3.17 *RMI_RTT_FOLD command*

B3.3.15.1 Interface

B3.3.15.1.1 Input values

Name	Register	Bits	Type	Description
fid	X0	63:0	UInt64	FID, value `0xC400015D`
rtt	X1	63:0	Address	PA of the target RTT
rd	X2	63:0	Address	PA of the RD for the target Realm
ipa	X3	63:0	Address	Base of the IPA range described by the RTT
level	X4	63:0	Int64	RTT level

B3.3.15.1.2 Context

The RMI_RTT_CREATE command operates on the following context.

Name	Type	Value	Before	Description

walk	RmmRttWalkResult	`RttWalk(`
		`  rd, ipa,`
		`  level - 1)`
			false	RTT walk result

entry_idx	UInt64	`RttEntryIndex(` `  ipa, walk.level)`	false	RTTE index

unfold	RmmRttEntry	`RttWalk(`
		`  rd, ipa,`
		`  level - 1).entry`
			true	RTTE before command execution

B3.3.15.1.3 Output values

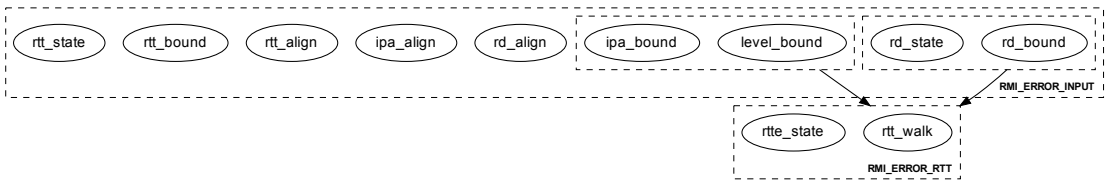
Name	Register	Bits	Type	Description
result	X0	63:0	RmiCommandReturnCode	Command return status

B3.3.15.2 Failure conditions

ID	Condition
rd_align	pre: !AddrIsGranuleAligned(rd) post: ResultEqual(result, RMI_ERROR_INPUT)
rd_bound	pre: !PaIsDelegable(rd) post: ResultEqual(result, RMI_ERROR_INPUT)
rd_state	pre: Granule(rd).state != RD post: ResultEqual(result, RMI_ERROR_INPUT)
level_bound	pre: (!RttLevelIsValid(rd, level) || RttLevelIsStarting(rd, level)) post: ResultEqual(result, RMI_ERROR_INPUT)
ipa_align	pre: !AddrIsRttLevelAligned(ipa, level - 1) post: ResultEqual(result, RMI_ERROR_INPUT)
ipa_bound	pre: UInt(ipa) >= (2 ^ Realm(rd).ipa_width) post: ResultEqual(result, RMI_ERROR_INPUT)
rtt_align	pre: !AddrIsGranuleAligned(rtt) post: ResultEqual(result, RMI_ERROR_INPUT)
rtt_bound	pre: !PaIsDelegable(rtt) post: ResultEqual(result, RMI_ERROR_INPUT)
rtt_state	pre: Granule(rtt).state != DELEGATED post: ResultEqual(result, RMI_ERROR_INPUT)
rtt_walk	pre: walk.level < level - 1 post: ResultEqual(result, RMI_ERROR_RTT, walk.level)
rtte_state	pre: walk.entry.state == TABLE post: ResultEqual(result, RMI_ERROR_RTT, walk.level)

B3.3.15.2.1 Failure condition ordering

```
[rd_bound, rd_state] < [rtt_walk, rtte_state]
[level_bound, ipa_bound] < [rtt_walk, rtte_state]
```



B3.3.15.3 Success conditions

ID	Condition
rtt_state	Granule(rtt).state == RTT
rtte_state	walk.entry.state == TABLE

ID	Condition
rtte_addr	`walk.entry.addr == rtt`
rtte_c_ripas	`RttAllEntriesRipas(Rtt(rtt), unfold.ripas)`
rtte_c_state	`RttAllEntriesState(Rtt(rtt), unfold.state)`
rtte_c_addr	`pre: (unfold.state != UNASSIGNED && unfold.state != DESTROYED)` `post: RttAllEntriesContiguous(Rtt(rtt), unfold.addr, level)`

B3.3.15.4 Footprint

ID	Value
rtt_state	`Granule(rtt).state`
rtte	`RttEntry(walk.rtt_addr, entry_idx)`

B3.3.16 RMI_RTT_DESTROY command

Destroys an RTT.

See also:

- A5.5 *Realm Translation Table*
- A5.5.9 *RTT destruction*
- B3.3.15 *RMI_RTT_CREATE command*
- B3.3.17 *RMI_RTT_FOLD command*

B3.3.16.1 Interface

B3.3.16.1.1 Input values

Name	Register	Bits	Type	Description
fid	X0	63:0	UInt64	FID, value `0xC400015E`
rtt	X1	63:0	Address	PA of the target RTT
rd	X2	63:0	Address	PA of the RD for the target Realm
ipa	X3	63:0	Address	Base of the IPA range described by the RTT
level	X4	63:0	Int64	RTT level

X_{FXWPM} In addition to the `ipa` and `level` input values which identify the target RTTE, the RMI_RTT_DESTROY command also takes the `rtt` address as an input value. The reason for this is to enable concurrent RTT modification from different Host threads.

As an example, consider a parent RTT, within which the entry for IPA `x` points to a child RTT `c1`. Host thread A destroys the child RTT at IPA `x` (`c1`) and replaces it with a new child RTT `c2`. Host thread B wishes to destroy the child RTT at IPA `x`.

In a traditional hypervisor, atomic check-and-modify behavior could be achieved through the use of fine-grained locking, for example:

```
    spin_lock(x);
    pte = read_pte(x);
    if (extract_output_address(pte) != c1) {
        ret = ERROR;
        goto unlock;
    }
    write_pte(x, NULL); // Clear table entry
unlock:
    spin_unlock(x);
```

A naive RMM equivalent would be:

```
    spin_lock(x);
    pte = RMI_RTT_READ(..., ipa=x);
    if (extract_output_address(pte) != c1) {
        ret = ERROR;
        goto unlock;
    }
```

```
    ret = RMI_RTT_DESTROY(..., ipa=x);
unlock:
    spin_unlock(x);
```

However, because execution of each RMI command may take a significant number of cycles, it may not be practical for the Host to hold a lock across this sequence.

By passing the expected RTT address as an input value to the RMI_RTT_DESTROY command, detection of an unexpected output address can be deferred to the RMM, and the Host-side locks can be removed:

```
RMI_RTT_DESTROY(..., ipa=x, rtt=c1);
```

B3.3.16.1.2 Context

The RMI_RTT_DESTROY command operates on the following context.

Name	Type	Value	Before	Description
walk	RmmRttWalkResult	`RttWalk( rd, ipa, level - 1)`	false	RTT walk result
entry_idx	UInt64	`RttEntryIndex( ipa, walk.level)`	false	RTTE index

B3.3.16.1.3 Output values

Name	Register	Bits	Type	Description
result	X0	63:0	RmiCommandReturnCode	Command return status

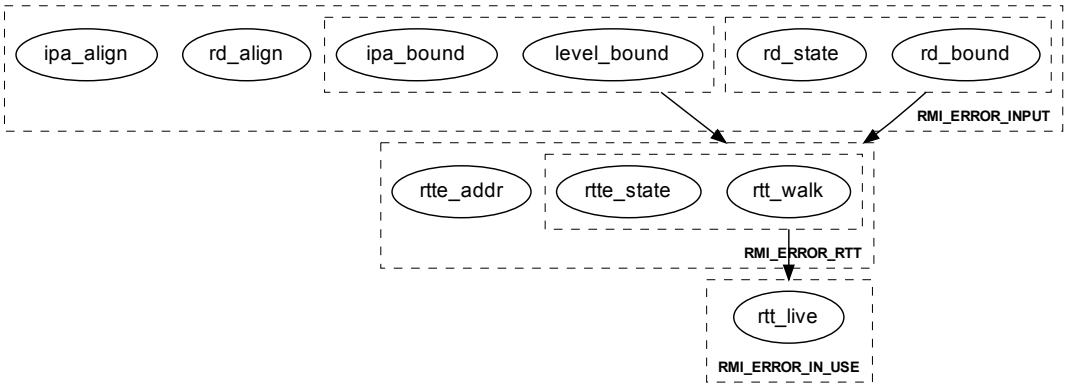
B3.3.16.2 Failure conditions

ID	Condition
rd_align	`pre: !AddrIsGranuleAligned(rd)` `post: ResultEqual(result, RMI_ERROR_INPUT)`
rd_bound	`pre: !PaIsDelegable(rd)` `post: ResultEqual(result, RMI_ERROR_INPUT)`
rd_state	`pre: Granule(rd).state != RD` `post: ResultEqual(result, RMI_ERROR_INPUT)`
level_bound	`pre: (!RttLevelIsValid(rd, level) || RttLevelIsStarting(rd, level))` `post: ResultEqual(result, RMI_ERROR_INPUT)`
ipa_align	`pre: !AddrIsRttLevelAligned(ipa, level - 1)` `post: ResultEqual(result, RMI_ERROR_INPUT)`
ipa_bound	`pre: UInt(ipa) >= (2 ^ Realm(rd).ipa_width)` `post: ResultEqual(result, RMI_ERROR_INPUT)`

ID	Condition
rtt_walk	pre: walk.level < level - 1 post: ResultEqual(result, RMI_ERROR_RTT, walk.level)
rtte_state	pre: walk.entry.state != TABLE post: ResultEqual(result, RMI_ERROR_RTT, walk.level)
rtte_addr	pre: walk.entry.addr != rtt post: ResultEqual(result, RMI_ERROR_RTT, walk.level)
rtt_live	pre: RttIsLive(Rtt(rtt)) post: ResultEqual(result, RMI_ERROR_IN_USE)

B3.3.16.2.1 Failure condition ordering

```
[rd_bound, rd_state] < [rtt_walk, rtte_state, rtte_addr]
[rtt_walk, rtte_state] < [rtt_live]
[level_bound, ipa_bound] < [rtt_walk, rtte_state, rtte_addr]
```



B3.3.16.3 Success conditions

ID	Condition
rtte_state	walk.entry.state == DESTROYED
rtt_state	Granule(rtt).state == DELEGATED

B3.3.16.4 Footprint

ID	Value
rtt_state	Granule(rtt).state
rtte	RttEntry(walk.rtt_addr, entry_idx)

B3.3.17 RMI_RTT_FOLD command

Destroys a homogeneous RTT.

See also:

- A5.5 *Realm Translation Table*
- A5.5.6 *RTT folding*
- B3.3.15 *RMI_RTT_CREATE command*
- B3.3.16 *RMI_RTT_DESTROY command*

B3.3.17.1 Interface

B3.3.17.1.1 Input values

Name	Register	Bits	Type	Description
fid	X0	63:0	UInt64	FID, value `0xC4000166`
rtt	X1	63:0	Address	PA of the target RTT
rd	X2	63:0	Address	PA of the RD for the target Realm
ipa	X3	63:0	Address	Base of the IPA range described by the RTT
level	X4	63:0	Int64	RTT level

B3.3.17.1.2 Context

The RMI_RTT_FOLD command operates on the following context.

Name	Type	Value	Before	Description
walk	RmmRttWalkResult	`RttWalk( rd, ipa, level - 1)`	false	RTT walk result
entry_idx	UInt64	`RttEntryIndex( ipa, walk.level)`	false	RTTE index
fold	RmmRttEntry	`RttFold(Rtt(rtt))`	true	Result of folding RTT

B3.3.17.1.3 Output values

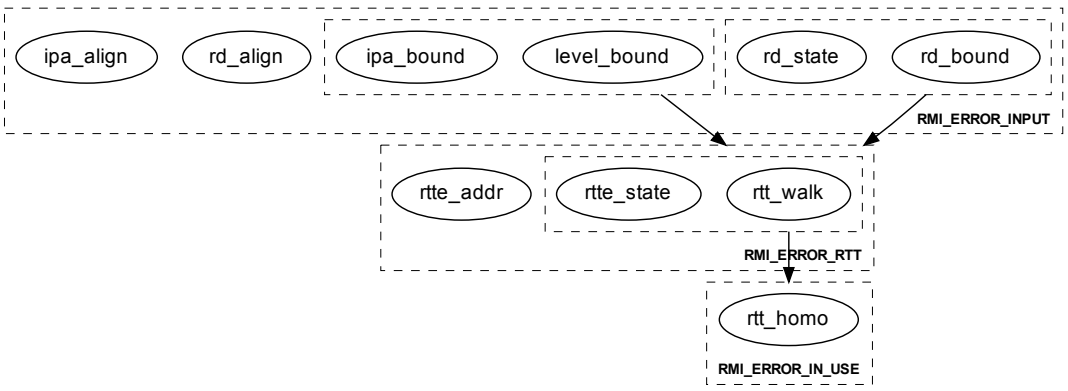
Name	Register	Bits	Type	Description
result	X0	63:0	RmiCommandReturnCode	Command return status

B3.3.17.2 Failure conditions

ID	Condition
rd_align	pre: !AddrIsGranuleAligned(rd) post: ResultEqual(result, RMI_ERROR_INPUT)
rd_bound	pre: !PaIsDelegable(rd) post: ResultEqual(result, RMI_ERROR_INPUT)
rd_state	pre: Granule(rd).state != RD post: ResultEqual(result, RMI_ERROR_INPUT)
level_bound	pre: (!RttLevelIsValid(rd, level) || RttLevelIsStarting(rd, level)) post: ResultEqual(result, RMI_ERROR_INPUT)
ipa_align	pre: !AddrIsRttLevelAligned(ipa, level - 1) post: ResultEqual(result, RMI_ERROR_INPUT)
ipa_bound	pre: UInt(ipa) >= (2 ^ Realm(rd).ipa_width) post: ResultEqual(result, RMI_ERROR_INPUT)
rtt_walk	pre: walk.level < level - 1 post: ResultEqual(result, RMI_ERROR_RTT, walk.level)
rtte_state	pre: walk.entry.state != TABLE post: ResultEqual(result, RMI_ERROR_RTT, walk.level)
rtte_addr	pre: walk.entry.addr != rtt post: ResultEqual(result, RMI_ERROR_RTT, walk.level)
rtt_homo	pre: !RttIsHomogeneous(Rtt(rtt)) post: ResultEqual(result, RMI_ERROR_IN_USE)

B3.3.17.2.1 Failure condition ordering

```
[rd_bound, rd_state] < [rtt_walk, rtte_state, rtte_addr]
[rtt_walk, rtte_state] < [rtt_homo]
[level_bound, ipa_bound] < [rtt_walk, rtte_state, rtte_addr]
```



B3.3.17.3 Success conditions

ID	Condition
rtte_state	`walk.entry.state == fold.state`
rtte_addr	`pre: (fold.state != UNASSIGNED && fold.state != DESTROYED)` `post: walk.entry.addr == fold.addr`
rtte_attr	`pre: fold.state == VALID_NS` `post: (walk.entry.MemAttr == fold.MemAttr && walk.entry.S2AP == fold.S2AP && walk.entry.SH == fold.SH)`
rtt_state	`Granule(rtt).state == DELEGATED`

B3.3.17.4 Footprint

ID	Value
rtt_state	`Granule(rtt).state`
rtte	`RttEntry(walk.rtt_addr, entry_idx)`

B3.3.18 RMI_RTT_INIT_RIPAS command

Set the RIPAS of a target IPA range to RAM, for a Realm in the NEW state.

See also:

- A5.2.2 *Realm IPA state*
- D1.2.3 *Initialize memory of New Realm flow*

B3.3.18.1 Interface

B3.3.18.1.1 Input values

Name	Register	Bits	Type	Description
fid	X0	63:0	UInt64	FID, value `0xC4000168`
rd	X1	63:0	Address	PA of the RD for the target Realm
ipa	X2	63:0	Address	IPA
level	X3	63:0	Int64	RTT level

B3.3.18.1.2 Context

The RMI_RTT_INIT_RIPAS command operates on the following context.

Name	Type	Value	Before	Description
realm	RmmRealm	`Realm(rd)`	true	Realm
walk	RmmRttWalkResult	`RttWalk(rd, ipa, level)`	false	RTT walk result
entry_idx	UInt64	`RttEntryIndex(` `  ipa, walk.level)`	false	RTTE index

B3.3.18.1.3 Output values

Name	Register	Bits	Type	Description
result	X0	63:0	RmiCommandReturnCode	Command return status

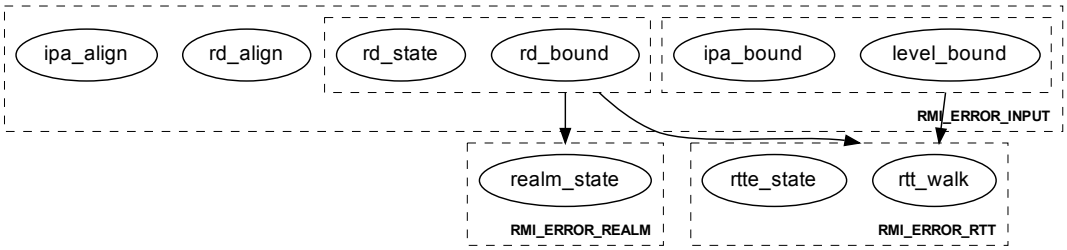
B3.3.18.2 Failure conditions

ID	Condition
rd_align	`pre:  !AddrIsGranuleAligned(rd)` `post: ResultEqual(result, RMI_ERROR_INPUT)`
rd_bound	`pre:  !PaIsDelegable(rd)` `post: ResultEqual(result, RMI_ERROR_INPUT)`
rd_state	`pre:  Granule(rd).state != RD` `post: ResultEqual(result, RMI_ERROR_INPUT)`

ID	Condition
level_bound	pre: !RttLevelIsValid(rd, level) post: ResultEqual(result, RMI_ERROR_INPUT)
ipa_align	pre: !AddrIsRttLevelAligned(ipa, level) post: ResultEqual(result, RMI_ERROR_INPUT)
ipa_bound	pre: !AddrIsProtected(ipa, realm) post: ResultEqual(result, RMI_ERROR_INPUT)
realm_state	pre: realm.state != NEW post: ResultEqual(result, RMI_ERROR_REALM)
rtt_walk	pre: walk.level < level post: ResultEqual(result, RMI_ERROR_RTT, walk.level)
rtte_state	pre: walk.entry.state != UNASSIGNED post: ResultEqual(result, RMI_ERROR_RTT, walk.level)

B3.3.18.2.1 Failure condition ordering

```
[rd_bound, rd_state] < [realm_state]
[rd_bound, rd_state] < [rtt_walk, rtte_state]
[level_bound, ipa_bound] < [rtt_walk, rtte_state]
```



B3.3.18.3 Success conditions

ID	Condition
rtte_ripas	walk.entry.ripas == RAM
rim	Realm(rd).measurements[0] == RimExtendRipas(realm, ipa, level)

B3.3.18.4 RMI_RTT_INIT_RIPAS extension of RIM

On successful execution of RMI_RTT_INIT_RIPAS, the new RIM value of the target Realm is calculated by the RMM as follows:

1. Allocate an RmmMeasurementDescriptorRipas data structure.

2. Populate the measurement descriptor:

- Set the desc_type field to the descriptor type.
- Set the len field to the descriptor length.
- Set the ipa field to the IPA value
- Set the level field to the RTT level.

3. Using the RHA of the target Realm, compute the hash of the measurement descriptor. Set the RIM of the target Realm to this value, zero filling upper bytes if the RHA output is smaller than the size of the RIM.

See also:

- A7.1.1 *Realm Initial Measurement*
- B2.37 *RimExtendRipas function*
- C1.7 *RmmMeasurementDescriptorRipas type*

B3.3.18.5 Footprint

ID	Value
rtte_ripas	`RttEntry(walk.rtt_addr, entry_idx).ripas`
rim	`Realm(rd).measurements[0]`

B3.3.19 RMI_RTT_MAP_UNPROTECTED command

Creates a mapping from an Unprotected IPA to a Non-secure PA.

See also:

- A5.5 *Realm Translation Table*
- B3.3.22 *RMI_RTT_UNMAP_UNPROTECTED command*

B3.3.19.1 Interface

B3.3.19.1.1 Input values

Name	Register	Bits	Type	Description
fid	X0	63:0	UInt64	FID, value `0xC400015F`
rd	X1	63:0	Address	PA of the RD for the target Realm
ipa	X2	63:0	Address	IPA at which the Granule will be mapped in the target Realm
level	X3	63:0	Int64	RTT level
desc	X4	63:0	Bits64	RTTE descriptor

The layout and encoding of fields in the `desc` input value match "Attribute fields in stage 2 VMSAv8-64 Block and Page descriptors" in *Arm Architecture Reference Manual for Armv8-A architecture profile* [3].

See also:

- *Arm Architecture Reference Manual for Armv8-A architecture profile* [3]
- A5.5.11 *RTT entry attributes*
- B2.51 *RttDescriptorIsValidForUnprotected function*

B3.3.19.1.2 Context

The RMI_RTT_MAP_UNPROTECTED command operates on the following context.

Name	Type	Value	Before	Description
walk	RmmRttWalkResult	`RttWalk(` `  rd, ipa, level)`	false	RTT walk result
entry_idx	UInt64	`RttEntryIndex(` `  ipa, walk.level)`	false	RTTE index
rtte	RmmRttEntry	`RttEntryFromDescriptor(` `  ↪desc)`	false	RTT entry

B3.3.19.1.3 Output values

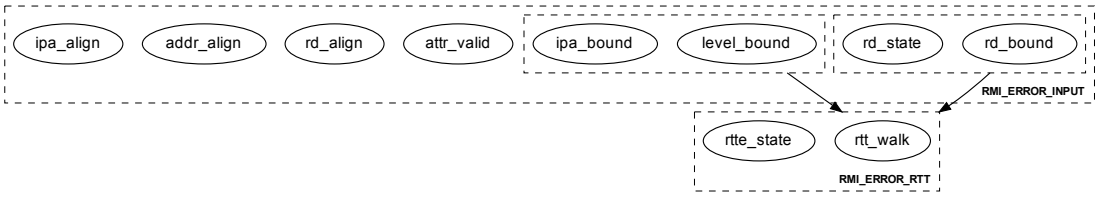
Name	Register	Bits	Type	Description
result	X0	63:0	RmiCommandReturnCode	Command return status

B3.3.19.2 Failure conditions

ID	Condition
attr_valid	pre: !RttDescriptorIsValidForUnprotected(desc) post: ResultEqual(result, RMI_ERROR_INPUT)
rd_align	pre: !AddrIsGranuleAligned(rd) post: ResultEqual(result, RMI_ERROR_INPUT)
rd_bound	pre: !PaIsDelegable(rd) post: ResultEqual(result, RMI_ERROR_INPUT)
rd_state	pre: Granule(rd).state != RD post: ResultEqual(result, RMI_ERROR_INPUT)
level_bound	pre: !RttLevelIsBlockOrPage(rd, level) post: ResultEqual(result, RMI_ERROR_INPUT)
addr_align	pre: !AddrIsRttLevelAligned(rtte.addr, level) post: ResultEqual(result, RMI_ERROR_INPUT)
ipa_align	pre: !AddrIsRttLevelAligned(ipa, level) post: ResultEqual(result, RMI_ERROR_INPUT)
ipa_bound	pre: (UInt(ipa) >= (2 ^ Realm(rd).ipa_width) || AddrIsProtected(ipa, Realm(rd))) post: ResultEqual(result, RMI_ERROR_INPUT)
rtt_walk	pre: walk.level < level post: ResultEqual(result, RMI_ERROR_RTT, walk.level)
rtte_state	pre: (walk.entry.state != UNASSIGNED && walk.entry.state != DESTROYED) post: ResultEqual(result, RMI_ERROR_RTT, walk.level)

B3.3.19.2.1 Failure condition ordering

```
[rd_bound, rd_state] < [rtt_walk, rtte_state]
[level_bound, ipa_bound] < [rtt_walk, rtte_state]
```



B3.3.19.3 Success conditions

ID	Condition
rtte_state	walk.entry.state == VALID_NS

ID	Condition

rtte_contents	`(walk.entry.MemAttr == rtte.MemAttr`
	`&& walk.entry.S2AP == rtte.S2AP`
	`&& walk.entry.SH == rtte.SH`
	`&& walk.entry.addr == rtte.addr)`

B3.3.19.4 Footprint

ID	Value
rtte	`RttEntry(walk.rtt_addr, entry_idx)`

B3.3.20 RMI_RTT_READ_ENTRY command

Reads an RTTE.

See also:

- A5.5 *Realm Translation Table*

B3.3.20.1 Interface

B3.3.20.1.1 Input values

Name	Register	Bits	Type	Description
fid	X0	63:0	UInt64	FID, value `0xC4000161`
rd	X1	63:0	Address	PA of the RD for the target Realm
ipa	X2	63:0	Address	Realm Address for which to read the RTTE
level	X3	63:0	Int64	RTT level at which to read the RTTE

B3.3.20.1.2 Context

The RMI_RTT_READ_ENTRY command operates on the following context.

Name	Type	Value	Before	Description
walk	RmmRttWalkResult	`RttWalk( rd, ipa, level)`	false	RTT walk result
rtte	RmmRttEntry	`RttEntryFromDescriptor( ↪desc)`	false	RTT entry

B3.3.20.1.3 Output values

Name	Register	Bits	Type	Description
result	X0	63:0	RmiCommandReturnCode	Command return status
walk_level	X1	63:0	UInt64	RTT level reached by the RTT walk
state	X2	7:0	RmiRttEntryState	State of RTTE reached by the walk
desc	X3	63:0	Bits64	RTTE descriptor
ripas	X4	7:0	RmiRipas	RIPAS of RTTE reached by the walk

Unused bits of RMI_RTT_READ_ENTRY output values must be zero.

The layout and encoding of fields in the `rtte` output value match “Attribute fields in stage 2 VMSAv8-64 Block and Page descriptors” in *Arm Architecture Reference Manual for Armv8-A architecture profile* [3].

See also:

- *Arm Architecture Reference Manual for Armv8-A architecture profile* [3]
- A5.5.11 *RTT entry attributes*

B3.3.20.2 Failure conditions

ID	Condition
rd_align	`pre: !AddrIsGranuleAligned(rd)` `post: ResultEqual(result, RMI_ERROR_INPUT)`
rd_bound	`pre: !PaIsDelegable(rd)` `post: ResultEqual(result, RMI_ERROR_INPUT)`
rd_state	`pre: Granule(rd).state != RD` `post: ResultEqual(result, RMI_ERROR_INPUT)`
level_bound	`pre: !RttLevelIsValid(rd, level)` `post: ResultEqual(result, RMI_ERROR_INPUT)`
ipa_align	`pre: !AddrIsRttLevelAligned(ipa, level)` `post: ResultEqual(result, RMI_ERROR_INPUT)`
ipa_bound	`pre: UInt(ipa) >= (2 ^ Realm(rd).ipa_width)` `post: ResultEqual(result, RMI_ERROR_INPUT)`

B3.3.20.2.1 Failure condition ordering

The RMI_RTT_READ_ENTRY command does not have any failure condition orderings.

B3.3.20.3 Success conditions

ID	Condition

state_invalid	`pre: (walk.entry.state == UNASSIGNED`
	`|| walk.entry.state == DESTROYED)`
	`post: X3 == Zeros()`

state_prot	`pre: (walk.entry.state == ASSIGNED`
	`|| walk.entry.state == TABLE)`
	`post: (rtte.MemAttr == Zeros()`
	`&& rtte.S2AP == Zeros()`
	`&& rtte.SH == Zeros()`
	`&& rtte.addr == walk.entry.addr)`

state_unprot	`pre: walk.entry.state == VALID_NS`
	`post: (rtte.MemAttr == walk.entry.MemAttr`
	`&& rtte.S2AP == walk.entry.S2AP`
	`&& rtte.SH == walk.entry.SH`
	`&& rtte.addr == walk.entry.addr)`

ripas_unprot	`pre: (!AddrIsProtected(ipa, Realm(rd))`
	`|| (walk.entry.state != UNASSIGNED`
	`&& walk.entry.state != ASSIGNED))`
	`post: X4 == Zeros()`

B3.3.20.4 Footprint

The RMI_RTT_READ_ENTRY command does not have any footprint.

B3.3.21 RMI_RTT_SET_RIPAS command

Completes a request made by the Realm to change the RIPAS of a target IPA range.

See also:

- A5.4 *RIPAS change*

B3.3.21.1 Interface

B3.3.21.1.1 Input values

Name	Register	Bits	Type	Description
fid	X0	63:0	UInt64	FID, value `0xC4000169`
rd	X1	63:0	Address	PA of the RD for the target Realm
rec	X2	63:0	Address	PA of the target REC
ipa	X3	63:0	Address	IPA
level	X4	63:0	Int64	RTT level
ripas	X5	7:0	RmiRipas	RIPAS value

Unused bits of RMI_RTT_SET_RIPAS input values must be zero.

B3.3.21.1.2 Context

The RMI_RTT_SET_RIPAS command operates on the following context.

Name	Type	Value	Before	Description
walk	RmmRttWalkResult	`RttWalk( rd, ipa, level)`	false	RTT walk result
entry_idx	UInt64	`RttEntryIndex( ipa, walk.level)`	false	RTTE index
ripas_addr	Address	`Rec(rec).ripas_addr`	true	Target address

B3.3.21.1.3 Output values

Name	Register	Bits	Type	Description
result	X0	63:0	RmiCommandReturnCode	Command return status

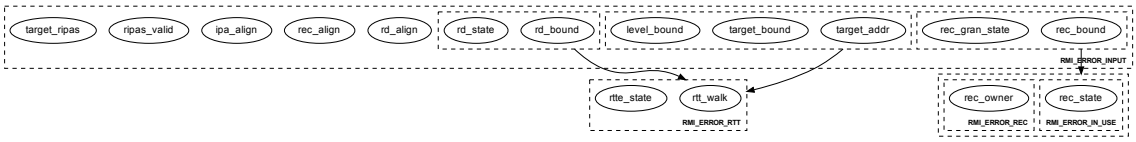
B3.3.21.2 Failure conditions

ID	Condition
rd_align	`pre: !AddrIsGranuleAligned(rd)` `post: ResultEqual(result, RMI_ERROR_INPUT)`

ID	Condition
rd_bound	pre: !PaIsDelegable(rd) post: ResultEqual(result, RMI_ERROR_INPUT)
rd_state	pre: Granule(rd).state != RD post: ResultEqual(result, RMI_ERROR_INPUT)
level_bound	pre: !RttLevelIsValid(rd, level) post: ResultEqual(result, RMI_ERROR_INPUT)
rec_align	pre: !AddrIsGranuleAligned(rec) post: ResultEqual(result, RMI_ERROR_INPUT)
rec_bound	pre: !PaIsDelegable(rec) post: ResultEqual(result, RMI_ERROR_INPUT)
rec_gran_state	pre: Granule(rec).state != REC post: ResultEqual(result, RMI_ERROR_INPUT)
rec_state	pre: Rec(rec).state == RUNNING post: ResultEqual(result, RMI_ERROR_IN_USE)
rec_owner	pre: Rec(rec).owner != rd post: ResultEqual(result, RMI_ERROR_REC)
ipa_align	pre: !AddrIsRttLevelAligned(ipa, level) post: ResultEqual(result, RMI_ERROR_INPUT)
ripas_valid	pre: !RmiRipasIsValid(X5[31:0]) post: ResultEqual(result, RMI_ERROR_INPUT)
target_addr	pre: ipa != Rec(rec).ripas_addr post: ResultEqual(result, RMI_ERROR_INPUT)
target_bound	pre: ((UInt(ipa) + RttLevelSize(level)) > UInt(Rec(rec).ripas_top)) post: ResultEqual(result, RMI_ERROR_INPUT)
target_ripas	pre: !Equal(ripas, Rec(rec).ripas_value) post: ResultEqual(result, RMI_ERROR_INPUT)
rtt_walk	pre: walk.level < level post: ResultEqual(result, RMI_ERROR_RTT, walk.level)
rtte_state	pre: (walk.entry.state != UNASSIGNED && walk.entry.state != ASSIGNED) post: ResultEqual(result, RMI_ERROR_RTT, walk.level)

B3.3.21.2.1 Failure condition ordering

```
[rd_bound, rd_state] < [rtt_walk, rtte_state]
[rec_bound, rec_gran_state] < [rec_state, rec_owner]
[target_addr, target_bound, level_bound] < [rtt_walk, rtte_state]
```



B3.3.21.3 Success conditions

ID	Condition
rtte_ripas	`Equal(walk.entry.ripas, ripas)`
ripas_addr	`Rec(rec).ripas_addr == ToAddress(` `  UInt(ripas_addr) + RttLevelSize(level))`

B3.3.21.4 Footprint

ID	Value
rtte	`RttEntry(walk.rtt_addr, entry_idx)`
ripas_addr	`Rec(rec).ripas_addr`

B3.3.22 RMI_RTT_UNMAP_UNPROTECTED command

Removes a mapping at an Unprotected IPA.

See also:

- A5.5 *Realm Translation Table*
- B3.3.19 *RMI_RTT_MAP_UNPROTECTED command*

B3.3.22.1 Interface

B3.3.22.1.1 Input values

Name	Register	Bits	Type	Description
fid	X0	63:0	UInt64	FID, value `0xC4000162`
rd	X1	63:0	Address	PA of the RD for the target Realm
ipa	X2	63:0	Address	IPA at which the Granule is mapped in the target Realm
level	X3	63:0	Int64	RTT level

B3.3.22.1.2 Context

The RMI_RTT_UNMAP_UNPROTECTED command operates on the following context.

Name	Type	Value	Before	Description
walk	RmmRttWalkResult	`RttWalk(rd, ipa, level)`	false	RTT walk result
entry_idx	UInt64	`RttEntryIndex(ipa, walk.level)`	false	RTTE index

B3.3.22.1.3 Output values

Name	Register	Bits	Type	Description
result	X0	63:0	RmiCommandReturnCode	Command return status

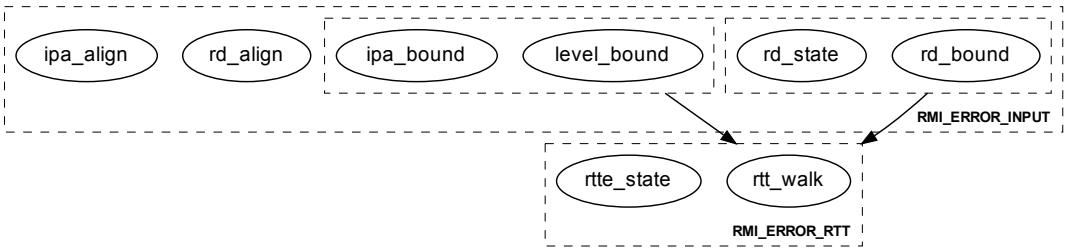
B3.3.22.2 Failure conditions

ID	Condition
rd_align	`pre: !AddrIsGranuleAligned(rd)` `post: ResultEqual(result, RMI_ERROR_INPUT)`
rd_bound	`pre: !PaIsDelegable(rd)` `post: ResultEqual(result, RMI_ERROR_INPUT)`
rd_state	`pre: Granule(rd).state != RD` `post: ResultEqual(result, RMI_ERROR_INPUT)`

ID	Condition
level_bound	`pre: !RttLevelIsBlockOrPage(rd, level)` `post: ResultEqual(result, RMI_ERROR_INPUT)`
ipa_align	`pre: !AddrIsRttLevelAligned(ipa, level)` `post: ResultEqual(result, RMI_ERROR_INPUT)`
ipa_bound	`pre: UInt(ipa) >= (2 ^ Realm(rd).ipa_width)` `post: ResultEqual(result, RMI_ERROR_INPUT)`
rtt_walk	`pre: walk.level < level` `post: ResultEqual(result, RMI_ERROR_RTT, walk.level)`
rtte_state	`pre: walk.entry.state != VALID_NS` `post: ResultEqual(result, RMI_ERROR_RTT, walk.level)`

B3.3.22.2.1 Failure condition ordering

```
[rd_bound, rd_state] < [rtt_walk, rtte_state]
[level_bound, ipa_bound] < [rtt_walk, rtte_state]
```



B3.3.22.3 Success conditions

ID	Condition
rtte_state	`walk.entry.state == UNASSIGNED`

B3.3.22.4 Footprint

ID	Value
rtte	`RttEntry(walk.rtt_addr, entry_idx)`

B3.3.23 RMI_VERSION command

Returns RMI version.

B3.3.23.1 Interface

B3.3.23.1.1 Input values

Name	Register	Bits	Type	Description
fid	X0	63:0	UInt64	FID, value `0xC4000150`

B3.3.23.1.2 Output values

Name	Register	Bits	Type	Description
result	X0	63:0	RmiInterfaceVersion	Interface version

See also:

- B3.1 *RMI version*

B3.3.23.2 Failure conditions

The RMI_VERSION command does not have any failure conditions.

B3.3.23.3 Success conditions

The RMI_VERSION command does not have any success conditions.

B3.3.23.4 Footprint

The RMI_VERSION command does not have any footprint.

B3.4 RMI types

This section defines types which are used in the RMI interface.

B3.4.1 RmiCommandReturnCode type

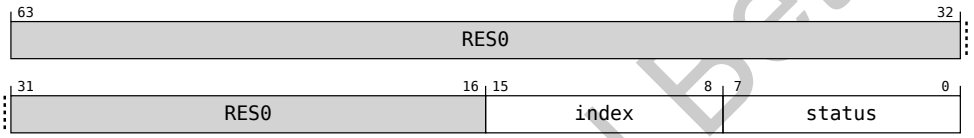
The RmiCommandReturnCode fieldset contains a return code from an RMI command.

The width of the RmiCommandReturnCode fieldset is 64 bits.

See also:

- Chapter B1 *Commands*

The fields of the RmiCommandReturnCode fieldset are shown in the following diagram.



The fields of the RmiCommandReturnCode fieldset are shown in the following table.

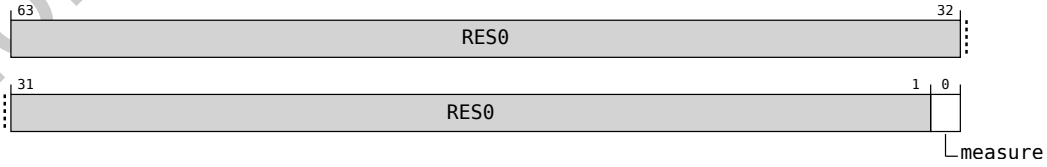
Name	Bits	Description	Value
status	7:0	Status of the command	RmiStatusCode
index	15:8	Index which identifies the reason for a command failure	UInt8
	63:16	Reserved	Must be zero

B3.4.2 RmiDataFlags type

The RmiDataFlags fieldset contains flags provided by the Host during DATA Granule creation.

The width of the RmiDataFlags fieldset is 64 bits.

The fields of the RmiDataFlags fieldset are shown in the following diagram.



The fields of the RmiDataFlags fieldset are shown in the following table.

Name	Bits	Description	Value
measure	0:0	Whether to measure DATA Granule contents	RmiDataMeasureContent
	63:1	Reserved	Must be zero

B3.4.3 RmiDataMeasureContent type

The RmiDataMeasureContent enumeration represents whether to measure DATA Granule contents.

The width of the RmiDataMeasureContent enumeration is 1 bits.

The values of the RmiDataMeasureContent enumeration are shown in the following table.

Encoding	Name	Description
0	RMI_NO_MEASURE_CONTENT	Do not measure DATA Granule contents.
1	RMI_MEASURE_CONTENT	Measure DATA Granule contents.

B3.4.4 RmiEmulatedMmio type

The RmiEmulatedMmio enumeration represents whether the host has completed emulation for an Emulatable Abort.

The width of the RmiEmulatedMmio enumeration is 1 bits.

The values of the RmiEmulatedMmio enumeration are shown in the following table.

Encoding	Name	Description
0	RMI_NOT_EMULATED_MMIO	Host has not completed emulation for an Emulatable Abort.
1	RMI_EMULATED_MMIO	Host has completed emulation for an Emulatable Abort.

B3.4.5 RmiFeature type

The RmiFeature enumeration represents whether a feature is supported or enabled.

The width of the RmiFeature enumeration is 1 bits.

The values of the RmiFeature enumeration are shown in the following table.

Encoding	Name	Description
0	RMI_NOT_SUPPORTED	• During discovery: Feature is not supported. • During selection: Feature is not enabled.
1	RMI_SUPPORTED	• During discovery: Feature is supported. • During selection: Feature is enabled.

B3.4.6 RmiFeatureRegister0 type

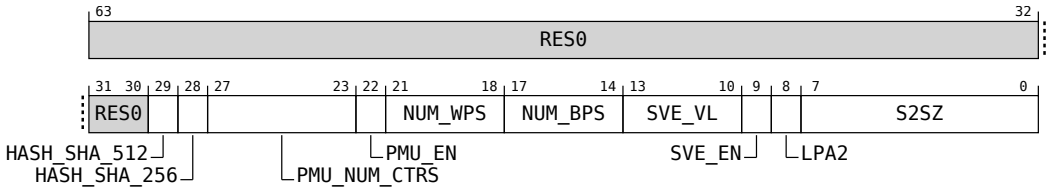
The RmiFeatureRegister0 fieldset contains feature register 0.

The width of the RmiFeatureRegister0 fieldset is 64 bits.

See also:

- A3.1 *Realm feature discovery and selection*
- B3.3.4 *RMI_FEATURES command*

The fields of the RmiFeatureRegister0 fieldset are shown in the following diagram.



The fields of the RmiFeatureRegister0 fieldset are shown in the following table.

Name	Bits	Description	Value
S2SZ	7:0	Specifies the input address size for stage 2 translation to be `2 ^ S2SZ`. Note this format expresses the IPA width directly and is therefore different from the `VTCR_EL2.T0SZ` encoding. • During discovery: maximum Realm IPA width supported by the RMM. • During selection: requested Realm IPA width.	UInt8
LPA2	8:8	• During discovery: Whether LPA2 is supported. • During selection: Whether LPA2 is enabled.	RmiFeature
SVE_EN	9:9	• During discovery: Whether SVE is supported. • During selection: Whether SVE is enabled.	RmiFeature
SVE_VL	13:10	• During discovery: maximum SVE vector length supported by the RMM. The effective vector length supported by the RMM is `(SVE_VL + 1)*128`, similar to the value of `ZCR_ELx.LEN`. • During selection: requested SVE vector length.	UInt4
NUM_BPS	17:14	• During discovery: number of breakpoints available. • During selection: requested number of breakpoints.	UInt4
NUM_WPS	21:18	• During discovery: number of watchpoints available. • During selection: requested number of watchpoints.	UInt4

Name	Bits	Description	Value
PMU_EN	22:22	• During discovery: Whether PMU is supported. • During selection: Whether PMU is enabled.	RmiFeature
PMU_NUM_CTRS	27:23	• During discovery: number of PMU counters available. • During selection: requested number of PMU counters.	UInt5
HASH_SHA_256	28:28	• During discovery: Whether SHA-256 is supported. • During selection: ignored.	RmiFeature
HASH_SHA_512	29:29	• During discovery: Whether SHA-512 is supported. • During selection: ignored.	RmiFeature
	63:30	Reserved	Must be zero

B3.4.7 RmiHashAlgorithm type

The RmiHashAlgorithm enumeration represents hash algorithm.

The width of the RmiHashAlgorithm enumeration is 8 bits.

The values of the RmiHashAlgorithm enumeration are shown in the following table.

Encoding	Name	Description
0	RMI_HASH_SHA_256	SHA-256 (*Secure Hash Standard (SHS)* [15])
1	RMI_HASH_SHA_512	SHA-512 (*Secure Hash Standard (SHS)* [15])

Unused encodings for the RmiHashAlgorithm enumeration are reserved for use by future versions of this specification.

B3.4.8 RmiInjectSea type

The RmiInjectSea enumeration represents whether to inject a Synchronous External Abort into the Realm.

The width of the RmiInjectSea enumeration is 1 bits.

The values of the RmiInjectSea enumeration are shown in the following table.

Encoding	Name	Description
0	RMI_NO_INJECT_SEA	Do not inject an SEA into the Realm.
1	RMI_INJECT_SEA	Inject an SEA into the Realm.

B3.4.9 RmiInterfaceVersion type

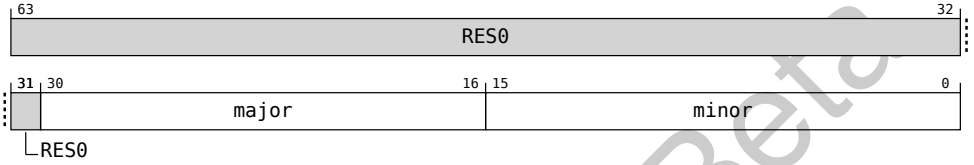
The RmiInterfaceVersion fieldset contains an RMI interface version.

The width of the RmiInterfaceVersion fieldset is 64 bits.

See also:

- B3.1 *RMI version*
- B3.3.23 *RMI_VERSION command*

The fields of the RmiInterfaceVersion fieldset are shown in the following diagram.



The fields of the RmiInterfaceVersion fieldset are shown in the following table.

Name	Bits	Description	Value
minor	15:0	Interface minor version number	UInt16
major	30:16	Interface major version number	UInt15
	63:31	Reserved	Must be zero

B3.4.10 RmiRealmParams type

The RmiRealmParams structure contains parameters provided by the Host during Realm creation.

The width of the RmiRealmParams structure is 4096 (`0x1000`) bytes.

See also:

- A2.1.6 *Realm parameters*
- B3.3.9 *RMI_REALM_CREATE command*

The members of the RmiRealmParams structure are shown in the following table.

Name	Byte offset	Type	Description
features_0	`0x0`	RmiFeatureRegister0	Feature register 0
hash_algo	`0x100`	RmiHashAlgorithm	Algorithm used to measure the initial state of the Realm
rpv	`0x400`	Bits512	Realm Personalization Value
vmid	`0x800`	Bits16	Virtual Machine Identifier
rtt_base	`0x808`	Address	Realm Translation Table base
rtt_level_start	`0x810`	Int64	RTT starting level
rtt_num_start	`0x818`	UInt32	Number of starting level RTTs

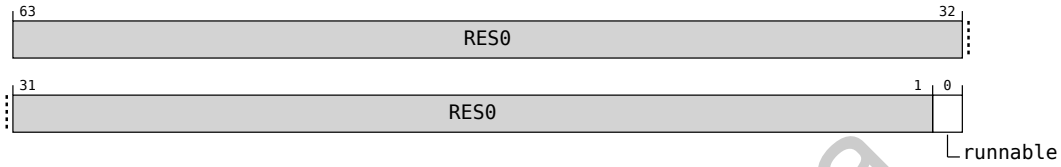
Unused bits of the RmiRealmParams structure should be zero.

B3.4.11 RmiRecCreateFlags type

The RmiRecCreateFlags fieldset contains flags provided by the Host during REC creation.

The width of the RmiRecCreateFlags fieldset is 64 bits.

The fields of the RmiRecCreateFlags fieldset are shown in the following diagram.



The fields of the RmiRecCreateFlags fieldset are shown in the following table.

Name	Bits	Description	Value
runnable	0:0	Whether REC is eligible for execution	RmiRecRunnable
	63:1	Reserved	Must be zero

B3.4.12 RmiRecEntry type

The RmiRecEntry structure contains data passed from the Host to the RMM on REC entry.

The width of the RmiRecEntry structure is 2048 (`0x800`) bytes.

See also:

- A4.2.1 *RecEntry object*
- B3.3.14 *RMI_REC_ENTER command*
- B3.4.14 *RmiRecExit type*

The members of the RmiRecEntry structure are shown in the following table.

Name	Byte offset	Type	Description
flags	`0x0`	RmiRecEntryFlags	Flags
gprs[0]	`0x200`	Bits64	Registers
gprs[1]	`0x208`	Bits64	Registers
gprs[2]	`0x210`	Bits64	Registers
gprs[3]	`0x218`	Bits64	Registers
gprs[4]	`0x220`	Bits64	Registers
gprs[5]	`0x228`	Bits64	Registers
gprs[6]	`0x230`	Bits64	Registers
gprs[7]	`0x238`	Bits64	Registers
gprs[8]	`0x240`	Bits64	Registers
gprs[9]	`0x248`	Bits64	Registers
gprs[10]	`0x250`	Bits64	Registers
gprs[11]	`0x258`	Bits64	Registers

Name	Byte offset	Type	Description
gprs[12]	0x260	Bits64	Registers
gprs[13]	0x268	Bits64	Registers
gprs[14]	0x270	Bits64	Registers
gprs[15]	0x278	Bits64	Registers
gprs[16]	0x280	Bits64	Registers
gprs[17]	0x288	Bits64	Registers
gprs[18]	0x290	Bits64	Registers
gprs[19]	0x298	Bits64	Registers
gprs[20]	0x2a0	Bits64	Registers
gprs[21]	0x2a8	Bits64	Registers
gprs[22]	0x2b0	Bits64	Registers
gprs[23]	0x2b8	Bits64	Registers
gprs[24]	0x2c0	Bits64	Registers
gprs[25]	0x2c8	Bits64	Registers
gprs[26]	0x2d0	Bits64	Registers
gprs[27]	0x2d8	Bits64	Registers
gprs[28]	0x2e0	Bits64	Registers
gprs[29]	0x2e8	Bits64	Registers
gprs[30]	0x2f0	Bits64	Registers
gicv3_hcr	0x300	Bits64	GICv3 Hypervisor Control Register value
gicv3_lrs[0]	0x308	Bits64	GICv3 List Register values
gicv3_lrs[1]	0x310	Bits64	GICv3 List Register values
gicv3_lrs[2]	0x318	Bits64	GICv3 List Register values
gicv3_lrs[3]	0x320	Bits64	GICv3 List Register values
gicv3_lrs[4]	0x328	Bits64	GICv3 List Register values
gicv3_lrs[5]	0x330	Bits64	GICv3 List Register values
gicv3_lrs[6]	0x338	Bits64	GICv3 List Register values
gicv3_lrs[7]	0x340	Bits64	GICv3 List Register values
gicv3_lrs[8]	0x348	Bits64	GICv3 List Register values
gicv3_lrs[9]	0x350	Bits64	GICv3 List Register values
gicv3_lrs[10]	0x358	Bits64	GICv3 List Register values
gicv3_lrs[11]	0x360	Bits64	GICv3 List Register values
gicv3_lrs[12]	0x368	Bits64	GICv3 List Register values
gicv3_lrs[13]	0x370	Bits64	GICv3 List Register values
gicv3_lrs[14]	0x378	Bits64	GICv3 List Register values

Name	Byte offset	Type	Description
gicv3_lrs[15]	0x380	Bits64	GICv3 List Register values

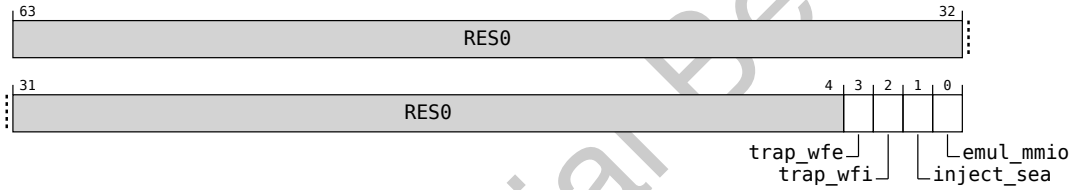
Unused bits of the RmiRecEntry structure should be zero.

B3.4.13 RmiRecEntryFlags type

The RmiRecEntryFlags fieldset contains flags provided by the Host during REC entry.

The width of the RmiRecEntryFlags fieldset is 64 bits.

The fields of the RmiRecEntryFlags fieldset are shown in the following diagram.



The fields of the RmiRecEntryFlags fieldset are shown in the following table.

Name	Bits	Description	Value
emul_mmio	0:0	Whether the host has completed emulation for an Emulatable Data Abort	RmiEmulatedMmio
inject_sea	1:1	Whether to inject a Synchronous External Abort into the Realm.	RmiInjectSea
trap_wfi	2:2	Whether to trap WFI execution by the Realm.	RmiTrap
trap_wfe	3:3	Whether to trap WFE execution by the Realm.	RmiTrap
	63:4	Reserved	Must be zero

B3.4.14 RmiRecExit type

The RmiRecExit structure contains data passed from the RMM to the Host on REC exit.

The width of the RmiRecExit structure is 2048 (0x800) bytes.

See also:

- A4.3.1 *RecExit object*
- B3.3.14 *RMI_REC_ENTER command*
- B3.4.12 *RmiRecEntry type*

The members of the RmiRecExit structure are shown in the following table.

Name	Byte offset	Type	Description
exit_reason	0x0	RmiRecExitReason	Exit reason
esr	0x100	Bits64	Exception Syndrome Register
far	0x108	Bits64	Fault Address Register

Name	Byte offset	Type	Description
hpfar	0x110	Bits64	Hypervisor IPA Fault Address register
gprs[0]	0x200	Bits64	Registers
gprs[1]	0x208	Bits64	Registers
gprs[2]	0x210	Bits64	Registers
gprs[3]	0x218	Bits64	Registers
gprs[4]	0x220	Bits64	Registers
gprs[5]	0x228	Bits64	Registers
gprs[6]	0x230	Bits64	Registers
gprs[7]	0x238	Bits64	Registers
gprs[8]	0x240	Bits64	Registers
gprs[9]	0x248	Bits64	Registers
gprs[10]	0x250	Bits64	Registers
gprs[11]	0x258	Bits64	Registers
gprs[12]	0x260	Bits64	Registers
gprs[13]	0x268	Bits64	Registers
gprs[14]	0x270	Bits64	Registers
gprs[15]	0x278	Bits64	Registers
gprs[16]	0x280	Bits64	Registers
gprs[17]	0x288	Bits64	Registers
gprs[18]	0x290	Bits64	Registers
gprs[19]	0x298	Bits64	Registers
gprs[20]	0x2a0	Bits64	Registers
gprs[21]	0x2a8	Bits64	Registers
gprs[22]	0x2b0	Bits64	Registers
gprs[23]	0x2b8	Bits64	Registers
gprs[24]	0x2c0	Bits64	Registers
gprs[25]	0x2c8	Bits64	Registers
gprs[26]	0x2d0	Bits64	Registers
gprs[27]	0x2d8	Bits64	Registers
gprs[28]	0x2e0	Bits64	Registers
gprs[29]	0x2e8	Bits64	Registers
gprs[30]	0x2f0	Bits64	Registers
gicv3_hcr	0x300	Bits64	GICv3 Hypervisor Control Register value
gicv3_lrs[0]	0x308	Bits64	GICv3 List Register values
gicv3_lrs[1]	0x310	Bits64	GICv3 List Register values

Name	Byte offset	Type	Description
gicv3_lrs[2]	0x318	Bits64	GICv3 List Register values
gicv3_lrs[3]	0x320	Bits64	GICv3 List Register values
gicv3_lrs[4]	0x328	Bits64	GICv3 List Register values
gicv3_lrs[5]	0x330	Bits64	GICv3 List Register values
gicv3_lrs[6]	0x338	Bits64	GICv3 List Register values
gicv3_lrs[7]	0x340	Bits64	GICv3 List Register values
gicv3_lrs[8]	0x348	Bits64	GICv3 List Register values
gicv3_lrs[9]	0x350	Bits64	GICv3 List Register values
gicv3_lrs[10]	0x358	Bits64	GICv3 List Register values
gicv3_lrs[11]	0x360	Bits64	GICv3 List Register values
gicv3_lrs[12]	0x368	Bits64	GICv3 List Register values
gicv3_lrs[13]	0x370	Bits64	GICv3 List Register values
gicv3_lrs[14]	0x378	Bits64	GICv3 List Register values
gicv3_lrs[15]	0x380	Bits64	GICv3 List Register values
gicv3_misr	0x388	Bits64	GICv3 Maintenance Interrupt State Register value
gicv3_vmcr	0x390	Bits64	GICv3 Virtual Machine Control Register value
cntp_ctl	0x400	Bits64	Counter-timer Physical Timer Control Register value
cntp_cval	0x408	Bits64	Counter-timer Physical Timer CompareValue Register value
cntv_ctl	0x410	Bits64	Counter-timer Virtual Timer Control Register value
cntv_cval	0x418	Bits64	Counter-timer Virtual Timer CompareValue Register value
ripas_base	0x500	Bits64	Base address of pending RIPAS change
ripas_size	0x508	UInt64	Size of pending RIPAS change
ripas_value	0x510	RmiRipas	RIPAS value of pending RIPAS change
imm	0x600	Bits16	Host call immediate value
pmu_ovf	0x700	Bits64	PMU overflow
pmu_intr_en	0x708	Bits64	PMU interrupt enable
pmu_cntr_en	0x710	Bits64	PMU counter enable

Unused bits of the RmiRecExit structure must be zero.

B3.4.15 RmiRecExitReason type

The RmiRecExitReason enumeration represents the reason for a REC exit.

The width of the RmiRecExitReason enumeration is 8 bits.

The values of the RmiRecExitReason enumeration are shown in the following table.

Encoding	Name	Description
0	RMI_EXIT_SYNC	REC exit due to synchronous exception
1	RMI_EXIT_IRQ	REC exit due to IRQ
2	RMI_EXIT_FIQ	REC exit due to FIQ
3	RMI_EXIT_PSCI	REC exit due to PSCI
4	RMI_EXIT_RIPAS_CHANGE	REC exit due to RIPAS change pending
5	RMI_EXIT_HOST_CALL	REC exit due to Host call
6	RMI_EXIT_SERROR	REC exit due to SError

Unused encodings for the RmiRecExitReason enumeration are reserved for use by future versions of this specification.

B3.4.16 RmiRecMpidr type

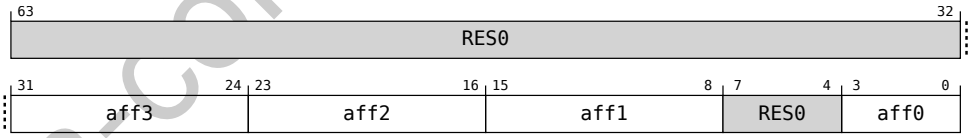
The RmiRecMpidr fieldset contains MPIDR value which identifies a REC.

The width of the RmiRecMpidr fieldset is 64 bits.

See also:

- A2.3.3 *REC index and MPIDR value*
- B3.3.12 *RMI_REC_CREATE command*

The fields of the RmiRecMpidr fieldset are shown in the following diagram.



The fields of the RmiRecMpidr fieldset are shown in the following table.

Name	Bits	Description	Value
aff0	3:0	Affinity level 0	Bits4
	7:4	Reserved	Must be zero
aff1	15:8	Affinity level 1	Bits8
aff2	23:16	Affinity level 2	Bits8
aff3	31:24	Affinity level 3	Bits8
	63:32	Reserved	Must be zero

B3.4.17 RmiRecParams type

The RmiRecParams structure contains parameters provided by the Host during REC creation.

The width of the RmiRecParams structure is 4096 (`0x1000`) bytes.

The number of valid entries in the `aux` array is determined by the return value from the RMI_REC_AUX_COUNT command.

See also:

- B3.3.11 *RMI_REC_AUX_COUNT command*

The members of the RmiRecParams structure are shown in the following table.

Name	Byte offset	Type	Description
flags	`0x0`	RmiRecCreateFlags	Flags
mpidr	`0x100`	RmiRecMpidr	MPIDR of the REC
pc	`0x200`	Bits64	Program counter
gprs[0]	`0x300`	Bits64	General-purpose registers
gprs[1]	`0x308`	Bits64	General-purpose registers
gprs[2]	`0x310`	Bits64	General-purpose registers
gprs[3]	`0x318`	Bits64	General-purpose registers
gprs[4]	`0x320`	Bits64	General-purpose registers
gprs[5]	`0x328`	Bits64	General-purpose registers
gprs[6]	`0x330`	Bits64	General-purpose registers
gprs[7]	`0x338`	Bits64	General-purpose registers
num_aux	`0x800`	UInt64	Number of auxiliary Granules
aux[0]	`0x808`	Address	Addresses of auxiliary Granules
aux[1]	`0x810`	Address	Addresses of auxiliary Granules
aux[2]	`0x818`	Address	Addresses of auxiliary Granules
aux[3]	`0x820`	Address	Addresses of auxiliary Granules
aux[4]	`0x828`	Address	Addresses of auxiliary Granules
aux[5]	`0x830`	Address	Addresses of auxiliary Granules
aux[6]	`0x838`	Address	Addresses of auxiliary Granules
aux[7]	`0x840`	Address	Addresses of auxiliary Granules
aux[8]	`0x848`	Address	Addresses of auxiliary Granules
aux[9]	`0x850`	Address	Addresses of auxiliary Granules
aux[10]	`0x858`	Address	Addresses of auxiliary Granules
aux[11]	`0x860`	Address	Addresses of auxiliary Granules
aux[12]	`0x868`	Address	Addresses of auxiliary Granules
aux[13]	`0x870`	Address	Addresses of auxiliary Granules
aux[14]	`0x878`	Address	Addresses of auxiliary Granules
aux[15]	`0x880`	Address	Addresses of auxiliary Granules

Unused bits of the RmiRecParams structure should be zero.

B3.4.18 RmiRecRun type

The RmiRecRun structure contains used to share information between RMM and Host during REC entry and REC exit.

The width of the RmiRecRun structure is 4096 (`0x1000`) bytes.

See also:

- A4.2.1 *RecEntry object*
- A4.3.1 *RecExit object*
- B3.3.14 *RMI_REC_ENTER command*

The members of the RmiRecRun structure are shown in the following table.

Name	Byte offset	Type	Description
entry	`0x0`	RmiRecEntry	Entry information
exit	`0x800`	RmiRecExit	Exit information

B3.4.19 RmiRecRunnable type

The RmiRecRunnable enumeration represents whether a REC is eligible for execution.

The width of the RmiRecRunnable enumeration is 1 bits.

The values of the RmiRecRunnable enumeration are shown in the following table.

Encoding	Name	Description
0	RMI_NOT_RUNNABLE	Not eligible for execution.
1	RMI_RUNNABLE	Eligible for execution.

B3.4.20 RmiRipas type

The RmiRipas enumeration represents realm IPA state.

The width of the RmiRipas enumeration is 8 bits.

The values of the RmiRipas enumeration are shown in the following table.

Encoding	Name	Description
0	RMI_EMPTY	Unused IPA location.
1	RMI_RAM	Private code or data owned by the Realm.

Unused encodings for the RmiRipas enumeration are reserved for use by future versions of this specification.

B3.4.21 RmiRttEntryState type

The RmiRttEntryState enumeration represents the state of an RTTE.

The width of the RmiRttEntryState enumeration is 8 bits.

The values of the RmiRttEntryState enumeration are shown in the following table.

Encoding	Name	Description
0	RMI_UNASSIGNED	This RTTE is not associated with any Granule.
1	RMI_DESTROYED	This RTTE cannot be used for the rest of the lifetime of the Realm.
2	RMI_ASSIGNED	The output address of this RTTE points to a DATA Granule.
3	RMI_TABLE	The output address of this RTTE points to the next-level RTT.
4	RMI_VALID_NS	The output address of this RTTE is in the Non- secure PAS. The mapping is valid.

Unused encodings for the RmiRttEntryState enumeration are reserved for use by future versions of this specification.

B3.4.22 RmiStatusCode type

The RmiStatusCode enumeration represents the status of an RMI operation.

The width of the RmiStatusCode enumeration is 8 bits.

See also:

- B1.3 *Command registers*
- B1.5 *Command context values*

The values of the RmiStatusCode enumeration are shown in the following table.

Encoding	Name	Description
0	RMI_SUCCESS	Command completed successfully
1	RMI_ERROR_INPUT	The value of a command input value caused the command to fail
2	RMI_ERROR_REALM	An attribute of a Realm does not match the expected value
3	RMI_ERROR_REC	An attribute of a REC does not match the expected value
4	RMI_ERROR_RTT	An RTT walk terminated before reaching the target RTT level, or reached an RTTE with an unexpected value
5	RMI_ERROR_IN_USE	An operation cannot be completed because a resource is in use

Unused encodings for the RmiStatusCode enumeration are reserved for use by future versions of this specification.

B3.4.23 RmiTrap type

The RmiTrap enumeration represents whether a trap is enabled.

The width of the RmiTrap enumeration is 1 bits.

The values of the RmiTrap enumeration are shown in the following table.

Encoding	Name	Description
0	RMI_NO_TRAP	Trap is disabled.
1	RMI_TRAP	Trap is enabled.

Chapter B4
Realm Services Interface

This chapter defines the interface used by Realm software to request services from the RMM.

B4.1 RSI version

R_{QKLGZ} This specification defines version 1.0 of the Realm Services Interface.

See also:

- B4.3.9 *RSI_VERSION command*

B4.2 RSI command return codes

I_{CYQDJ} An RSI command return code indicates whether the command

- succeeded, or
- failed, and the reason for the failure.

I_{DQJSP} If an RSI command succeeds then it returns RSI_SUCCESS.

I_{YMHKC} Multiple failure conditions in an RSI command may return the same return code.

R_{MLBDM} If an input to an RSI command uses an invalid encoding then the command fails and returns RSI_ERROR_INPUT.

Command inputs include registers and in-memory data structures.

Invalid encodings include:

- setting a “must be zero” bit to `'1'`
- using a reserved encoding in an enumeration

See also:

- B4.4.1 *RsiCommandReturnCode type*

B4.3 RSI commands

The following table summarizes the FIDs of commands in the RSI interface.

FID	Command
0xC4000195	RSI_ATTESTATION_TOKEN_CONTINUE
0xC4000194	RSI_ATTESTATION_TOKEN_INIT
0xC4000199	RSI_HOST_CALL
0xC4000198	RSI_IPA_STATE_GET
0xC4000197	RSI_IPA_STATE_SET
0xC4000193	RSI_MEASUREMENT_EXTEND
0xC4000192	RSI_MEASUREMENT_READ
0xC4000196	RSI_REALM_CONFIG
0xC4000190	RSI_VERSION

B4.3.1 RSI_ATTESTATION_TOKEN_CONTINUE command

Continue the operation to retrieve an attestation token.

See also:

- A7.2 *Realm attestation*
- B4.3.2 *RSI_ATTESTATION_TOKEN_INIT command*

B4.3.1.1 Interface

B4.3.1.1.1 Input values

Name	Register	Bits	Type	Description
fid	X0	63:0	UInt64	FID, value `0xC4000195`
addr	X1	63:0	Address	IPA of the Granule to which the token will be written

B4.3.1.1.2 Context

The RSI_ATTESTATION_TOKEN_CONTINUE command operates on the following context.

Name	Type	Value	Before	Description
realm	RmmRealm	`CurrentRealm()`	false	Current Realm
rec	RmmRec	`CurrentRec()`	false	Current REC

B4.3.1.1.3 Output values

Name	Register	Bits	Type	Description
result	X0	63:0	RsiCommandReturnCode	Command return status
size	X1	63:0	UInt64	Token size in bytes

B4.3.1.2 Failure conditions

ID	Condition
addr	`pre: addr != rec.attest_addr` `post: result == RSI_ERROR_INPUT`
state	`pre: rec.attest_state != ATTEST_IN_PROGRESS` `post: result == RSI_ERROR_STATE`

B4.3.1.2.1 Failure condition ordering

The RSI_ATTESTATION_TOKEN_CONTINUE command does not have any failure condition orderings.

B4.3.1.3 Success conditions

ID	Condition
incomplete	`pre: Token generation is not complete.` `post: result == RSI_INCOMPLETE`
complete	`pre: Token generation is complete.` `post: rec.attest_state == NO_ATTEST_IN_PROGRESS`

B4.3.1.4 Footprint

ID	Value
state	`rec.attest_state`

B4.3.2 RSI_ATTESTATION_TOKEN_INIT command

Initialize the operation to retrieve an attestation token.

See also:

- A7.2 *Realm attestation*
- B4.3.1 *RSI_ATTESTATION_TOKEN_CONTINUE command*

B4.3.2.1 Interface

B4.3.2.1.1 Input values

Name	Register	Bits	Type	Description
fid	X0	63:0	UInt64	FID, value `0xC4000194`
addr	X1	63:0	Address	IPA of the Granule to which the token will be written
challenge_0	X2	63:0	Bits64	Doubleword 0 of the challenge value
challenge_1	X3	63:0	Bits64	Doubleword 1 of the challenge value
challenge_2	X4	63:0	Bits64	Doubleword 2 of the challenge value
challenge_3	X5	63:0	Bits64	Doubleword 3 of the challenge value
challenge_4	X6	63:0	Bits64	Doubleword 4 of the challenge value
challenge_5	X7	63:0	Bits64	Doubleword 5 of the challenge value
challenge_6	X8	63:0	Bits64	Doubleword 6 of the challenge value
challenge_7	X9	63:0	Bits64	Doubleword 7 of the challenge value

B4.3.2.1.2 Context

The RSI_ATTESTATION_TOKEN_INIT command operates on the following context.

Name	Type	Value	Before	Description
realm	RmmRealm	`CurrentRealm()`	false	Current Realm
rec	RmmRec	`CurrentRec()`	false	Current REC

B4.3.2.1.3 Output values

Name	Register	Bits	Type	Description
result	X0	63:0	RsiCommandReturnCode	Command return status

B4.3.2.2 Failure conditions

ID	Condition
addr_align	`pre: !AddrIsGranuleAligned(addr)` `post: result == RSI_ERROR_INPUT`
addr_bound	`pre: !AddrIsProtected(addr, realm)` `post: result == RSI_ERROR_INPUT`

B4.3.2.2.1 Failure condition ordering

The RSI_ATTESTATION_TOKEN_INIT command does not have any failure condition orderings.

B4.3.2.3 Success conditions

ID	Condition
state	`rec.attest_state == ATTEST_IN_PROGRESS`
addr	`rec.attest_addr == addr`

challenge	`rec.attest_challenge == [`
	`  challenge_0,`
	`  challenge_1,`
	`  challenge_2,`
	`  challenge_3,`
	`  challenge_4,`
	`  challenge_5,`
	`  challenge_6,`
	`  challenge_7`
	`]`

B4.3.2.4 Footprint

ID	Value
state	`rec.attest_state`
addr	`rec.attest_addr`
challenge	`rec.attest_challenge`

B4.3.3 RSI_HOST_CALL command

Make a Host call.

See also:

- A4.5 *Host call*

B4.3.3.1 Interface

B4.3.3.1.1 Input values

Name	Register	Bits	Type	Description
fid	X0	63:0	UInt64	FID, value `0xC4000199`
addr	X1	63:0	Address	IPA of the Host call data structure

B4.3.3.1.2 Context

The RSI_HOST_CALL command operates on the following context.

Name	Type	Value	Before	Description
realm	RmmRealm	`CurrentRealm()`	false	Current Realm
rec	RmmRec	`CurrentRec()`	false	Current REC
data	RsiHostCall	`RealmHostCall(addr)`	false	Host call data structure

B4.3.3.1.3 Output values

Name	Register	Bits	Type	Description
result	X0	63:0	RsiCommandReturnCode	Command return status

B4.3.3.2 Failure conditions

ID	Condition
addr_align	`pre: !AddrIsGranuleAligned(addr)` `post: result == RSI_ERROR_INPUT`
addr_bound	`pre: !AddrIsProtected(addr, realm)` `post: result == RSI_ERROR_INPUT`

B4.3.3.2.1 Failure condition ordering

The RSI_HOST_CALL command does not have any failure condition orderings.

B4.3.3.3 Success conditions

The RSI_HOST_CALL command does not have any success conditions.

B4.3.3.4 Footprint

ID	Value
host_call	`rec.host_call_pending`

B4.3.4 RSI_IPA_STATE_GET command

Get RIPAS of a target page.

See also:

- A5.2 *Realm view of memory management*
- B4.3.5 *RSI_IPA_STATE_SET command*

B4.3.4.1 Interface

B4.3.4.1.1 Input values

Name	Register	Bits	Type	Description
fid	X0	63:0	UInt64	FID, value `0xC4000198`
addr	X1	63:0	Address	IPA of target page

B4.3.4.1.2 Context

The RSI_IPA_STATE_GET command operates on the following context.

Name	Type	Value	Before	Description
realm	RmmRealm	`CurrentRealm()`	false	Current Realm

B4.3.4.1.3 Output values

Name	Register	Bits	Type	Description
result	X0	63:0	RsiCommandReturnCode	Command return status
ripas	X1	7:0	RsiRipas	RIPAS value

Unused bits of RSI_IPA_STATE_GET output values must be zero.

B4.3.4.2 Failure conditions

ID	Condition
addr_align	`pre: !AddrIsGranuleAligned(addr)` `post: result == RSI_ERROR_INPUT`
addr_bound	`pre: !AddrIsProtected(addr, realm)` `post: result == RSI_ERROR_INPUT`

B4.3.4.2.1 Failure condition ordering

The RSI_IPA_STATE_GET command does not have any failure condition orderings.

B4.3.4.3 Success conditions

The RSI_IPA_STATE_GET command does not have any success conditions.

B4.3.4.4 Footprint

The RSI_IPA_STATE_GET command does not have any footprint.

B4.3.5 RSI_IPA_STATE_SET command

Request RIPAS of a target IPA range to be changed to a specified value.

See also:

- A5.2 *Realm view of memory management*
- A5.4 *RIPAS change*
- B4.3.4 *RSI_IPA_STATE_GET command*

B4.3.5.1 Interface

B4.3.5.1.1 Input values

Name	Register	Bits	Type	Description
fid	X0	63:0	UInt64	FID, value `0xC4000197`
base	X1	63:0	Address	Base of target IPA region
size	X2	63:0	UInt64	Size of target IPA region
ripas	X3	7:0	RsiRipas	RIPAS value

Unused bits of RSI_IPA_STATE_SET input values must be zero.

B4.3.5.1.2 Context

The RSI_IPA_STATE_SET command operates on the following context.

Name	Type	Value	Before	Description
realm	RmmRealm	`CurrentRealm()`	false	Current Realm
rec	RmmRec	`CurrentRec()`	false	Current REC

B4.3.5.1.3 Output values

Name	Register	Bits	Type	Description
result	X0	63:0	RsiCommandReturnCode	Command return status
new_base	X1	63:0	Address	Base of IPA region which was not modified by the command

B4.3.5.2 Failure conditions

ID	Condition
addr_align	`pre: !AddrIsGranuleAligned(base)` `post: result == RSI_ERROR_INPUT`
size_bound	`pre: !AddrIsGranuleAligned(size)` `post: result == RSI_ERROR_INPUT`

ID	Condition
rgn_bound	`pre: !AddrRangeIsProtected(base, size, realm)` `post: result == RSI_ERROR_INPUT`
ripas_valid	`pre: !RsiRipasIsValid(X3[7:0])` `post: result == RSI_ERROR_INPUT`

B4.3.5.2.1 Failure condition ordering

The RSI_IPA_STATE_SET command does not have any failure condition orderings.

B4.3.5.3 Success conditions

ID	Condition
new_base	`new_base == rec.ripas_addr`

B4.3.5.4 Footprint

The RSI_IPA_STATE_SET command does not have any footprint.

B4.3.6 RSI_MEASUREMENT_EXTEND command

Extend Realm Extensible Measurement (REM) value.

B4.3.6.1 Interface

B4.3.6.1.1 Input values

Name	Register	Bits	Type	Description
fid	X0	63:0	UInt64	FID, value `0xC4000193`
index	X1	63:0	UInt64	Measurement index
size	X2	63:0	UInt64	Measurement size in bytes
value_0	X3	63:0	Bits64	Doubleword 0 of the measurement value
value_1	X4	63:0	Bits64	Doubleword 1 of the measurement value
value_2	X5	63:0	Bits64	Doubleword 2 of the measurement value
value_3	X6	63:0	Bits64	Doubleword 3 of the measurement value
value_4	X7	63:0	Bits64	Doubleword 4 of the measurement value
value_5	X8	63:0	Bits64	Doubleword 5 of the measurement value
value_6	X9	63:0	Bits64	Doubleword 6 of the measurement value
value_7	X10	63:0	Bits64	Doubleword 7 of the measurement value

B4.3.6.1.2 Context

The RSI_MEASUREMENT_EXTEND command operates on the following context.

Name	Type	Value	Before	Description
realm	RmmRealm	`CurrentRealm()`	false	Current Realm
meas_old	RmmRealmMeasuremen	`CurrentRealm().measurements[index]`	true	Previous measurement value

B4.3.6.1.3 Output values

Name	Register	Bits	Type	Description
result	X0	63:0	RsiCommandReturnCode	Command return status

B4.3.6.2 Failure conditions

ID	Condition
index_bound	`pre: index < 1 || index > 4` `post: result == RSI_ERROR_INPUT`
size_bound	`pre: size > 64` `post: result == RSI_ERROR_INPUT`

B4.3.6.2.1 Failure condition ordering

The RSI_MEASUREMENT_EXTEND command does not have any failure condition orderings.

B4.3.6.3 Success conditions

ID	Condition

realm_meas	`realm.measurements[index] == RemExtend(`
	`realm.hash_algo, meas_old,`
	`[value_0, value_1, value_2, value_3,`
	`value_4, value_5, value_6, value_7],`
	`size)`

B4.3.6.4 Footprint

ID	Value
realm_meas	`realm.measurements[index]`

B4.3.7 RSI_MEASUREMENT_READ command

Read measurement for the current Realm.

See also:

- A7.1 *Realm measurements*
- D1.2.1 *Realm creation flow*

B4.3.7.1 Interface

B4.3.7.1.1 Input values

Name	Register	Bits	Type	Description
fid	X0	63:0	UInt64	FID, value `0xC4000192`
index	X1	63:0	UInt64	Measurement index

`index` 0 selects the RIM. An `index` of 1 or greater selects the corresponding REM.

B4.3.7.1.2 Output values

Name	Register	Bits	Type	Description
result	X0	63:0	RsiCommandReturnCode	Command return status
value_0	X1	63:0	Bits64	Doubleword 0 of the Realm measurement identified by “index”
value_1	X2	63:0	Bits64	Doubleword 1 of the Realm measurement identified by “index”
value_2	X3	63:0	Bits64	Doubleword 2 of the Realm measurement identified by “index”
value_3	X4	63:0	Bits64	Doubleword 3 of the Realm measurement identified by “index”
value_4	X5	63:0	Bits64	Doubleword 4 of the Realm measurement identified by “index”
value_5	X6	63:0	Bits64	Doubleword 5 of the Realm measurement identified by “index”
value_6	X7	63:0	Bits64	Doubleword 6 of the Realm measurement identified by “index”
value_7	X8	63:0	Bits64	Doubleword 7 of the Realm measurement identified by “index”

If the size of the measurement value is smaller than 512 bits, the output values are padded with zeroes.

B4.3.7.2 Failure conditions

ID	Condition
index_bound	`pre: index > 4` `post: result == RSI_ERROR_INPUT`

B4.3.7.3 Success conditions

The RSI_MEASUREMENT_READ command does not have any success conditions.

B4.3.7.4 Footprint

The RSI_MEASUREMENT_READ command does not have any footprint.

B4.3.8 RSI_REALM_CONFIG command

Read configuration for the current Realm.

B4.3.8.1 Interface

B4.3.8.1.1 Input values

Name	Register	Bits	Type	Description
fid	X0	63:0	UInt64	FID, value `0xC4000196`
addr	X1	63:0	Address	IPA of the Granule to which the configuration data will be written

B4.3.8.1.2 Context

The RSI_REALM_CONFIG command operates on the following context.

Name	Type	Value	Before	Description
realm	RmmRealm	`CurrentRealm()`	false	Current Realm
cfg	RsiRealmConfig	`RealmConfig(addr)`	false	Realm configuration

B4.3.8.1.3 Output values

Name	Register	Bits	Type	Description
result	X0	63:0	RsiCommandReturnCode	Command return status

B4.3.8.2 Failure conditions

ID	Condition
addr_align	`pre: !AddrIsGranuleAligned(addr)` `post: result == RSI_ERROR_INPUT`
addr_bound	`pre: !AddrIsProtected(addr, realm)` `post: result == RSI_ERROR_INPUT`

B4.3.8.2.1 Failure condition ordering

The RSI_REALM_CONFIG command does not have any failure condition orderings.

B4.3.8.3 Success conditions

ID	Condition
ipa_width	`cfg.ipa_width == realm.ipa_width`

B4.3.8.4 Footprint

The RSI_REALM_CONFIG command does not have any footprint.

B4.3.9 RSI_VERSION command

Returns RSI version.

B4.3.9.1 Interface

B4.3.9.1.1 Input values

Name	Register	Bits	Type	Description
fid	X0	63:0	UInt64	FID, value `0xC4000190`

B4.3.9.1.2 Output values

Name	Register	Bits	Type	Description
result	X0	63:0	RsiInterfaceVersion	Interface version

See also:

- B4.1 *RSI version*

B4.3.9.2 Failure conditions

The RSI_VERSION command does not have any failure conditions.

B4.3.9.3 Success conditions

The RSI_VERSION command does not have any success conditions.

B4.3.9.4 Footprint

The RSI_VERSION command does not have any footprint.

B4.4 RSI types

This section defines types which are used in the RSI interface.

B4.4.1 RsiCommandReturnCode type

The RsiCommandReturnCode enumeration represents a return code from an RSI command.

The width of the RsiCommandReturnCode enumeration is 64 bits.

See also:

- Chapter B1 *Commands*

The values of the RsiCommandReturnCode enumeration are shown in the following table.

Encoding	Name	Description
0	RSI_SUCCESS	Command completed successfully
1	RSI_ERROR_INPUT	The value of a command input value caused the command to fail
2	RSI_ERROR_STATE	The state of the current Realm or current REC does not match the state expected by the command
3	RSI_INCOMPLETE	The operation requested by the command is not complete

Unused encodings for the RsiCommandReturnCode enumeration are reserved for use by future versions of this specification.

B4.4.2 RsiHostCall type

The RsiHostCall structure contains data structure used to pass Host call arguments and return values.

The width of the RsiHostCall structure is 4096 (`0x1000`) bytes.

See also:

- A4.5 *Host call*
- B4.3.3 *RSI_HOST_CALL command*

The members of the RsiHostCall structure are shown in the following table.

Name	Byte offset	Type	Description
imm	`0x0`	UInt16	Immediate value
gprs[0]	`0x8`	Bits64	Registers
gprs[1]	`0x10`	Bits64	Registers
gprs[2]	`0x18`	Bits64	Registers
gprs[3]	`0x20`	Bits64	Registers
gprs[4]	`0x28`	Bits64	Registers
gprs[5]	`0x30`	Bits64	Registers
gprs[6]	`0x38`	Bits64	Registers

Unused bits of the RsiHostCall structure should be zero.

B4.4.3 RsiInterfaceVersion type

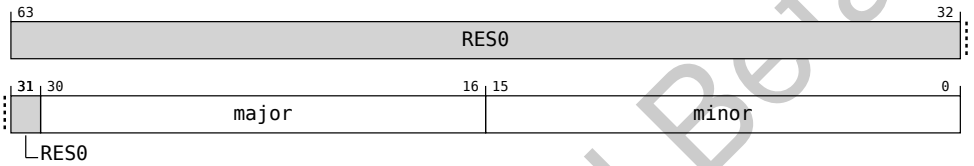
The RsiInterfaceVersion fieldset contains an RSI interface version.

The width of the RsiInterfaceVersion fieldset is 64 bits.

See also:

- B4.1 *RSI version*
- B4.3.9 *RSI_VERSION command*

The fields of the RsiInterfaceVersion fieldset are shown in the following diagram.



The fields of the RsiInterfaceVersion fieldset are shown in the following table.

Name	Bits	Description	Value
minor	15:0	Interface minor version number	UInt16
major	30:16	Interface major version number	UInt15
	63:31	Reserved	Must be zero

B4.4.4 RsiRealmConfig type

The RsiRealmConfig structure contains realm configuration.

The width of the RsiRealmConfig structure is 4096 (`0x1000`) bytes.

See also:

- B4.3.8 *RSI_REALM_CONFIG command*

The members of the RsiRealmConfig structure are shown in the following table.

Name	Byte offset	Type	Description
ipa_width	`0x0`	UInt64	IPA width in bits

Unused bits of the RsiRealmConfig structure should be zero.

B4.4.5 RsiRipas type

The RsiRipas enumeration represents realm IPA state.

The width of the RsiRipas enumeration is 8 bits.

See also:

- A5.4 *RIPAS change*
- B4.3.4 *RSI_IPA_STATE_GET command*

- B4.3.5 *RSI_IPA_STATE_SET command*

The values of the RsiRipas enumeration are shown in the following table.

Encoding	Name	Description
0	RSI_EMPTY	Unused IPA location.
1	RSI_RAM	Private code or data owned by the Realm.

Unused encodings for the RsiRipas enumeration are reserved for use by future versions of this specification.

Chapter B5
Power State Control Interface

This section describes how Power State Control Interface (PSCI) function execution by a Realm execution of SMC instructions is handled.

B5.1 PSCI overview

I_{GBVWX} In this section,

- `rec` refers to the currently executing REC
- `exit` refer to the RecExit object which was provided to the RMI_REC_ENTER command
- `target_rec` refers to the REC identified by an MPIDR value passed to a PSCI function.

I_{GHKCJ} The RMM provides a trusted implementation of parts of the PSCI ABI. This section describes the checks performed by the RMM when a Realm executes a PSCI command, and the internal RMM state changes which result from a successful PSCI command execution. Successful execution by the RMM of some PSCI commands results in a *REC exit due to PSCI*, which allows the Host to perform further processing of the command.

I_{XHDQF} The HVC conduit for PSCI is not supported for Realms.

See also:

- *Arm Power State Coordination Interface (PSCI)* [16]
- A2.3.2 *REC attributes*
- A4.3.7 *REC exit due to PSCI*
- A4.5 *Host call*
- D1.4 *PSCI flows*

B5.2 PSCI version

R_{TFCVF} The RMM must support version >= 1.1 of the Power State Control Interface.

See also:

- B5.3.8 *PSCI_VERSION command*

B5.3 PSCI commands

The following table summarizes the FIDs of commands in the PSCI interface.

FID	Command
0xC4000004	PSCI_AFFINITY_INFO
0x84000002	PSCI_CPU_OFF
0xC4000003	PSCI_CPU_ON
0xC4000001	PSCI_CPU_SUSPEND
0x8400000A	PSCI_FEATURES
0x84000008	PSCI_SYSTEM_OFF
0x84000009	PSCI_SYSTEM_RESET
0x84000000	PSCI_VERSION

B5.3.1 PSCI_AFFINITY_INFO command

Query status of a VPE.

This command causes a REC exit due to PSCI. In response, the Host should provide the target REC (identified by `target_affinity`) by calling RMI_PSCI_COMPLETE.

See also:

- A2.3.2 *REC attributes*
- A4.3.7 *REC exit due to PSCI*
- B3.3.7 *RMI_PSCI_COMPLETE command*
- B5.3.2 *PSCI_CPU_OFF command*
- B5.3.3 *PSCI_CPU_ON command*

B5.3.1.1 Interface

B5.3.1.1.1 Input values

Name	Register	Bits	Type	Description
fid	X0	63:0	UInt64	FID, value `0xC4000004`
target_affinity	X1	63:0	Bits64	This parameter contains a copy of the affinity fields of the MPIDR register
lowest_affinity_leve l	X2	31:0	UInt32	Denotes the lowest affinity level field that is valid in the target_affinity parameter

Unused bits of PSCI_AFFINITY_INFO input values must be zero.

B5.3.1.1.2 Context

The PSCI_AFFINITY_INFO command operates on the following context.

Name	Type	Value	Before	Description
target_rec	RmmRec	`RecFromMpidr( target_affinity)`	false	Target REC

B5.3.1.1.3 Output values

Name	Register	Bits	Type	Description
result	X0	31:0	PsciReturnCode	Command return code

Unused bits of PSCI_AFFINITY_INFO output values must be zero.

B5.3.1.2 Failure conditions

ID	Condition
target_bound	`pre: lowest_affinity_level != 0` `post: result == PSCI_INVALID_PARAMETERS`

ID	Condition
target_match	`pre: !MpidrIsUsed(target_affinity)` `post: result == PSCI_INVALID_PARAMETERS`

B5.3.1.2.1 Failure condition ordering

The PSCI_AFFINITY_INFO command does not have any failure condition orderings.

B5.3.1.3 Success conditions

ID	Condition
runnable	`pre: target_rec.flags.runnable == RUNNABLE` `post: result == PSCI_SUCCESS`
not_runnable	`pre: target_rec.flags.runnable == NOT_RUNNABLE` `post: result == PSCI_OFF`

B5.3.1.4 Footprint

The PSCI_AFFINITY_INFO command does not have any footprint.

B5.3.2 PSCI_CPU_OFF command

Power down the calling core.

This command causes a REC exit due to PSCI.

See also:

- A2.3.2 *REC attributes*
- A4.3.7 *REC exit due to PSCI*
- B5.3.3 *PSCI_CPU_ON command*
- B5.3.4 *PSCI_CPU_SUSPEND command*

B5.3.2.1 Interface

B5.3.2.1.1 Input values

Name	Register	Bits	Type	Description
fid	X0	63:0	UInt64	FID, value `0x84000002`

B5.3.2.1.2 Context

The PSCI_CPU_OFF command operates on the following context.

Name	Type	Value	Before	Description
rec	RmmRec	`CurrentRec()`	false	Current REC

B5.3.2.1.3 Output values

The PSCI_CPU_OFF command does not have any output values.

Following execution of PSCI_CPU_OFF, control does not return to the caller.

B5.3.2.2 Failure conditions

The PSCI_CPU_OFF command does not have any failure conditions.

B5.3.2.3 Success conditions

The PSCI_CPU_OFF command does not have any success conditions.

Following execution of PSCI_CPU_OFF, control does not return to the caller.

B5.3.2.4 Footprint

The PSCI_CPU_OFF command does not have any footprint.

B5.3.3 PSCI_CPU_ON command

Power up a core.

This command causes a REC exit due to PSCI. In response, the Host should provide the target REC (identified by `target_cpu`) by calling RMI_PSCI_COMPLETE.

See also:

- A2.3.2 *REC attributes*
- A4.3.7 *REC exit due to PSCI*
- B3.3.7 *RMI_PSCI_COMPLETE command*
- B5.3.2 *PSCI_CPU_OFF command*
- B5.3.4 *PSCI_CPU_SUSPEND command*
- D1.4.1 *PSCI_CPU_ON flow*

B5.3.3.1 Interface

B5.3.3.1.1 Input values

Name	Register	Bits	Type	Description
fid	X0	63:0	UInt64	FID, value `0xC4000003`
target_cpu	X1	63:0	Bits64	This parameter contains a copy of the affinity fields of the MPIDR register
entry_point_address	X2	63:0	Address	Address at which the core must resume execution
context_id	X3	31:0	UInt32	This parameter is only meaningful to the caller (must be present in X0 of the target PE upon first entry to Non-Secure exception level)

Unused bits of PSCI_CPU_ON input values must be zero.

B5.3.3.1.2 Context

The PSCI_CPU_ON command operates on the following context.

Name	Type	Value	Before	Description
realm	RmmRealm	`CurrentRealm()`	false	Current Realm
target_rec	RmmRec	`RecFromMpidr(target_cpu)`	false	Target REC

B5.3.3.1.3 Output values

Name	Register	Bits	Type	Description
result	X0	31:0	PsciReturnCode	Command return code

Unused bits of PSCI_CPU_ON output values must be zero.

B5.3.3.2 Failure conditions

ID	Condition
entry	`pre: !AddrIsProtected(entry_point_address, realm)` `post: result == PSCI_INVALID_ADDRESS`
mpidr	`pre: !MpidrIsUsed(target_cpu)` `post: result == PSCI_INVALID_PARAMETERS`
runnable	`pre: target_rec.flags.runnable == RUNNABLE` `post: result == PSCI_ALREADY_ON`

B5.3.3.2.1 Failure condition ordering

The PSCI_CPU_ON command does not have any failure condition orderings.

B5.3.3.3 Success conditions

ID	Condition
entry	`target_rec.pc == entry_point_address`
runnable	`target_rec.flags.runnable == RUNNABLE`
sysreg	`// REVISIT: specify sysreg reset values which are applied to rec`

B5.3.3.4 Footprint

ID	Value
runnable	`target_rec.flags.runnable`

B5.3.4 PSCI_CPU_SUSPEND command

Suspend execution on the calling VPE.

This command causes a REC exit due to PSCI.

See also:

- A4.3.7 *REC exit due to PSCI*
- B5.3.2 *PSCI_CPU_OFF command*
- B5.3.3 *PSCI_CPU_ON command*

B5.3.4.1 Interface

B5.3.4.1.1 Input values

Name	Register	Bits	Type	Description
fid	X0	63:0	UInt64	FID, value `0xC4000001`
power_state	X1	31:0	UInt32	Identifier for a specific local state
entry_point_address	X2	63:0	Address	Address at which the core must resume execution
context_id	X3	63:0	UInt64	This parameter is only meaningful to the caller (must be present in X0 upon first entry to Non- Secure exception level)

Unused bits of PSCI_CPU_SUSPEND input values must be zero.

The RMM treats all target power states as suspend requests, and therefore the `entry_point_address` and `context_id` arguments are ignored.

B5.3.4.1.2 Output values

The PSCI_CPU_SUSPEND command does not have any output values.

Following execution of PSCI_CPU_SUSPEND, control does not return to the caller.

B5.3.4.2 Failure conditions

The PSCI_CPU_SUSPEND command does not have any failure conditions.

B5.3.4.3 Success conditions

The PSCI_CPU_SUSPEND command does not have any success conditions.

Following execution of PSCI_CPU_SUSPEND, control does not return to the caller.

B5.3.4.4 Footprint

The PSCI_CPU_SUSPEND command does not have any footprint.

B5.3.5 PSCI_FEATURES command

Query whether a specific PSCI feature is implemented.

See also:

- B5.3.1 *PSCI_AFFINITY_INFO command*
- B5.3.2 *PSCI_CPU_OFF command*
- B5.3.3 *PSCI_CPU_ON command*
- B5.3.4 *PSCI_CPU_SUSPEND command*
- B5.3.6 *PSCI_SYSTEM_OFF command*
- B5.3.7 *PSCI_SYSTEM_RESET command*

B5.3.5.1 Interface

B5.3.5.1.1 Input values

Name	Register	Bits	Type	Description
fid	X0	63:0	UInt64	FID, value `0x8400000A`
psci_func_id	X1	31:0	UInt32	Function ID for a PSCI Function

Unused bits of PSCI_FEATURES input values must be zero.

B5.3.5.1.2 Output values

Name	Register	Bits	Type	Description
result	X0	31:0	PsciReturnCode	Command return code

Unused bits of PSCI_FEATURES output values must be zero.

B5.3.5.2 Failure conditions

The PSCI_FEATURES command does not have any failure conditions.

B5.3.5.3 Success conditions

ID	Condition
func_ok	`pre: psci_func_id is a supported PSCI function.` `post: result == PSCI_SUCCESS`
func_not_ok	`pre: psci_func_id is not a supported PSCI function.` `post: result == PSCI_NOT_SUPPORTED`

B5.3.5.4 Footprint

The PSCI_FEATURES command does not have any footprint.

B5.3.6 PSCI_SYSTEM_OFF command

Shut down the system.

This command causes a REC exit due to PSCI.

See also:

- A2.3.2 *REC attributes*
- A4.3.7 *REC exit due to PSCI*
- B5.3.7 *PSCI_SYSTEM_RESET command*

B5.3.6.1 Interface

B5.3.6.1.1 Input values

Name	Register	Bits	Type	Description
fid	X0	63:0	UInt64	FID, value `0x84000008`

B5.3.6.1.2 Context

The PSCI_SYSTEM_OFF command operates on the following context.

Name	Type	Value	Before	Description
realm	RmmRealm	`CurrentRealm()`	false	Current Realm

B5.3.6.1.3 Output values

The PSCI_SYSTEM_OFF command does not have any output values.

Following execution of PSCI_SYSTEM_OFF, control does not return to the caller.

B5.3.6.2 Failure conditions

The PSCI_SYSTEM_OFF command does not have any failure conditions.

B5.3.6.3 Success conditions

The PSCI_SYSTEM_OFF command does not have any success conditions.

Following execution of PSCI_SYSTEM_OFF, control does not return to the caller.

B5.3.6.4 Footprint

The PSCI_SYSTEM_OFF command does not have any footprint.

B5.3.7 PSCI_SYSTEM_RESET command

Shut down the system.

This command causes a REC exit due to PSCI.

See also:

- A2.3.2 *REC attributes*
- A4.3.7 *REC exit due to PSCI*
- B5.3.6 *PSCI_SYSTEM_OFF command*

B5.3.7.1 Interface

B5.3.7.1.1 Input values

Name	Register	Bits	Type	Description
fid	X0	63:0	UInt64	FID, value `0x84000009`

B5.3.7.1.2 Context

The PSCI_SYSTEM_RESET command operates on the following context.

Name	Type	Value	Before	Description
realm	RmmRealm	`CurrentRealm()`	false	Current Realm

B5.3.7.1.3 Output values

The PSCI_SYSTEM_RESET command does not have any output values.

Following execution of PSCI_SYSTEM_RESET, control does not return to the caller.

B5.3.7.2 Failure conditions

The PSCI_SYSTEM_RESET command does not have any failure conditions.

B5.3.7.3 Success conditions

The PSCI_SYSTEM_RESET command does not have any success conditions.

Following execution of PSCI_SYSTEM_RESET, control does not return to the caller.

B5.3.7.4 Footprint

The PSCI_SYSTEM_RESET command does not have any footprint.

B5.3.8 PSCI_VERSION command

Query the version of PSCI implemented.

B5.3.8.1 Interface

B5.3.8.1.1 Input values

Name	Register	Bits	Type	Description
fid	X0	63:0	UInt64	FID, value `0x84000000`

B5.3.8.1.2 Output values

Name	Register	Bits	Type	Description
result	X0	63:0	PsciInterfaceVersion	Interface version

See also:

- B5.2 *PSCI version*

B5.3.8.2 Failure conditions

The PSCI_VERSION command does not have any failure conditions.

B5.3.8.3 Success conditions

The PSCI_VERSION command does not have any success conditions.

B5.3.8.4 Footprint

The PSCI_VERSION command does not have any footprint.

B5.4 PSCI types

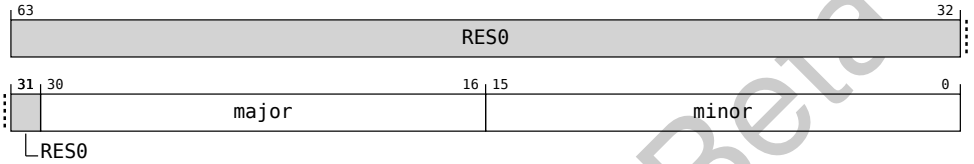
This section defines types which are used in the PSCI interface.

B5.4.1 PsciInterfaceVersion type

The PsciInterfaceVersion fieldset contains an PSCI interface version.

The width of the PsciInterfaceVersion fieldset is 64 bits.

The fields of the PsciInterfaceVersion fieldset are shown in the following diagram.



The fields of the PsciInterfaceVersion fieldset are shown in the following table.

Name	Bits	Description	Value
minor	15:0	Interface minor version number	UInt16
major	30:16	Interface major version number	UInt15
	63:31	Reserved	Must be zero

B5.4.2 PsciReturnCode type

The PsciReturnCode enumeration represents the return code of a PSCI command.

The width of the PsciReturnCode enumeration is 32 bits.

The values of the PsciReturnCode enumeration are shown in the following table.

Encoding	Name	Description
-9	PSCI_INVALID_ADDRESS	Refer to PSCI specification
-8	PSCI_DISABLED	Refer to PSCI specification
-7	PSCI_NOT_PRESENT	Refer to PSCI specification
-6	PSCI_INTERNAL_FAILURE	Refer to PSCI specification
-5	PSCI_ON_PENDING	Refer to PSCI specification
-4	PSCI_ALREADY_ON	Refer to PSCI specification
-3	PSCI_DENIED	Refer to PSCI specification
-2	PSCI_INVALID_PARAMETERS	Refer to PSCI specification
-1	PSCI_NOT_SUPPORTED	Refer to PSCI specification
0	PSCI_SUCCESS	Refer to PSCI specification
1	PSCI_OFF	Refer to PSCI specification

Unused encodings for the PsciReturnCode enumeration are reserved for use by future versions of this specification.

Part C
Types

Chapter C1
RMM types

This section describes types which are used to model the abstract state of the RMM.

C1.1 RmmGranule type

The RmmGranule structure contains attributes of a Granule.

The members of the RmmGranule structure are shown in the following table.

Name	Type	Description
pas	RmmPhysicalAddressSpace	Physical Address Space
state	RmmGranuleState	Lifecycle state

C1.2 RmmGranuleState type

The RmmGranuleState enumeration represents the state of a granule.

The values of the RmmGranuleState enumeration are shown in the following table.

Name	Description
DATA	Realm code or data.
DELEGATED	Delegated for use by the RMM.

Name	Description
RD	Realm Descriptor.
REC	Realm Execution Context.
REC_AUX	Realm Execution Context auxiliary Granule.
RTT	Realm Translation Table.
UNDELEGATED	Not delegated for use by the RMM.

C1.3 RmmHashAlgorithm type

The RmmHashAlgorithm enumeration represents hash algorithm.

The values of the RmmHashAlgorithm enumeration are shown in the following table.

Name	Description
HASH_SHA_256	SHA-256 (*Secure Hash Standard (SHS)* [15])
HASH_SHA_512	SHA-512 (*Secure Hash Standard (SHS)* [15])

C1.4 RmmHostCallPending type

The RmmHostCallPending enumeration represents whether a Host call is pending.

The values of the RmmHostCallPending enumeration are shown in the following table.

Name	Description
HOST_CALL_PENDING	No Host call is pending.
NO_HOST_CALL_PENDING	A Host call is pending.

C1.5 RmmMeasurementDescriptorData type

The RmmMeasurementDescriptorData structure contains data structure used to calculate the contribution to the RIM of a DATA Granule.

The width of the RmmMeasurementDescriptorData structure is 256 (`0x100`) bytes.

See also:

- B3.3.1.4 *RMI_DATA_CREATE extension of RIM*

The members of the RmmMeasurementDescriptorData structure are shown in the following table.

Name	Byte offset	Type	Description
desc_type	`0x0`	Bits8	Measurement descriptor type, value `0x0`
len	`0x8`	UInt64	Length of this data structure in bytes

Name	Byte offset	Type	Description
rim	0x10	RmmRealmMeasurement	Current RIM value
ipa	0x50	Address	IPA at which the DATA Granule is mapped in the Realm
flags	0x58	RmiDataFlags	Flags provided by Host
content	0x60	RmmRealmMeasurement	Hash of contents of DATA Granule, or zero if flags indicate DATA Granule contents are unmeasured

Unused bits of the RmmMeasurementDescriptorData structure must be zero.

C1.6 RmmMeasurementDescriptorRec type

The RmmMeasurementDescriptorRec structure contains data structure used to calculate the contribution to the RIM of a REC.

The width of the RmmMeasurementDescriptorRec structure is 256 (0x100) bytes.

See also:

- B3.3.12.4 *RMI_REC_CREATE extension of RIM*

The members of the RmmMeasurementDescriptorRec structure are shown in the following table.

Name	Byte offset	Type	Description
desc_type	0x0	Bits8	Measurement descriptor type, value 0x1
len	0x8	UInt64	Length of this data structure in bytes
rim	0x10	RmmRealmMeasurement	Current RIM value
content	0x50	RmmRealmMeasurement	Hash of 4KB page which contains REC parameters data structure

Unused bits of the RmmMeasurementDescriptorRec structure must be zero.

C1.7 RmmMeasurementDescriptorRipas type

The RmmMeasurementDescriptorRipas structure contains data structure used to calculate the contribution to the RIM of a RIPAS change.

The width of the RmmMeasurementDescriptorRipas structure is 256 (0x100) bytes.

See also:

- B3.3.18.4 *RMI_RTT_INIT_RIPAS extension of RIM*

The members of the RmmMeasurementDescriptorRipas structure are shown in the following table.

Name	Byte offset	Type	Description
desc_type	0x0	Bits8	Measurement descriptor type, value 0x2

Name	Byte offset	Type	Description
len	0x8	UInt64	Length of this data structure in bytes
rim	0x10	RmmRealmMeasurement	Current RIM value
ipa	0x50	Address	IPA at which the RIPAS change occurred
level	0x58	Int8	RTT level at which the RIPAS change occurred

Unused bits of the RmmMeasurementDescriptorRipas structure must be zero.

C1.8 RmmPhysicalAddressSpace type

The RmmPhysicalAddressSpace enumeration represents the PAS of a Granule.

The values of the RmmPhysicalAddressSpace enumeration are shown in the following table.

Name	Description
NS	Non-secure PAS.
OTHER	PAS other than Non-secure or Realm.
REALM	Realm PAS.

C1.9 RmmPsciPending type

The RmmPsciPending enumeration represents whether a PSCI request is pending.

The values of the RmmPsciPending enumeration are shown in the following table.

Name	Description
NO_PSCI_REQUEST_PENDING	A PSCI request is pending.
PSCI_REQUEST_PENDING	No PSCI request is pending.

C1.10 RmmRealm type

The RmmRealm structure contains attributes of a Realm.

See also:

- A2.1 *Realm*

The members of the RmmRealm structure are shown in the following table.

Name	Type	Description
ipa_width	UInt8	IPA width in bits
measurements	RmmRealmMeasurement[5]	Realm measurements

Name	Type	Description
hash_algo	RmmHashAlgorithm	Algorithm used to compute Realm measurements
rec_index	UInt64	Index of next REC to be created
rtt_base	Address	Realm Translation Table base address
rtt_level_start	Int64	RTT starting level
rtt_num_start	UInt64	Number of physically contiguous starting level RTTs
state	RmmRealmState	Lifecycle state
vmid	Bits16	Virtual Machine Identifier
rpv	Bits512	Realm Personalization Value

C1.11 RmmRealmMeasurement type

The RmmRealmMeasurement type is realm measurement.

The width of the RmmRealmMeasurement type is 512 bits.

C1.12 RmmRealmState type

The RmmRealmState enumeration represents the state of a Realm.

The values of the RmmRealmState enumeration are shown in the following table.

Name	Description
ACTIVE	Eligible for execution.
NEW	Under construction. Not eligible for execution.
SYSTEM_OFF	System has been turned off. Not eligible for execution.

C1.13 RmmRec type

The RmmRec structure contains attributes of a REC.

See also:

- A2.3 *Realm Execution Context*

The members of the RmmRec structure are shown in the following table.

Name	Type	Description
attest_state	RmmRecAttestState	Attestation token generation state
attest_addr	Address	Address of under-construction attestation token
attest_challenge	Bits512	Challenge for under-construction attestation token
aux	Address[16]	Addresses of auxiliary Granules

Name	Type	Description
emulatable_abort	RmmRecEmulatableAbort	Whether the most recent exit from this REC was due to an Emulatable Data Abort
flags	RmmRecFlags	Flags which control REC behavior
gprs	Bits64[32]	General-purpose register values
mpidr	Bits64	MPIDR value
owner	Address	PA of RD of Realm which owns this REC
pc	Address	Program counter value
psci_pending	RmmPsciPending	Whether a PSCI request is pending
state	RmmRecState	Lifecycle state
sysregs	RmmSystemRegisters	EL1 and EL0 system register values
ripas_addr	Address	Next address to be processed in RIPAS change
ripas_top	Address	Top address of pending RIPAS change
ripas_value	RmmRipas	RIPAS value of pending RIPAS change
host_call_pending	RmmHostCallPending	Whether a Host call is pending

C1.14 RmmRecAttestState type

The RmmRecAttestState enumeration represents whether an attestation token generation operation is ongoing on this REC.

The values of the RmmRecAttestState enumeration are shown in the following table.

Name	Description
ATTEST_IN_PROGRESS	An attestation token generation operation is in progress.
NO_ATTEST_IN_PROGRESS	No attestation token generation operation is in progress.

C1.15 RmmRecEmulatableAbort type

The RmmRecEmulatableAbort enumeration represents whether the most recent exit from a REC was due to an Emulatable Data Abort.

The values of the RmmRecEmulatableAbort enumeration are shown in the following table.

Name	Description
EMULATABLE_ABORT	The most recent exit from a REC was due to an Emulatable Data Abort.
NOT_EMULATABLE_ABORT	The most recent exit from a REC was not due to an Emulatable Data Abort.

C1.16 RmmRecFlags type

The RmmRecFlags structure contains REC flags.

The members of the RmmRecFlags structure are shown in the following table.

Name	Type	Description
runnable	RmmRecRunnable	Whether the REC is elgible to run

C1.17 RmmRecRunnable type

The RmmRecRunnable enumeration represents whether a REC is eligible for execution.

The values of the RmmRecRunnable enumeration are shown in the following table.

Name	Description
NOT_RUNNABLE	Not eligible for execution.
RUNNABLE	Eligible for execution.

C1.18 RmmRecState type

The RmmRecState enumeration represents the state of a REC.

The values of the RmmRecState enumeration are shown in the following table.

Name	Description
READY	REC is not currently running.
RUNNING	REC is currently running.

C1.19 RmmRipas type

The RmmRipas enumeration represents realm IPA state.

The values of the RmmRipas enumeration are shown in the following table.

Name	Description
EMPTY	Unused IPA location.
RAM	Private code or data owned by the Realm.

C1.20 RmmRtt type

The RmmRtt structure contains an RTT.

The members of the RmmRtt structure are shown in the following table.

Name	Type	Description
entries	RmmRttEntry[512]	Entries

C1.21 RmmRttEntry type

The RmmRttEntry structure contains attributes of an RTT Entry.

See also:

- A5.5 *Realm Translation Table*

The members of the RmmRttEntry structure are shown in the following table.

Name	Type	Description
addr	Address	Output address
ripas	RmmRipas	RIPAS
state	RmmRttEntryState	State
MemAttr	Bits3	MemAttr
S2AP	Bits2	S2AP
SH	Bits2	SH

C1.22 RmmRttEntryState type

The RmmRttEntryState enumeration represents the state of an RTTE.

The values of the RmmRttEntryState enumeration are shown in the following table.

Name	Description
ASSIGNED	The output address of this RTTE points to a DATA Granule.
DESTROYED	This RTTE cannot be used for the rest of the lifetime of the Realm.
TABLE	The output address of this RTTE points to the next-level RTT.
UNASSIGNED	This RTTE is not associated with any Granule.
VALID_NS	The output address of this RTTE is in the Non- secure PAS. The mapping is valid.

C1.23 RmmRttWalkResult type

The RmmRttWalkResult structure contains result of an RTT walk.

See also:

- A5.5.10 *RTT walk*

The members of the RmmRttWalkResult structure are shown in the following table.

Name	Type	Description
level	Int8	RTT level reached by the walk
rtt_addr	Address	Address of RTT reached by the walk
entry	RmmRttEntry	RTTE reached by the walk

C1.24 RmmSystemRegisters type

The RmmSystemRegisters structure contains EL0 and EL1 system registers.

The members of the RmmSystemRegisters structure are shown in the following table.

Name	Type	Description
SCTLR_EL1	Bits64	System control register

Chapter C2
Generic types

This section defines types which are shared between RMM interfaces and descriptions of RMM abstract state.

See also:

- B3.4 *RMI types*
- B4.4 *RSI types*
- B5.4 *PSCI types*
- Chapter C1 *RMM types*

C2.1 Address type

The Address type is an address.

The width of the Address type is 64 bits.

C2.2 BitsN type

The BitsN type is an N-bit field.

The width of the BitsN type is N bits.

C2.3 IntN type

The IntN type is an signed N-bit integer.

The width of the IntN type is N bits.

C2.4 UIntN type

The UIntN type is an unsigned N-bit integer.

The width of the UIntN type is N bits.

Part D
Usage

Chapter D1
Flows

This section presents flows which explain how the RMM architecture can be used by the Host, and by Realm software.

Note that parts of the sequences below are for illustration only. For example, in the Realm creation flows, the RMI_GRANULE_DELEGATE and RMI_GRANULE_UNDELEGATE commands are called immediately before or after the RMI_X_CREATE and RMI_X_DESTROY commands respectively. An alternative flow would be for the Host to maintain a pool of Granules in the DELEGATED state, from which RMM data structures and Realm data can be allocated on demand.

D1.1 Granule delegation flows

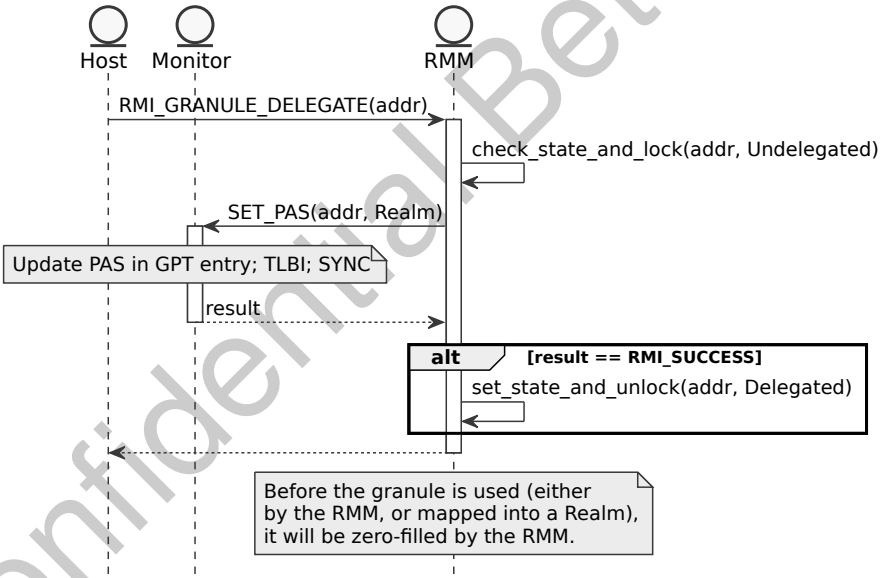
D1.1.1 Granule delegation flow

The following diagram shows how the PAS of a Granule is changed from NS to REALM.

Provisional

See *Arm Architecture Reference Manual Supplement, The Realm Management Extension (RME), for Armv9-A* [2] for example software flows for the operations performed by the Monitor in this flow.

It is anticipated that the Monitor software will be required to use synchronization mechanisms to serialize access to the GPT.



See also:

- A2.2.1 *Granule attributes*
- B3.3.5 *RMI_GRANULE_DELEGATE command*
- D1.1.2 *Granule undelegation flow*

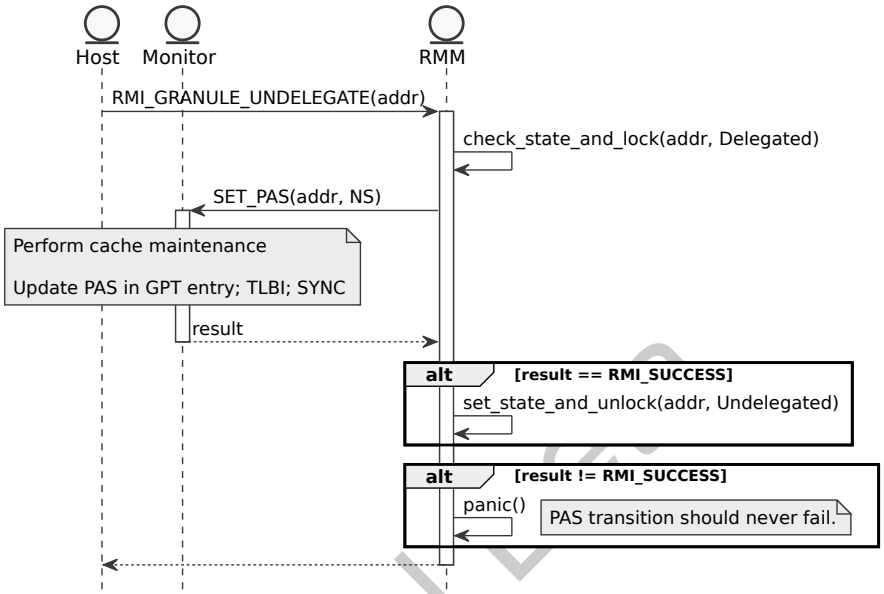
D1.1.2 Granule undelegation flow

The following diagram shows how the PAS of a Granule is changed from REALM to NS.

Provisional

See *Arm Architecture Reference Manual Supplement, The Realm Management Extension (RME), for Armv9-A* [2] for example software flows for the operations performed by the Monitor in this flow.

It is anticipated that the Monitor software will be required to use synchronization mechanisms to serialize access to the GPT.



See also:

- A2.2.1 *Granule attributes*
- B3.3.6 *RMI_GRANULE_UNDELEGATE command*
- D1.1.1 *Granule delegation flow*

D1.2 Realm lifecycle flows

This section contains flows which relate to the Realm lifecycle.

See also:

- A2.1.5 *Realm lifecycle*

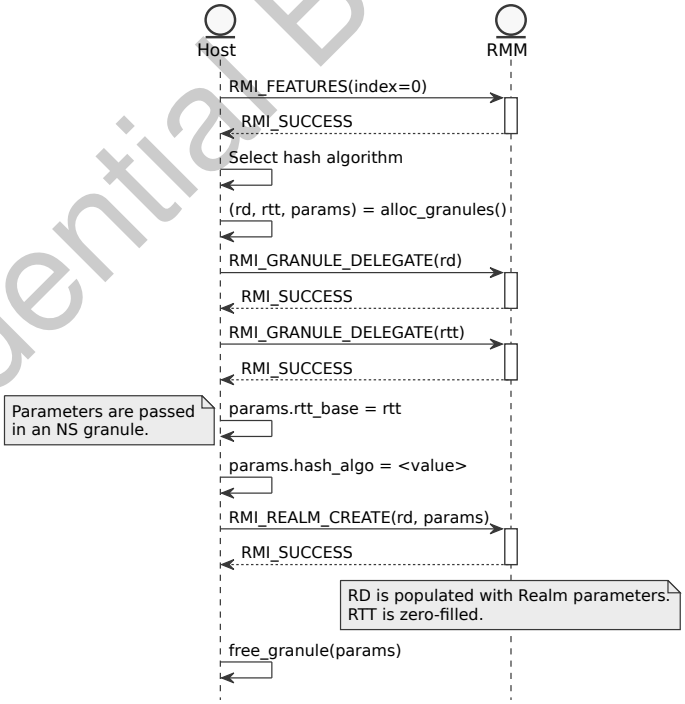
D1.2.1 Realm creation flow

The following diagram shows the flow for creating a Realm.

To create a Realm, the Host must allocate and delegate three Granules:

- `rd` to store the Realm Descriptor
- `rtt` which will be the starting level Realm Translation Table (RTT)

The Host also provides an NS Granule (`params`) containing Realm creation parameters.



See also:

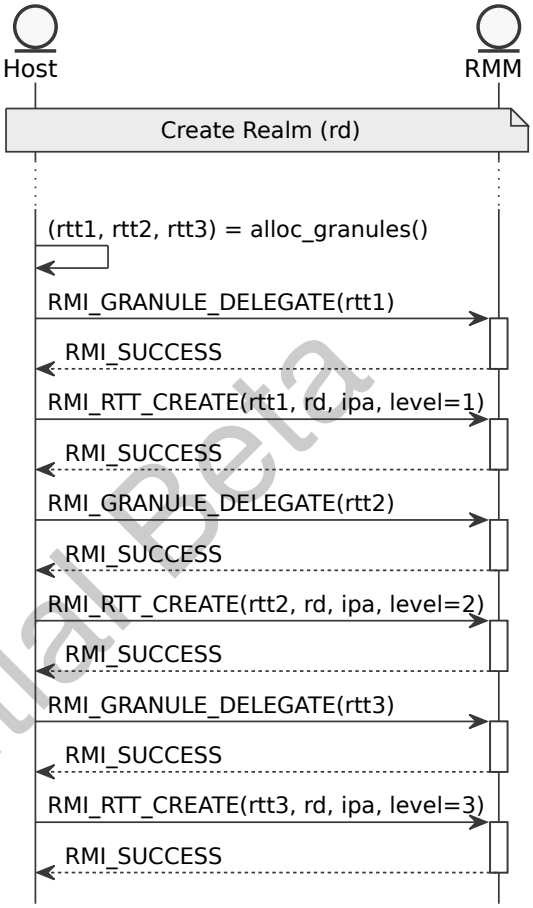
- B3.3.5 *RMI_GRANULE_DELEGATE command*
- B3.3.9 *RMI_REALM_CREATE command*
- D1.2.5 *Realm destruction flow*

D1.2.2 Realm Translation Table creation flow

The following diagram shows the flow for populating the Realm Translation Tables (RTTs).

The starting level Realm Translation Tables (RTTs) are provided at Realm creation time.

Subsequent levels of RTT are added using the RMI_RTT_CREATE command. This can be performed when the state of the Realm is NEW or ACTIVE.



See also:

- Chapter A5 *Realm memory management*
- B3.3.15 *RMI_RTT_CREATE command*
- D1.2.1 *Realm creation flow*
- D1.2.3 *Initialize memory of New Realm flow*

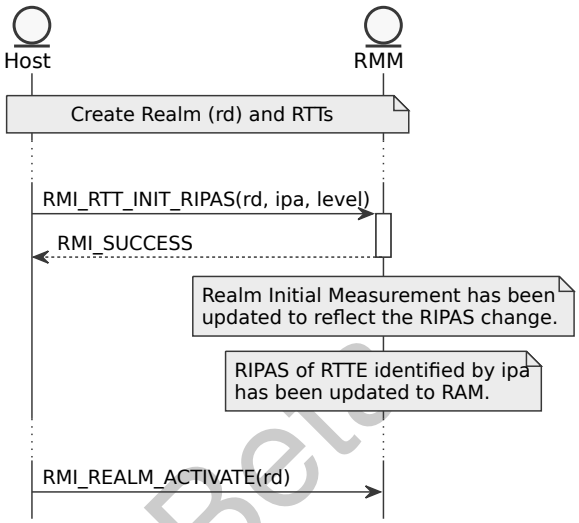
D1.2.3 Initialize memory of New Realm flow

Immediately following Realm creation, every page in the Protected IPA space has its RIPAS set to EMPTY. There are two ways in which the Host can set the RIPAS of a given page of Protected IPA space to RAM:

1. Change the RIPAS by executing RMI_RTT_INIT_RIPAS, but do not populate the contents of the page. The RIM is extended to reflect the RIPAS change.
2. Populate the page with contents provided by the Host. The RIPAS is changed to RAM, and the RIM is extended to reflect the contents added by the Host.

Once the Host has performed either of these actions for a given page of Protected IPA space, that page cannot be further modified prior to Realm activation.

The following diagram shows the flow for initializing the RIPAS without providing contents.

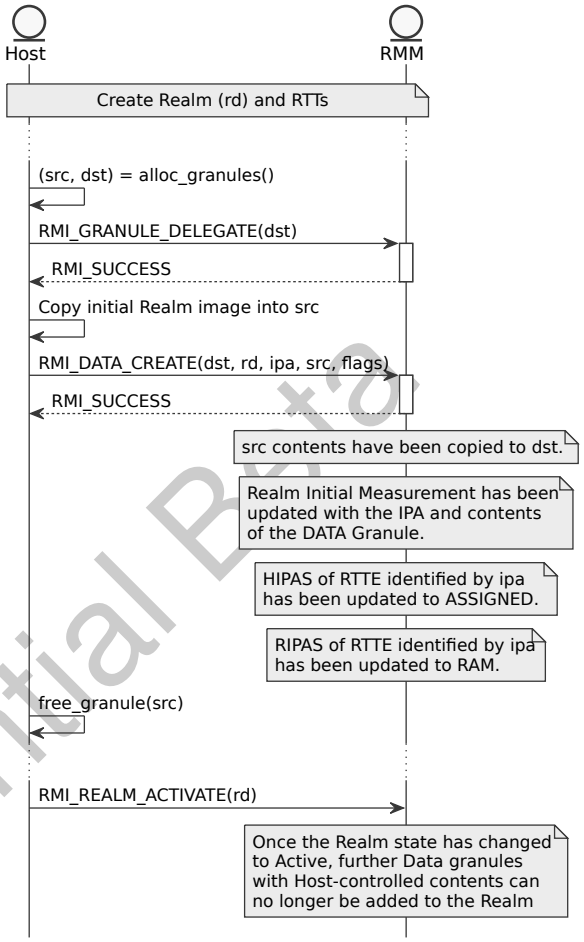


The following diagram shows the flow for populating the page with contents provided by the Host.

To do this, the Host must:

- Delegate a destination Granule (`dst`).
- Provide an NS Granule (`src`), whose contents will be copied into the destination Granule.
- Provide an NS Granule (`params`), which contains parameters. These include the Protected IPA at which the `dst` Granule will be mapped in the Realm's IPA space.
- Ensure that the level 3 RTT which contains the RTTE identified by the Protected IPA has been created.

Once the Data Granule has been created, the `src` and `params` Granules can be reallocated by the Host.



See also:

- A2.2.1 *Granule attributes*
- A5.2.2 *Realm IPA state*
- A7.1.1 *Realm Initial Measurement*
- B3.3.1 *RMI_DATA_CREATE command*
- B3.3.5 *RMI_GRANULE_DELEGATE command*
- B3.3.18 *RMI_RTT_INIT_RIPAS command*
- D1.2.1 *Realm creation flow*
- D1.2.2 *Realm Translation Table creation flow*
- D1.2.5 *Realm destruction flow*

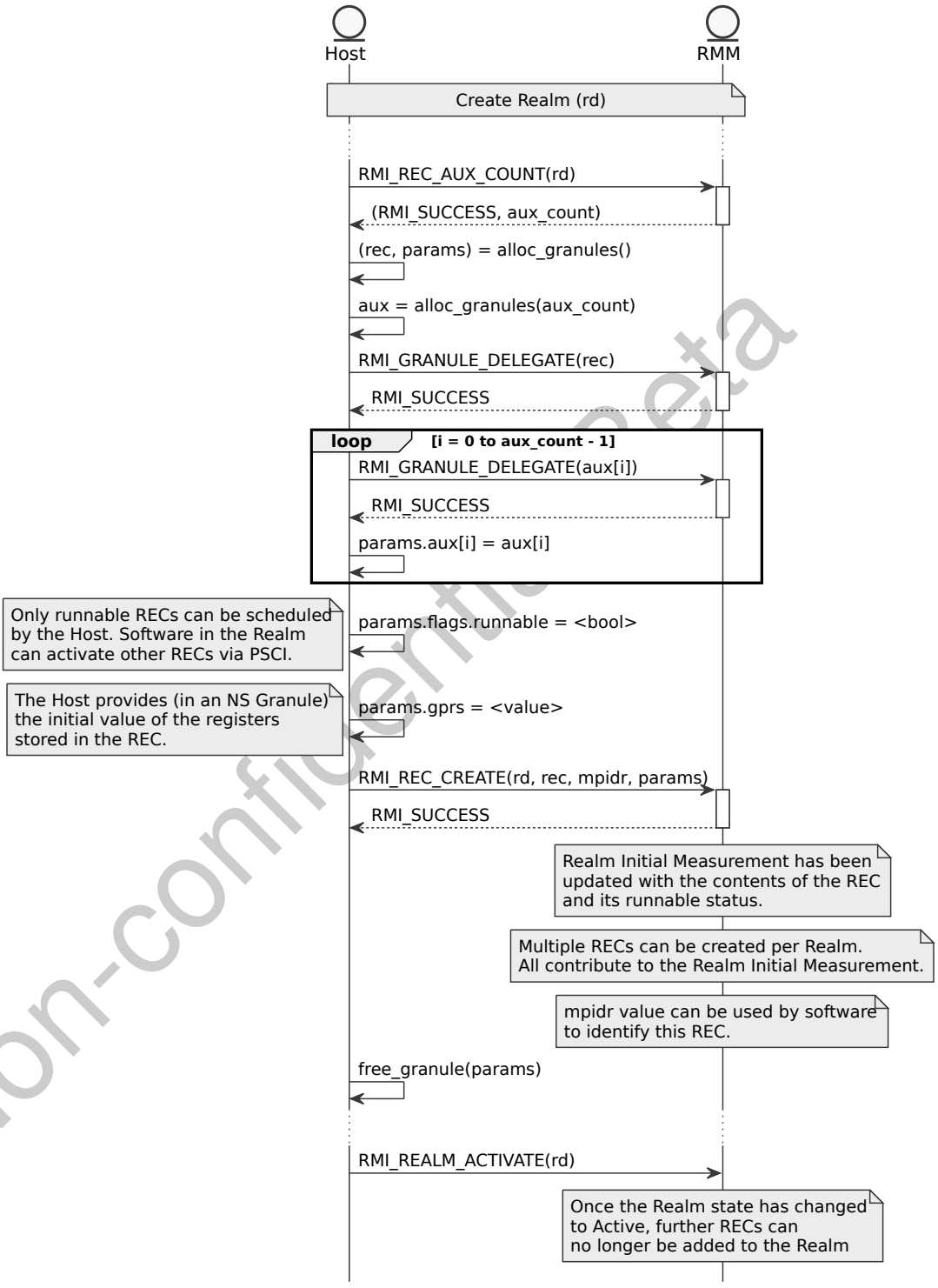
D1.2.4 REC creation flow

The following diagram shows the flow for creating a REC during Realm creation.

To create a REC, the Host must:

- Delegate a destination Granule (`rec`).
- Query the number of auxiliary Granules required, by calling RMI_REC_AUX_COUNT
- Delegate the required number of auxiliary Granules (`aux`)
- Provide auxiliary Granule addresses, register values and REC activation status in an NS Granule (`params`).

Once the REC has been created, the `params` Granule can be reallocated by the Host.



See also:

- B3.3.5 *RMI_GRANULE_DELEGATE command*
- B3.3.11 *RMI_REC_AUX_COUNT command*
- B3.3.12 *RMI_REC_CREATE command*
- D1.2.1 *Realm creation flow*
- D1.2.5 *Realm destruction flow*

D1.2.5 Realm destruction flow

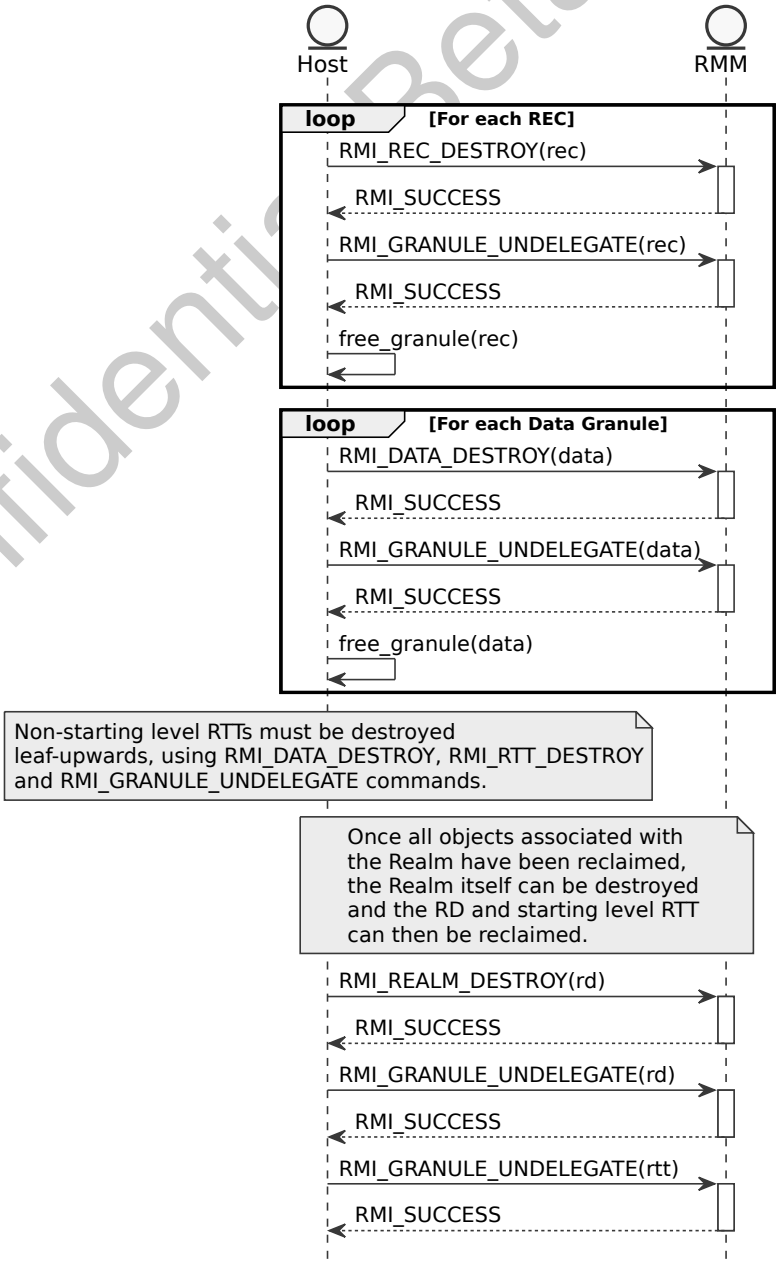
The following diagram shows the flow for destroying a Realm.

To destroy a Realm, the Host must first make the Realm non-live. This is done by destroying (in any order) the objects which are associated with the Realm:

- Data Granules
- RECs
- RTTs

Finally, the Realm itself can be destroyed.

Once each of these objects has been destroyed, the corresponding Granules can be undelegated and reallocated by the Host.



See also:

- A2.1.4 *Realm liveness*
- B3.3.3 *RMI_DATA_DESTROY command*
- B3.3.6 *RMI_GRANULE_UNDELEGATE command*
- B3.3.10 *RMI_REALM_DESTROY command*
- B3.3.13 *RMI_REC_DESTROY command*
- D1.2.1 *Realm creation flow*

D1.3 Realm exception model flows

This section contains flows which relate to the Realm exception model.

See also:

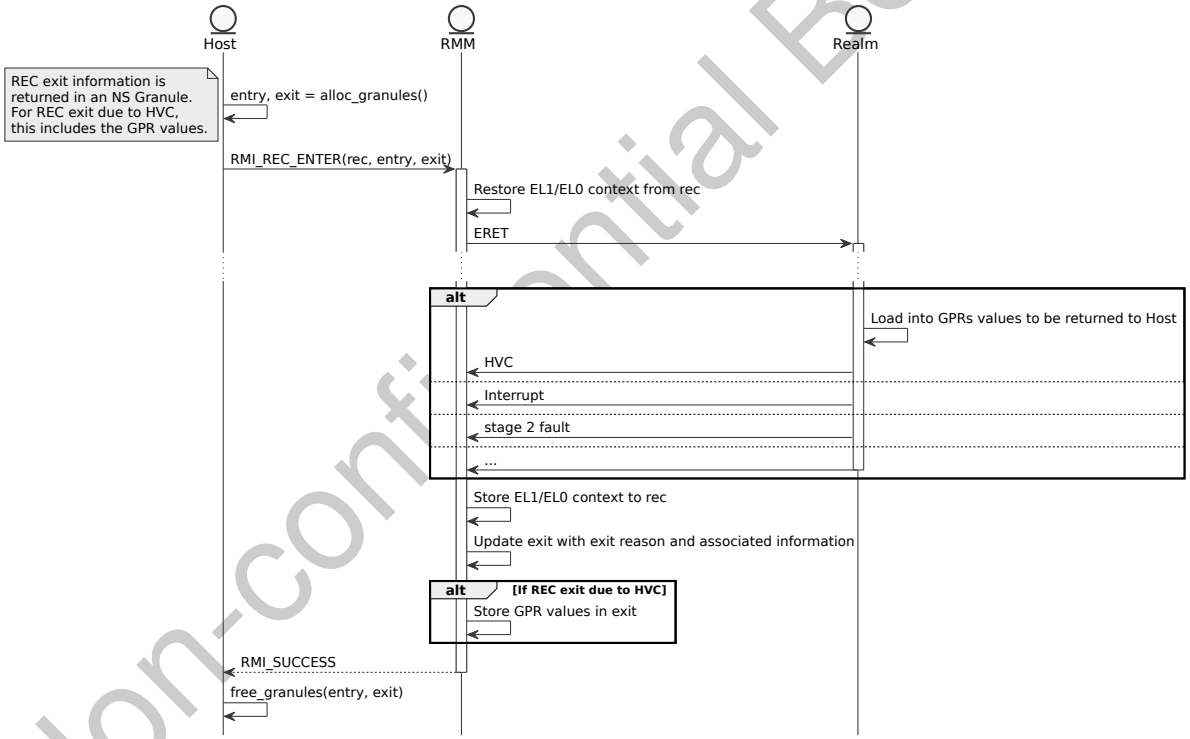
- Chapter A4 *Realm exception model*

D1.3.1 Realm entry and exit flow

The following diagram shows how a Realm is executed, and illustrates the different reasons for exiting the Realm and returning control to the Host.

A REC is entered using the RMI_REC_ENTER command. The parameters to this command include:

- a *RecEntry object*, which is a data structure used to pass values from the Host to the RMM on REC entry
- a *RecExit object*, which is a data structure used to pass values from the RMM to the Host on REC exit



See also:

- Chapter A4 *Realm exception model*
- D1.3.2 *Host call flow*
- D1.3.3 *REC exit due to Data Abort fault flow*
- D1.3.4 *MMIO emulation flow*

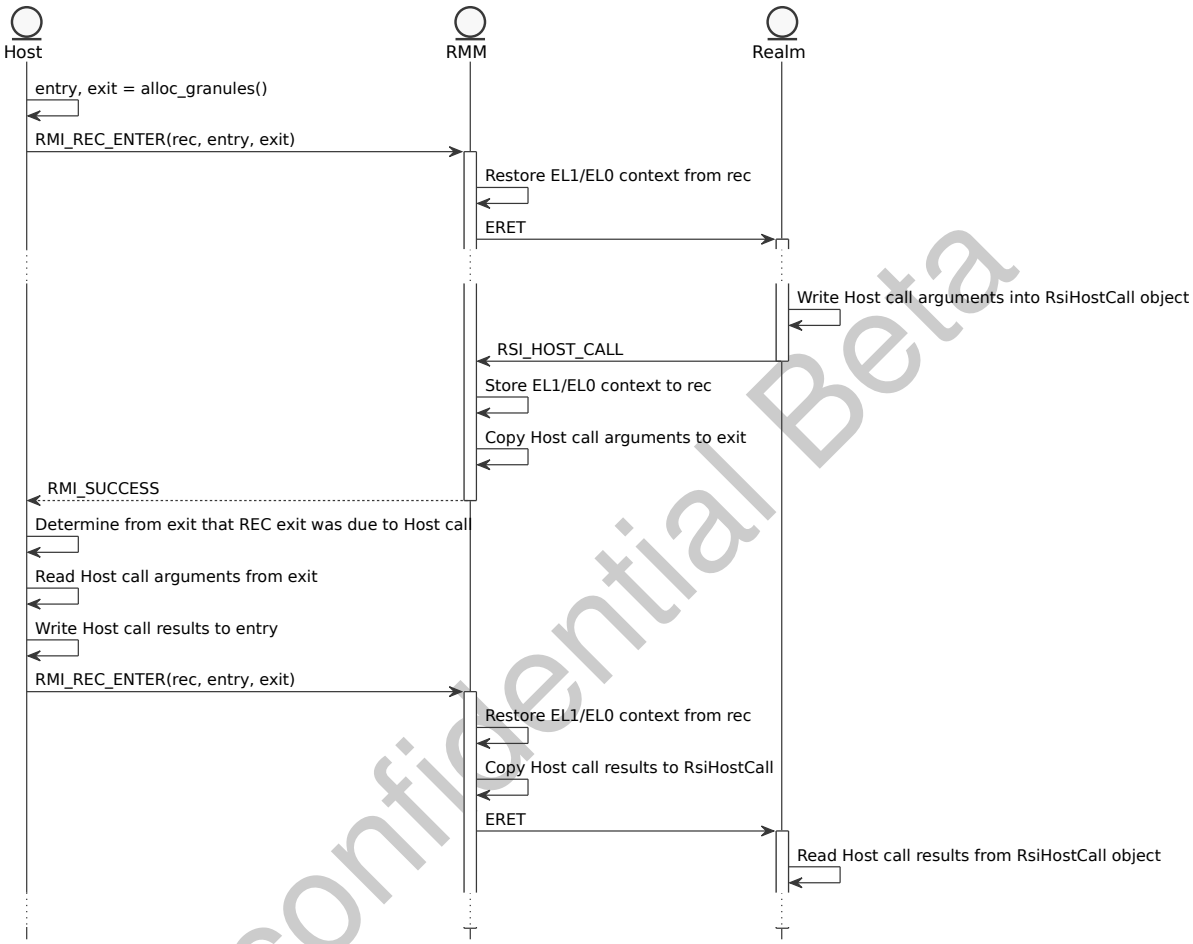
D1.3.2 Host call flow

The following diagram shows how software executing inside the Realm can voluntarily yield control back to the Host by making a Host call.

A REC is entered using the RMI_REC_ENTER command. The parameters to this command include:

- a *RecEntry object*, which is a data structure used to pass values from the Host to the RMM on REC entry
- a *RecExit object*, which is a data structure used to pass values from the RMM to the Host on REC exit

On execution of RSI_HOST_CALL, arguments are copied from the RsiHostCall object in Realm memory into the RecExit object in NS memory. On the subsequent RMI_REC_ENTER, return values are copied from the RecEntry object in NS memory into the RsiHostCall object in Realm memory.



See also:

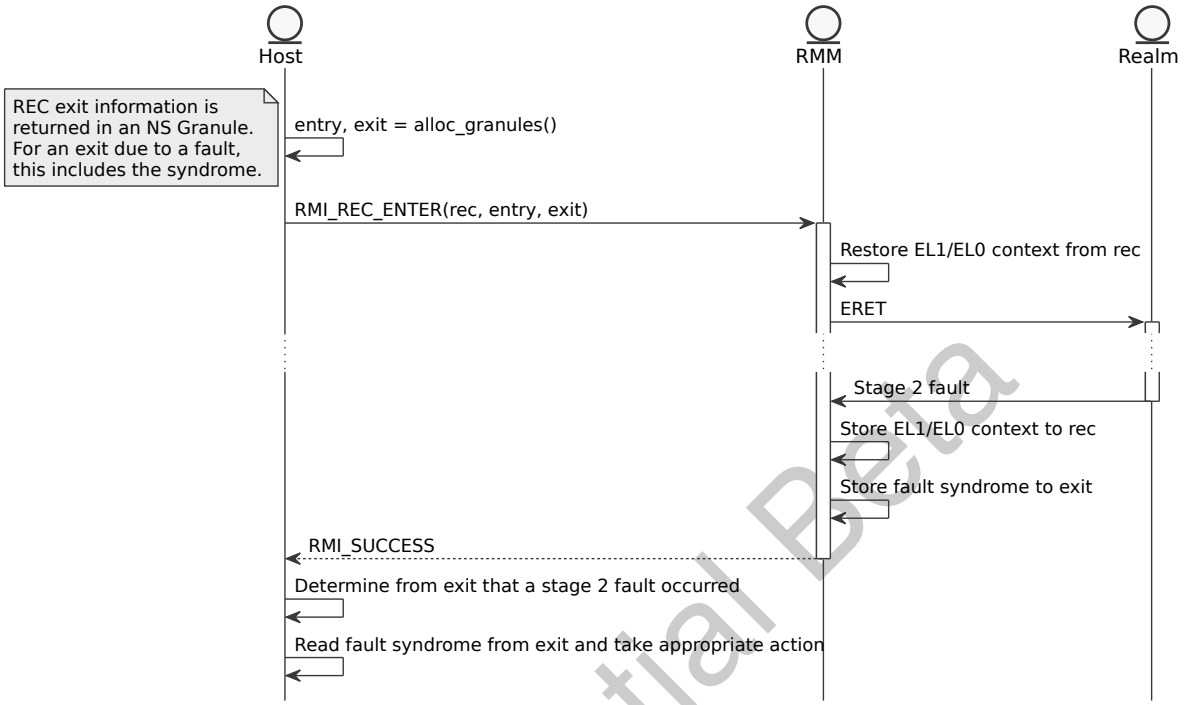
- A4.5 *Host call*

D1.3.3 REC exit due to Data Abort fault flow

The following diagram shows how a Data Abort due to a Realm access is taken to the Host.

A REC is entered using the RMI_REC_ENTER command. The parameters to this command include:

- a *RecEntry object*, which is a data structure used to pass values from the Host to the RMM on REC entry
- a *RecExit object*, which is a data structure used to pass values from the RMM to the Host on REC exit

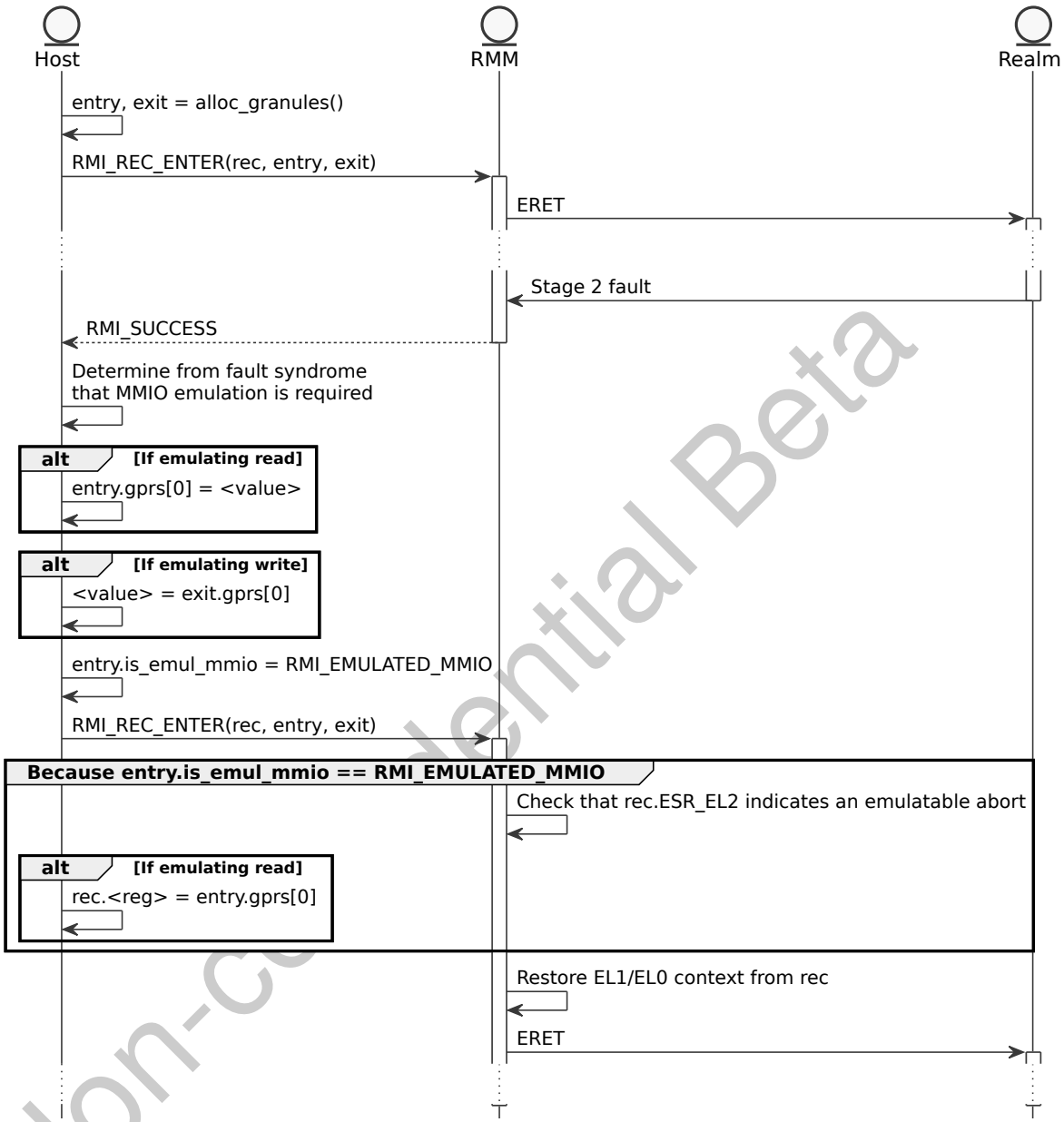


See also:

- Chapter A4 *Realm exception model*

D1.3.4 MMIO emulation flow

The following diagram shows how an MMIO access by a Realm can be emulated by the Host.



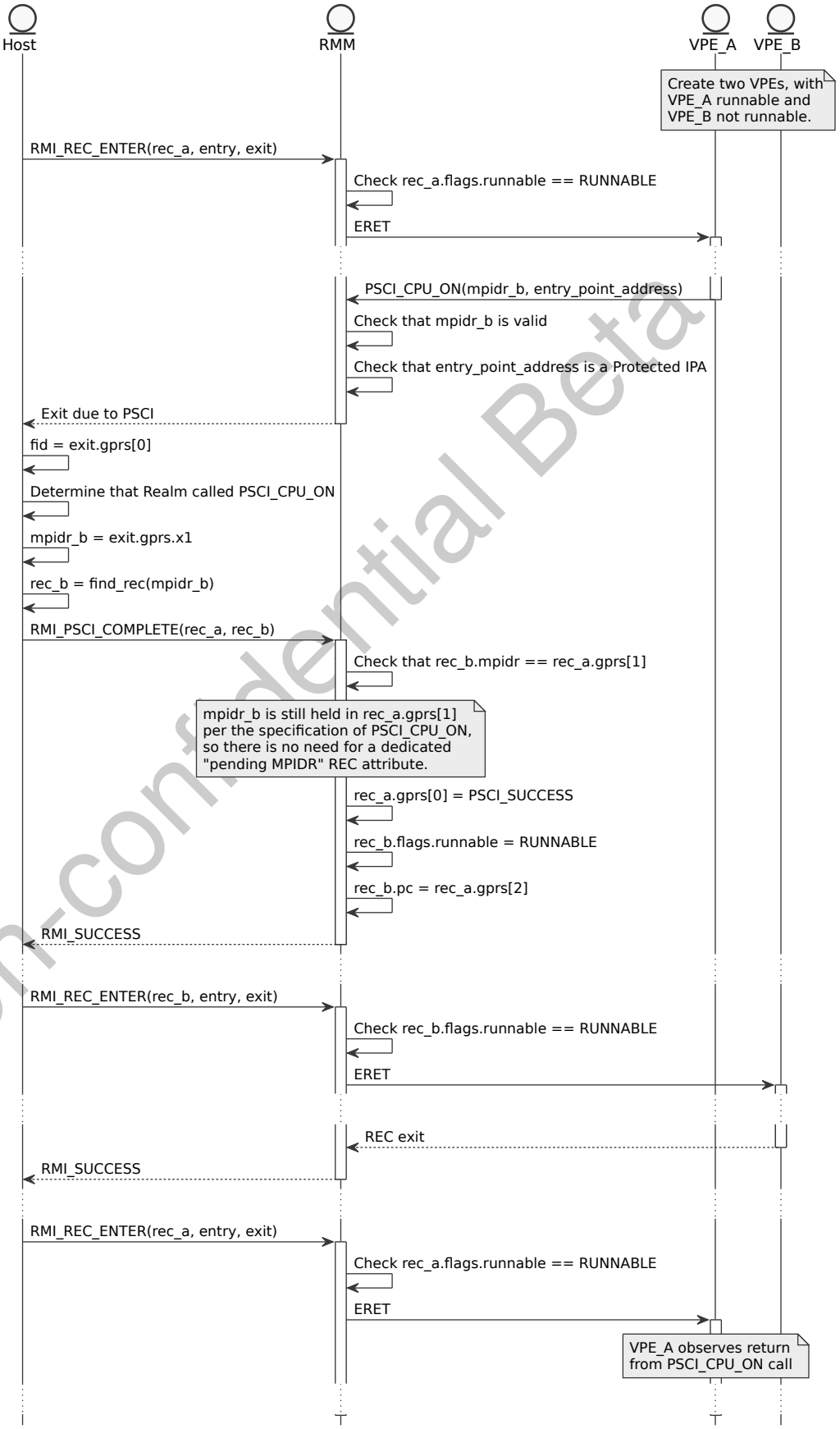
See also:

- Chapter A4 *Realm exception model*

D1.4 PSCI flows

D1.4.1 PSCI_CPU_ON flow

The following diagram shows how one Realm VPE can set the “runnable” flag in another Realm VPE by executing PSCI_CPU_ON.



See also:

- B3.3.7 *RMI_PSCI_COMPLETE command*
- B5.3.3 *PSCI_CPU_ON command*

D1.5 Realm memory management flows

This section contains flows which relate to management of Realm memory.

See also:

- Chapter A5 *Realm memory management*

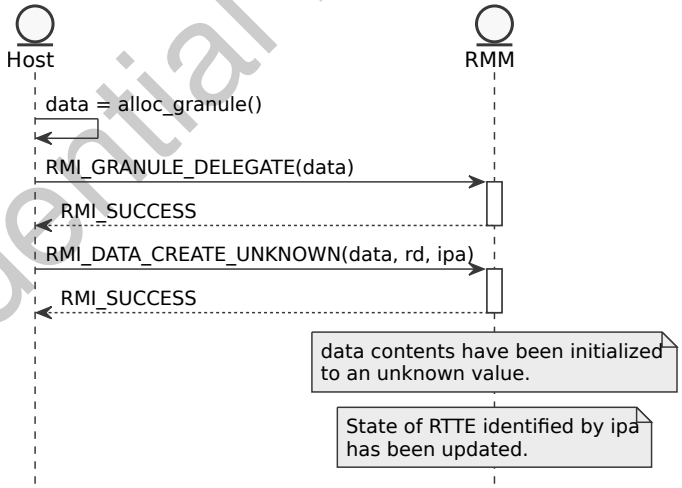
D1.5.1 Add memory to Active Realm flow

The following diagram shows the flow for adding memory to a Realm whose state is ACTIVE.

To add memory to a Realm whose state is ACTIVE, the Host must:

- Delegate a destination Granule (`dst`).
- Specify the Protected IPA at which the `dst` Granule will be mapped in the Realm's IPA space.
- Ensure that the level 3 RTT which contains the RTTE identified by the Protected IPA has been created.
- Ensure that the RIPAS of the Protected IPA is RAM.

Once a given Protected IPA has been populated with unknown content, it cannot be repopulated.

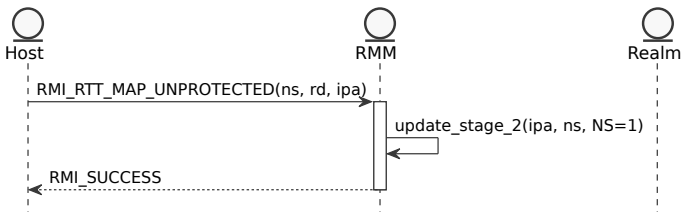


See also:

- A2.1.5 *Realm lifecycle*
- Chapter A5 *Realm memory management*
- B3.3.2 *RMI_DATA_CREATE_UNKNOWN command*
- B3.3.5 *RMI_GRANULE_DELEGATE command*

D1.5.2 NS memory flow

The following diagram describes how NS memory can be mapped into a Realm.



See also:

- Chapter A5 *Realm memory management*

- B3.3.19 *RMI_RTT_MAP_UNPROTECTED command*
- B3.3.22 *RMI_RTT_UNMAP_UNPROTECTED command*

D1.5.3 RIPAS change flow

The following diagram describes how a Realm requests a RIPAS change, and how that request is handled by the Host.

- The Realm calls RSI_IPA_STATE_SET to request a RIPAS change for IPA range `[base, base + size)`.
- This causes a REC exit due to RIPAS change pending.

If the Host accepts the RIPAS change request:

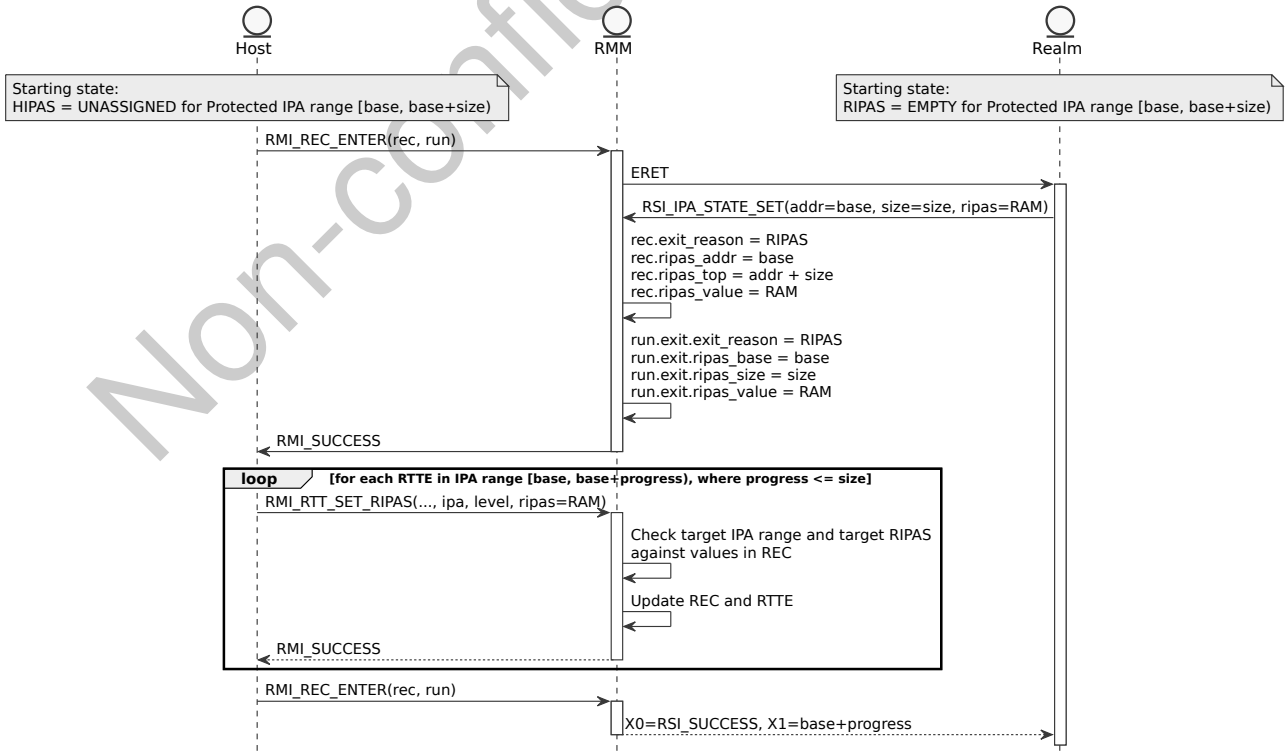
- The Host reads the region base address and size from the RecExit object.
- The Host calls RMI_RTT_SET_RIPAS for each RTTE within the target IPA region.
- The Host calls RMI_REC_ENTER to re-enter the REC.
- The Realm receives a response of ACCEPT from RSI_IPA_STATE_SET.
- The Realm observes that the RIPAS of the target IPA region has changed.

If the Host rejects the RIPAS change request:

- The Host calls RMI_REC_ENTER to re-enter the REC.
- The Realm receives a response of REJECT from RSI_IPA_STATE_SET.
- The Realm observes that the RIPAS of the target IPA region has not changed.

If the Host partially applies the RIPAS change:

- The Host reads the region base address and size from the RecExit object.
- The Host calls RMI_RTT_SET_RIPAS for a subset of RTTEs within the target IPA region.
- The Host calls RMI_REC_ENTER to re-enter the REC.
- RMI_REC_ENTER fails with error code RMI_ERROR_REC.



See also:

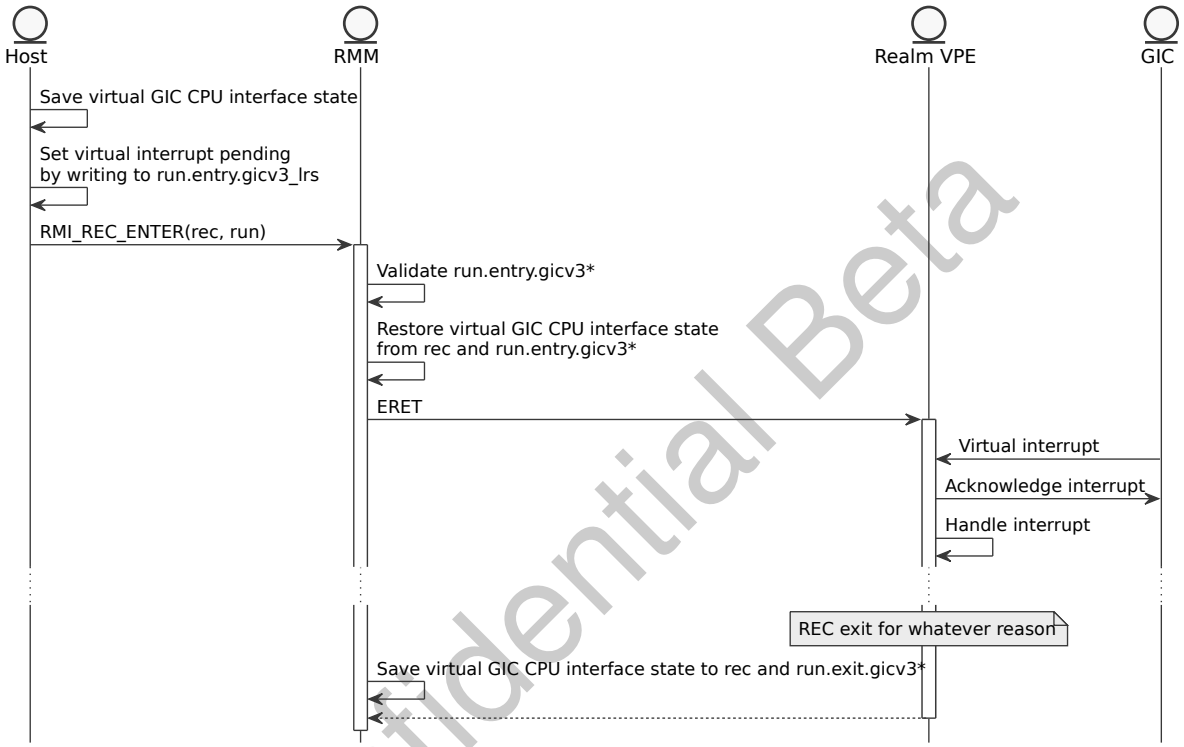
- A5.4 *RIPAS change*

- B3.3.14 *RMI_REC_ENTER command*
- B3.3.21 *RMI_RTT_SET_RIPAS command*
- B4.3.5 *RSI_IPA_STATE_SET command*
- D2.2 *Realm shared memory protocol flow*

D1.6 Realm interrupts and timers flows

D1.6.1 Interrupt flow

The following diagram shows how a virtual interrupt is injected into a Realm by the Host.

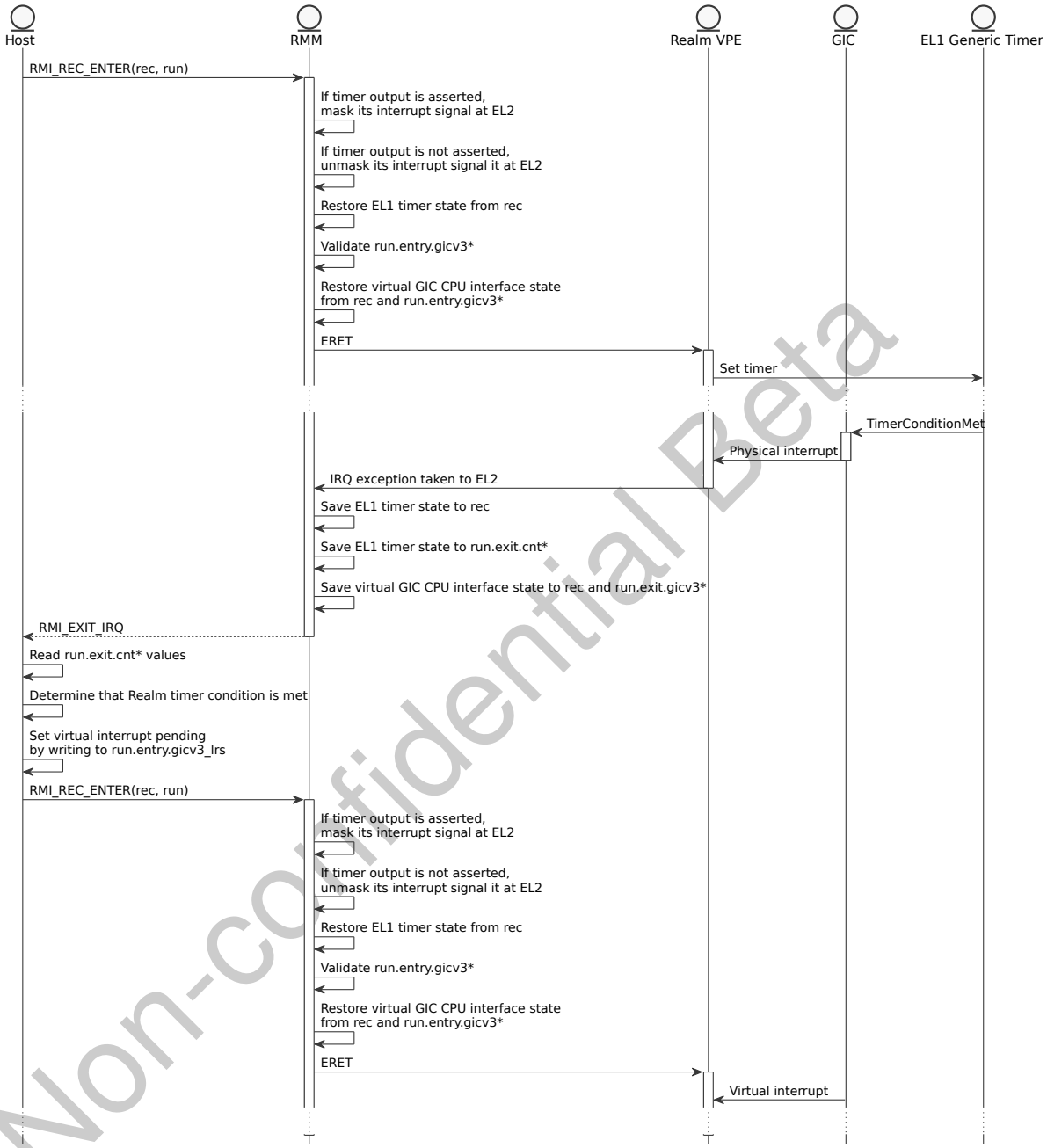


See also:

- A6.1 *Realm interrupts*

D1.6.2 Timer interrupt delivery flow

The following diagram shows how a timer interrupt is delivered to and handled by a Realm.



See also:

- A6.2 *Realm timers*

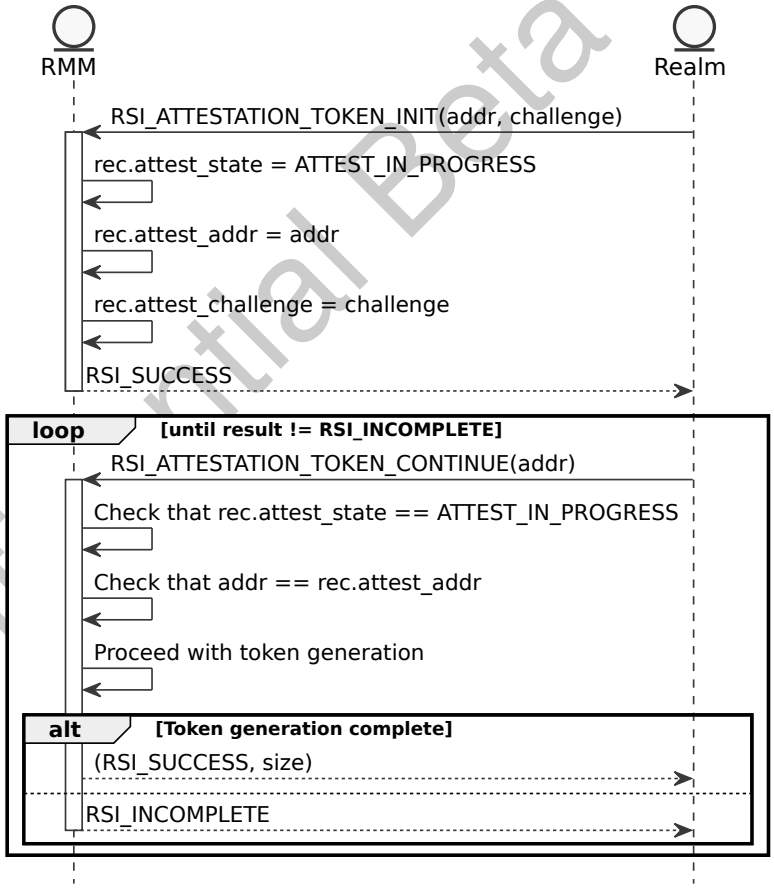
D1.7 Realm attestation flows

D1.7.1 Attestation token generation flow

The following diagram shows the flow for a Realm to obtain an attestation token.

The Realm first calls RSI_ATTESTATION_TOKEN_INIT, providing the address where the attestation token will be written, and a challenge value.

The Realm then calls RSI_ATTESTATION_TOKEN_CONTINUE, providing the same address. This command is called in a loop, until the result is not RSI_INCOMPLETE.



See also:

- A7.2.2 *Attestation token generation*
- B4.3.1 *RSI_ATTESTATION_TOKEN_CONTINUE command*
- B4.3.2 *RSI_ATTESTATION_TOKEN_INIT command*

D1.7.2 Handling interrupts during attestation token generation flow

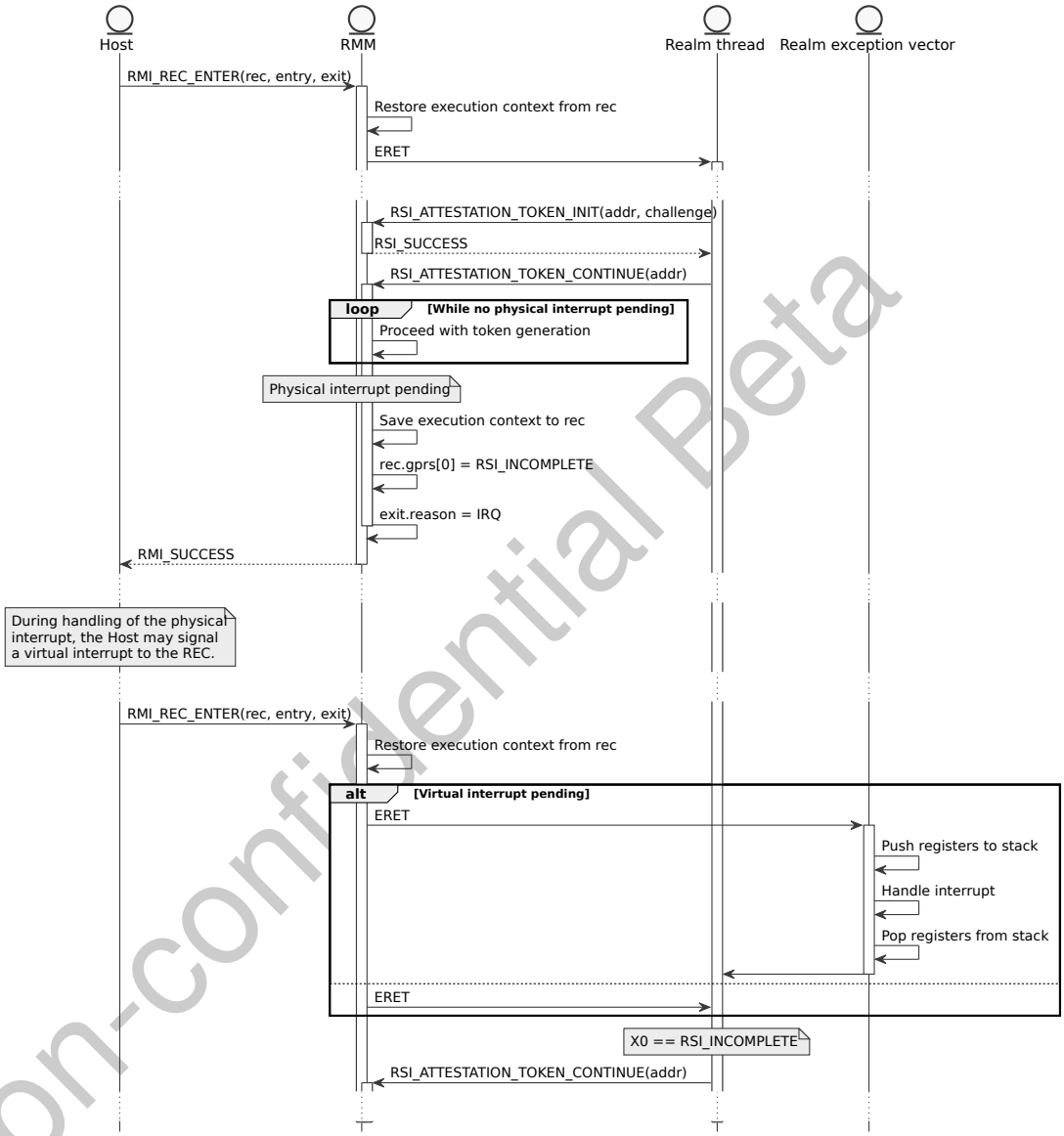
The following diagram shows how interrupts are handled during generation of an attestation token.

If the RMM detects that a physical interrupt is pending during execution of RSI_ATTESTATION_TOKEN_CONTINUE, it saves the execution context to the REC, and performs a REC exit due to IRQ.

During handling of the IRQ, the Host may signal a virtual interrupt to the REC.

On the next entry to the REC, if a virtual interrupt is pending, it is taken to the REC's exception vector.

Whether or not a virtual interrupt was taken, on return to the original thread, the REC determines that X0 is RSI_INCOMPLETE, and therefore calls RSI_ATTESTATION_TOKEN_CONTINUE again.



See also:

- A4.3.5 *REC exit due to IRQ*
- A6.1 *Realm interrupts*
- A7.2.2 *Attestation token generation*
- B4.3.1 *RSI_ATTESTATION_TOKEN_CONTINUE command*
- B4.3.2 *RSI_ATTESTATION_TOKEN_INIT command*
- D1.3.1 *Realm entry and exit flow*

Chapter D2
Realm shared memory protocol

This section describes a protocol for management of memory which is shared between a Realm and the Host. This protocol makes use of the primitives described in this specification. However, the protocol itself is not part of the RMM architecture. Use of this protocol is subject to a contract between the Realm and Host software agents.

Provisional

Arm plans to publish a standard interface via which a Realm can discover whether the Host supports this protocol.

See also:

- Chapter A5 *Realm memory management*

D2.1 Realm shared memory protocol description

The Host agrees to provide the Realm with a certain amount of memory. This memory is referred to below as the Realm's "memory footprint".

The memory footprint is described to the Realm, for example via firmware tables. The Realm can choose, at any point during its execution, how much of its memory footprint is protected (accessible only to the Realm) and how much is shared with the Host.

Realm software treats the most significant IPA bit as a "protection attribute" bit. This means that for every Protected IPA (in which the most significant bit is `'0'`), there exists a corresponding Unprotected IPA alias, which is generated by setting the most significant bit to `'1'`.

The choice of whether a given page is protected or shared at a given time is expressed by setting the RIPAS of the Protected IPA:

- If the RIPAS of the Protected IPA is RAM, the page is protected and access to the Unprotected IPA alias causes a Synchronous External Abort taken to the Realm.
- If the RIPAS of the Protected IPA is EMPTY, the page is shared and access to the Unprotected IPA alias does not cause a Synchronous External Abort taken to the Realm.

The initial RIPAS for every page in the Realm's memory footprint is described to the Realm, for example via firmware tables. The Host agrees that during Realm execution, it will accept a RIPAS change request on any page within the Realm's memory footprint.

See also:

- A5.2.1 *Realm IPA space*
- A5.2.2 *Realm IPA state*
- A5.4 *RIPAS change*

D2.2 Realm shared memory protocol flow

The following diagram illustrates how the protocol is used to set up and tear down a shared memory buffer.

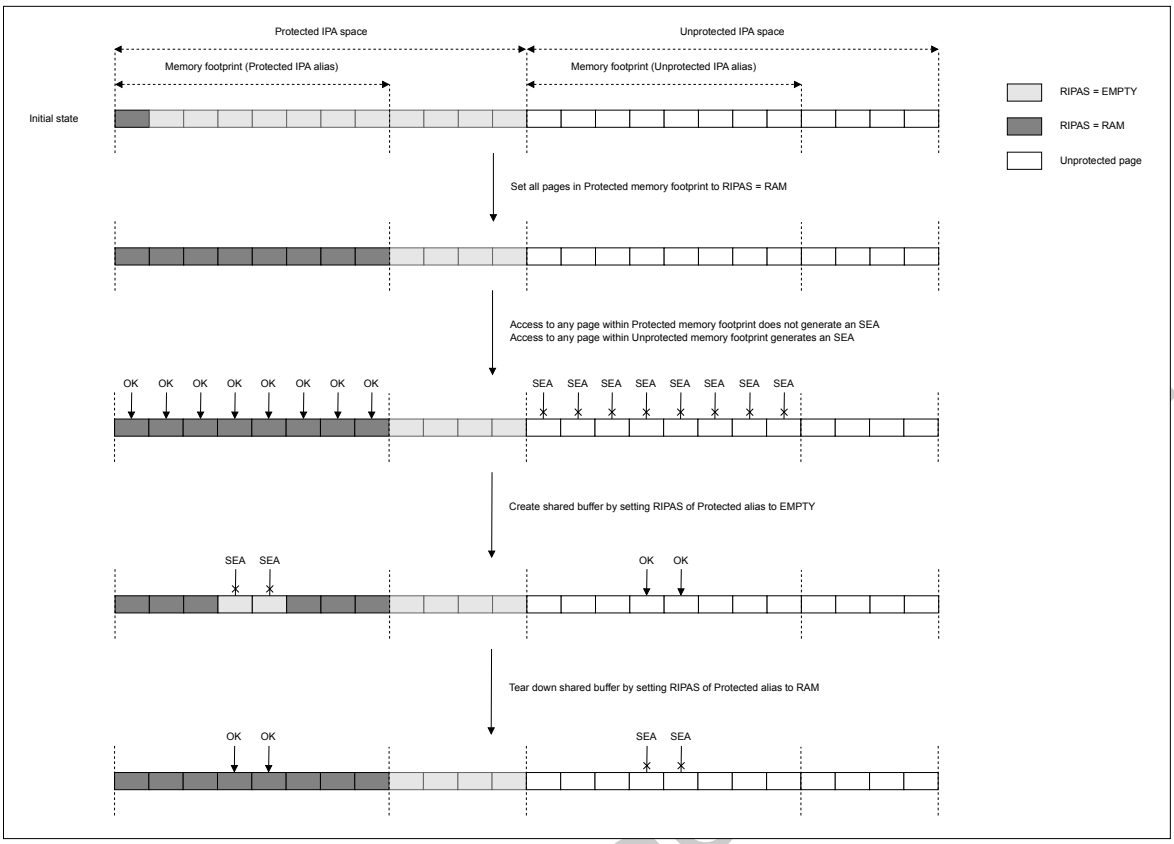


Figure D2.1: Realm shared memory protocol flow

See also:

- D1.5.3 *RIPAS change flow*

Glossary

ASL

Arm Specification Language
Language used to express pseudocode implementations. Formal language definition can be found in *Arm Specification Language Reference Manual* [14].

CBOR

Concise Binary Object Representation

CCA

Confidential Compute Architecture

CCA platform

All hardware and firmware components which are involved in delivering the CCA security guarantee. See *Arm CCA Security model* [4].

CDDL

Concise Data Definition Language

COSE

CBOR Object Signing and Encryption

EAT

Entity Attestation Token

FID

Function Identifier

GIC

Generic Interrupt Controller
See *Arm Generic Interrupt Controller (GIC) Architecture Specification version 3 and version 4* [5]

GPF

Granule Protection Fault

GPT

Granule Protection Table
Table which determines the Physical Address Space of each Granule.

HIPAS

Host IPA state

Host

Software executing in Non-secure Security state which manages resources used by Realms

IAK

Initial Attestation Key Key used to sign the CCA platform attestation token.

IPA

Intermediate Physical Address
Address space visible to software executing at EL1 in the Realm.

IPI

Inter-processor interrupt

IRI

Interrupt Routing Infrastructure
A subset of the components which make up the GIC.

ITS

Interrupt Translation Service
A service provided by the GIC.

MMIO

Memory-mapped I/O

MPIDR

Multiprocessor Affinity Register

NS

Non-secure

PAS

Physical Address Space

PE

Processing Element

PMU

Performance Monitor Unit

PSCI

Power State Control Interface
See *Arm Power State Coordination Interface (PSCI)* [16]

RAK

Realm Attestation Key Key used to sign the Realm attestation token.

RD

Realm Descriptor
Object which stores attributes of a Realm.

Realm

A protected execution environment

REC

Realm Execution Context
Object which stores PE state associated with a thread of execution within a Realm.

REM

Realm Extensible Measurement Measurement value which can be extended during the lifetime of a Realm.

RHA

Realm Hash Algorithm

RIM

Realm Initial Measurement Measurement of the state of a Realm at the time of activation.

RIPAS

Realm IPA state

RMI

Realm Management Interface The ABI exposed by the RMM for use by the Host.

RMM

Realm Management Monitor

RNVS

Root Non-volatile Storage

RPV

Realm Personalization Value

RSI

Realm Services Interface The ABI exposed by the RMM for use by the Realm.

RTT

Realm Translation Table
Object which describes the IPA space of a Realm.

RTTE

Realm Translation Table Entry

SEA

Synchronous External Abort

SGI

Software Generated Interrupt

SMCCC

SMC Calling Convention
See *Arm SMC Calling Convention* [13]

SPM

Secure Partition Manager

TA

Trusted Application

TOS

Trusted OS

VMM

Virtual Machine Monitor

VMSA

Virtual Memory System Architecture

VPE

Virtual Processing Element

Wiping

An operation which changes the value of a memory location from *X* to *Y*, such that the value *X* cannot be determined from the value *Y*