Arm® RMM Architecture Compliance

Revision: r0p1

Command Test Scenario

Issue 01
PJDOC-1505342170-664114

Arm CCA-RMM-ACS Command Test Scenario Document

Release information

Document history

Issue	Date	Confidentiality	Change
01	27-03-2023	Non-Confidential	RMM 1.0 beta1 compliance scenario

Non-Confidential Proprietary Notice

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349)

Confidentiality Status

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

Unrestricted Access is an Arm internal classification.

Product Status

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

Progressive terminology commitment

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

This document includes terms that can be offensive. We will replace these terms in a future issue of this document. If you find offensive terms in this document, please email **terms@arm.com**.

Web Address

www.arm.com.

Contents

1 Introduction

In this document the test scenarios for all RMM ABI commands are detailed. First, an overview of the general test strategy is provided, which forms the basis of the overall flow and rationale of the various scenarios. Unless otherwise specified, this scenario doc is compliant to Bet1 of RMM spec

1.1 Product revision status

The *rmpn* identifier indicates the revision status of the product described in this book, for example, r*1*p*2*, where:

rm Identifies the major revision of the product, for example, r*1*.

pn Identifies the minor revision or modification status of the product, for example, p*2*.

1.2 Intended audience.

This document is for engineers who are verifying the implementation compliance of Arm® Realm Management Monitor 1.0

1.3 Conventions

The following subsections describe conventions used in Arm documents.

1.3.1 Glossary

The Arm Glossary is a list of terms used in Arm documentation, together with definitions for those terms. The Arm Glossary does not contain terms that are industry standard unless the Arm meaning differs from the generally accepted meaning.

See the Arm Glossary for more information: **https://developer.arm.com/glossary**.

1.3.2 Typographical Conventions

Convention	Use
italic	Introduces citations.
bold	Highlights interface elements, such as menu names. Denotes signal names. Also used for terms in descriptive lists, where appropriate.
`monospace`	Denotes text that you can enter at the keyboard, such as commands, file and program names, and source code.
`monospace` **`bold`**	Denotes language keywords when used outside example code.
`monospace underline`	Denotes a permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.
<and>	Encloses replaceable terms for assembler syntax where they appear in code or code fragments. For example: `MRC p15, 0, <Rd>, <CRn>, <CRm>, <Opcode_2>`
SMALL CAPITALS	Used in body text for a few terms that have specific technical meanings, that are defined in the Arm® Glossary. For example, IMPLEMENTATION DEFINED, IMPLEMENTATION SPECIFIC, UNKNOWN, and UNPREDICTABLE.

1.4 Additional Reading

This document contains information that is specific to this product. See the following documents for other relevant information:

Table 1-1 Arm publications

Document name	Document ID	Licensee only
Realm Management Monitor specification	DEN0137	No
Arm® Architecture Reference Manual Armv8, for Armv8-A Architecture	ARM DDI 0487H.a	No
Arm® System Memory Management Unit Architecture Specification	IHI0070	No
Arm® Realm Management Extension (RME) System Architecture	ARM DEN 0129	No
Arm® Power State Coordination Interface (PSCI) - Arm Ltd.	ARM DEN 0022 D.b	No

1.5 Feedback

Arm welcomes feedback on this product and its documentation.

1.5.1 Feedback on content

If you have comments on content, please raise a GitHub issue or mail it to support-cca-rmm-acs@arm.com

2 Arm Realm Management Monitor Specification

The Realm Management Monitor (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.

2.1 CCA-RMM-ACS - Command Suite

The Command Suite in the CCA-RMM ACS tests the Interface section of the Realm Management Specification. The RMM exposes three interfaces of which one is facing the Host (RMI), and two are facing the Realm (RSI and PSCI). For each of the commands a set of input stimuli with which all the failure conditions can be verified, and instructions to observe the command footprint are provided.

The tests are classified as:

- RMI - Realm Management Interface
- RSI - Realm Services Interface
- PSCI - Power State Coordination Interface

2.1.1 Test Strategy

ABI command testing can be regarded as the 'unit-level like' testing of each ABI command. The goal here is to verify whether execution of the command produces the expected outcome. For RMM ABI command testing there are three aspects that require elaboration:

1. The scope and generation of input stimuli
2. The observability of the command footprints
3. The general test flow

Each of the aforementioned aspects will be discussed in a dedicated sections below.

2.1.2 Stimulus

In order to properly exercise the commands, sets of input arguments must be selected such that all corner cases are covered. Each command will be tested on failure conditions, failure priority ordering, and finally, successful execution.

Failure condition testing

Typically, each ABI command has a set of failure conditions that can arise from invalid input argument values. Such conditions have an associated error code, and the same error can often be triggered by multiple values of that input argument. In this part of the test, we will explore all the input values with which the failure condition can be triggered. For example, we might test whether an address is out of bounds by testing boundary conditions, but not all possible regions that would meet the criteria. In each case we then provide a set of input arguments that should trigger only a single failure condition at a time.

Exceptions:

Although the strategy is to cover all corner cases in failure condition testing, there are exceptions:

- Circular Dependencies: Here we would require ABI commands that are yet to be verified to generate the stimulus (note that these are sometimes unavoidable)
- Multicausal Stimuli: This is unavoidable in certain cases. For example, an out-of-bound `rd` in RMI_DATA_CREATE ABI would also result in `Granule(rd).state` error. However, we will exclude stimuli that trigger multiple failure conditions which have both: I) different error codes, and II) no architected priority ordering relative to each other

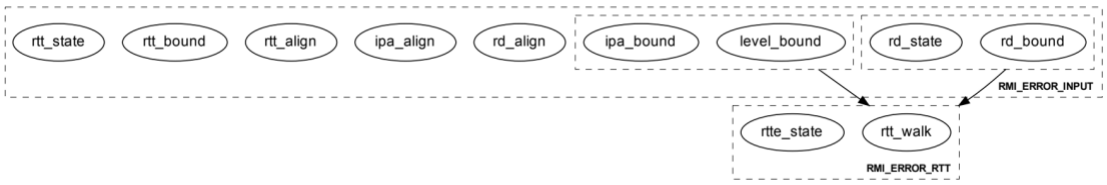
Failure Priority Ordering

It is also important to test that the correct order is maintained between these failure conditions. To test this, we trigger multiple failure conditions on one or multiple input arguments, and verify that the error code of the higher priority condition is observed. In contrast to failure condition testing, each condition will only be triggered in a single way to limit the problem size. This methodology of triggering multiple faults at a time will be referred to as Pairwise Failure Ordering Testing (PFOT).

A failure condition ordering A < B can be grouped into two categories:

- Well-formedness (Not Tested)

These orderings exist because failure condition B can only be evaluated if failure condition A is false. In the figure below `[rd_state, rd_bound] < [rtte_state, rtt_walk]` are well-formedness orderings as we cannot define an RTT walk if the RD granule is not an actual Realm Descriptor granule. As the name suggests, these orderings exist to ensure the mathematical "well-formedness" of the RMM specification and we will not verify these.



- Behavioural (Tested)

These orderings exist to prevent security leaks and to ensure that the returned error code is deterministic across RMM implementations and will be verified, An example in the figure above is `level_bound < rtt_walk`, which implies that an out-of-bounds level parameter should be reported ahead of an RTT walk failure. Furthermore, while it is mathematically acceptable to derive the order between failure conditions based on hierarchy (i.e. if A < B and B < C, then A < C), we will also verify these transitive priority relationships (i.e. A < C) as these orderings must be honoured by the implementer, but are not explicitly mentioned in the specification.

Success condition testing

At the end of each ABI command test we will execute the command with valid input parameters to check that the command executed successfully.

2.1.3 Observability

At the end of each ABI command test we will execute the command with valid input parameters to check that the command executed successfully.

Footprint	Category	Can it be queried by the host
Properties of a Granule	**state** (UNDELEGATED, DELEGATED, RD, REC, AUX_REC, DATA, RTT)	No
	substate (RD.new/Active/NULL, REC.Running/Ready, RTTE.state/ripas)	In general, not, except: RTTE.state/ripas through RTT_READ_ENTRY
	ownership (RD)	No
	PAS (NS / Root / Realm)	No
Contents of a Granule	(RD / REC / DATA / RTT / NS)	No, these are provisioned by the Host but outside of the Realm's TCB (except for NS granules)

As many of these properties and contents of Granules cannot directly by queried by the Host, we need to detect these indirectly. For example, we can determine the Granule states and substates by ascertaining which state transitions are possible, or not possible. Since each state transition is associated with a successful ABI call, some of which would still be subject to verification, this gives rise to so called circular dependencies. The general strategy here is to prevent circular dependencies as much as possible and defer the residual observation of command footprints to other ACS test scenarios. Hence, we will employ the following strategies where the properties and contents of Granules cannot be queried by the Host.

Observing Properties of a Granule

Scenario	Strategy
When the command fails	In general, we will not check for changes in properties of a Granule. We will only check the error code in command ACS. The expectation that a failing command must not cause footprint changes is validated indirectly, to a large extent, as part of testing the command for success criteria. This is because of the valid arguments being same across failure testing (other than the argument that causes a particular failure condition) and successful execution of the command. Consider also checking `rtte_state` & `rtte_addr` for applicable commands
	state We will not test this in CCA-RMM-ACS as the logic to ascertain that the state is unchanged (due to command failure) is not trivial and typically falls in DV space. Also see above comment.
	substate Where they can be queried, we will execute the query ABI (i.e. RTT_READ_ENTRY). Where it cannot be queried, we will follow the same strategy for granule states listed above.
	ownership We will follow the same strategy as for granule state.

Scenario	Strategy
	PAS For PAS checks we will do testing in memory management ACS scenarios to ensure that the GPI encodings in the GPT has not changed.
Summary	When command fails, ACS will check for return error code, and RTTE for some of the failure conditions.
When the command succeeds	In general, we will check that the returned error code is equal to zero.
	state We will not test this in command ACS as it doesn't fit into a typical command ACS test flow. This is covered outside of the command ACS, for example, as part of typical realm creation flow or winding the state of RMM/Realm as part of rollback logic by VAL that's needed to run ACS as a single ELF.
	substate Where they can be queried, we will execute the query ABI. Where it cannot be queried, we will follow the strategy for observing granule states listed above.
	ownership Unless otherwise specified we will follow the same strategy as specified for granule state.
	PAS There will be testing in memory management ACS scenarios to probe the GPI encodings in the GPT.
Summary	When command succeeds, ACS will check for return status, and RTTE wherever applicable.

Observing Contents of a Granule

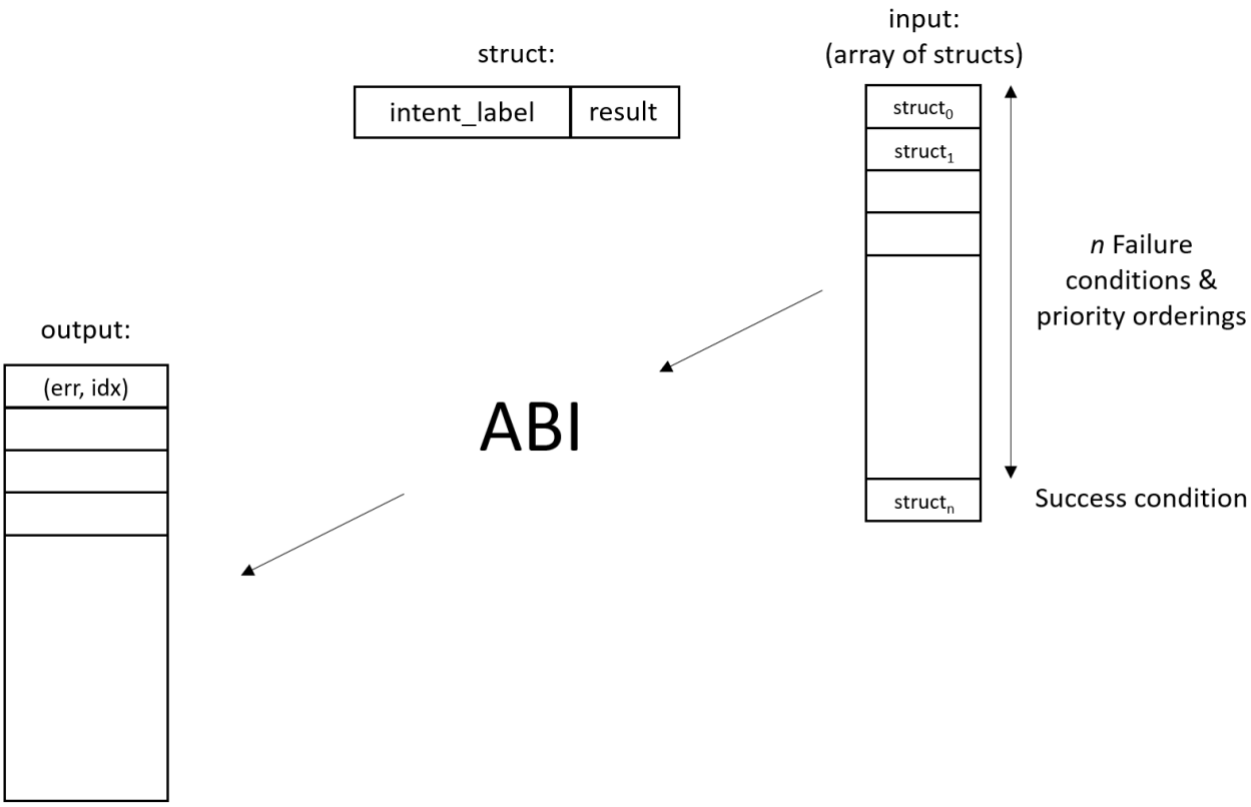
Scenario	Strategy
When the command fails	We will, in general, not check for forbidden changes in contents of a Granule. There is an exception for NS Granules. For example, while testing *REC_ENTER*, we can check whether the `exit_ptr` remains unchanged and does not contain the fields populated in `entry_ptr` through testing outside of the ACS command scenarios.
When the command succeeds	• Host Provisioned For contents that are provisioned by the Host through parameters we do know the expected value, but still need to verify whether the RMM correctly handled the parameters. These will be verified through testing outside of the command ACS. • Non-Host Provisioned For contents that are not provisioned by the Host, if the expected values are architected, we will verify this through testing outside of the ACS command scenarios.

2.1.4 General Test Flow

Having defined the overall test strategy and scoping, the general flow of ABI command tests is as follows:

1. Enter the test from NS-EL2
2. Initialize the input structure (as depicted in the figure below)
3. Iteratively load the intent labels from the input structure and perform the corresponding parameter preparation sequence
4. Execute the ABI command with the prepared set of parameters and check for the expected error code

5. If all reported error codes are as expected, check the command footprint
6. Undo any footprint changes caused by the successful ABI execution and observability tests
7. Return to the test dispatcher



Disclaimer: Only invalid values that cause a failure condition are specified. All other attributes of an input argument must be set to valid values, if applicable, as defined in argument list table above.

2.2 RMI Commands

2.2.1 RMI_DATA_CREATE

Argument List:

Input Parameter	Valid Values
data	granule(data) = 4K_ALIGNED granule(data).state = Delegated
rd	granule(rd).state = RD Realm(rd).state = New

Input Parameter	Valid Values
ipa	ipa = 4K_ALIGNED ipa = protected walk(ipa).level = LEAF_LEVEL RTTE[ipa].state = Unassigned RTTE[ipa].ripas = EMPTY, RAM
src	granule(src) = 4K_ALIGNED granule(src).PAS = NS
flags	flags = RMI_MEASURE_CONTENT

Failure condition testing:

Input Parameter	Input Values	Remarks
data	granule(data) = unaligned_addr, mmio_region (A), outside_of_permitted_pa (B), not_backed_by_encryption, raz or wi (C) granule(data).state = UNDELEGATED, REC, RD, RTT, DATA granule(data).pas = Secure	(A) Memory that behaves like mmio (i.e. read or write-sensitive region) (B) Pick an address that is outside the permitted PA range (lowest of RmiFeatureRegister0.S2SZ and ID_AA64MMFR0_EL1.PARange) (C) Memory address that reads as zero and ignores writes
rd	granule(rd) = unaligned_addr, mmio_region (A), outside_of_permitted_pa (B), not_backed_by_encryption, raz or wi (C) granule(rd).state = UNDELEGATED, DELEGATED, REC, RTT, DATA Realm(rd).state = Active, Null, System_Off	
ipa	ipa = unaligned_addr, unprotected_ipa, outside_of_permitted_ipa (info) walk(ipa).level != LEAF_LEVEL RTTE[ipa].state = Assigned (circular), Destroyed (circular) RTTE[ipa].ripas = EMPTY	*unprotected_ipa := ipa >= 2**(IPA_WIDTH - 1) **IPA_WIDTH = RmiFeatureRegister0.S2SZ (info) Must cover rule - R0254 The input address to an RTT walk is always less than 2^w, where w is the IPA width of the target Realm. No way to prevent circular dependencies here
src	granule(src) = unaligned_addr, mmio_region (A), outside_of_permitted_pa (B), not_backed_by_encryption, raz or wi (C) granule(src).PAS == Secure, Realm	

Failure priority Ordering:

Input Parameter	Input value	Remarks
ipa	unprotected_ipa && walk(ipa).level != LEAF_LEVEL unprotected_ipa && RTTE[ipa].state = VALID_NS	

Observability:

Footprint	Verification
• Command Failure	
RTTE.state, RTTE.addr	Refer **Observing Properties of a Granule** and **Observing Contents of a Granule** for details
rim	This needs to be tested outside of ACS command scenarios (Security Model / Attestation scenarios)
granule(data).state, granule(data).content	This is outside the scope of CCA-RMM-ACS. Refer **Observing Properties of a Granule** and **Observing Contents of a Granule** for details
• Command Success	
RTTE.state, RTTE.addr	Execute RTT_READ_ENTRY and compare the outcome with expected value (as defined by the architecture) Do this for RTTE[ipa].ripas = {EMPTY, RAM}
granule(data).state, granule(data).content	This is already tested outside of ACS command scenarios, as part of Realm creation with payload.
rim	This needs to be tested outside of ACS command scenarios (Security Model / Attestation scenarios)

2.2.2 RMI_DATA_CREATE_UNKNOWN

Argument List:

Input Parameter	Valid Values
data	granule(data) = 4K_ALIGNED granule(data).state = Delegated
ipa	ipa = 4K_ALIGNED ipa = protected walk(ipa).level = LEAF_LEVEL RTTE[ipa].state = Unassigned RTTE[ipa].ripas = EMPTY, RAM
rd	granule(rd).state = RD Realm(rd).state = New, Active.

Failure condition testing:

Input Parameter	Input Values	Remarks
rd	granule(rd) = unaligned_addr, mmio_region (A), outside_of_permitted_pa (B), not_backed_by_encryption, raz or wi (C) granule(rd).state = UNDELEGATED, DELEGATED, REC, DATA, RTT	
ipa	ipa = unaligned_addr, unprotected_ipa, outside_of_permitted_ipa walk(ipa).level != LEAF_LEVEL RTTE[ipa].state = Assigned (circular), Destroyed (circular)	*unprotected_ipa := ipa >= 2**(IPA_WIDTH - 1) **IPA_WIDTH = RmiFeatureRegister0.S2SZ Must cover rule - R0254 The input address to an RTT walk is always less than 2^w, where w is the IPA width of the target Realm. No way to prevent circular dependencies here
data	granule(data) = unaligned_addr, mmio_region (A), outside_of_permitted_pa (B), not_backed_by_encryption, raz or wi (C) granule(data).state = UNDELEGATED, REC, RD, RTT, DATA	See RMI_DATA_CREATE for the specifics behind these stimuli

Failure priority Ordering:

Input Parameters	Input Values	Remarks
ipa	unprotected_ipa && walk(ipa).level != LEAF_LEVEL unprotected_ipa && RTTE[ipa].state = VALID_NS	

Observability:

Footprint	Verification
Command Failure	
RTTE.state, RTTE.addr	Follow the same strategy as in RMI_DATA_CREATE
granule(data).state, granule(data).content	This is outside the scope of CCA-RMM-ACS. . Refer **Observing Properties of a Granule** and **Observing Contents of a Granule** for details.
Command Success	
RTTE.state, RTTE.addr	Execute RTT_READ_ENTRY and compare the outcome with expected value (as defined by the architecture) Do this for Realm(rd).state = {New, Active} & RTTE[ipa].ripas = {EMPTY, RAM}
granule(data).content, granule(data).state	For granule(data).content, it needs to be tested outside of ACS command scenarios as part of verifying "granule wiping" security property. For granule(data).state, it is already tested outside of command ACS, for example as part of RMM/Realm state rollback at the end of each test.

2.2.3 RMI_DATA_DESTROY

Argument List:

Input Parameter	Valid Values
rd	granule(rd).state = RD Realm(rd).state = New, Active
ipa	ipa = 4K_ALIGNED ipa = protected walk(ipa).level = LEAF_LEVEL RTTE[ipa].state = Assigned RTTE[ipa].ripas = EMPTY, RAM

Failure condition testing:

Input Parameter	Input Values	Remarks
rd	granule(rd) = unaligned_addr, mmio_region (A), outside_of_permitted_pa (B), not_backed_by_encryption, raz or wi (C) granule(rd).state = UNDELEGATED, DELEGATED, REC, DATA, RTT	See RMI_DATA_CREATE for the specifics behind these stimuli
ipa	ipa = unaligned_addr, unprotected_ipa, outside_of_permitted_ipa walk(ipa).level != LEAF_LEVEL RTTE[ipa].state = Unassigned, Destroyed (Circular or dependency on RMI_RTT_DESTROY)	*unprotected_ipa := ipa >= 2**(IPA_WIDTH - 1) **IPA_WIDTH = RmiFeatureRegister0.S2SZ Must cover rule - R0254 The input address to an RTT walk is always less than 2^w, where w is the IPA width of the target Realm.

Failure priority Ordering:

Input Parameters	Input Values	Remarks
ipa	unprotected_ipa && walk(ipa).level != LEAF_LEVEL unprotected_ipa && RTTE[ipa].state = Unassigned, Assigned (VALID_NS)	

Observability:

Footprint	Verification	Remarks
Command Failure		
RTTE.state, RTTE.addr	Follow the same strategy as in RMI_DATA_CREATE	
granule(data).state	This is outside the scope of CCA-RMM-ACS. Refer **Observing Properties of a Granule** and **Observing Contents of a Granule** for details	granule(data) = PA of the target data mapped @ ipa
Command Success		
RTTE.state, RTTE.addr	Execute RTT_READ_ENTRY and compare the outcome with expected value (as defined by the architecture) Do this for Realm(rd).state = {New, Active} & RTTE[ipa].ripas = {EMPTY, RAM}	
granule(data).state	This needs to be tested outside of ACS command scenarios (as part of security scenarios)	

2.2.4 RMI_FEATURES

Argument List:

Input Parameter	Valid Values
index	index = any_integer (64b)

Failure condition testing:

This command has no failure conditions.

Failure priority Ordering:

This command has no failure priority orderings.

Observability:

Footprint	Verification
Command Success	
X1 (command return value)	Check that RES0 fields are zero (TBD: sanity checks on the fields inside features_0)

2.2.5 RMI_GRANULE_DELEGATE

Argument List:

Input Parameter	Valid Values
addr	granule(addr) = 4K_ALIGNED granule(addr).state = Undelegated granule(addr).PAS = Non-Secure

Failure condition testing:

Input Parameter	Input Values	Remarks
addr	granule(addr) = unaligned_addr, mmio_region (A), outside_of_permitted_pa (B), not_backed_by_encryption, raz or wi (C) granule(addr).state = Delegated, REC, DATA, RTT, RD granule(addr).PAS = Secure, Realm	See RMI_DATA_CREATE for the specifics behind these stimuli. granule(addr).PAS = Root is outside the scope of ACS.

Failure priority Ordering:

This command has no failure priority orderings.

Observability:

Footprint	Verification
Command Failure	
granule(addr).PAS	For granule(addr).PAS=Non-secure, an access to addr from the NS world should be successful and should be tested in command ACS. For granule(addr).PAS=Secure and Realm (granule(addr).state = DATA), this needs to be tested outside of ACS command scenarios.
granule(addr).state	This is outside the scope of CCA-RMM-ACS. Refer **Observing Properties of a Granule** and **Observing Contents of a Granule** for details.
Command Success	

Footprint	Verification
granule(addr).PAS	This needs to be tested outside of ACS command scenarios - Accesses from different worlds. Note: A NS-world access to addr in this case would result in GFP. The target EL for this GPF depends on SCR_EL3.GPF (which is the DUT). If SCR_EL3.GPF =1, the fault is reported as GPC exception and is taken to EL3. EL3 may choose to delegate this exception to NS-EL2. If this delegation scheme is supported by the implementation, we can validate changes in PAS in ACS. If not, the test needs to be able to exit gracefully, for example, using watchdog interrupt. If this is not possible, we won't be able to verify changes in PAS in ACS. If SCR_EL3.GPF=0, the GPF is reported as instruction or data abort at EL2 itself, and this can be validated in ACS. Need to be wary of the above while writing ACS. 2. Using realm creation flow that's already tested outside of ACS command scenarios Post RMI_GRANULE_DELEGATE, such a flow would create realm/rec/rtt/data and be able to execute from realm successfully. Point 2 proves that the granule can be accessed from the target granule(addr).PAS and Point 1 proves that the granule access is forbidden from the current state. Conclusion - do option1 outside of command ACS and keep such testing to a limited set of tests.
granule(addr).state	This is already tested outside of ACS command scenarios (Realm creation with payload).

2.2.6 RMI_GRANULE_UNDELEGATE

Argument List:

Input Parameter	Valid Values
addr	granule(addr) = 4K_ALIGNED granule(addr).state = Delegated granule(addr).PAS = Realm

Failure condition testing:

Input Parameter	Input Values	Remarks
addr	granule(addr) = unaligned_addr, mmio_region (A), outside_of_permitted_pa (B), not_backed_by_encryption, raz or wi (C) granule(addr).state = Undelegated, REC, DATA, RTT, RD	See RMI_DATA_CREATE for the specifics behind these stimuli

Failure priority Ordering:

This command has no failure priority orderings

Observability:

Footprint	Verification
Command Failure	

Footprint	Verification
granule(addr).PAS	For granule(addr).PAS = Realm && granule(addr).state = REC/DATA/RD/RTT execute the respective destroy command and verify that it is successful
granule(addr).content	For granule(addr).state = REC/DATA/RTT verify that the content is unchanged. This needs to be tested outside of ACS command scenarios
granule(addr).state	For granule(addr).state = Undelegated, access this granule from the NS world, and this access should be successful. This is in scope for command ACS
Command Success	
granule(addr).PAS, granule(addr).content	Can be tested through accesses from the NS world (should succeed and content be wiped) Sequence: RMI_DATA_CREATE(ipa, src, data) → RMI_DATA_DESTROY(rd, ipa) → RMI_GRANULE_UNDELEGATE(addr=data) Verify: src != data This will be covered within the command ACS
granule(addr).state	This is already tested outside of ACS command scenarios (Realm teardown sequence).

2.2.7 RMI_PSCI_COMPLETE

To test this command, unless otherwise specified below, the pre-requsite is that the realm needs to initiate corresponding PSCI request (PSCI_AFFINITY_INFO and PSCI_CPU_ON) through RSI.

Argument List:

Input Parameter	Valid Values
calling_rec	granule(calling_rec) = 4K_ALIGNED granule(calling_rec).state = REC Rec(calling_rec).psci_pending = PSCI_REQUEST_PENDING calling rec != target rec
target_rec	granule(target_rec) = 4K_ALIGNED granule(target_rec).state = REC Rec(target_rec).owner = Rec(calling_rec).owner Rec(target_rec).owner = Rec(calling_rec).gprs[1]

Failure condition testing:

Input Parameter	Input Values	Remarks
calling_rec	granule(calling_rec) = granule(target_rec), unaligned_addr, mmio_region (A), outside_of_permitted_pa (B), not_backed_by_encryption, raz or wi (C), granule(calling_rec).state = Undelegated, Delegated, RD, RTT Rec(calling_rec).psci_pending != PSCI_REQUEST_PENDING (E)	(E) - This can be achieved in two ways. (1) Execute RMI_PSCI_COMPLETE without a request from realm (2) provide incorrect calling_rec arg value (same realm but didn't initiate RSI request, REC belonging to different realm) in RMI_PSCI_COMPLETE
target_rec	granule(target_rec) = unaligned_addr, mmio_region, outside of permitted pa, not backed by encryption, raz or wi, other_realm_owned rec wrong_target (D) granule(target_rec).state = Undelegated, Delegated, RD, RTT	(D) wrong_target implies that calling_rec has a different mpidr value stored in gprs[1] than target_rec.mpidr

Failure priority Ordering:

This command has no failure priority orderings

Observability:

Footprint	Verification
Command Failure	
target_rec.content, calling_rec.content	Refer **Observing Properties of a Granule** and **Observing Contents of a Granule** for details.
Command Success	
target_rec.content, calling_rec.content	Tested outside of ACS command scenarios. Overlap with a scenario in Exception Model section - REC Exit due to PSCI

2.2.8 RMI_REALM_ACTIVATE

Argument List:

Input Parameter	Valid Values
rd	granule(rd) = 4K_ALIGNED granule(rd).state = RD Realm(rd).state = New

Failure condition testing:

Input Parameter	Input Values	Remarks
rd	granule(rd) = unaligned_addr, mmio_region (A), outside_of_permitted_pa (B), not_backed_by_encryption, raz or wi (C) granule(rd).state = Undelegated, Delegated, REC, DATA, RTT Realm(rd).state = Active (D), Null, System_off (D)	(A - C) See RMI_DATA_CREATE for the specifics behind these stimuli (D) Active requires a valid REALM_ACTIVATE call (circular dependency) → Do this as part of the positive observability check (D) System_Off is outside of the scope of command ACS, as the Realm needs to be active to get to this state. Also, the scope of "SYSTEM" in PSCI_SYSTEM_OFF is imp def.

Failure priority Ordering:

Input Parameter	Input Values	Remarks
rd	granule(rd).state = Null && Realm(rd).state = Delegated	This is already covered with Realm(rd).state = Null in the failure condition stimulus above

Observability:

Footprint	Verification
Command Failure	
Realm(rd).state	This is outside the scope of CCA-RMM-ACS. Refer **Observing Properties of a Granule** and **Observing Contents of a Granule** for details.
Command Success	
Realm(rd).state	This is already tested outside of ACS command scenarios (as part of Realm entry test flows)

2.2.9 RMI_REALM_CREATE

Argument List:

Input Parameter	Valid Values
rd	granule(rd) = 4K_ALIGNED granule(rd).state = Delegated
params_ptr	granule(params_ptr) = 4K_ALIGNED granule(params_ptr).PAS = NS granule(params_ptr).content(rtt_base) = 4K_ALIGNED granule(params_ptr).content(rtt_base).state = Delegated granule(params_ptr).content(features0) = valid granule(params_ptr).content(vmid) = valid, not_in_use granule(params_ptr).content(hash_algo) = valid, supported granule(params_ptr).content(rtt_level_start, rtt_num_start) = consistent with features0.S2SZ !(granule(rd) >= granule(params_ptr).content(rtt_base) && granule(rd) <= granule(params_ptr).content(rtt_base+rtt size))

Failure condition testing:

Input Parameter	Input Values	Remarks
rd	granule(rd) = unaligned_addr, mmio_region (A), outside_of_permitted_pa (B), not_backed_by_encryption, raz or wi (C), params_ptr.content(rtt_base) granule(rd).state = Undelegated, RD, RTT, DATA, REC	

Input Parameter	Input Values	Remarks
params_ptr	granule(params_ptr) = unaligned_addr, mmio_region (A), outside_of_permitted_pa (B), not_backed_by_encryption, raz or wi (C) granule(params_ptr).content(hash_algo) = encoding_reserved (D), not_supported (E) granule(params_ptr).content(rtt_base) = unaligned_addr, granule(params_ptr).content(rtt_num_start, rtt_level_start) = incompatible (F), granule(params_ptr).content(rtt_base, rttsize*).state = Undelegated, granule(params_ptr).content(features_0) = ipa_w_unsupported, granule(params_ptr).content(vmid) = invalid, in_use granule(params_ptr).PAS = Realm, Secure, Root (may be outside the scope of ACS as we may not get to know Root memory from platform memory map) *rtt size = rtt base<addr<rtt_num_start*RMM granule size so, cover RMIRealmparams.rtt_num_start =1 and >1. For the latter, for example, if read of RMIFeatureregister0.S2SZ=16 (that is implementation supports 48-bit PA/IPA), program RMIRealmparams.features0.S2SZ= 24 (that is 40-bit IPA), RMIRealmparams.rtt_num_start ~2, RMIRealmparams.rtt_level_start ~1, choose rtt_base ~8K aligned address with first 4KB in delegate state and the next 4KB in undelegate state	(D) encoding_reserved refers to values that are reserved for future implementations (i.e., not in the table in spec) (E) not_supported refers to a valid encoding that is not supported by current implementation - To achieve this error, perform following sequence read RMI Featureregister0.HASH_SHA_256 and HASH_SHA_512 and figure out which one of these is supported by the underlying platform Provide the unsupported value from previous step in granule(params_ptr).hash_algo (F) params_ptr.content(rtt_num_start, rtt_level_start) = incompatible with RMIFeatureregister0.S2SZ Scenario: Host to choose rtt_level_start and ipa_width such that number of starting level RTTs is greater than one. Host to populate correct rtt_num_start value in realmParam, expect SUCCESS. Host to choose rtt_level_start and ipa_width such that number of starting level RTTs is greater than one. Host to populate incorrect rtt_num_start value in realmParam and expect ERROR Perform following steps in argument preparation phase (intent to sequence block) to achieve above conditions (for generating ERROR) read RMIFeatureregister0.S2SZ if S2SZ ~ [12-15], set RMIRealmparams.rtt_level_start ~ 1/2/3 or set RMIRealmparams.rtt_num_start ~ >16 if S2SZ ~ [16-24], set RMIRealmparams.rtt_level_start ~ 2/3 or set RMIRealmparams.rtt_num_start ~ >16 if S2SZ ~ [25-33], set RMIRealmparams.rtt_level_start ~ 3 or set RMIRealmparams.rtt_num_start ~ >16 if S2SZ ~ [34-42], set RMIRealmparams.rtt_num_start ~ >16 Note that S2SZ format in RMIFeatureregister0 is different from VTCR_EL2.T0SZ in that the former expects the actual IPA width to be programmed (or returned during quering) as against specifying the equivalent of T0SZ value.

Failure priority Ordering:

This command has no failure priority orderings.

Observability:

Footprint	Verification
Command Failure	
rd.state	This is outside the scope of CCA-RMM-ACS. Refer **Observing Properties of a Granule** and **Observing Contents of a Granule** for details.
rd.substate, rd.content	Same as above
Command Success	
rd.state	This is already tested outside of ACS command scenarios (as part of realm creation flow).
rd.substate, rd.content	same as above

2.2.10 RMI_REALM_DESTROY

Argument List:

Input Parameter	Valid Values
rd	granule(rd) = 4K_ALIGNED granule(rd).state = RD Realm liveliness = False

Failure condition testing:

Input Parameter	Input Values	Remarks
rd	granule(rd) = unaligned_addr, mmio_region (A), outside_of_permitted_pa (B), not_backed_by_encryption, raz or wi (C), alive (D) granule(rd).state = Delegated*, Undelegated, REC, DATA, RTT *-create a realm, destroy a realm. The state of granule is in delegated state. Use this granule to destroy an already destroyed realm. The command should fail due to rd_state error.	(A - C) See RMI_DATA_CREATE for the specifics behind these stimuli (D) alive = has 1 REC OR has 1 RTT above the start level RTT

Failure priority Ordering:

Input Parameter	Input Values	Remarks
rd	Orderings between granule(rd) or granule(rd).state & realm liveliness	These are outside of the scope of CCA-RMM-ACS as these fall under well-formedness orderings

Observability:

Footprint	Verification
Command Failure	
granule(rd).state	This is outside of the scope of CCA-RMM-ACS and falls into DV space

Footprint	Verification
Realm(rd).state	For Realm(rd).state = Active, System_Off: This is outside the scope of CCA-RMM-ACS, as the only sensible thing we can do, is to destroy the Realm, which is the ABI we are currently testing. For System_Off, since the scope of system is imp def, we won't be able to test this in ACS. For Realm(rd).state = New: This is outside of the scope of CCA-RMM-ACS and falls into DV space.
granule(rtt).state	This is already tested in other command ACS scenarios (RMI_GRANULE_UNDELEGATE or RMI_REALM_CREATE)
Command Success	
vmid	All of this is already tested outside of ACS command scenarios . For example, VMID being freed up is tested as part of running ACS as a single ELF (that is, VAL winds up state of RMM/a test before start of another test). For granule(rd).state, Realm(rd).state, verify this is command ACS by performing RMI_REALM_CREATE again with identical attributes.
granule(rd).state	
Realm(rd).state	

2.2.11 RMI_REC_AUX_COUNT

Argument List:

Input Parameter	Valid Values
rd	granule(rd) = 4K_ALIGNED granule(rd).state = RD

Failure condition testing:

Input Parameter	Input Values	Remarks
rd	granule(rd) = unaligned_addr, mmio_region (A), outside_of_permitted_pa (B), not_backed_by_encryption, raz or wi (C) granule(rd).state = Delegated, Undelegated, DATA, RTT	See RMI_DATA_CREATE for the specifics behind these stimuli

Failure priority Ordering:

This command has no failure priority orderings.

Observability:

Footprint	Verification
Command Success	
X1 (command return value)	This is already tested outside of ACS command scenarios as part of the Realm creation flow (when the various feature: SVE, etc. are implemented)

2.2.12 RMI_REC_CREATE

Argument List:

Input Parameter	Valid Values
rec	granule(rec) = 4K_ALIGNED granule(rec).state = Delegated
rd	granule(rd) = 4K_ALIGNED granule(rd).state = RD Realm(rd).state = New
params_ptr	granule(params_ptr) = 4K_ALIGNED granule(params_ptr).PAS = NS granule(params_ptr).content(mpidr) = in_range (where in_range = 0, 1, 2, ...) granule(params_ptr).content(aux) = 4K_ALIGNED granule(params_ptr).content(num_aux) = RMI_REC_AUX_COUNT(rd) granule(params_ptr).content(aux).state = Delegated granule(params_ptr).content/content(name).MBZ/SBZ = 0, where name can be flags. Try with flag = runnable and not runnable

Failure condition testing:

Input Parameter	Input Values	Remarks
rec	granule(rec) = unaligned_addr, mmio_region (A), outside_of_permitted_pa (B), not_backed_by_encryption, raz or wi (C) granule(rec).state = Undelegated, REC, RTT, RD, DATA	
rd	granule(rd) = unaligned_addr, mmio_region (A), outside_of_permitted_pa (B), not_backed_by_encryption, raz or wi (C) granule(rd).state = Undelegated, Delegated*, REC, RTT, DATA *-create a realm, destroy the realm and use the granule that's in delegated state as an input to this ABI to test above failure condition. Realm(rd).state = Active	Prepare granule whose granule(rd).state=Delegated and realm(rd).state=Null Realm(rd).state = Null will result in more than one failure condition whose error codes are different and priority order ing is not defined Realm(rd).state = System_Off is outside of the scope of command ACS (DV space)

Input Parameter	Input Values	Remarks
params_ptr	granule(params_ptr) = unaligned_addr, mmio_region (A), outside_of_permitted_pa (B), not_backed_by_encryption, raz or wi (C) granule(params_ptr).content(num_aux) != RMI_REC_AUX_COUNT(rd) granule(params_ptr).content(aux...num_aux) = unaligned_addr (F) granule(params_ptr).content(aux...num_aux) = granule(rec) (G) granule(params_ptr).content(aux) = mmio_region (A), outside_of_permitted_pa (B), not_backed_by_encryption, raz or wi (C) granule(params_ptr).content(aux...num_aux).state = Undelegated, REC, RTT, DATA, RD granule(params_ptr).content(flags) = invalid_res0 (D) granule(params_ptr).content(mpidr) = non_zero_mpidr (set MBZ fields to 1) (D), provide mpidr value starts from 2 granule(params_ptr).pas = Realm, Secure, Root (E)	(D) content(flags) = invalid_res0 the RES0 fields are set to 1 in the flags field content(mpidr) = invalid_res0 the RES0 fields are set to 1 in the MPIDR field (E) Root (may be outside the scope of ACS as we may not get to know Root memory from platform memory map) (F) at least one aux_granule must be unaligned (G) Provide granule(rec) address to one of aux address

Failure priority Ordering:

Input Parameter	Remarks
rd	The priority ordering as defined in the spec is already covered with granule(rd) = mmio in the failure condition stimulus above

Observability:

Footprint	Verification
Command Failure	
Realm(rd).rec_index	Refer **Observing Properties of a Granule** and **Observing Contents of a Granule** for details.
granule(rec).content	This is outside of the scope of CCA-RMM-ACS
rim	This needs to be tested outside of command ACS (Attestation Scenarios)
granule(rec).state	Refer **Observing Properties of a Granule** and **Observing Contents of a Granule** for details.
granule(rec).attest	TBD
granule(rec_aux).state	Refer **Observing Properties of a Granule** and **Observing Contents of a Granule** for details.
Command Success	
Realm(rd).rec_index	This is already tested outside of command ACS (a Realm with multiple RECs)
granule(rec).content	This is already tested outside of command ACS (a Realm entry)
granule(rec).ripas_addr granule(rec).ripas_top granule(rec).host_call	This is already tested outside of command ACS in one of the Exception Model scenario
rim	This needs to be tested outside of command ACS (Attestation Scenarios)
granule(rec).attest	TBD

2.2.13 RMI_REC_DESTROY

Argument List:

Input Parameter	Valid Values
rec	granule(rec) = 4K_ALIGNED granule(rec).state = REC Rec(rec).state = READY

Failure condition testing:

Input Parameter	Input Values	Remarks
rec	granule(rec) = unaligned_addr, mmio_region (A), outside_of_permitted_pa (B), not_backed_by_encryption, raz or wi (C) granule(rec).state = Undelegated, Delegated, RTT, DATA, RD, REC_AUX Rec(rec).state = Running (D)	(D) This can be verified only in an MP envrionment and need to be tested outside of command ACS.

Failure priority Ordering:

Input Parameter	Remarks
rec	The priority ordering as defined in the spec is already covered with granule(rec) = mmio and granule(rec).state in the failure condition stimulus above

Observability:

Footprint	Verification
Command Failure	
granule(rec).state granule(rec_aux).state	Refer **Observing Properties of a Granule** and **Observing Contents of a Granule** for details.
granule(rec).content	This needs to be tested outside of ACS command scenarios
Command Success	
granule(rec).state granule(rec_aux).state	This is already tested outside of ACS command scenarios, as part of RMM/Realm state rollback that's needed to run ACS as a single ELF.
granule(rec).content	This needs to be tested outside of ACS command scenarios (in memory management scenarios)

2.2.14 RMI_REC_ENTER

Argument List:

Input Parameter	Valid Values
rec	granule(rec) = 4K_ALIGNED granule(rec).state = REC Rec(rec).state = READY Rec(rec).content(flags.runnable) = RUNNABLE Rec(rec).content(psci_pending) = NO_PSCI_REQUEST_PENDING Realm(Rec(rec).owner).state = Active
run_ptr	granule(run_ptr) = 4K_ALIGNED granule(run_ptr).PAS = NS granule(run_ptr).content(entry.flags.emul_mmio) = NOT_RMI_EMULATED_MMIO granule(run_ptr).content(entry.gicv3_hcr) = valid (RES0) granule(run_ptr).content(entry.gicv3_lrs) = valid (HW = 0)

Failure condition testing:

Input Parameter	Input Values	Remarks
rec	granule(rec) = unaligned_addr, mmio_region , outside_of_permitted_pa (B), not_backed_by_encryption, raz or wi (C) granule(rec).content(flags.runnable) = NOT_RUNNABLE granule(rec).content(psci_pending) = PSCI_REQUEST_PENDING (F) granule(rec).state = Undelegated, Delegated, RTT, RD, DATA, REC_AUX Rec(rec).state = Running (E) Realm(Rec(rec).owner).state = New, System_Off (D)	(D) May not be possible to do in ACS and is outside the scope of ACS (E) This is an MP scenario as one thread (REC) will be running inside the Realm, while another will attempt to enter into realm using the same REC. This needs to be tested outside of command ACS. (F) This needs to be tested outside of command ACS as it requires entering into realm and execute a PSCI command.
run_ptr	granule(run_ptr) = unaligned_addr, mmio_region (A), outside_of_permitted_pa (B), not_backed_by_encryption, raz or wi (C) granule(run_ptr).pas = Realm, Secure, Root (H) granule(run_ptr).content(is_emulated_mmio) = RMI_EMULATED_MMIO (F) granule(run_ptr).content(gicv3_hcr/gcv3_lrs) = invalid_encoding (G)	(F) assumes rec.content(emulatable_abort) = NOT_EMULATABLE_ABORT (this is the case before even entering into realm for the first time) (G) Exhaustive testing to follow in **GIC scenarios** (H) Root (may be outside the scope of ACS as we may not get to know Root memory from platform memory map)

Failure priority Ordering:

This is tested as part of single failure condition testing.

Observability:

Footprint	Verification
Command Failure	
Rec(rec).content	This needs to be tested outside of ACS command scenarios Overlap with scenarios in Exception Model section REC Exit Security & Data Abort scenarios
Command Success	
Rec(rec).content	This needs to be tested outside of ACS command scenarios Overlap with scenarios in Exception Model section REC Exit Security & Data Abort scenarios

2.2.15 RMI_RTT_CREATE

Argument List:

Input Parameter	Valid Values
rtt	granule(rtt) = 4K_ALIGNED granule(rtt).state = Delegated
rd	granule(rd) = 4K_ALIGNED granule(rd).state = RD Realm(rd).state = New, Active, System_Off
ipa	ipa = (level-1)_aligned ipa = within_permissible_ipa (< 2^features0.S2SZ) walk(ipa).level = level - 1 RTTE[ipa].state = Unassigned
level	level = {1, 2, 3} if start level is level 0.

Failure condition testing:

Input Parameter	Input Values	Remarks
rtt	granule(rtt) = unaligned_addr (<4KB aligned), mmio_region (A), outside_of_permitted_pa (B), not_backed_by_encryption, raz or wi (C) granule(rtt).state = Undelegated, REC, RD, RTT, DATA	
rd	granule(rd) = unaligned_addr, mmio_region (A), outside_of_permitted_pa (B), not_backed_by_encryption, raz or wi (C) granule(rd).state = Undelegated, Delegated, REC, RTT, DATA	

Input Parameter	Input Values	Remarks
ipa	ipa = unaligned_addr (for example, a 4KB aligned IPA to create level2 RTT), outside_permissible_ipa (*) walk(ipa).level < level - 1 RTTE[ipa].state = Table (circular → same as positive Observability check)	(*)Must cover rule - R0254 The input address to an RTT walk is always less than 2^w, where w is the IPA width of the target Realm as defined by RMIFeatureregister0.S2SZ
level	level = start_level (for example 0 if S2SZ supports an IPA width compatible to level0), 4	

Failure priority Ordering:

All failure priority ordering conditions specified in spec are tested as part of failure condition testing (multi-causal stimuli)

Observability:

Footprint	Verification
Command Failure	
rtt.state, RTTE.state, RTTE.addr, RTT[ipa].content	Refer **Observing Properties of a Granule** and **Observing Contents of a Granule** for details.
Command Success	
rtt.state, RTTE.state, RTTE.addr, RTT[ipa].content	Execute Valid RTT_CREATE → success Execute RTT_READ_ENTRY and verify the outcome is as expected by the architecture.

2.2.16 RMI_RTT_DESTROY

Argument List:

Input Parameter	Valid Values
rtt	granule(rtt) = 4K_ALIGNED granule(rtt).state = RTT Rtt(rtt) = not_live (should cover situations wherein all RTTE.state for all RTTEs in the target RTT is unassigned or destroyed or assigned && for unprotected IPA (RTT liveliness definition spec did not include the last case and ACS must prove this)
rd	granule(rd) = 4K_ALIGNED granule(rd).state = RD Realm(rd).state = New, Active, System_Off
ipa	ipa = (level-1)_aligned ipa = within_permissible_ipa (< 2^ (features0.S2SZ)) walk(ipa).level = level - 1 walk(ipa).level.entry.addr == rtt RTTE[ipa].state = Table

Input Parameter	Valid Values
level	level = {1, 2, 3} if start level is level 0.

Failure condition testing:

Input Parameter	Input Values	Remarks
rtt	Rtt(rtt) = not_empty*	*not_empty refers to an RTT that has at least one Assigned entry Try to destroy RTT whose one of the entry state is ASSIGNED (for protected IPA). Try to destroy RTT whose one of the entry state in the RTT is TABLE (points to next level RTT).
rd	granule(rd) = unaligned_addr, mmio_region (A), outside_of_permitted_pa (B), not_backed_by_encryption, raz or wi (C) Granule(rd).state = Undelegated, Delegated, DATA, REC, RTT	
ipa	ipa = unaligned_addr(for example, a 4KB aligned IPA to destroy level2 RTT), outside_permissible_ipa, walk(ipa,level).addr != rtt(D) walk(ipa).level < level - 1 RTTE[ipa].state = Assigned, Unassigned, Destroyed (circular depedency)	(D) not_rtt refers to an address that does not point to the target rtt that we want to destroy Must cover rule - R0254 The input address to an RTT walk is always less than 2^w, where w is the IPA width of the target Realm.
level	level = start_level (for example 0 if S2SZ supports an IPA width compatible to level0), 4	

Failure priority Ordering:

All failure priority ordering conditions specified in spec are tested as part of failure condition testing (multi-causal stimuli) and some fall under well-formed ordering.

Observability:

Footprint	Verification
Command Failure	
rtt.state, RTTE.state	Refer **Observing Properties of a Granule** and **Observing Contents of a Granule** for details.
Command Success	
rtt.state, RTTE.state	Execute Valid RTT_DESTROY → success Execute RTT_READ_ENTRY and verify the outcome is as expected by the architecture.

2.2.17 RMI_RTT_FOLD

Argument List:

Input Parameter	Valid Values
rtt	granule(rtt) = 4K_ALIGNED granule(rtt).state = RTT Rtt(rtt) = homogeneous
rd	granule(rd) = 4K_ALIGNED granule(rd).state = RD Realm(rd).state = New, Active, System_Off
ipa	ipa = (level-1)_aligned ipa = within_permissible_ipa (< 2^Features0.S2SZ) walk(ipa).level = level-1 walk(ipa).level.entry.addr == rtt RTTE[ipa].state = Table
level	level = {1 when RMIFeatureregister0.LPA2 is True, 2, 3}

Failure condition testing:

Input Parameter	Input Values	Remarks
rtt	Rtt(rtt) = not_homogeneous (D)	(D) not_homogeneous refers to an RTT that has RTTEs in different states. For example, an RTTE is in assigned state and another RTTE in destroyed state.
rd	granule(rd) = unaligned_addr, mmio_region (A), outside_of_permitted_ipa (B), not_backed_by_encryption, raz or wi (C) granule(rd).state = Undelegated, Delegated, REC, DATA, RTT	
ipa	ipa = unaligned_addr (for example, a 4KB aligned IPA to fold level3 RTT), outside_permissible_ipa, walk(ipa,level-1).addr != rtt(E) walk(ipa).level < level - 1 RTTE[ipa].state = Assigned, Unassigned, Destroyed	(E) not_rtt refers to an address that does not point to the target rtt that we want to fold
level	level = SL (-1 when RMIFeatureregister0.LPA2 is supported),4,1(when RMIFeatureregister0.LPA2 is not supported)	

Failure priority Ordering:

All failure priority ordering conditions specified in spec are tested as part of failure condition testing (multi-causal stimuli) and some fall under well-formed ordering

Observability:

Footprint	Verification	
Command Failure		

Footprint	Verification	
rtt.state, RTTE.state	Refer **Observing Properties of a Granule** and **Observing Contents of a Granule** for details.	
Command Success		
rtt.state, RTTE.state	Execute Valid RTT_FOLD → success Execute RTT_READ_ENTRY and verify the outcome is as expected by the architecture.	

2.2.18 RMI_RTT_INIT_RIPAS

Argument List:

Input Parameter	Valid Values
rd	granule(rd) = 4K_ALIGNED granule(rd).state = RD Realm(rd).state = New
ipa	ipa = level_aligned ipa = protected_ipa walk(ipa).level = level RTTE[ipa].state = Unassigned
level	level = 3. For level 0,1, and 2, the preparation sequence requires RTT_FOLD. Level0 can only be covered when RMIFeatureregister0.LPA2 is supported.

Failure condition testing:

Input Parameter	Input Values	Remarks
rd	granule(rd) = unaligned_addr, mmio_region (A), outside_of_permitted_ipa (B), not_backed_by_encryption, raz or wi (C) granule(rd).state = Undelegated, Delegated, DATA, RTT, REC Realm(rd).state = Active, Null, System_off	
ipa	ipa = unaligned_addr(wrt "level" argument value supplied to the command. For example, if level = 2, provide an IPA that's 4KB aligned), unprotected_ipa walk(ipa).level < level RTTE[ipa].state = Assigned, Destroyed, Table	Must cover rule - R0254 The input address to an RTT walk is always less than 2^w, where w is the IPA width of the target Realm.
level	level = 0 when RMIFeatureregister0.LPA2 is not supported, else >3.	

Failure priority Ordering:

All failure priority ordering conditions specified in spec are tested as part of failure condition testing (multi-causal stimuli)

Observability:

Footprint	Verification
Command Failure	
RTTE.ripas	Refer **Observing Properties of a Granule** and **Observing Contents of a Granule** for details.
RIM	This needs to be tested outside of ACS command scenarios.
Command Success	
RTTE.ripas	Execute Valid RMI_RTT_INIT_RIPAS → success Execute RTT_READ_ENTRY and verify the outcome is as expected by the architecture.
RIM	This needs to be tested outside of ACS command scenarios.

2.2.19 RMI_RTT_MAP_UNPROTECTED

Argument List:

Input Parameter	Valid Values
rd	granule(rd) = 4K_ALIGNED granule(rd).state = RD
ipa	ipa = (level) aligned ipa = unprotected_ipa and within_permissible_ipa (< 2^Features0.S2SZ) walk(ipa).level = level RTTE[ipa].state = Unassigned or Destroyed
level	level = 3
desc	desc = attr_valid, output_addr_aligned to level

Failure condition testing:

Input Parameter	Input Values	Remarks
rd	granule(rd) = unaligned_addr, mmio_region (A), outside_of_permitted_ipa (B), not_backed_by_encryption, raz or wi (C) granule(rd).state = Undelegated, Delegated, REC, DATA, RTT	
ipa	ipa = unaligned_addr(wrt "level" argument value supplied to the command. For example, if level = 3, provide an IPA that's < 4KB aligned),protected_ipa, outside_permissible_ipa walk(ipa).level != level RTTE[ipa].state = Assigned (NS Granule)	
level	level = 0 (when RMIFeatureregister0.LPA2 is not supported), 1 (assuming this is pointing to a Table entry, that is there is no prior RTT_FOLD operation), 4	
desc	desc = rtte_addr_unaligned to level, attr_invalid (a value 1 in RES0 field, for example MemAttr[3] = 1)	

Failure priority Ordering:

All failure priority ordering conditions specified in spec are tested as part of failure condition testing (multi-causal stimuli)

Observability:

Footprint	Verification
Command Failure	
RTTE.state, RTTE.content	Refer **Observing Properties of a Granule** and **Observing Contents of a Granule** for details.
Command Success	
RTTE.state, RTTE.content	Execute Valid RTT_MAP_UNPROTECTED → success Execute RTT_READ_ENTRY → RTTE.state =ASSIGNED → RTTE.MemAttr = desc.MemAttr → RTTE.s2ap = desc.s2ap → RTTE.sh = desc.sh → RTTE.addr = desc.addr

2.2.20 RMI_RTT_READ_ENTRY

Argument List:

Input Parameter	Valid Values
rd	granule(rd) = 4K_ALIGNED granule(rd).state = RD
ipa	ipa = level_aligned ipa = within_permissible_ipa (< 2^Features0.S2SZ)
level	level = SL/SL+1 (depending on the value read from RMIFeatureregister0.S2SZ),2, 3

Failure condition testing:

Input Parameter	Input Values	Remarks
rd	granule(rd) = unaligned_addr, mmio_region (A), outside_of_permitted_ipa (B), not_backed_by_encryption, raz or wi (C) granule(rd).state = Undelegated, Delegated, REC, RTT, DATA	
ipa	ipa = unaligned_addrr(wrt "level" argument value supplied to the command. For example, if level = 3, provide an IPA that's < 4KB aligned), outside_permissible_ipa	

Input Parameter	Input Values	Remarks
level	level = 4.	

Failure priority Ordering:

This command has no failure priority orderings.

Observability:

Footprint	Verification
Command Failure	- Check for X1-X4 = zeros()
Command Success	
	Follow below steps for output values, Execute Valid RTT_READ_ENTRY → success Execute RTT_READ_ENTRY and verify the outcome is as expected by the architecture.
X1 (walk_level)	Create a maximum RTT depth (say, till level3 mapping) and use the same IPA aligned to corresponding level to cover various walk levels.
X2 (state)	Although some of this is tested in other commands as part of their successful execution, we will do this for completeness' sake in this command. see last row in this table.
X3 (desc)	Although some of this is tested in other commands as part of their successful execution, we will do this for completeness' sake in this command. see last row in this table.
X4 (ripas)	Although some of this is tested in other commands as part of their successful execution, we will do this for completeness' sake in this command. see last row in this table.
rtte(state) rtte(ripas)	Provide IPA whose state is UNASSIGNED/DESTROYED and validate X3(desc) as per spec Provide protected IPA whose state is ASSIGNED or IPA whose state is TABLE and validate X3(desc).MemAtt, X3(desc). S2AP, X3(desc).SH, and X3(desc).addr as per spec. Provide Unprotected IPA whose state is ASSIGNED and validate that X3(desc).MemAttr, X3(desc).S2AP, X3(desc).SH, and X3(desc).addr as per spec. Provide unprotected IPA/ provide IPA whose state is in DESTROYED/TABLE and validate X4 as per spec.

2.2.21 RMI_RTT_SET_RIPAS

Should be tested outside of command ACS (requires entry into Realm)

Argument List:

Input Parameter	Valid Values
rd	granule(rd) = 4K_ALIGNED granule(rd).state = RD

Input Parameter	Valid Values
rec	granule(rec) = 4K_ALIGNED granule(rec).state = REC Rec(rec).state = READY Rec(rec).owner = rd
ipa	ipa = level_aligned ipa = Rec(rec).ripas_addr ipa + size(level) < Rec(rec).ripas_top walk(ipa).level = level rtte_state = UNASSIGNED/ASSIGNED
level	level = 3. For level 0,1, and 2, the preparation sequence requires RTT_FOLD. Level0 can only be covered when RMIFeatureregister0.LPA2 is supported.
ripas	matches RSI_SET_RIPAS command value

Failure condition testing:

Input Parameter	Input Values	Remarks
rd	granule(rd) = unaligned_addr, mmio_region (A), outside_of_permitted_ipa (B), not_backed_by_encryption, raz or wi (C) granule(rd).state = Undelegated, Delegated, REC, RTT, DATA	
rec	granule(rec) = unaligned_addr, mmio_region (A), outside_of_permitted_ipa (B), not_backed_by_encryption, raz or wi (C) granule(rec).state = Undelegated, Delegated, RD, DATA, RTT Rec(rec).state = Running Rec(rec).owner = not_rd (D)	(D) Other RD than the one used to create "rec" should be provided here
ipa	ipa = unaligned_addr(wrt "level" argument value supplied to the command. For example, if level = 3, provide an IPA that's < 4KB aligned), not_ripas_addr walk(ipa).level < level RTTE[ipa].state = Table, Destroyed	
level	level = 0 (when RMIFeatureregister0.LPA2 is not supported), 1 (assuming this is pointing to a Table entry, that is there is no prior RTT_FOLD operation), 4, overflow (E)	(E) to trigger target_bound: ipa + size(level) > Rec(rec).ripas_top && ipa = ripas_addr
ripas	ripas = not_rec_ripas_value	

Failure priority Ordering:

Input Parameters	Input Values	Remarks
ipa	ipa = not_ripas_addr && [walk(ipa).level != level, RTTE[ipa].state = Table]	
ipa && level	ipa + size(level) > Rec(rec).ripas_top && [walk(ipa).level != level, RTTE[ipa].state = Table]	

Observability:

Footprint	Verification
Command Failure	
RTTE.ripas	Refer **Observing Properties of a Granule** and **Observing Contents of a Granule** for details.
Command Success	
RTTE.ripas	Execute Valid RMI_SET_RIPAS→ success Execute RTT_READ_ENTRY and verify the outcome is as expected by the architecture.
ripas_addr	

2.2.22 RMI_RTT_UNMAP_UNPROTECTED

Argument List:

Input Parameter	Valid Values
rd	granule(rd) = 4K_ALIGNED granule(rd).state = RD
ipa	ipa = level_aligned, unprotected_ipa and within_permissible_ipa (< 2^Features0.S2SZ) walk(ipa).level = level RTTE[ipa].state = ASSIGNED
level	level = 3

Failure condition testing:

Input Parameter	Input Values	Remarks
rd	rd = unaligned_addr, mmio_region (A), outside_of_permitted_ipa (B), not_backed_by_encryption, raz or wi (C) rd.state = Undelegated, Delegated, REC, DATA, RTT	
ipa	ipa = unaligned_addr(wrt "level" argument value supplied to the command. For example, if level = 3, provide an IPA that's < 4KB aligned), protected_ipa, outside_permissible_ipa walk(ipa).level != level RTTE[ipa].state = Unassigned, Destroyed	
level	level = 0 (when RMIFeatureregister0.LPA2 is not supported), 4	

Failure priority Ordering:

All failure priority ordering conditions specified in spec are tested as part of failure condition testing.

Observability:

Footprint	Verification
Command Failure	

Footprint	Verification
RTTE.state	Refer **Observing Properties of a Granule** and **Observing Contents of a Granule** for details.
Command Success	
RTTE.state	Execute Valid RMI_UNMAP_UNPROTECTED→ success Execute RTT_READ_ENTRY and verify the outcome is as expected by the architecture.

2.2.23 RMI_VERSION

Argument List:

- <none>

Failure condition testing:

This command has no failure conditions.

Failure priority Ordering:

This command has no failure priority orderings.

Observability:

This command has no footprint.

2.3 RSI Commands

2.3.1 RSI_ATTESTATION_TOKEN_CONTINUE

Tested as part of Security Model / Attestation scenarios.

Argument List:

Input Parameter	Valid Values
addr	addr=rec.attest_addr

Failure condition testing:

Input Parameter	Input Values	Remarks
addr	addr = not_rec_attest_gran Current.rec(attest_state) = NO_ATTEST_IN_PROGRESS	

Failure priority Ordering:

This command has no failure priority orderings.

Observability:

Footprint	Verification
Command Failure	
addr.owner.rec(attest_state)	
Command Success	
addr.owner.rec(attest_state)	Execute Valid RSI_ATTESTATION_TOKEN_CONTINUE until output = RSI_SUCCESS

2.3.2 RSI_ATTESTATION_TOKEN_INIT

Tested as part of Security Model / Attestation scenarios.

Argument List:

Input Parameter	Valid Values
addr	addr= 4K_ALIGNED addr = within_permissible_ipa (< 2^(IPA_WIDTH - 1))
[challenge_0 : 7]	

Failure condition testing:

Input Parameter	Input Values	Remarks
addr	addr = unaligned_addr, >= 2**(IPA_WIDTH - 1)	

Failure priority Ordering:

This command has no failure priority orderings.

Observability:

Footprint	Verification
Command Failure	
addr.owner.rec(attest_state)	Execute Invalid RSI_ATTESTATION_TOKEN_INIT → failure Execute Valid RSI_ATTESTATION_TOKEN_INIT → RSI_INCOMPLETE Execute Valid RSI_ATTESTATION_TOKEN_INIT → success
Command Success	
addr.owner.rec(attest_state)	Execute Valid RSI_ATTESTATION_TOKEN_INIT → success Execute Valid RSI_ATTESTATION_TOKEN_INIT → failure (will this give an error? → No, the process will be started anew)

2.3.3 RSI_HOST_CALL

Argument List:

Input Parameter	Valid Values
addr	addr= 4K_ALIGNED addr = within_permissible_ipa (< 2^(IPA_WIDTH - 1))

Failure condition testing:

Input Parameter	Input Values	Remarks
addr	addr = unaligned_addr, >= 2**(IPA_WIDTH - 1)	

Failure priority Ordering:

This command has no failure priority orderings.

Observability:

Footprint	Verification
Command Failure	
host_call	
Command Success	
host_call	

2.3.4 RSI_IPA_STATE_GET

Tested as part of Exception Model scenarios: Exit due to RIPAS (info)

Argument List:

Input Parameter	Valid Values
addr	addr= 4K_ALIGNED addr = Protected

Failure condition testing:

Input Parameter	Input Values	Remarks
addr	addr = unaligned_addr, addr >= 2**(IPA_WIDTH - 1)	

Failure priority Ordering:

This command has no failure priority orderings.

Observability:

This command has no footprint.

2.3.5 RSI_IPA_STATE_SET

Tested as part of Exception Model scenarios: Exit due to RIPAS (info)

Argument List:

Input Parameter	Valid Values
addr	addr= 4K_ALIGNED addr = Protected
size	size = 4K_ALIGNED
ripas	0 (EMPTY) or 1 (RAM)

Failure condition testing:

Input Parameter	Input Values	Remarks
addr	addr = unaligned_addr	
size	size = unaligned_size , size >	
ripas	Invalid encoding (ex: 0x2)	

Failure priority Ordering:

This command has no failure priority orderings.

Observability:

This command has no footprint.

2.3.6 RSI_MEASUREMENT_EXTEND

Tested as part of Security Model / Attestation scenarios.

Argument List:

Input Parameter	Valid Values
index	1 < index < 4
size	Size <= 64 bytes
value_0 : 7	

Failure condition testing:

Input Parameter	Input Values	Remarks
index	index = 0, 7	
size	size = 65	

Failure priority Ordering:

This command has no failure priority orderings

Observability:

Footprint	Verification
Command Failure	
realm.measurements[index]	Tested outside of ACS commands scenarios
Command Success	
realm.measurements[index]	Tested outside of ACS commands scenarios

2.3.7 RSI_MEASUREMENT_READ

Tested as part of Security Model / Attestation scenarios.

Argument List:

Input Parameter	Valid Values
index	Index = valid

Failure condition testing:

Input Parameter	Input Values	Remarks
index	index = 7	

Failure priority Ordering:

This command has no failure priority orderings.

Observability:

This command has no footprint.

2.3.8 RSI_REALM_CONFIG

Can be tested as part of any functional test within Realm side code.

Argument List:

Input Parameter	Valid Values
addr	addr = 4K_ALIGNED addr = Protected

Failure condition testing:

Input Parameter	Input Values	Remarks
addr	addr = unaligned_addr , addr >= 2**(IPA_WIDTH - 1)	

Failure priority Ordering:

This command has no failure priority orderings.

Observability:

This command has no footprint.

2.3.9 RSI_VERSION

Argument List:

- <none>

Failure condition testing:

This command has no failure conditions.

Failure priority Ordering:

This command has no failure priority orderings.

Observability:

This command has no footprint.

2.4 PSCI Commands

The following commands will be tested outside of the ACS command scenarios.

2.4.1 PSCI_AFFINITY_INFO

Tested as part of Exception Model scenarios: Exit due to PSCI.

Argument List:

Input Parameter	Valid Values
target_affinity	Assigned MPIDR.
Lowest_affinity_level	0

Failure condition testing:

Input Parameter	Input Values	Remarks
target_affinity	target_affinity = unassigned_mpidr	
lowest_affinity_level	lowest_affinity_level = 1	

Failure priority Ordering:

This command has no failure priority orderings.

Observability:

This command has no footprint.

2.4.2 PSCI_CPU_OFF

Argument List:

- <none>

Failure condition testing:

This command has no failure conditions.

Failure priority Ordering:

This command has no failure priority orderings.

Observability:

This command has no footprint.

2.4.3 PSCI_CPU_ON

Tested as part of Exception Model scenarios: Exit due to PSCI

not yet included

Argument List:

Input Parameter	Valid Values
target_cpu	target_cpu = assigned mpidr
entry_point_address	entry_point_address = Protected
context_id	

Failure condition testing:

Input Parameter	Input Values	Remarks
target_cpu	target_cpu = unassigned_mpidr target_cpu.flags.runnable = Runnable	
entry_point_address	entry_point_address = Unprotected	

Failure priority Ordering:

This command has no failure priority orderings.

Observability:

This command has no footprint.

2.4.4 PSCI_CPU_SUSPEND

Argument List:

Input Parameter	Valid Values
power_state	
entry_point_address	
context_id	

Failure condition testing:

This command has no failure conditions.

Failure priority Ordering:

This command has no failure priority orderings.

Observability:

This command has no footprint.

2.4.5 PSCI_FEATURES

Argument List:

Input Parameter	Valid Values
psci_func_id	

Failure condition testing:

This command has no failure conditions.

Failure priority Ordering:

This command has no failure priority orderings.

Observability:

This command has no footprint.

2.4.6 PSCI_SYSTEM_OFF

Argument List:

- <none>

Failure condition testing:

This command has no failure conditions.

Failure priority Ordering:

This command has no failure priority orderings.

Observability:

This command has no footprint.

2.4.7 PSCI_SYSTEM_RESET

Argument List:

- <none>

Failure condition testing:

This command has no failure conditions.

Failure priority Ordering:

This command has no failure priority orderings.

Observability:

This command has no footprint.

2.4.8 PSCI_VERSION

Argument List:

- <none>

Failure condition testing:

This command has no failure conditions.

Failure priority Ordering:

This command has no failure priority orderings.

Observability:

This command has no footprint.