



Firmware Interfaces for Realm Management Extension

Document number	DEN0149
Document quality	ALP2
Document version	1.0
Document confidentiality	Non-confidential

Copyright © 2026 Arm Limited or its affiliates. All rights reserved.

Firmware Interfaces for Realm Management Extension

Release information

1.0 ALP2 (2026-02-23)

- Changes to common guidance
 - Split FIRME ABIs into independently versioned services
 - Clarify description of common error codes
 - Clarify register usage to support forward and backward compatibility
 - Add guidance for TOCTOU attacks associated with shared buffers
 - State compliance rules for FIRME ABIs of each independently versioned service
 - Downgrade rule about duration for which a shared buffer remains mapped in EL3 into an information statement
 - Clarify GPI encoding of shared buffer must correspond to its PAS
 - Clarify scenarios when a FIRME ABI returns an error if EL3 firmware is unable to map a shared buffer
 - Remove INVALID_REQUEST error code and rationalise usage of other error codes
 - Miscellaneous fixes and language clarifications
- Changes to attestation token management guidance
 - Remove RAT_SIGN field from FIRME_FEATURES feature register 2
 - Return INCOMPLETE instead of SUCCESS in key and token retrieval ABIs when retrieval is incomplete
- Changes to granule management guidance
 - Mandate scrubbing of L1 GPT object during FIRME_GM_L1_GPT_DESTROY
 - Clarify FIRME_GM_L1_GPT_CREATE has no effect if DENIED is returned
 - Reject FIRME_GM_L1_GPT_CREATE if invoked for same PAR with different L1 GPT object instead of terminating the old operation
 - Clarify permitted GPI transitions relative to available RME extensions
 - Clarify why scrubbing of granules during a PAS transition is applicable to other transitions apart from only NSO <-> NS
 - Add error code to FIRME_L1_GPT_CREATE when L1 GPT is already present
 - Add error code to FIRME_GM_GPI_SET when L1 GPT is missing
 - Add FIRME_GM_GPI_OP_CONTINUE to support pause and resume of GPI transition operations
- Changes to IDE key management guidance
 - Add a segment ID field to the keyset ID
- Changes to Integrated device management guidance
 - Add ABIs to support DA flows for integrated devices/PCIe RCiEPs.
- Changes to MEC management guidance
 - Mark FIRME_MEC_REFRESH ABI's quality as BETA.

1.0 ALP1 (2025-10-24)

- Changes to common guidance
 - Replace MFI prefix with FIRME
 - Correct FIRME_FEATURES ABI definition to include output register that holds the value of the feature register
 - Specify the PAS of a shared buffer at a FIRME instance
 - Add rule to state that a shared buffer is mapped in EL3 for at least the duration of a FIRME call that uses it
 - Clarify SBZ/MBZ behaviour of unused input/output registers and unused fields in registers
 - Clarify the role of a caller and a callee at a FIRME instance
 - Split the top level Concept section into sub-sections for readability
 - Clarify that use of FIRME on systems that do not implement FEAT_RME is not precluded
 - Renamed MAX_SH_BUF_SZ parameter to MAX_SH_BUF_PG_CNT to better reflect its purpose
 - Miscellaneous fixes and language clarifications
- Changes to attestation token management guidance

- Add attestation ABIs to extend platform token for CXL-T3 devices
- List types of ECC curve in FIRME_ATTEST_RAK_GET
- Reorganise guidance for the ABORTED error code w.r.t attestation ABIs
- Enable implementation of FIRME_ATTEST_RAT_SIGN to return the ABORTED error code
- Clarify that the CCA HES provides a generic attestation service instead of only the delegated attestation service
- Renamed MAX_PAT_SZ parameter to MAX_PAT_PG_CNT to better reflect its purpose
- Added error condition in FIRME_ATTEST_PAT_GET and FIRME_ATTEST_RAK_GET when retrieval is continued without being initiated
- Added SUCCESS return status code to pseudocode for FIRME_ATTEST_PAT_GET and FIRME_ATTEST_RAK_GET
- Correct size of shared buffer in pseudocode for FIRME_ATTEST_RAK_GET
- Clarified why retrieval of the private portion and public portion of RAK is mutually exclusive
- Clarified that FIRME_ATTEST_RAT_SIGN is optional
- Changes to granule management guidance
 - Add ABIs to create and destroy L1 GPTs for a physical address range
 - Allow GPI transitions between NSO and NSP when FEAT_RME_GDI is implemented
 - Remove the *Current GPI encoding* field from FIRME_GM_GPI_SET
 - Clarify why output granule count could be less than the input granule count in FIRME_GM_GPI_SET
 - Add a reference to software flows of FIRME_GM_GPI_SET where this ABI is introduced for better readability
- Changes to IDE key management guidance
 - Specify encoding of an IDE key in the input registers of the FIRME_IDE_KEYSET_PROG ABI
 - Renamed *Request type* field in all IDE key configuration ABIs to *IDE type*.
 - Add rule to state that IDE key configuration interface is either available at the NS FIRME instance or the Realm FIRME instance
 - Add rule to state that either all or no IDE key configuration ABIs are implemented at a FIRME instance
 - Update IDE key programming software flows to reflect RMM enables selective IDE stream at endpoint instead of the Host
 - Clarify that limit on IDE key configuration requests in non-blocking mode is IMPLEMENTATION DEFINED and RETRY is returned once exceeded
- Changes to MEC management guidance
 - Add a rule to state ciphertext requirements after a MECID refresh
 - Clarify usage of the MEC refresh reason field in FIRME_MEC_REFRESH

1.0 ALP0 (2025-07-09)

- Replace FIRME prefix with MFI (Monitor Firmware Interface) in ABI and register names to enable usage on systems that do not implement FEAT_RME
- Add guidance for using a shared buffer with EL3 firmware
- Add MFI_FEATURES ABI to discover implemented ABIs
- Add support to discover Physical Granule size via the MFI_FEATURES ABI
- Add support to discover value of Level 0 GPT entry size as encoded in GPCCR_EL3.LOGPTSZ via the MFI_FEATURES ABI
- Add support to discover value of Protected Physical Address Size as encoded in GPCCR_EL3.PPS via the MFI_FEATURES ABI
- Add support to discover Common MECID width via the MFI_FEATURES ABI
- Add support to discover minimum size and alignment boundary of a shared buffer via the MFI_FEATURES ABI
- Add support to discover maximum size of a shared buffer via the MFI_FEATURES ABI
- Add support to discover maximum size of the platform attestation token via the MFI_FEATURES ABI
- Add support to discover support for retrieval of the public portion of the Realm attestation key via the MFI_FEATURES ABI
- Add support to discover support to discover encoding of Realm attestation key via the MFI_FEATURES ABI
- Add support to discover support for signing a Realm attestation token via the MFI_FEATURES ABI
- Clarify that the NSO GPI encoding is available only when FEAT_RME_GPC2 is implemented
- Add pseudocode for GPI transition flows between NS and NSO
- Add support for GPI transition flows for FEAT_RME_GDI
- Add pseudocode for GPI transition flows between NS and NSP
- Add ABIs to support the Arm CCA delegated attestation flow

- RETRY error code is used in lieu of the BUSY error code
- INCOMPLETE error code is used in lieu of PENDING error code
- Remove NO_MEMORY and OUT_OF_RESOURCE error codes as they are unused
- Add concept of granule security policy to better describe permitted GPI transitions
- Clarify that terms used in Keyset ID are defined in the PCIe spec
- Tweak MFI_GM_GPI_SET ABI to avoid scanning all GPI encoding of all granules before performing a GPI transition
- Extend MFI IDE key configuration ABIs to support IDE key management for CXL devices
- Allow IDE key configuration ABIs to be invoked from the Normal world
- Add IMPLEMENTATION_DEFINED cookie parameters to IDE key configuration ABIs to facilitate matching of requests to responses
- Miscellaneous fixes and language clarifications

1.0 DEV0 (2025-04-11)

- First DEV version that includes ABIs for MEC management, Granule management and IDE key management

Non-Confidential Proprietary Notice

This document is protected by copyright and other related rights and the practice or implementation of the information contained in this document may be protected by one or more patents or pending patent applications. No part of this document may be reproduced in any form by any means without the express prior written permission of Arm. **No license, express or implied, by estoppel or otherwise to any intellectual property rights is granted by this document unless specifically stated.**

Your access to the information in this document is conditional upon your acceptance that you will not use or permit others to use the information for the purposes of determining whether implementations infringe any third party patents.

THIS DOCUMENT IS PROVIDED “AS IS”. ARM PROVIDES NO REPRESENTATIONS AND NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, SATISFACTORY QUALITY, NON-INFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO THE DOCUMENT. For the avoidance of doubt, Arm makes no representation with respect to, and has undertaken no analysis to identify or understand the scope and content of, patents, copyrights, trade secrets, or other rights.

This document may include technical inaccuracies or typographical errors.

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

This document consists solely of commercial items. You shall be responsible for ensuring that any use, duplication or disclosure of this document complies fully with any relevant export laws and regulations to assure that this document or any portion thereof is not exported, directly or indirectly, in violation of such export laws. Use of the word “partner” in reference to Arm’s customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

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

The Arm corporate logo and words marked with ® or ™ are registered trademarks or trademarks of Arm Limited (or its affiliates) in the US and/or elsewhere. All rights reserved. Other brands and names mentioned in this document may be the trademarks of their respective owners. Please follow Arm’s trademark usage guidelines at <http://www.arm.com/company/policies/trademarks>.

Copyright © 2026 Arm Limited (or its affiliates). All rights reserved.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20349 version 21.0

Quality level

The following table identifies features that are designated at a quality level different from the overall specification. Any feature not explicitly listed in the table remains at the ALPHA quality level.

Feature	Quality level
FIRME_MEC_REFRESH	BETA

Contents

Firmware Interfaces for Realm Management Extension

Firmware Interfaces for Realm Management Extension	ii
Release information	ii
Non-Confidential Proprietary Notice	v
Quality level	v

Preface

About this book	ix
Using this book	x
Conventions	xi
Typographical conventions	xi
Numbers	xi
Pseudocode descriptions	xi
Assembler syntax descriptions	xi
Rules-based writing	xii
Content item identifiers	xii
Content item rendering	xii
Content item classes	xii
Additional reading	xiv
Feedback	xv
Feedback on this book	xv

Chapter 1

Overview

Chapter 2

Concepts

2.1	Instances	18
2.2	SMC interface	18
2.3	FIRME services	21
2.4	Versioning policy	22
2.5	Feature discovery	24
2.6	Compliance	27
2.7	Long-running operations	27
2.8	Shared buffer	28
2.8.1	Data retrieval in chunks	30

Chapter 3

Granule management service

3.1	Overview	31
3.2	Scope	31
3.3	Concepts	32
3.3.1	GPI transitions	32
3.3.2	GPI management ABIs	38
3.3.3	L1 GPT management	44
3.3.4	Feature discovery	51
3.3.5	Compliance	51

Chapter 4

IDE key management service

4.1	Overview	53
4.2	Scope	53
4.3	Concepts	53
4.3.1	Feature discovery	59

	4.3.2	Compliance	60
Chapter 5		MECID management service	
	5.1	Overview	61
	5.2	Scope	61
	5.3	Concepts	62
	5.3.1	Feature discovery	62
	5.3.2	Compliance	63
Chapter 6		Attestation service	
	6.1	Overview	64
	6.2	Scope	64
	6.3	Concepts	64
	6.3.1	Shared buffer usage	65
	6.3.2	Platform attestation token retrieval	66
	6.3.3	Realm attestation key retrieval	68
	6.3.4	Realm attestation token signing	71
	6.3.5	Platform attestation token extension	73
	6.3.6	Aborted operations	76
	6.3.7	Feature discovery	77
	6.3.8	Compliance	78
Chapter 7		Integrated device management service	
	7.1	Overview	80
	7.2	Scope	80
	7.3	Concepts	80
	7.3.1	Feature discovery	85
	7.3.2	Compliance	86
Chapter 8		Interface	
	8.1	FIRME_SERVICE_VERSION	88
	8.2	FIRME_SERVICE_FEATURES	89
	8.3	FIRME_GM_GPI_SET	90
	8.4	FIRME_GM_GPI_OP_CONTINUE	91
	8.5	FIRME_GM_L1_GPT_CREATE	92
	8.6	FIRME_GM_L1_GPT_DESTROY	93
	8.7	FIRME_IDE_KEYSET_PROG	94
	8.8	FIRME_IDE_KEYSET_GO	95
	8.9	FIRME_IDE_KEYSET_STOP	96
	8.10	FIRME_IDE_KEYSET_POLL	97
	8.11	FIRME_MEC_REFRESH	99
	8.12	FIRME_ATTEST_PAT_GET	100
	8.13	FIRME_ATTEST_RAK_GET	101
	8.14	FIRME_ATTEST_RAT_SIGN	103
	8.15	FIRME_ATTEST_PAT_EXT_CLAIMS_STAGE	105
	8.16	FIRME_ATTEST_PAT_EXT_CLAIMS_CLEAR	106
	8.17	FIRME_ATTEST_PAT_EXT_CLAIMS_FINALISE	107
	8.18	FIRME_IDEV_OP_START	108
	8.19	FIRME_IDEV_OP_CONTINUE	109

Glossary

Preface

About this book

Using this book

Conventions

Typographical conventions

The typographical conventions are:

italic

Introduces special terminology, and denotes citations.

bold

Denotes signal names, and is used for terms in descriptive lists, where appropriate.

`monospace`

Used for assembler syntax descriptions, pseudocode, and source code examples.

Also used in the main text for instruction mnemonics and for references to other items appearing in assembler syntax descriptions, 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.

Assembler syntax descriptions

This book contains numerous syntax descriptions for assembler instructions and for components of assembler instructions. These are shown in a monospace font.

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_{iiii}

- L is a label that indicates the content class of the content item.
- $iiii$ 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] *Arm® Architecture Reference Manual Armv8, for Armv8-A architecture profile*. See <https://developer.arm.com/documentation/ddi0487/latest>
- [2] *SMC Calling Convention v1.2*. See <https://developer.arm.com/documentation/den0028/c>
- [3] *Realm Management Monitor specification*. See <https://developer.arm.com/documentation/den0137/latest>
- [4] *Arm® FF-A memory management protocol version 1.2*. See <https://developer.arm.com/documentation/den0140/a>
- [5] *PCI Express 6.0 specification*. See <https://pcisig.com/pci-express-6.0-specification>
- [6] *CXL 3.2 specification*. See <https://computeexpresslink.org/cxl-specification/>
- [7] *RME system architecture spec*. See <https://documentation-service.arm.com/static/60d3309b677cf7536a55bae0>
- [8] *Arm CCA Security Model*.
- [9] *CBOR Object Signing and Encryption (COSE)*. See <https://datatracker.ietf.org/doc/html/rfc8152>

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 (Firmware Interfaces for Realm Management Extension).
- The number (DEN0149 1.0).
- 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.

Chapter 1

Overview

I_0001

The Realm Management Extension (FEAT_RME) [1] is a key component of the Arm Confidential Compute Architecture (Arm CCA). Together with the other components of Arm CCA, FEAT_RME enables support for dynamic, attestable, and trusted execution environments (Realms) to be run on an Arm PE. The software flows for Realm management have dependencies on EL3 firmware e.g., flows for construction of Realms, allocation and isolation of resources to Realms, attestation of initial state of Realms. This document (henceforth referred to as the FIRME specification) defines standard interfaces presented by EL3 firmware to fulfil the following dependencies:

- Granule Protection Table management.
- IDE key management at PCIe and CXL Root ports.
- Memory Encryption Context management.
- Attestation token management.
- Integrated device management.

Each set of standard interfaces that fulfils a dependency constitutes a FIRME service. Additionally, a FIRME Base service is specified in this chapter for the following purposes:

- Specify concepts that are common to all FIRME services.
- Discover the presence of a FIRME service.
- Discover the features of an implemented FIRME service.

An implementation of the FIRME specification on a system that does not implement FEAT_RME is permitted. E.g., Attestation token management interfaces could be used by the Host to obtain an attestation report from the subsystem that serves as the hardware root-of-trust.

[Figure 1.1](#) illustrates that the FIRME specification is implemented at the EL boundary between EL3 and the Non-secure, Realm and Secure security states.

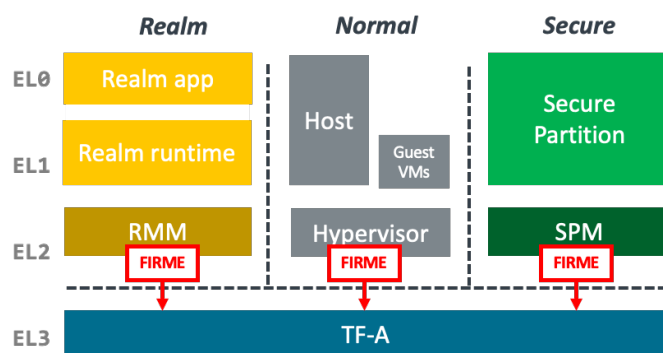


Figure 1.1: Firmware interfaces for RME

Chapter 2

Concepts

2.1 Instances

- D₀₀₀₂ An implementation of one or more FIRME services at a valid Exception level boundary between EL3 and the highest Exception level in a different Security state is called a *FIRME instance*. The software component resident in the latter Exception level is the *caller* of the ABIs implemented by the FIRME service. Firmware resident in EL3 is the *callee* of these ABIs.
- I₀₀₀₃ An [FIRME instance](#) uniquely identifies an implementation of one or more FIRME services in EL3 firmware based on the Security state of the caller.
- D₀₀₀₄ An implementation of one or more FIRME services at the Exception level boundary between EL3 and the highest implemented Exception level in the Non-secure Security state is called the *NS FIRME instance*.
- D₀₀₀₅ An implementation of one or more FIRME services at the Exception level boundary between EL3 and the highest implemented Exception level in the Secure Security state is called the *Secure FIRME instance*.
- D₀₀₀₆ An implementation of one or more FIRME services at the Exception level boundary between EL3 and the highest implemented Exception level in the Realm Security state is called the *Realm FIRME instance*.

2.2 SMC interface

- R₀₀₀₇ Each ABI implemented at any [FIRME instance](#) is compliant with SMCCC version ≥ 1.2 . See also [2].
- R₀₀₀₈ If FEAT_SVE is implemented, each ABI implemented at any [FIRME instance](#) is compliant with SMCCC version ≥ 1.3 .
- R₀₀₀₉ If FEAT_SME is implemented, each ABI implemented at any [FIRME instance](#) is compliant with SMCCC version ≥ 1.4 .

- R₀₀₁₀ Each ABI implemented at any [FIRME instance](#) uses the SMC64 calling convention.
- R₀₀₁₁ ABIs defined in this version of the FIRME specification use the status codes defined in [Table 2.1](#).

Table 2.1: FIRME error status codes

Status code	Name	Description
0	SUCCESS	- If the requested operation is synchronous, one of the following applies: - The operation is complete and its results are visible to the caller. - The operation is partially complete such that no internal state related to the remainder of the operation persists in the callee, that can be observed by the caller. * The caller subsequently requests completion of the remainder of the operation. - If the requested operation is asynchronous, the operation is in progress. - The caller polls for completion of the operation.
-1	NOT_SUPPORTED	- The ABI is not implemented at this FIRME instance. - The ABI invocation had no effect.
-2	INVALID_PARAMETERS	- The requested operation could not start because at least one input parameter of the ABI is invalid. - The ABI invocation had no effect.
-3	ABORTED	- The requested operation was terminated before completion. The state of the system pertaining to the operation is UNKNOWN. - Recovery from this state requires a system reset unless the operation description specifies an alternative recovery method.
-4	INCOMPLETE	- The requested synchronous operation is incomplete such that internal state related to the incomplete operation persists in the callee, and could be observed by the caller. - The caller subsequently requests completion of the remainder of the operation. The caller may have to include a cookie as an input parameter to reference the saved internal state. - The requested asynchronous operation is incomplete. - The caller polls for the completion of the operation.
-5	DENIED	- The requested operation could not complete because the caller is not permitted to request the entire operation, or a part of it. - If no part of the operation completed, the ABI invocation had no effect. - Otherwise, the ABI description specifies one or more of the following: * The progress of the operation. * The presence of any internal state in the callee related to the remainder of the operation.

Status code	Name	Description
-6	RETRY	- The requested operation could not start because a resource required to perform the requested operation is unavailable temporarily. - The caller can retry the operation. - The ABI invocation had no effect.
-7	IN_PROGRESS	- Another operation is already in progress on a subset of objects required by the requested operation. - The caller should allow the other operation to complete before retrying its operation. - If no part of the requested operation completed, the ABI invocation had no effect. - Otherwise, the ABI description specifies one or more of the following: * The progress of the operation. * The presence of any internal state in the callee related to the remainder of the operation.
-8	EXISTS	- The requested operation attempts to create an object that already exists. - The ABI invocation had no effect.
-9	NO_ENTRY	- An object required to complete the requested operation does not exist. - The ABI invocation had no effect.
-10	NO_MEMORY	- The callee was unable to allocate memory required to complete the requested operation. - The ABI invocation had no effect.
-11	BAD_DATA	- The callee was unable to complete the requested operation because the input data is malformed. - The ABI invocation had no effect.

- R₀₀₁₂ An unused input parameter register in a FIRME ABI is Reserved and SBZ.
- R₀₀₁₃ An unused bit in an input parameter register in a FIRME ABI is Reserved and SBZ.
- R₀₀₁₄ An unused output parameter register in a FIRME ABI is Reserved, IGNORED by the caller, and has a value that cannot be used to compromise the security of the callee.
- R₀₀₁₅ An unused bit in an output parameter register in a FIRME ABI is Reserved, IGNORED by the caller, and has a value that cannot be used to compromise the security of the callee.
- X₀₀₁₆ A FIRME ABI is backwards compatible as described below:
- Backward compatibility implies that a callee at a higher compatible version is able to correctly interpret information from a caller at a lower compatible version.
 - The SMC64 calling convention in [2] specifies 17 input parameter registers ranging from x1-x17 (excluding x0 which encodes the function identifier).

- An ABI definition in a version of a FIRME service specifies the input parameter registers that are being used.
- An unused input parameter register is Reserved, and should have a default value of 0.
- A callee treats the default value as a nop, that is, this value has no effect on the behaviour of the ABI.
- It is strongly recommended that the caller populates the unused input parameter registers with the default value. This prevents unexpected behaviour in an ABI invocation because of the following reasons:
 - The caller is at a lower compatible version, and does not zero the unused input parameter registers.
 - The callee is at a higher compatible version that defines one or more unused input parameter registers, it is not aware of the caller's version, and it interprets the junk values in the unused input parameter registers as valid input.

X₀₀₁₇ The SBZ constraint on unused input parameter registers reflects the strong recommendation to use the default value of 0. This is preferred over the MBZ constraint where the caller must zero all unused input parameter registers. It is possible that even though the callee is at a higher compatible version than the caller, it does not use all 17 input parameter registers. The MBZ constraint requires the callee to validate registers that are unused by both the caller and that callee, are zeroed. The SBZ constraint avoids this unnecessary additional check.

X₀₀₁₈ A FIRME ABI is forwards compatible as described below:

- Forward compatibility implies that a caller at a lower compatible version is able to correctly interpret information from a callee at a higher compatible version.
- The SMC64 calling convention in [2] specifies 17 output parameter registers ranging from x1-x17 (excluding w0 which encodes the return status).
- An ABI definition in a version of a FIRME service specifies the output parameter registers that are being used.
- An unused output parameter register is Reserved, must be ignored by the caller, and populated with a value that cannot be used to compromise the security of the callee at a higher Exception level.
- The caller ignores any output parameter register that is unused in its version of the ABI, but is used in the callee's version of the ABI. This prevents unexpected behaviour in an ABI completion because of the following reasons:
 - The callee is at a higher compatible version that defines one or more output parameter registers that the caller considers unused, it is not aware of the caller's version, and it populates valid values in these output parameter registers.
 - The caller is at a lower compatible version, and neither ignores the unused output parameter registers nor is able to interpret them correctly.

I₀₀₁₉ The contents of unused input parameter registers are not guaranteed to be preserved when an invocation of a FIRME ABI completes. The caller should not rely on this behaviour for correct operation.

2.3 FIRME services

D₀₀₂₀ A *FIRME service* is a collection of related interfaces presented by EL3 firmware.

I₀₀₂₁ This version of the FIRME specification specifies the following services apart from the Base service:

- Granule management service. See also:
 - [Chapter 3 Granule management service.](#)
- IDE key management service. See also:
 - [Chapter 4 IDE key management service.](#)
- MECID management service. See also:
 - [Chapter 5 MECID management service.](#)
- Attestation service. See also:
 - [Chapter 6 Attestation service.](#)

- Integrated device management service. See also:
– [Chapter 7 Integrated device management service](#).

D₀₀₂₂ A FIRME service is identified by its *Service ID*.

R₀₀₂₃ A FIRME [service ID](#) is an 8-bit unsigned integer.

R₀₀₂₄ Each FIRME service is allocated a [service ID](#) as specified in [Table 2.2](#). All unallocated values are unused.

Table 2.2: FIRME service ID allocation

Service name	Service ID
Base service	0x0
Granule management service	0x1
IDE key management service	0x2
MECID management service	0x3
Attestation service	0x4
Integrated device management service	0x5

R₀₀₂₅ The list of FIRME services that can be implemented at each [FIRME instance](#) is specified in [Table 2.3](#).

Table 2.3: Presence of FIRME services

Service name	NS FIRME instance	Secure FIRME instance	Realm FIRME instance
Base	Yes	Yes	Yes
Granule management	Yes	Yes	Yes
IDE key management	Yes	No	Yes
MECID management	No	No	Yes
Attestation	Yes	No	Yes
Integrated device management	No	No	Yes

R₀₀₂₆ The FIRME base service is implemented at all [FIRME instances](#).

I₀₀₂₇ A usable FIRME implementation at a [FIRME instance](#) comprises of the FIRME base service and at least another FIRME service. EL3 firmware chooses the latter service as per [Table 2.3](#).

For example, EL3 firmware can choose to implement all FIRME services at the [Realm FIRME instance](#), and only the Base service and the Granule management service at the remaining instances.

2.4 Versioning policy

I₀₀₂₈ Each FIRME service has a distinct version. The version of the FIRME specification is derived from the version of the services it specifies.

I₀₀₂₉ A version is a 31-bit unsigned integer that comprises of a major version number and a minor version number. The upper 15 bits encode the major revision, and the lower 16 bits encode the minor revision.

- I₀₀₃₀ Each version of the FIRME specification identifies a unique combination of versions of the FIRME services it specifies.
- R₀₀₃₁ The minor version number of the FIRME specification is incremented if any of the following is true:
- The minor version number of one or more FIRME services is incremented.
 - A new service is added to the FIRME specification.
- The major version number of the FIRME specification is incremented if any of the following is true:
- The major version number of one or more FIRME services is incremented.
 - An existing service is removed from the FIRME specification.
- I₀₀₃₂ A minor version number increment of the FIRME specification implies that all functionality from the preceding version of the specification is preserved, and new functionality is added to the specification. For example:
- One or more ABIs in one or more FIRME services undergo a backwards compatible change.
 - One or more ABIs are added to one or more FIRME services.
 - One or more FIRME services are added.
- A major version number increment of the FIRME specification implies that some functionality from the preceding version of the specification is not preserved. For example:
- One or more ABIs in one or more FIRME services undergo a backwards incompatible change.
 - One or more ABIs from one or more FIRME services are removed.
 - One or more FIRME services are removed.
- The following table illustrates an example evolution of the version of the FIRME specification and the versions of the services it specifies.

Service A version	Service B version	Service C version	Service D version	Specification version
1.0	1.0	1.0	Absent	1.0
1.1	1.1	1.0	Absent	1.1
1.1	1.1	1.1	1.0	1.2
1.2	2.0	1.1	1.0	2.0
1.3	2.0	1.2	2.0	3.0
1.3	3.0	2.0	3.0	4.0
Absent	3.0	2.0	3.0	5.0

- R₀₀₃₃ The following rules apply to the major version and minor version numbers in a version of a FIRME service:
- Different major revision values indicate possibly incompatible functions. A newer major revision might:
 - Introduce new functions
 - Deprecate older functions
 - Change behavior of existing functions
 - For two revisions, A and B, for which the major revision values are identical, if the minor revision value of revision B is greater than the minor revision value of revision A, then every function in revision A must work in a compatible way with revision B. However, revision B can have a higher function count than revision A.
- R₀₀₃₄ The versions of each component in this revision of the FIRME specification are listed in [Table 2.5](#).

Table 2.5: FIRME specification and service versions

Component	Major version number	Minor version number
Base service	1	0
Granule management service	1	0
IDE key management service	1	0
MECID management service	1	0
Attestation service	1	0
Integrated device management service	1	0

R₀₀₃₅ Each version of a FIRME service depends upon a version of the FIRME base service as listed in [Table 2.6](#).

Table 2.6: FIRME base service version dependencies

Component	FIRME base service major version number	FIRME base service minor version number
Granule management service	1	0
IDE key management service	1	0
MECID management service	1	0
Attestation service	1	0
Integrated device management service	1	0

I₀₀₃₆ EL3 firmware is permitted to implement different versions of a FIRME service at different [FIRME instances](#). Furthermore, presence of a FIRME service at one [FIRME instance](#) does not imply its presence at other instances. If different versions of a FIRME service are implemented at different [FIRME instances](#), the compatibility between these versions is IMPLEMENTATION DEFINED.

For example, the implementations of the Host and the RMM could be tightly coupled such that both expect the same version of the FIRME granule management service at the [NS FIRME instance](#) and the [Realm FIRME instance](#). The software integrator of such a system ensures that the FIRME implementation in EL3 firmware fulfils this constraint.

I₀₀₃₇ The [FIRME_SERVICE_VERSION](#) interface is used to obtain the version of a FIRME service implemented at the [FIRME instance](#). The *Service ID* input parameter specifies the FIRME service whose version must be returned.

R₀₀₃₈ An invocation of the [FIRME_SERVICE_VERSION](#) ABI returns the version of a FIRME service, and completes with the [SUCCESS](#) return status code, if all of the following are false:

- The FIRME service corresponding to the *Service ID* input parameter is not implemented at this [FIRME instance](#).
 - In this case, the ABI invocation completes with the [NOT_SUPPORTED](#) return status code.
 - See also [Table 2.3](#).

2.5 Feature discovery

D₀₀₃₉ The features implemented by a FIRME service at a [FIRME instance](#) are discovered by reading a feature register.

- I₀₀₄₀ Each FIRME service specifies a set of features that enables its users to correctly use the service. For example:
- The Base service specifies the minimum and maximum size of a buffer that any other service can share with EL3 firmware for data transfer.
 - The Granule management service exports the Protected Granule size used by EL3 firmware which is used to determine the size and alignment of granules that must undergo a [GPI transition](#).

Also, a FIRME service at one [FIRME instance](#) may support a different set of features from its implementation at another instance. Such properties of a service can be discovered via the [feature registers](#).

R₀₀₄₁ The size of a [feature register](#) is a non-zero multiple of 64-bits.

X₀₀₄₂ The value of a [feature register](#) is returned in one or more 64-bit general purpose registers.

I₀₀₄₃ The maximum size of a [feature register](#) is bounded by the maximum number of 64-bit result registers specified by the SMC64 calling convention in [2]. For example, in SMCCC version ≥ 1.2 , x1-x17 are result registers. The maximum size of a [feature register](#) is 136 bytes.

R₀₀₄₄ A 64-bit result register N of the SMC64 convention encodes $\text{Bits}[64N-1:64(N-1)]$ of a [feature register](#).

I₀₀₄₅ In SMCCC version ≥ 1.2 , the encoding of a [feature register](#) in SMC64 result registers is as follows:

Feature register bits	SMC64 result register
Bits[63:0]	x1
Bits[127:64]	x2
Bits[191:128]	x3
.	.
.	.
.	.
Bits[1081:1024]	x17

D₀₀₄₆ A feature register is identified by a feature register index.

R₀₀₄₇ A feature register index is an 8-bit unsigned integer.

R₀₀₄₈ If a FIRME service implemented at a [FIRME instance](#), the feature register 0 reports which ABIs of that service are implemented as follows:

- Each ABI is assigned a bit in the register.
- If the value of the bit is 1, the ABI is implemented. Otherwise, the ABI is not implemented.

I₀₀₄₉ The FIRME base service specifies the feature register 0 as a common mechanism to discover the ABIs implemented by a FIRME service. The size of this register can vary across FIRME services, but the encoding remains the same. See also:

- [Table 2.8](#).
- [Table 3.7](#).
- [Table 4.1](#).
- [Table 5.1](#).
- [Table 6.3](#).
- [Table 7.6](#).

I₀₀₅₀ This version of the FIRME Base service specifies the following feature registers:

- Feature register 0.
- Feature register 1.

R₀₀₅₁ The size of each feature register, its index, and its encoding is specified in [Table 2.8](#). All unallocated values are unused.

Table 2.8: FIRME Base service feature registers

Feature register name	Index	Size	Encoding
Feature register 0	0x1	64-bit	• Bits[63:2]: Unused. See also 2.2 SMC interface. • Bits[1]: Indicates presence of the FIRME_SERVICE_FEATURES ABI. • Bits[0]: Indicates presence of the FIRME_SERVICE_VERSION ABI. • Value of each bit is determined from Table 2.9.
Feature register 1	0x0	64-bit	• Bits[63:32]: Unused. See also 2.2 SMC interface. • Bits[31:16]: FIRME service list. – Each bit indicates whether a service is implemented at this FIRME instance as follows: * 0b0: Service is not implemented. * 0b1: Service is implemented. – A bit is allocated to a FIRME service as follows: * Bit[4]: Integrated device management service. * Bit[3]: Attestation service. * Bit[2]: MECID management service. * Bit[1]: IDE key management service. * Bit[0]: Granule management service. – All unallocated bits are unused. See also 2.2 SMC interface. • Value of each bit is determined from Table 2.3. • Bits[15:2]: MAX_SH_BUF_PG_CNT. – Value of this field is used to determine the maximum size of a shared buffer as follows: * If P = Maximum size of shared buffer (in KB). * If Q = Minimum size of shared buffer (in KB). · This is derived from the value of the MIN_SH_BUF_SZ field. * Then $P = (MAX_SH_BUF_PG_CNT + 1) * Q.$

Feature register name	Index	Size	Encoding
			• Bits[1:0]: MIN_SH_BUF_SZ. – Value of this field is used to determine the minimum size, and alignment boundary of a shared buffer as follows: * 0b00: 4KB. * 0b01: 64KB. * 0b10: 16KB. – All other values are unused. See also 2.2 SMC interface

- I₀₀₅₂ A [feature register](#) is read via the [FIRME_SERVICE_FEATURES](#) ABI. The *Service ID* input parameter specifies the FIRME service whose feature register must be returned. The *Feature register index* input parameter specifies the feature register that must be returned.
- R₀₀₅₃ An invocation of the [FIRME_SERVICE_FEATURES](#) ABI returns the FIRME [feature register](#) requested by the caller, and completes with the [SUCCESS](#) return status code, if all of the following are false:
- The FIRME [feature register](#) corresponding to the combination of the *Service ID*, and *Feature register index* input parameters is not implemented at this [FIRME instance](#).
 - In this case, the ABI invocation completes with the [NOT_SUPPORTED](#) return status code.
 - See also:
 - * [Table 2.3](#).

2.6 Compliance

- R₀₀₅₄ If a FIRME service is implemented at a [FIRME instance](#), for each ABI of this service, exactly one of the following is true:
- The ABI is mandatory, and it must be implemented at this instance.
 - The ABI is optional, and it could be implemented at this instance.
 - The ABI is not supported, and it cannot be implemented at this instance.
- R₀₀₅₅ The compliance rules for ABIs specified in this version of the FIRME base service are listed in [Table 2.9](#).

Table 2.9: FIRME base service ABI compliance rules

ABI name	NS FIRME instance	Secure FIRME instance	Realm FIRME instance
FIRME_SERVICE_VERSION	Mandatory	Mandatory	Mandatory
FIRME_SERVICE_FEATURES	Mandatory	Mandatory	Mandatory

2.7 Long-running operations

- D₀₀₅₆ An operation that requires multiple invocations of FIRME ABIs to complete is called a *long-running* operation.
- I₀₀₅₇ Some examples of scenarios that lead to [long-running](#) operations are listed below:
- It is possible that the size of the output data of a FIRME ABI exceeds the size of the [shared buffer](#). In this case, the ABI is invoked multiple times such that each invocation returns a chunk of the output data that can fit in the buffer. This process is repeated until the entire output data is returned in the [shared buffer](#). See also:

– 2.8.1 Data retrieval in chunks.

- Execution of the FIRME ABI may complete before the requested operation is complete if there is a pending interrupt targeted to the Non-secure, Realm or Secure Security states. The operation is resumed by the caller after the interrupt is handled. This process could be repeated multiple times before the operation is complete.

D0058

If the callee pauses the operation and saves enough state to resume the operation subsequently, it is called a *stateful long-running* operation. The operation is resumed by the caller.

See also:

- [FIRME_GM_GPI_SET](#).
- [FIRME_GM_L1_GPT_CREATE](#).
- [FIRME_GM_L1_GPT_DESTROY](#).
- [FIRME_GM_GPI_OP_CONTINUE](#).
- [FIRME_IDE_KEYSET_PROG](#).
- [FIRME_IDE_KEYSET_GO](#).
- [FIRME_IDE_KEYSET_STOP](#).
- [FIRME_IDE_KEYSET_POLL](#).
- [FIRME_ATTEST_PAT_GET](#).
- [FIRME_ATTEST_RAK_GET](#).
- [FIRME_IDEV_OP_START](#).
- [FIRME_IDEV_OP_CONTINUE](#).

D0059

In some FIRME ABIs, the callee returns an IMPLEMENTATION DEFINED value to the caller, to identify, and track progress of a *stateful long-running* operation. The caller passes the value back to the callee when it resumes the operation. This value is called a *cookie*.

See also:

- [FIRME_GM_GPI_SET](#).
- [FIRME_GM_L1_GPT_CREATE](#).
- [FIRME_GM_L1_GPT_DESTROY](#).
- [FIRME_GM_GPI_OP_CONTINUE](#).
- [FIRME_IDEV_OP_START](#).
- [FIRME_IDEV_OP_CONTINUE](#).

R0060

A *cookie* becomes valid when the callee returns it to the caller to identify an in-progress operation. A *cookie* becomes invalid when the caller passes it back to the callee, and successfully resumes the operation.

D0061

If the callee does not save any state, terminates the in-progress operation, and reports the progress to the caller, it is called a *stateless long-running* operation. The operation is restarted by the caller from the point where it was terminated.

See also:

- [FIRME_GM_GPI_SET](#).

I0062

An invocation of a FIRME ABI can either run to completion or use a *long-running* operation to fulfil a request. The EL3 firmware implementation determines the choice of programming model.

For example, an invocation of [FIRME_GM_GPI_SET](#) runs to completion if *GPI transition* of all granules is completed successfully. If EL3 firmware is unable to transition all granules, it starts a *long-running* operation to complete the request across multiple calls.

2.8 Shared buffer

D0063

A FIRME ABI can use a memory buffer to transmit input parameters to, and receive output parameters from EL3 firmware. The memory buffer is called a *shared buffer*.

- R₀₀₆₄ A [shared buffer](#) is mapped in the EL3 translation regime and the applicable translation regime of the caller of a FIRME ABI with the following memory region attributes:
- Normal memory.
 - Write-Back Cacheable.
 - Outer-shareable.
- R₀₀₆₅ The PAS of a [shared buffer](#), and its GPI encoding in the GPT, corresponds to the Security state of the [FIRME instance](#) at which it exists.
- I₀₀₆₆ The correspondence between the PAS of a [shared buffer](#) and the Security state of the [FIRME instance](#) at which it exists is listed below:
- A [shared buffer](#) at the [NS FIRME instance](#) belongs to the NS PAS, and the value of its GPI encoding is 0b1001.
 - A [shared buffer](#) at the [Realm FIRME instance](#) belongs to the Realm PAS, and the value of its GPI encoding is 0b1011.
 - A [shared buffer](#) at the [Secure FIRME instance](#) belongs to the Secure PAS, and the value of its GPI encoding is 0b1000.
- I₀₀₆₇ A [shared buffer](#) is mapped in the EL3 translation regime for at least the duration of an invocation of FIRME ABI that uses the buffer. There are two usage models to ensure this:
- The buffer is mapped in the EL3 translation regime before the FIRME ABI that uses the buffer is invoked. The buffer is unmapped from the EL3 translation regime after the invocation of the FIRME ABI that uses the buffer completes. Alternatively, the buffer remains mapped in the EL3 translation regime. The mechanism used to map and unmap the buffer at a [FIRME instance](#) is IMPLEMENTATION DEFINED.

E.g., EL3 firmware could establish a [shared buffer](#) with the RMM via an IMPLEMENTATION DEFINED boot protocol at the [Realm FIRME instance](#). The RMM uses this buffer in applicable FIRME ABIs for transmitting input parameters, and receiving output parameters.
 - The memory buffer is mapped into the EL3 translation regime upon invocation of the ABI. The memory buffer is unmapped from the EL3 translation regime upon completion of the invocation of the ABI.

E.g., the Host dynamically allocates memory for a shared buffer at the [NS FIRME instance](#). The Host uses this buffer in applicable FIRME ABIs for transmitting input parameters, and receiving output parameters.
- In both usage models the FIRME ABI that uses the buffer specifies the physical base address, and the size of the buffer.
- U₀₀₆₈ EL3 firmware must mitigate against TOCTOU (Time-of-Check to Time-of-Use) attacks while consuming information passed by the caller in a [shared buffer](#). For example, it should copy a payload into a private buffer before validating its contents.
- I₀₀₆₉ The layout of the [shared buffer](#) used by a FIRME ABI is described by the *Shared buffer base address* and the *Shared buffer page count* input parameters.
- R₀₀₇₀ The size of the [shared buffer](#) used by a FIRME ABI fulfils the following constraints:
- The size is $\geq$ Minimum shared buffer size determined from the *MIN_SH_BUF_SZ* field in feature register 1 of the FIRME base service.
 - The size is $\leq$ Maximum shared buffer size determined from the *MAX_SH_BUF_PG_CNT* field in feature register 1 of the FIRME base service.
- See also:
- [Table 2.8](#).
- R₀₀₇₁ The physical base address of the [shared buffer](#) used by a FIRME ABI is aligned to the minimum shared buffer size determined from the *MIN_SH_BUF_SZ* field in feature register 1 of the FIRME base service.

2.8.1 Data retrieval in chunks

- D₀₀₇₂ A FIRME ABI that uses a [shared buffer](#) to transmit output data in chunks in a [long-running](#) operation supports *partial retrieval* of output data.
- I₀₀₇₃ FIRME ABIs that support [partial retrieval](#) include the following output parameters to let the caller track progress of this transmission:
- *Written size*.
 - *Remaining size*.
- R₀₀₇₄ Upon completion, a FIRME ABI that supports [partial retrieval](#), indicates retrieval of the last chunk of output data as follows:
- *Written size* is non-zero.
 - *Remaining size* is zero.
- R₀₀₇₅ Upon completion, a FIRME ABI that supports [partial retrieval](#), indicates retrieval of the first chunk or an intermediate chunk of output data as follows:
- *Written size* is non-zero.
 - *Remaining size* is non-zero.
- R₀₀₇₆ Upon completion, a FIRME ABI that supports [partial retrieval](#), indicates that no chunk of output data was retrieved as follows:
- *Written size* is zero.
 - *Remaining size* is zero.

Chapter 3

Granule management service

3.1 Overview

I0077

The Realm Management Extension (FEAT_RME) [1] adds the Granule Protection Check (GPC) mechanism to the Arm A-profile architecture. This mechanism uses the Granule Protection Table (GPT) to manage the association of physical granules of memory with physical address (PA) spaces. This association is specified via Granule Protection Information (GPI) stored in one or more of the following GPT descriptors:

- Level 0 GPT Block descriptors.
- Level 1 Contiguous or Granules descriptors.

The GPT is an in-memory data structure that is accessible only in the Root PA space. The GPI encoding of a granule is changed only if it is permitted by the granule security policy, enforced by EL3 firmware.

The FIRME Granule Management Service is a set of ABIs implemented by EL3 firmware for the following purposes:

- Enable software running in the Non-secure, Realm and Secure security states to change the GPI encoding of granules as per the granule security policy. See also:
 - [3.3.1 GPI transitions](#).
- Enable software running in the Realm security state to create and destroy a level 1 GPT for an address range in the protected address space. See also:
 - [3.3.3 L1 GPT management](#).

3.2 Scope

- I₀₀₇₈ This version of the FIRME Granule Management Service supports changes to the GPI encoding of a granule within the scope of features added by the following architecture extensions:
- FEAT_RME
 - FEAT_RME_GPC2
 - FEAT_RME_GDI

3.3 Concepts

3.3.1 GPI transitions

- D₀₀₇₉ A *GPI transition* is a change of the GPI encoding of a granule from its current GPI encoding in the GPT, to a target GPI encoding.
- R₀₀₈₀ The size of a granule that can undergo a [GPI transition](#) is equal to the PGS.
- I₀₀₈₁ The PGS is discovered via the *PGS* field in feature register 1. See also:
- [Table 3.7](#).
- I₀₀₈₂ The FIRME specification defines the granule security policy which specifies the [GPI transitions](#) that are permitted for a granule.
- I₀₀₈₃ The [FIRME_GM_GPI_SET](#) ABI is used to request a [GPI transition](#) for a set of physically contiguous granules as follows:
- The physical base address of one or more physically contiguous granules is specified in the *Base Address* input parameter.
 - The count of granules starting from the *Base address* is specified in the *Granule count* input parameter.
 - The [GPI transition](#) to perform is described via the *Target GPI encoding* field of the *Attributes* input parameter.
- See also:
- [3.3.1.1 Software flows](#).
- R₀₀₈₄ The *Base address* input parameter of the [FIRME_GM_GPI_SET](#) ABI is aligned to the PGS.
- D₀₀₈₅ The GPI encoding 0b1000 is called the *Secure* GPI encoding.
The GPI encoding 0b1001 is called the *Non-secure* GPI encoding.
The GPI encoding 0b1010 is called the *Root* GPI encoding.
The GPI encoding 0b1011 is called the *Realm* GPI encoding.
The GPI encoding 0b1101 is called the *NSO* GPI encoding if FEAT_RME_GPC2 is implemented.
The GPI encoding 0b0100 is called the *SA* GPI encoding if FEAT_RME_GDI is implemented.
The GPI encoding 0b0101 is called the *NSP* GPI encoding if FEAT_RME_GDI is implemented.
- R₀₀₈₆ When FEAT_RME is implemented, the granule security policy permits the following [GPI transitions](#) via the [FIRME_GM_GPI_SET](#) ABI:

Table 3.1: Permitted GPI transitions for FEAT_RME via FIRME_GM_GPI_SET

Caller's Security state	Current GPI encoding	Target GPI encoding
Secure	Non-secure	Secure
Secure	Secure	Non-secure
Realm	Non-secure	Realm

Caller's Security state	Current GPI encoding	Target GPI encoding
Realm	Realm	Non-secure

If only FEAT_RME is implemented, no other [GPI transition](#) is permitted via the [FIRME_GM_GPI_SET](#) ABI, apart from those listed in [Table 3.1](#).

R0087

When FEAT_RME_GPC2 is implemented, in addition to the [GPI transitions](#) permitted via the [FIRME_GM_GPI_SET](#) ABI when FEAT_RME is implemented, the granule security policy permits the following [GPI transitions](#) via the same ABI:

Table 3.2: Permitted GPI transitions for FEAT_RME_GPC2 via FIRME_GM_GPI_SET

Caller's Security state	Current GPI encoding	Target GPI encoding
Non-secure	Non-secure	NSO
Non-secure	NSO	Non-secure

If only FEAT_RME_GPC2 is implemented, no other [GPI transitions](#) are permitted via the [FIRME_GM_GPI_SET](#) ABI, apart from those listed in [Table 3.1](#), and [Table 3.2](#).

S0088

The following pseudocode illustrates the [GPI transition](#) of a single granule between the [Non-secure](#) and the [NSO](#) GPI encodings.

In a transition from [NSO](#) to [Non-secure](#), it is assumed that the caller is responsible for scrubbing the granule and flushing its contents to the Point of Coherency [1]. This ensures that any confidential information does not become accessible to the Realm world or the Secure world after the granule has transitioned to the NS PAS.

```
int32_t ns_nso_gpi_transition(uint64_t *addr, uint8_t target_gpi) {
    int ret;
    uint32_t current_gpi;

    // Read the current GPI encoding of the granule
    // from the GPT and check if one of the following is true:
    // - Current encoding is NS and target encoding is NSO.
    // - Current encoding is NSO and target encoding is NS.
    // Otherwise return an error.
    current_gpi = read_gpt(addr);
    if ((current_gpi != NSO || target_gpi != NS) && ((current_gpi != NS ||
        ↪target_gpi != NSO))
        return ERROR;

    // Update the GPI encoding to the target_gpi.
    ret = write_gpt(addr, target_gpi);
    if (ret != SUCCESS)
        return ret;
    DSB(OSHST);

    // Ensure that all agents observe the new GPI configuration
    TLBI_RPALOS(addr, PGS);
    DSB(OSH);

    return SUCCESS;
}
```

X₀₀₈₉ As per the granule security policy, only the Host can request a [GPI transition](#) of a granule from [NSO](#) to [Non-secure](#). The Host is responsible for preserving the confidentiality and integrity of information contained in the granule undergoing the transition. If a granule has the [NSO](#) GPI encoding, its contents in the caches and Conventional memory [1] are still associated with the NS PAS. When the granule undergoes this [GPI transition](#), it is adequate to perform cache maintenance to the Point of Coherency [1]. This makes it possible for the Host to perform correct cache maintenance by VA to ensure that the effects of scrubbing an [NSO](#) granule are visible to all observers in the system. Hence, no cache maintenance is done by EL3 firmware.

This is in contrast to other transitions where a different PAS is at least the source or target of the transition, and EL3 firmware is responsible for preserving the confidentiality and integrity of information contained in the granules undergoing the transition. A caller of the [FIRME_GM_GPI_SET](#) ABI is not trusted to perform the equivalent cache maintenance. The cache maintenance is done by PA to the Point of Physical Aliasing [1] via instructions that are only accessible in EL3.

R₀₀₉₀ When FEAT_RME_GDI is implemented, in addition to the [GPI transitions](#) permitted via the [FIRME_GM_GPI_SET](#) ABI when FEAT_RME_GPC2 is implemented, the granule security policy permits the following [GPI transitions](#) via the same ABI:

Table 3.3: Permitted GPI transitions for FEAT_RME_GDI via FIRME_GM_GPI_SET

Caller's Security state	Current GPI encoding	Target GPI encoding
Non-secure	Non-secure or NSO	NSP
Non-secure	Non-secure or NSO	SA
Non-secure	NSP	Non-secure or NSO
Non-secure	SA	Non-secure or NSO

If only FEAT_RME_GDI is implemented, no other [GPI transitions](#) are permitted via the [FIRME_GM_GPI_SET](#) ABI, apart from those listed in [Table 3.1](#), [Table 3.2](#), and [Table 3.3](#).

S₀₀₉₁ The following pseudocode illustrates the [GPI transition](#) of a single granule from [Non-secure](#) or [NSO](#), to [NSP](#).

```
int32_t granule_delegate_to_nsp(uint64_t *addr) {
    int ret;
    uint32_t current_gpi;

    // Read the current GPI encoding of the granule
    // from the GPT and check if it is NS or NSO.
    // Otherwise return an error.
    current_gpi = read_gpt(addr);
    if ((current_gpi != NSO || target_gpi != NSP) && ((current_gpi != NS ||
        ↪target_gpi != NSP))
        return ERROR;

    // Update the GPI encoding to NSP.
    ret = write_gpt(addr, NSP);
    if (ret != SUCCESS)
        return ret;
    DSB(OSHST);

    // Ensure that all agents observe the new GPI configuration
    TLBI_RPALOS(addr, PGS);
    DSB(OSH);

    // Flush any NS data in this granule to avoid corrupting NSP data
    for (i = 0; i < PGS; i += cache_line_size)
```

```
        DC_CIPAPA((addr+i), NS);

    DSB(OSH);
    return SUCCESS;
}
```

S0092

The following pseudocode illustrates the [GPI transition](#) of a single granule from [NSP](#) to [Non-secure](#).

```
int32_t granule_undelagate_from_nsp(uint64_t *addr) {
    int ret;

    // Read the current GPI encoding of the granule
    // from the GPT and check if it is NSP.
    // Otherwise return an error.
    if (read_gpt(addr) != NSP)
        return ERROR;

    // Make the granule inaccessible as its correct transition
    // relies on break before make semantics.
    write_gpt(addr, NO_ACCESS);
    DSB(OSHST);

    // Ensure that all agents observe the new GPI configuration
    TLBI_RPALOS(addr, PGS);
    DSB(OSH);

    // Flush any NSP data in this granule to prevent it from
    // becoming visible in the NS PAS
    for (i = 0; i < PGS; i += cache_line_size)
        DC_CIPAPA((addr+i), NSP);
    DSB(OSH);

    // Flush any NSP data in this granule that was speculatively
    // prefetched via the NS PAS into the PE caches
    for (i = 0; i < PGS; i += cache_line_size)
        DC_CIPAPA((addr+i), NS);
    DSB(OSH);

    // Update the GPI encoding
    write_gpt(addr, NS);
    DSB(OSHST);

    // Ensure that all agents observe the new GPI configuration
    TLBI_RPALOS(addr, PGS);
    DSB(OSH);

    return SUCCESS;
}
```

D0093

A range of addresses that lies within the protected address space defined by GPCCR_EL3.PPS is called a *protected address range*.

R0094

The base address of a [protected address range](#) is aligned to the size of the protected address space covered by a L0 GPT Table descriptor specified via GPCCR_EL3.L0GPTSZ.

R0095

The size of a [protected address range](#) is equal to the size of the protected address space covered by a L0 GPT Table descriptor specified via GPCCR_EL3.L0GPTSZ.

3.3.1.1 Software flows

I_0096

The Host can use the [FIRME_GM_GPI_SET](#) ABI to perform a [GPI transition](#) as follows:

- If FEAT_RME_GPC2 is implemented, the GPI encoding of a granule can be transitioned between the [Non-secure](#) and [NSO](#) GPI encodings.
- If FEAT_RME_GDI is implemented, the GPI encoding of a granule can be transitioned as follows:
 - Between the [Non-secure](#) and [NSP](#) GPI encodings.
 - Between the [Non-secure](#) and [SA](#) GPI encodings.
 - Between the [NSO](#) and [NSP](#) GPI encodings.
 - Between the [NSO](#) and [SA](#) GPI encodings.

I_0097

The Host uses the RMI commands to delegate and undelegate granules in [3] to request the RMM to change the GPI encoding of a granule as follows:

- From [Non-secure](#) to [Realm](#). An example flow where the Host delegates a single granule to the RMM via an RMI command for granule delegation is illustrated in [Figure 3.1](#).
- From [Realm](#) to [Non-secure](#). An example flow where the single granule is undelegated by the RMM back to the Host in response to an RMI command for granule undelegation is illustrated in [Figure 3.2](#).

The RMM uses the [FIRME_GM_GPI_SET](#) ABI to perform the [GPI transition](#) requested by the Host.

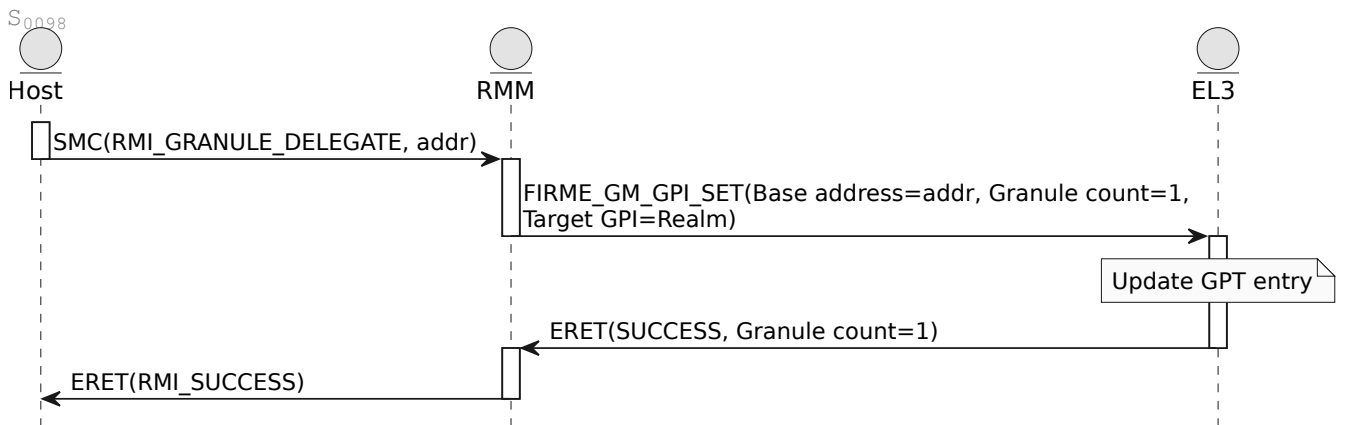


Figure 3.1: Example granule delegation flow with RMI_GRANULE_DELEGATE

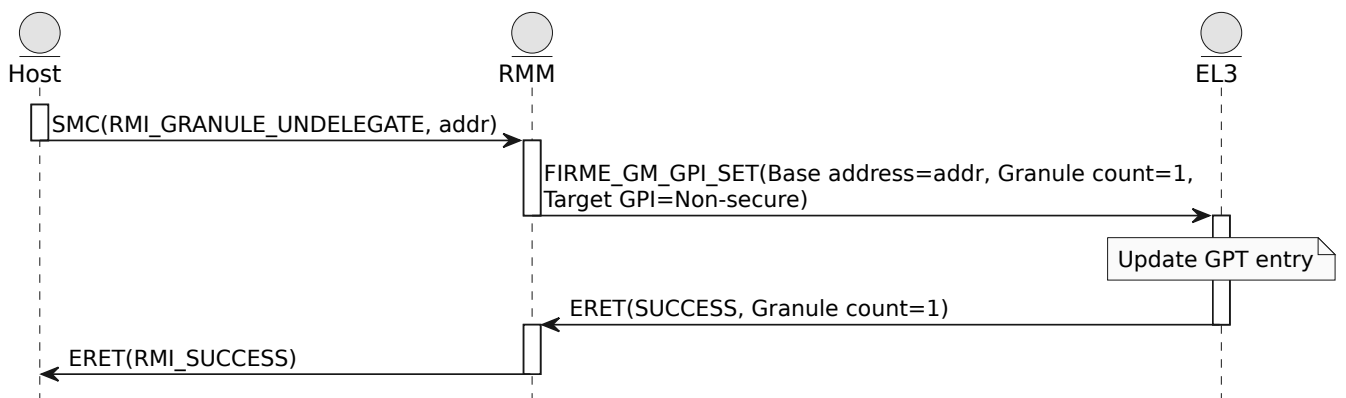


Figure 3.2: Example granule undelegation flow with RMI_GRANULE_UNDELEGATE

I_0099

The OS or Hypervisor uses the FFA_MEM_LEND ABI in [4] to request the SPMC to change the GPI encoding of a granule from [Non-secure](#) to [Secure](#). An example flow where the Host lends a list of physically discontinuous memory regions to an SP via FFA_MEM_LEND [4] is illustrated in [Figure 3.3](#).

The OS or Hypervisor uses the FFA_MEM_RECLAIM ABI to request the SPMC to change the GPI encoding of a granule from **Secure** to **Non-secure**. An example flow where the SP returns the memory regions back to the Host via FFA_MEM_RECLAIM [4] is illustrated in Figure 3.4.

The SPMC uses the FIRME_GM_GPI_SET ABI to perform the **GPI transition** requested by the OS or Hypervisor.

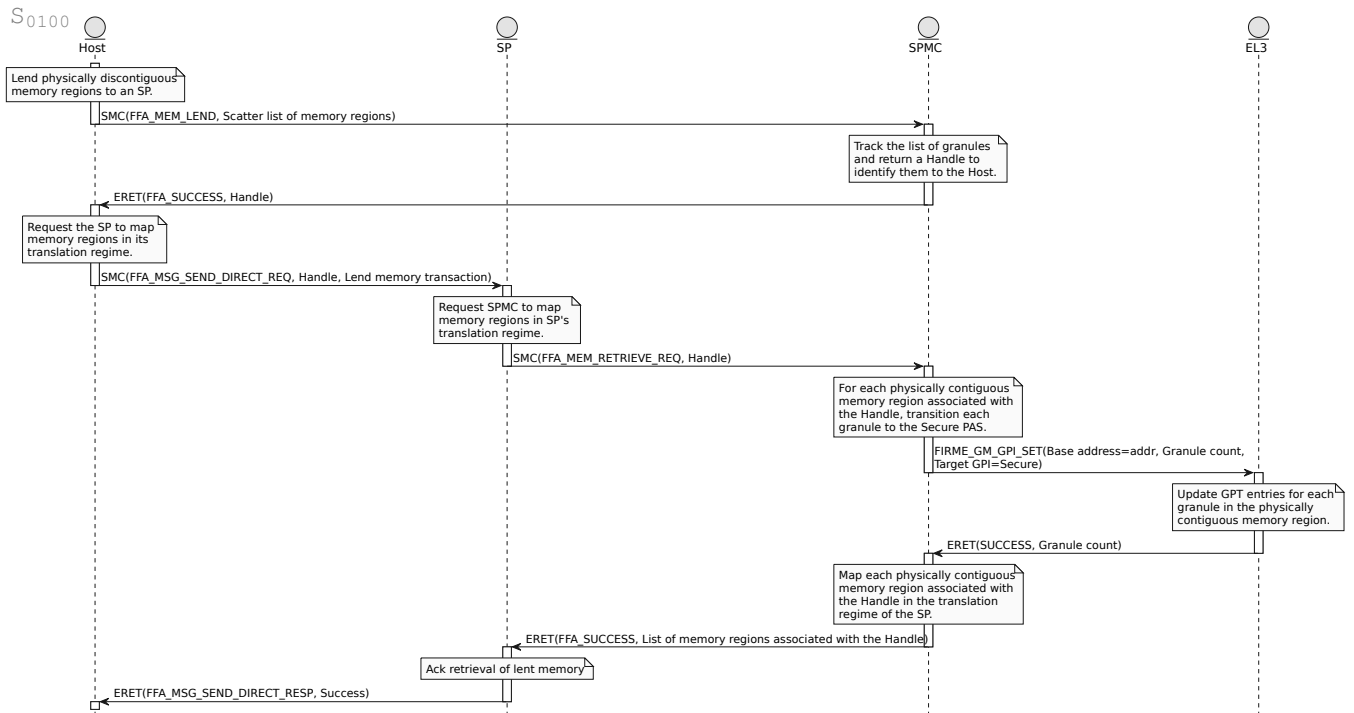


Figure 3.3: Example flow to lend memory from Non-secure to Secure with FFA_MEM_LEND

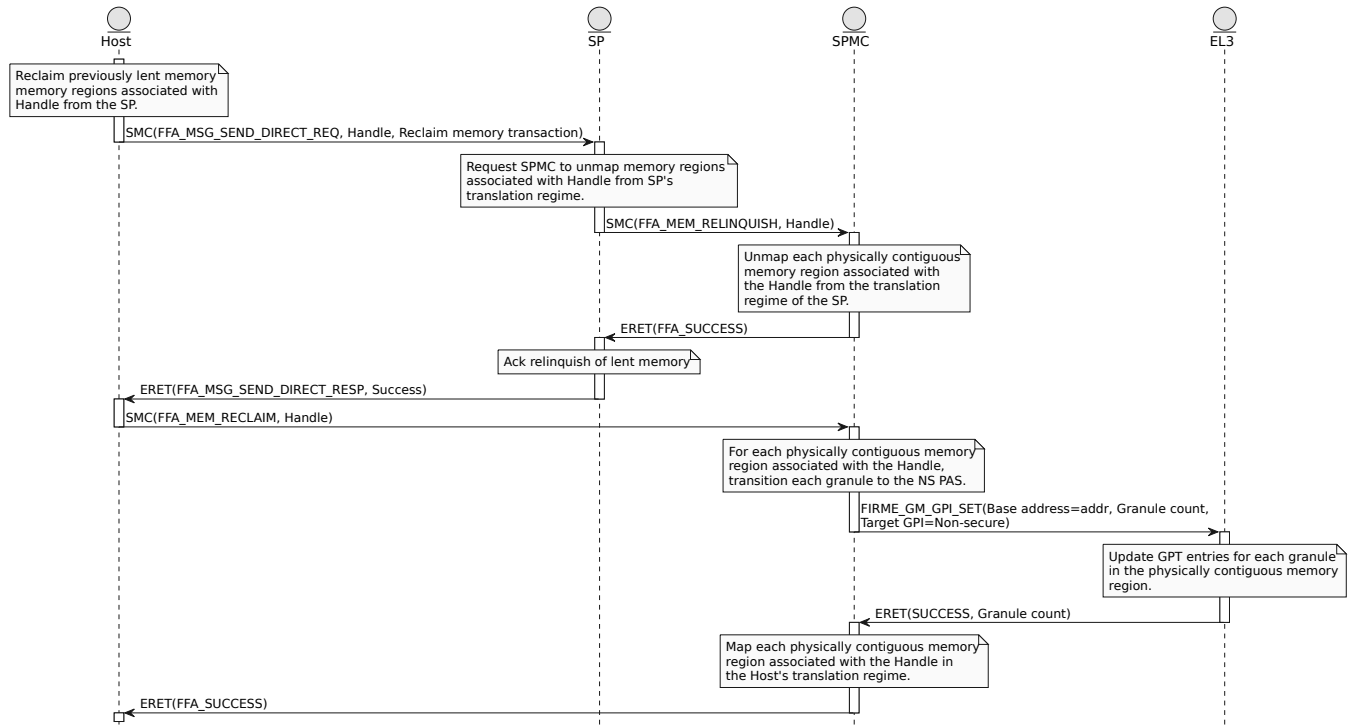


Figure 3.4: Example flow to return memory from Secure to Non-secure with FFA_MEM_RECLAIM

3.3.2 GPI management ABIs

D₀₁₀₁

A caller requests a **GPI transition** for a physically contiguous range of granules by invoking the **FIRME_GM_GPI_SET** ABI. The invocation can complete in one call, or as a **long-running** operation as follows:

- A **stateless long-running** operation. The callee reports progress but does not retain internal state. The caller restarts by re-invoking **FIRME_GM_GPI_SET** on the remaining range of granules.
- A **stateful long-running** operation. The callee pauses and retains internal state. It allocates and returns a **cookie** value to identify this state. The caller resumes the operation by invoking **FIRME_GM_GPI_OP_CONTINUE** ABI with the **cookie** value.

The **FIRME_GM_GPI_SET** ABI and the **FIRME_GM_GPI_OP_CONTINUE** ABI are collectively referred to as the **GPI management ABIs**.

I₀₁₀₂

When an invocation of these ABIs completes, the values of the return status code and the *Granule count* output parameter are used by the caller to determine progress of the requested operation.

R₀₁₀₃

The value of the *Granule count* output parameter is valid when an invocation of a **GPI management ABI** completes with any of the following return status codes:

- **SUCCESS.**
- **DENIED.**
- **INCOMPLETE.**
- **IN_PROGRESS.**

Otherwise, the value of the *Granule count* output parameter is UNKNOWN and should be ignored by the caller.

I₀₁₀₄

The value of the *Granule count* output parameter is UNKNOWN when an invocation of a **GPI management ABI** completes with any of the following return status codes:

- [NOT_SUPPORTED](#).
- [INVALID_PARAMETERS](#).
- [RETRY](#).
- [NO_ENTRY](#).

D ₀₁₀₅	The first granule that a GPI management ABI operates on is called the <i>start granule</i> .
R ₀₁₀₆	When the FIRME_GM_GPI_SET ABI is invoked, the start granule is the first granule at the <i>Base address</i> input parameter.
I ₀₁₀₇	When the FIRME_GM_GPI_SET ABI is invoked, the value of the start granule is 0. This value is the start index of the specified range of granules. if the <i>Granule count</i> input parameter is X , the start index is 0 and the end index is $X - 1$.
R ₀₁₀₈	When the FIRME_GM_GPI_OP_CONTINUE ABI is invoked, the start granule is the sum of all valid <i>Granule count</i> output parameter values returned by the first invocation of the FIRME_GM_GPI_SET ABI, and all previous invocations of the FIRME_GM_GPI_OP_CONTINUE ABI.
R ₀₁₀₉	If the <i>Granule count</i> output parameter is valid when an invocation of the FIRME_GM_GPI_SET ABI completes, it specifies the count of granules starting from start granule that underwent a successful GPI transition .
R ₀₁₁₀	If the <i>Granule count</i> output parameter is valid when an invocation of the FIRME_GM_GPI_OP_CONTINUE ABI completes, it specifies the count of granules starting from the start granule that underwent a successful GPI transition .
I ₀₁₁₁	If the <i>Granule count</i> output parameter is valid, its value is 0 if no granules underwent a GPI transition . If the parameter is valid, and has a non-zero value, at least one granule underwent a GPI transition as follows: • Let A be the count of granules that have not undergone the requested GPI transition such that: – If the FIRME_GM_GPI_SET ABI was invoked, A is the <i>Granule count</i> input parameter. – If the FIRME_GM_GPI_OP_CONTINUE ABI was invoked, $A = X - Y$ where: * X is the <i>Granule count</i> input parameter when the transition was started by invoking the FIRME_GM_GPI_SET ABI. * Y is the start granule when the transition was resumed by invoking the FIRME_GM_GPI_OP_CONTINUE ABI. • Let B be the <i>Granule count</i> output parameter. • If $A == B$, all granules in the requested range underwent the transition. • If $A > B$, all granules in the requested range could not undergo the transition. If the INCOMPLETE return status code is returned, this is a stateful long-running operation. The caller should invoke the FIRME_GM_GPI_OP_CONTINUE ABI with the cookie value to resume the paused operation. Otherwise, this is a stateless long-running operation. The caller should invoke the FIRME_GM_GPI_SET ABI again with the updated <i>Base address</i> and the <i>Granule count</i> input parameter to transition the remaining granules. The count of remaining granules is $A - B$.
R ₀₁₁₂	When an invocation of a GPI management ABI completes with the SUCCESS return status code, the value of the <i>Granule count</i> output parameter is ≥ 1 .
I ₀₁₁₃	The SUCCESS return status code implies that at least a single granule underwent the GPI transition . This is in contrast to the other return status codes where it is possible that no granule underwent the transition.
R ₀₁₁₄	When an invocation of a GPI management ABI completes with the NOT_SUPPORTED return status code, one of the following is true: • The ABI is not implemented by EL3 firmware. • The caller is not allowed to invoke the ABI.

- R₀₁₁₅ When an invocation of a [GPI management ABI](#) completes with the [INVALID_PARAMETERS](#) return status code, one of the following is true:
- If the [FIRME_GM_GPI_SET](#) ABI was invoked, one of the following is true:
 - The base address of the physically contiguous granules is not aligned to the GPCCR_EL3.PGS.
 - The physically contiguous granules do not lie in the region in the physical address map protected by the GPC mechanism and specified in GPCCR_EL3.PPS.
 - A field in the *Attributes* input parameter is incorrectly encoded.
 - If the [FIRME_GM_GPI_OP_CONTINUE](#) ABI was invoked, the *Cookie* input parameter contains an invalid [cookie](#) value.
- R₀₁₁₆ When an invocation of a [GPI management ABI](#) completes with the [RETRY](#) return status code, EL3 firmware cannot complete the request due to an IMPLEMENTATION DEFINED reason and the caller must retry the operation.
- R₀₁₁₇ When an invocation of the [FIRME_GM_GPI_SET](#) ABI completes with the [NO_ENTRY](#) return status code, EL3 firmware cannot complete the request because all of the following are true:
- One or more granules lie in a [protected address range](#) that is covered by an L0 GPT Block descriptor.
 - An L1 GPT object cannot be allocated to replace the L0 GPT Block descriptor with an L0 GPT Table descriptor.
- R₀₁₁₈ When an invocation of a [GPI management ABI](#) completes with the [DENIED](#) return status code, the [GPI transition](#) of a granule to the *Target GPI* encoding specified in the *Attributes* input parameter, is either not permitted or the granule already has the *Target GPI* encoding.
- I₀₁₁₉ Completion of a [GPI management ABI](#) with the [DENIED](#) return status code, prevents the implementation from checking that the [GPI transition](#) of all granules is permitted, before the [GPI transition](#) of any granule starting from the [start granule](#) is performed.
- Instead, starting from the [start granule](#), the implementation can perform a successful [GPI transition](#) of as many granules as possible until it encounters a granule that cannot undergo the requested transition.
- If the [start granule](#) cannot undergo the requested transition, the implementation is unable to transition any granule. In this case, the [GPI management ABI](#) completes with the [DENIED](#) return status code and the *Granule count* output parameter is 0.
- Otherwise, the [GPI management ABI](#) completes with the [DENIED](#) return status code, and the *Granule count* output parameter specifies the count of granules (starting from the [start granule](#)) that underwent a successful [GPI transition](#).
- R₀₁₂₀ When an invocation of a [GPI management ABI](#) completes with the [IN_PROGRESS](#) return status code, EL3 firmware cannot complete the request because one of the following is true:
- One or more granules are participating in another operation invoked via a [GPI management ABI](#).
 - One or more granules are participating in creation or destruction of an L1 GPT object via an invocation of the [FIRME_GM_L1_GPT_CREATE](#) ABI, or the [FIRME_GM_L1_GPT_DESTROY](#) ABI. For example:
 - A granule falls within a [protected address range](#) that is undergoing a [GPI transition](#).
 - A granule falls within the memory region used for the L1 GPT object being created or destroyed.
- R₀₁₂₁ When an invocation of a [GPI management ABI](#) completes with the [INCOMPLETE](#) return status code, EL3 firmware has started a [stateful long-running](#) operation as the [GPI transition](#) of one or more granules is incomplete.
- I₀₁₂₂ When an invocation of [GPI management ABI](#) ABI completes with the [INCOMPLETE](#) return status code, the *Granule count* output parameter specifies the count of granules starting from the [start granule](#) that underwent a successful [GPI transition](#). This is a [stateful long-running](#) operation. The caller should invoke the [FIRME_GM_GPI_OP_CONTINUE](#) ABI with the [cookie](#) value to resume the paused operation.
- I₀₁₂₃ An L0 GPT Block descriptor represents a [protected address range](#) whose granules share a single GPI encoding. Therefore, a [GPI transition](#) can be applied to granules described by an L0 GPT Block descriptor only if the request covers the entire [protected address range](#). Otherwise, if the granules do not cover the entire [protected address range](#), an L1 GPT object is required to perform the [GPI transition](#).

In an invocation of the `FIRME_GM_GPI_SET` ABI, the caller specifies the requested granules using the *Base address* and *Granule count* input parameters. If this request intersects a [protected address range](#) covered by an L0 GPT Block descriptor one of the following is true:

- If the request does not cover the entire [protected address range](#), EL3 firmware must create an L1 GPT object for that [protected address range](#) to transition only the intersecting granules. If allocation of the L1 GPT object fails, `FIRME_GM_GPI_SET` completes with `NO_ENTRY`.
- If the request does cover the entire [protected address range](#), EL3 firmware can perform the transition by updating the GPI encoding in the L0 GPT Block descriptor. The required cache and TLB maintenance for the entire [protected address range](#) is a [stateful long-running](#) operation, and EL3 firmware can make one of the following IMPLEMENTATION DEFINED choices:
 - Return `NO_ENTRY` to indicate that it cannot perform the transition because it was unable to allocate an L1 GPT object.
 - Return `INCOMPLETE` and a `cookie`. The caller resumes the operation using `FIRME_GM_GPI_OP_CONTINUE`. The *Granule count* output parameter may be returned as 0 if the count cannot be reported until completion.

S0124

The following pseudocode illustrates the usage of the [GPI management ABIs](#) by a caller at a [FIRME instance](#).

```
//
// Inputs:
//   base_pa      : Physical base address of the first granule in the range.
//   remaining    : Number of granules remaining to transition.
//   attributes    : Attributes input (includes target GPI encoding).
//
// Output:
//   Returns a FIRME status code (SUCCESS only if all granules transitioned).
//   Other outputs are returned via pointer parameters.
//
//   - out_granule_count is meaningful only when status is one of:
//     SUCCESS, DENIED, INCOMPLETE, IN_PROGRESS.
//   - out_cookie is meaningful when status == INCOMPLETE.
//
int32_t firme_gpi_set_range(uint64_t base_pa,
                           uint64_t remaining,
                           uint64_t attributes)
{
    const uint64_t granule_size_bytes = discover_pgs(); // PGS in bytes

    while (remaining > 0) {
        int32_t status;
        uint64_t out_granule_count = 0; // progress reported by this call
                                     // (if meaningful)
        uint64_t out_cookie        = 0; // cookie reported by this call
                                     // (if meaningful)

        status = FIRME_GM_GPI_SET(
            /* in */ base_pa,
            /* in */ remaining,
            /* in */ attributes,
            /* out */ &out_granule_count,
            /* out */ &out_cookie);

        if (status == SUCCESS) {
            // Completed, or stateless long-running with partial progress.
            // out_granule_count is meaningful and >= 1.
            base_pa += out_granule_count * granule_size_bytes;
            remaining -= out_granule_count;
            continue;
        }
    }
}
```

```
if (status == INCOMPLETE) {
    // Stateful long-running: resume using OP_CONTINUE(cookie).
    // out_granule_count is meaningful (may be 0).
    base_pa += out_granule_count * granule_size_bytes;
    remaining -= out_granule_count;
    uint64_t cookie = out_cookie;

    while (true) {
        int32_t cont_status;
        uint64_t cont_granule_count = 0; // progress since last CONTINUE
                                         // (if meaningful)
        uint64_t cont_cookie = 0; // updated cookie (if provided)

        cont_status = FIRME_GM_GPI_OP_CONTINUE(
            /* in */ cookie,
            /* out */ &cont_granule_count,
            /* out */ &cont_cookie);

        if (cont_status == SUCCESS) {
            // Stateful operation finished (may report final chunk
            // progress).
            base_pa += cont_granule_count * granule_size_bytes;
            remaining -= cont_granule_count;
            break; // exit CONTINUE loop; outer loop handles any
                  // remaining range
        }

        if (cont_status == INCOMPLETE) {
            // Still paused; apply reported progress and continue
            // with returned cookie.
            base_pa += cont_granule_count * granule_size_bytes;
            remaining -= cont_granule_count;
            cookie = cont_cookie;
            continue;
        }

        if (cont_status == RETRY) {
            // Temporary conflict/resource shortage: retry CONTINUE
            // later.
            backoff_or_yield();
            continue;
        }

        if (cont_status == IN_PROGRESS) {
            // Either
            // 1. The operation identified by the cookie value is
            //    already in progress. Eventually the cookie will
            //    become invalid and cause `INVALID_PARAMETERS`
            //    to be returned.
            // or
            // 2. One or more granules are participating in another
            //    stateful long-running operation.
            base_pa += cont_granule_count * granule_size_bytes;
            remaining -= cont_granule_count;
            backoff_or_yield();
            continue;
        }

        if (cont_status == DENIED) {
            // Not permitted for the remaining portion; a prefix

```

```
        // may have completed.
        base_pa += cont_granule_count * granule_size_bytes;
        remaining -= cont_granule_count;
        return DENIED;
    }

    // Other errors: cont_granule_count is UNKNOWN and
    // must be ignored.
    return cont_status;
}
continue; // continue outer while(remaining > 0)
}

if (status == IN_PROGRESS) {
    // Temporary conflict with another stateful operation.
    // out_granule_count is meaningful (may be 0).
    base_pa += out_granule_count * granule_size_bytes;
    remaining -= out_granule_count;
    backoff_or_yield();
    continue;
}
if (status == DENIED) {
    // Transition not permitted; a prefix may already have
    // transitioned. out_granule_count is meaningful (may be 0).
    base_pa += out_granule_count * granule_size_bytes;
    remaining -= out_granule_count;
    return DENIED;
}
if (status == RETRY) {
    // No progress made; out_granule_count is not meaningful for RETRY.
    backoff_or_yield();
    continue;
}
// Other errors: out_granule_count is UNKNOWN and must be ignored.
return status;
}

return SUCCESS;
}
```

3.3.3 L1 GPT management

I₀₁₂₅ The [FIRME_GM_L1_GPT_CREATE](#) and the [FIRME_GM_L1_GPT_DESTROY](#) ABIs are implemented only at the [Realm FIRME](#) instance. See also:

- [Table 3.8](#).

S₀₁₂₆ The Host uses RMI commands defined in [3] to request the RMM to create and destroy L1 GPT objects. The RMM uses the [FIRME_GM_L1_GPT_CREATE](#) and the [FIRME_GM_L1_GPT_DESTROY](#) ABIs to complete the request.

3.3.3.1 L1 GPT creation

I₀₁₂₇ The [FIRME_GM_L1_GPT_CREATE](#) ABI is used to replace an L0 GPT Block descriptor for a [protected address range](#) with an L1 GPT object as follows:

- The physical base address of the range is specified in the *PA range base* input parameter.
- The physical base address of the L1 GPT object for the range is specified in the *L1 GPT base* input parameter.

I₀₁₂₈ The alignment of the physical base address of the L1 GPT object, and its size depends upon the value of GPCCR_EL3.PGS and GPCCR_EL3.L0GPTSZ as described in [1]. A caller of the [FIRME_GM_L1_GPT_CREATE](#) ABI can discover both parameters via feature register 1 in order to correctly allocate memory for an L1 GPT object. See also:

- [Table 3.7](#).

S₀₁₂₉ EL3 firmware performs the following non-exhaustive list of operations in response to an invocation of the [FIRME_GM_L1_GPT_CREATE](#) ABI:

- The *PA range base* and *L1 GPT base* input parameters are validated for size and alignment.
- The presence of an L0 GPT Block descriptor for the specified [protected address range](#) is checked.
- The GPI encodings of granules occupied by the L1 GPT object are transitioned from the [Non-secure](#) GPI encoding to the [Root](#) GPI encoding in the GPT.
 - All applicable cache and TLB invalidation operations required to perform the GPI transition are also performed.
- The L1 GPT object is mapped in the EL3 translation regime.
- EL3 firmware creates GPT Granules descriptors and/or GPT contiguous descriptors in the L1 GPT object with the GPI value used in the L0 GPT Block descriptor for the [protected address range](#).
- The L1 GPT object is unmapped from the EL3 translation regime.
- The L0 GPT Block descriptor is replaced with a L0 GPT Table descriptor that contains the address of the L1 GPT object as the *Next-level table address*.

R₀₁₃₀ When FEAT_RME is implemented, the granule security policy permits the following [GPI transition](#) via the [FIRME_GM_L1_GPT_CREATE](#) ABI:

Caller's Security state	Current GPI encoding	Target GPI encoding
Realm	Non-secure	Root

Otherwise, the [GPI transition](#) is not permitted via the [FIRME_GM_L1_GPT_CREATE](#) ABI.

I₀₁₃₁ Execution of the [FIRME_GM_L1_GPT_CREATE](#) ABI can be a [stateful long-running](#) operation. If a physical interrupt becomes pending during execution of the ABI, its invocation can complete before EL3 firmware is able to create the L1 GPT object for the specified [protected address range](#). In this case, the ABI is invoked again after the interrupt is handled to continue creation of the L1 GPT object.

D₀₁₃₂ An invocation of the [FIRME_GM_L1_GPT_CREATE](#) ABI is a *request to start creation* of an L1 GPT object for the specified [protected address range](#) if the value of the *L1 GPT base* input parameter is non-zero.

D₀₁₃₃ An invocation of the [FIRME_GM_L1_GPT_CREATE](#) ABI is a *request to continue creation* of an L1 GPT object for the specified [protected address range](#) if the value of the *L1 GPT base* input parameter is zero.

I₀₁₃₄ The handling of a request to start creation, or continue creation of an L1 GPT object depends upon whether an operation is already in-progress for the specified [protected address range](#). Each scenario and its outcome is described in the below table:

In-progress request	L1 GPT base value	Description
No	Zero	Invalid request. The INVALID_PARAMETERS return status code is returned.
No	Non-zero	Request to start creation of an L1 GPT object in the memory referenced by <i>L1 GPT base</i> .
Yes	Zero	Request to continue creation of an L1 GPT object for the specified protected address range .
Yes	Non-zero	Invalid request. The IN_PROGRESS return status code is returned.

S₀₁₃₅ The following pseudocode illustrates creation of an L1 GPT object.

```
int32_t ll_gpt_create(uint64_t *prot_addr_range_base, uint64_t ll_gpt_sz) {

    int32 result;
    uint64_t *ll_gpt_base = alloc_mem(ll_gpt_sz);

    // Start creation of the L1 GPT object for the protected address range and
    // iterate until it is complete.
    result = FIRME_GM_L1_GPT_CREATE(prot_addr_range_base, ll_gpt_base);
    while (result == INCOMPLETE) {
        result = FIRME_GM_L1_GPT_CREATE(prot_addr_range_base, 0);
    };

    return result;
}
```

R₀₁₃₆ An invocation of the [FIRME_GM_L1_GPT_CREATE](#) ABI completes with the [SUCCESS](#) return status code, if none of the following are true:

- The ABI is either not implemented by EL3 firmware or the caller is not allowed to invoke the ABI.
 - In this case, the ABI invocation completes with the [NOT_SUPPORTED](#) return status code.
- The [protected address range](#) does not completely lie within the protected address space defined by GPCCR_EL3.PPS.
 - In this case, the ABI invocation completes with the [INVALID_PARAMETERS](#) return status code.
- The specified [protected address range](#) lies within the protected address space but is not valid for L1 GPT object creation.
 - In this case, the ABI invocation completes with the [INVALID_PARAMETERS](#) return status code.
- The base address of the [protected address range](#) is not correctly aligned.
 - In this case, the ABI invocation completes with the [INVALID_PARAMETERS](#) return status code.
- The L1 GPT object for the [protected address range](#) does not lie within the protected address space defined by GPCCR_EL3.PPS.
 - In this case, the ABI invocation completes with the [INVALID_PARAMETERS](#) return status code.
- The L1 GPT object for the [protected address range](#) is not correctly aligned.
 - In this case, the ABI invocation completes with the [INVALID_PARAMETERS](#) return status code.
- There is no request in-progress for the [protected address range](#) and the *L1 GPT base* input parameter is zero.

- In this case, the ABI invocation completes with the `INVALID_PARAMETERS` return status code.
- EL3 firmware cannot complete the request as the L1 GPT object lies in a `protected address range` that is covered by an L0 GPT Block descriptor.
 - In this case, the ABI invocation completes with the `NO_ENTRY` return status code.
- The `protected address range` is already covered by an L1 GPT object referenced by an L0 GPT Table descriptor.
 - In this case, the ABI invocation completes with the `EXISTS` return status code.
- This is a request to start creation of an L1 GPT object but another operation is already in-progress for the specified `protected address range`.
 - In this case, the ABI invocation completes with the `IN_PROGRESS` return status code and has no effect.
- EL3 firmware cannot complete the request due to an IMPLEMENTATION DEFINED reason and the caller must retry the operation.
 - In this case, the ABI invocation completes with the `RETRY` return status code.
- EL3 firmware could not allocate translation tables to map the L1 GPT in its translation regime.
 - In this case, the ABI invocation completes with the `NO_MEMORY` return status code.
- The caller is not allowed to create an L1 GPT object for the specified `protected address range`.
 - In this case, the ABI invocation completes with the `DENIED` return status code and has no effect.
- At least a single granule in the memory region occupied by the L1 GPT object does not have the `Non-secure` GPI encoding.
 - In this case, the ABI invocation completes with the `DENIED` return status code and has no effect.
- The request to create the L1 GPT object was interrupted due to an IMPLEMENTATION DEFINED reason and the caller must resume the operation.
 - In this case, the ABI invocation completes with the `INCOMPLETE` return status code.

R₀₁₃₇ When an invocation of the `FIRME_GM_L1_GPT_CREATE` ABI completes with the `SUCCESS` return status code, all of the following are true:

- The memory occupied by the L1 GPT object is accessible only via the Root PAS.
- For a `protected address range` PAR0, GPI_X == GPI_Y where:
 - GPI_X was the GPI value in the L0 GPT Block descriptor to perform granule protection checks on PAR0 before invocation of the ABI.
 - GPI_Y is the GPI value used by the L1 GPT object to perform granule protection checks on PAR0 after the invocation of the ABI completes.

R₀₁₃₈ When an invocation of the `FIRME_GM_L1_GPT_CREATE` ABI completes with the `INCOMPLETE` return status code, behaviour of granule protection checks on the specified `protected address range` is the same as it was before creation of the L1 GPT object was started.

X₀₁₃₉ An in-progress operation to create an L1 GPT object for a `protected address range` has no effect on the behaviour of granule protection checks on that range.

I₀₁₄₀ When an invocation of the `FIRME_GM_L1_GPT_CREATE` ABI completes with the `INCOMPLETE` return status code, it is UNKNOWN which granules occupied by the L1 GPT object are assigned to the Root PAS. The caller should not assume that any granules occupied by the L1 GPT object are accessible via the NS PAS.

I₀₁₄₁ As a part of L1 GPT object creation, the GPI encoding of granules that are occupied by the L1 GPT object is transitioned from the `Non-secure` GPI encoding to the `Root` GPI encoding. These granules could be a part of a `protected address range` that is covered by an L0 GPT Block descriptor, that is, it is not covered by an L1 GPT object.

This scenario highlights a recursive dependency where an L1 GPT object must be created for one `protected address range` to facilitate creation of an L1 GPT object for another `protected address range`.

It is possible that EL3 firmware is unable to allocate memory for the additional L1 GPT object. An invocation of the `FIRME_GM_L1_GPT_CREATE` ABI completes with the `NO_ENTRY` return status code in this scenario.

R₀₁₄₂ When an invocation of the `FIRME_GM_L1_GPT_CREATE` ABI completes with the `DENIED` return status code, no granule occupied by the L1 GPT object undergoes a `GPI transition`.

X₀₁₄₃

An invocation of the `FIRME_GM_L1_GPT_CREATE` ABI for a [protected address range](#) completes with the `DENIED` return status code when at least one granule in the memory region for the L1 GPT object does not have the [Non-secure](#) GPI encoding.

This is because as per the granule security policy, EL3 firmware cannot assign the [Root](#) GPI encoding to a granule whose current GPI encoding is not the [Non-secure](#) GPI encoding. EL3 firmware can detect this condition while iterating over granules in the L1 GPT object in order to assign them the [Root](#) GPI encoding.

When the condition is detected, one or more granules in the L1 GPT object could have been successfully assigned the [Root](#) GPI encoding. To ensure that the ABI invocation has no effect, EL3 firmware must restore the GPI encoding of the transitioned granules back to [Non-secure](#). The remaining granules do not undergo a [GPI transition](#).

U₀₁₄₄

If EL3 firmware implements fine grained locks over the granules occupied by the L1 GPT object, it might be unable to acquire all locks required to transition granules back to [Non-secure](#) before returning the `DENIED` return status code to the caller.

In this case, EL3 firmware can pause the on-going operation and complete the ABI invocation with the `INCOMPLETE` return status code. EL3 firmware can complete the ABI invocation with the `DENIED` return status code when the operation is resumed by the caller, and it is able to transition the granules back to [Non-secure](#) GPI encoding.

EL3 firmware can alternatively acquire a lock for all the granules occupied by the L1 GPT object, and transition them to the [Root](#) GPI encoding only after verifying that all of them have the [Non-secure](#) GPI encoding.

3.3.3.2 L1 GPT destruction

I₀₁₄₅ The [FIRME_GM_L1_GPT_DESTROY](#) ABI is used to replace an L1 GPT object for a [protected address range](#) with an L0 GPT Block descriptor as follows:

- The physical base address of the range is specified in the *PA range base* input parameter.
- The physical base address of the L1 GPT object for the range is returned in the *L1 GPT base* output parameter.

S₀₁₄₆ EL3 firmware performs the following non-exhaustive list of operations in response to an invocation of the [FIRME_GM_L1_GPT_DESTROY](#) ABI:

- The *PA range base* input parameter is validated for size and alignment.
- The presence of an L0 GPT Table descriptor for the specified [protected address range](#) is checked.
- A check is performed to ensure GPI encodings of all GPT Granules descriptors and/or GPT contiguous descriptors in the L1 GPT object are the same.
- The L0 GPT Table descriptor is replaced with a L0 GPT Block descriptor that has the same GPI value as all the descriptors in the L1 GPT object.
- All granules occupied by the L1 GPT object are scrubbed so that any confidential information from the Root world does not become accessible to the Non-Secure, Realm, or Secure worlds after the granule has transitioned to the NS PAS.
- The GPI encodings of granules in the L1 GPT object are transitioned from the [Root](#) GPI encoding to the [Non-secure](#) GPI encoding in the GPT.
 - All applicable cache and TLB invalidation operations required to perform the GPI transition are also performed to avoid leakage of information from the Root world into an untrusted Security state.
- The physical base address of the L1 GPT object is returned in the *L1 GPT base* output parameter.

R₀₁₄₇ When FEAT_RME is implemented, the granule security policy permits the following [GPI transition](#) via the [FIRME_GM_L1_GPT_DESTROY](#) ABI:

Caller's Security state	Current GPI encoding	Target GPI encoding
Realm	Root	Non-secure

Otherwise, the [GPI transition](#) is not permitted via the [FIRME_GM_L1_GPT_DESTROY](#) ABI.

R₀₁₄₈ During execution of the [FIRME_GM_L1_GPT_DESTROY](#) ABI, before any granule that is occupied by the L1 GPT object undergoes a [GPI transition](#) from the [Root](#) GPI encoding to the [Non-secure](#) GPI encoding, its contents are overwritten with a value that cannot be used to infer its original contents.

X₀₁₄₉ Granules that are occupied by an L1 GPT object contain information that is only accessible from the Root Security state. The information comprises of L1 GPT Granules descriptors or L1 GPT Contiguous descriptors. After a granule has undergone the [GPI transition](#), the ciphertext of this information becomes accessible to the Non-Secure, Realm, or Secure worlds. If the granules are not scrubbed as described in the above rule, ciphertext analysis is possible to break the memory encryption algorithm used for the Root PAS.

I₀₁₅₀ Execution of the [FIRME_GM_L1_GPT_DESTROY](#) ABI can be a long-running operation, during which interrupts may need to be handled in the Non-secure, Realm and Secure Security states. If a physical interrupt becomes pending during execution of the ABI, its invocation can complete before EL3 firmware is able to destroy the L1 GPT object for the specified [protected address range](#). In this case, the ABI is invoked again after the interrupt is handled to continue removal of the L1 GPT object.

S₀₁₅₁ The following pseudocode illustrates removal of an L1 GPT object.

```
int32_t l1_gpt_destroy(uint64_t *prot_addr_range_base) {
    int32 result;
    uint64_t *l1_gpt_base;

    // Start destruction of the L1 GPT object for the protected address range and
```

```
// iterate until it is complete.
do {
    result = FIRME_GM_L1_GPT_DESTROY(prot_addr_range_base, &l1_gpt_base);
} while (result == INCOMPLETE);

if (result == SUCCESS)
    free_mem(l1_gpt_base);

return result;
}
```

R0152 An invocation of the [FIRME_GM_L1_GPT_DESTROY](#) ABI completes with the [SUCCESS](#) return status code, if none of the following are true:

- The ABI is either not implemented by EL3 firmware or the caller is not allowed to invoke the ABI.
 - In this case, the ABI invocation completes with the [NOT_SUPPORTED](#) return status code.
- The [protected address range](#) does not completely lie within the protected address space defined by GPCCR_EL3.PPS.
 - In this case, the ABI invocation completes with the [INVALID_PARAMETERS](#) return status code.
- The base address of the [protected address range](#) is not correctly aligned.
 - In this case, the ABI invocation completes with the [INVALID_PARAMETERS](#) return status code.
- The [protected address range](#) is not covered by an L1 GPT object referenced by an L0 GPT Table descriptor.
 - In this case, the ABI invocation completes with the [NO_ENTRY](#) return status code.
- This is a request to destroy an L1 GPT object but another operation is already in-progress for the specified [protected address range](#).
 - In this case, the ABI invocation completes with the [IN_PROGRESS](#) return status code and has no effect.
- The caller is not allowed to destroy an L1 GPT object for the specified [protected address range](#).
 - In this case, the ABI invocation completes with the [DENIED](#) return status code and has no effect.
- All granules in the [protected address range](#) do not have the same GPI encoding.
 - In this case, the ABI invocation completes with the [DENIED](#) return status code and has no effect.
- EL3 firmware cannot complete the request due to an IMPLEMENTATION DEFINED reason and the caller must retry the operation.
 - In this case, the ABI invocation completes with the [RETRY](#) return status code.
- The request to destroy the L1 GPT object was interrupted due to an IMPLEMENTATION DEFINED reason and the caller must resume the operation.
 - In this case, the ABI invocation completes with the [INCOMPLETE](#) return status code.

R0153 When an invocation of the [FIRME_GM_L1_GPT_DESTROY](#) ABI completes with the [SUCCESS](#) return status code, all of the following are true:

- The memory occupied by the L1 GPT object is assigned to the NS PAS.
- For a [protected address range](#) PAR0, GPI_X == GPI_Y where:
 - GPI_X was the GPI value used by the L1 GPT object to perform granule protection checks on PAR0 before the invocation of the ABI.
 - GPI_Y is the GPI value in the L0 GPT Block descriptor to perform granule protection checks on PAR0 after the invocation of the ABI completes.

R0154 When an invocation of the [FIRME_GM_L1_GPT_DESTROY](#) ABI completes with the [SUCCESS](#) return status code, the physical base address of the L1 GPT object is returned in the *L1 GPT base* output parameter. Otherwise, the value of this parameter is UNKNOWN and is ignored by the caller.

R0155 When an invocation of the [FIRME_GM_L1_GPT_DESTROY](#) ABI completes with the [INCOMPLETE](#) return status code, behaviour of granule protection checks on the specified [protected address range](#) is the same as it was before removal of the L1 GPT object was started.

X0156 An in-progress operation to destroy an L1 GPT object for a [protected address range](#) has no effect on the behaviour of granule protection checks on that range.

- I₀₁₅₇ When an invocation of the [FIRME_GM_L1_GPT_DESTROY](#) ABI completes with the [INCOMPLETE](#) return status code, it is UNKNOWN which granules occupied by the L1 GPT object are assigned to the NS PAS. The caller should not assume that any granules occupied by the L1 GPT object are accessible via the NS PAS.
- R₀₁₅₈ There can be at most a single in-progress operation to either create or destroy an L1 GPT object for a [protected address range](#).
- X₀₁₅₉ If there is an ongoing operation to create or destroy an L1 GPT object for a [protected address range](#), another operation to create or destroy an L1 GPT object for the same range is not permitted. Also, an interrupted operation to create or destroy an L1 GPT object for a [protected address range](#) must be completed. It is not possible to start a new operation for the range when a previous operation is pending on the same range.

3.3.4 Feature discovery

- I₀₁₆₀** This version of the FIRME Granule Management service specifies the following feature registers:
- Feature register 0.
 - Feature register 1.
- See also:
- [2.5 Feature discovery](#).
- R₀₁₆₁** The size of each feature register, its index, and its encoding is specified in [Table 3.7](#). All unallocated values are unused.

Table 3.7: FIRME Granule Management service feature registers

Feature register name	Index	Size	Encoding
Feature register 0	0x1	64-bit	• Bits[63:4]: Unused. See also 2.2 SMC interface. • Bits[3]: Indicates presence of the FIRME_GM_L1_GPT_DESTROY ABI. • Bits[2]: Indicates presence of the FIRME_GM_L1_GPT_CREATE ABI. • Bits[1]: Indicates presence of the FIRME_GM_GPI_OP_CONTINUE ABI. • Bits[0]: Indicates presence of the FIRME_GM_GPI_SET ABI. • Value of each bit is determined from Table 3.8.
Feature register 1	0x0	64-bit	• Bits[63:9]: Unused. See also 2.2 SMC interface. • Bits[8:6]: PPS. – Value of Protected Physical Address size as encoded in GPCCR_EL3.PPS in [1]. • Bits[5:2]: LOGPTSZ. – Value of Level 0 GPT entry size as encoded in GPCCR_EL3.LOGPTSZ in [1]. • Bits[1:0]: PGS. – Value of Physical Granule size as encoded in GPCCR_EL3.PGS in [1].

3.3.5 Compliance

- R₀₁₆₂** The compliance rules for ABIs specified in this version of the FIRME Granule Management service are listed in [Table 3.8](#). See also:
- [2.6 Compliance](#).

Table 3.8: FIRME granule management service ABIs compliance rules

ABI name	NS FIRME instance	Secure FIRME instance	Realm FIRME instance
FIRME_GM_GPI_SET	Mandatory	Mandatory	Mandatory
FIRME_GM_GPI_OP_CONTINUE	Mandatory	Mandatory	Mandatory
FIRME_GM_L1_GPT_CREATE	Not supported	Not supported	Optional ¹
FIRME_GM_L1_GPT_DESTROY	Not supported	Not supported	Optional ¹

¹: Either both [FIRME_GM_L1_GPT_CREATE](#) and [FIRME_GM_L1_GPT_DESTROY](#) are implemented or neither is implemented.

Chapter 4

IDE key management service

4.1 Overview

I₀₁₆₃ The PCIe [5] and CXL [6] IDE Key programming mechanisms permit a Root Complex to use IMPLEMENTATION DEFINED key management to provide integrity and data encryption for TLPs transmitted through the link between a PCIe or CXL Root Port (RP) and an endpoint port. The RME System architecture specification [7] states that the IDE key programming register format for an RME-DA Root Port is IMPLEMENTATION DEFINED. RMSD access to the key programming interface is mediated through MSD firmware and implemented by IMPLEMENTATION DEFINED code that can execute either in MSD firmware or in a Trusted subsystem. The FIRME IDE key management service is a set of ABIs implemented by EL3 firmware i.e. MSD firmware to manage IDE keys in the IMPLEMENTATION DEFINED key programming interface at a Root port.

4.2 Scope

I₀₁₆₄ The scope of the FIRME IDE key management service is as follows:

- On a system that implements FEAT_RME, the Host or OS in the Normal world uses the IDE key management service ABIs exposed by the RMM to establish an IDE stream between the Root port and an Endpoint. In turn, the RMM uses the FIRME IDE key management service to establish an IDE stream between the Root port and an Endpoint.
- On a system that does not implement FEAT_RME, the Host or OS in the Normal world use the FIRME IDE key management service to establish an IDE stream between the Root port and an Endpoint.

4.3 Concepts

- I₀₁₆₅ The IDE key management service is implemented either at the *Realm FIRME instance* or at the *NS FIRME instance*. This service is not implemented at the *Secure FIRME instance*. See also:
- [Table 4.2](#).
- I₀₁₆₆ If the IDE key management service is implemented, all IDE key management ABIs described in this section are implemented.
- I₀₁₆₇ The IDE key management ABIs are used for configuring the following types of streams in a Root port:
- PCIe or CXL.io Selective IDE stream.
 - CXL.cachemem Link IDE stream.
- D₀₁₆₈ An invocation of an IDE key management ABI with the *Input Flags.IDE type* field set to 0b00 is a request to configure a PCIe or CXL.io Selective IDE stream.
- D₀₁₆₉ An invocation of an IDE key management ABI with the *Input Flags.IDE type* field set to 0b11 is a request to configure a CXL.cachemem Link IDE stream.
- D₀₁₇₀ The target keyset in an IDE key management ABI is identified by the *keyset ID*.
- R₀₁₇₁ *Keyset ID* is a 64-bit bitmap that is encoded as follows:
- Bits[63:38]: Unused.
 - Bits[37:30]: Segment number.
 - Bits[29:14]: Root port ID.
 - Bits[13:6]: Stream ID.
 - Bits[5:2]: Substream ID.
 - Bits[1]: Direction.
 - Bits[0]: Key set.
- I₀₁₇₂ The names of the fields in a *keyset ID* are terms defined in [5].
- R₀₁₇₃ The *Key Set* field in a *keyset ID* is unused in a request to configure a CXL.cachemem Link IDE stream. See also [2.2 SMC interface](#).
- X₀₁₇₄ CXL.cachemem IDE does not define a mechanism for directly updating the active keys. This is in contrast to PCIe IDE KM that allows programming of both the old and the new key sets.
- R₀₁₇₅ The value of the *Stream ID* field in a *keyset ID* is 0 in a request to configure a CXL.cachemem Link IDE stream.
- R₀₁₇₆ The value of the *Substream ID* field in a *keyset ID* is 0b1000 in a request to configure a CXL.cachemem Link IDE stream.
- R₀₁₇₇ The ECAM address space of a Root port defines the namespace for *keyset IDs*.
- R₀₁₇₈ A *keyset ID* is unique in the namespace defined by the ECAM address space of a Root port.
- I₀₁₇₉ The *FIRME_IDE_KEYSET_PROG* ABI programs a AES-GCM 256 bit key for the keyset associated with the *keyset ID* specified by the caller.
- Issue** The *FIRME_IDE_KEYSET_PROG* ABI does not specify the Invocation Field value (IFV) of the 96 bit initialisation vector (IV) for the AES-GCM key, as an input parameter. As per [5], the IV is deterministically constructed such that Bit[63:0] of the IV are set to 0000_0001h, for each Sub-Stream upon establishment of the Stream. As per [6], the IV could be generated locally in CXL.cachemem IDE port. Root ports compliant with the RME system architecture do not have this capability. If either is not IV generation capable, the Default IV is used as described in [6]. EL3 firmware can do the prescribed initialization without an input from the caller of this ABI. Arm requests feedback on the validity of this assumption.
- R₀₁₈₀ An invocation of the *FIRME_IDE_KEYSET_PROG* ABI successfully programs the key for the *keyset ID* specified by the caller, and completes with the *SUCCESS* return status code, if none of the following are true:
- The ABI is either not implemented by EL3 firmware or the caller is not allowed to invoke the ABI.
 - In this case, the ABI invocation completes with the *NOT_SUPPORTED* return status code.
 - The base address of the ECAM address space of the target Root port is invalid.

- In this case, the ABI invocation completes with the `INVALID_PARAMETERS` return status code.
- The *IDE type* field in the *Input Flags* input parameter is set to an unused value.
 - In this case, the ABI invocation completes with the `INVALID_PARAMETERS` return status code.
- The *keyset ID* is invalid.
 - In this case, the ABI invocation completes with the `INVALID_PARAMETERS` return status code.
- The FIRME ID key management service at the Root port is busy and the caller must retry the operation.
 - In this case, the ABI invocation completes with the `RETRY` return status code.
- An operation for the specified *keyset ID* is already in progress.
 - In this case, the ABI invocation completes with the `IN_PROGRESS` return status code and has no effect.
- The caller should poll for completion of the operation for the specified *keyset ID*.
 - In this case, the ABI invocation completes with the `INCOMPLETE` return status code. See also:
 - * `FIRME_IDE_KEYSET_POLL`.

I₀₁₈₁ The `FIRME_IDE_KEYSET_GO` ABI programs the Root port to start transmission and receipt of IDE TLPs for the keyset associated with the *keyset ID* specified by the caller.

R₀₁₈₂ An invocation of the `FIRME_IDE_KEYSET_GO` ABI successfully starts transmission and receipt of IDE TLPs using the *keyset ID* specified by the caller, and completes with the `SUCCESS` return status code, if none of the following are true:

- The ABI is either not implemented by EL3 firmware or the caller is not allowed to invoke the ABI.
 - In this case, the ABI invocation completes with the `NOT_SUPPORTED` return status code.
- The base address of the ECAM address space of the target Root port is invalid.
 - In this case, the ABI invocation completes with the `INVALID_PARAMETERS` return status code.
- The *IDE type* field in the *Input Flags* input parameter is set to an unused value.
 - In this case, the ABI invocation completes with the `INVALID_PARAMETERS` return status code.
- The *keyset ID* is invalid.
 - In this case, the ABI invocation completes with the `INVALID_PARAMETERS` return status code.
- The FIRME IDE key management service at the Root port is busy and the caller must retry the operation.
 - In this case, the ABI invocation completes with the `RETRY` return status code.
- An operation for the specified *keyset ID* is already in progress.
 - In this case, the ABI invocation completes with the `IN_PROGRESS` return status code and has no effect.
- No key was programmed for the specified *keyset ID*.
 - In this case, the ABI invocation completes with the `DENIED` return status code and has no effect.
- The caller should poll for completion of the operation for the specified *keyset ID*.
 - In this case, the ABI invocation completes with the `INCOMPLETE` return status code. See also:
 - * `FIRME_IDE_KEYSET_POLL`.

I₀₁₈₃ The `FIRME_IDE_KEYSET_STOP` ABI programs the Root port to stop transmission and receipt of IDE TLPs for the keyset associated with the *keyset ID* specified by the caller.

R₀₁₈₄ An invocation of the `FIRME_IDE_KEYSET_STOP` ABI successfully stops transmission and receipt of IDE TLPs using the *keyset ID* specified by the caller, and completes with the `SUCCESS` return status code, if none of the following are true:

- The ABI is either not implemented by EL3 firmware or the caller is not allowed to invoke the ABI.
 - In this case, the ABI invocation completes with the `NOT_SUPPORTED` return status code.
- The base address of the ECAM address space of the target Root port is invalid.
 - In this case, the ABI invocation completes with the `INVALID_PARAMETERS` return status code.
- The *IDE type* field in the *Input Flags* input parameter is set to an unused value.
 - In this case, the ABI invocation completes with the `INVALID_PARAMETERS` return status code.
- The *keyset ID* is invalid.
 - In this case, the ABI invocation completes with the `INVALID_PARAMETERS` return status code.
- The FIRME IDE key management service at the Root port is busy and the caller must retry the operation.
 - In this case, the ABI invocation completes with the `RETRY` return status code.
- An operation for the specified *keyset ID* is already in progress.
 - In this case, the ABI invocation completes with the `IN_PROGRESS` return status code and has no effect.
- No key was programmed for the specified *keyset ID*.

- In this case, the ABI invocation completes with the **DENIED** return status code and has no effect.
- The caller should poll for completion of the operation for the specified **keyset ID**.
 - In this case, the ABI invocation completes with the **INCOMPLETE** return status code. See also:
 - * **FIRME_IDE_KEYSET_POLL**.

D₀₁₈₅ A FIRME IDE key management operation is complete when the invocation of the corresponding ABI completes with the **SUCCESS** return status code. The caller is blocked while the operation is in progress. Alternatively, EL3 firmware can continue a requested operation in background, and complete the invocation of the ABI with the **INCOMPLETE** return status code, before the operation has completed. The caller of the ABI polls the status of the operation until it completes. This mechanism to complete an operation is called the *non-blocking* mode.

I₀₁₈₆ The number of FIRME IDE key management operations that EL3 firmware can complete in the *non-blocking* mode is IMPLEMENTATION DEFINED. EL3 firmware completes a request for an operation with the **RETRY** return status code if this IMPLEMENTATION DEFINED limit has exceeded.

I₀₁₈₇ The **FIRME_IDE_KEYSET_POLL** ABI is used to determine the status of a *non-blocking* IDE key management operation at a Root port as follows:

- The *Pending response type* field in the *Input flags* input parameter is set to 0 to retrieve a response for the keyset associated with the **keyset ID** specified by the caller.
- The *Pending response type* field in the *Input flags* input parameter is set to 1 to retrieve a response for any keyset. The **keyset ID** specified by the caller is ignored.

S₀₁₈₈ The FIRME IDE key management ABIs caters for a scenario where EL3 firmware delegates the responsibility for IDE key management to a Trusted subsystem via an IMPLEMENTATION DEFINED interface. It might not be acceptable to block the caller until the Trusted subsystem completes the request and sends the result back to EL3 firmware. In this case, EL3 firmware completes the invocation of the IDE key management ABI after delegating the operation to the Trusted subsystem. The caller polls the status of the operation via the **FIRME_IDE_KEYSET_POLL** ABI until the operation completes.

I₀₁₈₉ EL3 firmware does not provide a mechanism to determine which IDE key management operation is in progress in the *non-blocking* mode i.e. whether it is an operation to program a keyset, start use of a keyset, or stop use of a keyset. The caller of the FIRME IDE key management ABIs tracks the type of operation that was requested. Each IDE key management ABI provides the *Cookie1* and *Cookie2* parameters that can be used by a caller to track a pending operation as follows:

- The caller allocates, and specifies a non-zero value for one or both of these parameters to uniquely identify an IDE key management operation to program a keyset, start use of a keyset, or stop use of a keyset.
- If the requested operation completes via the *non-blocking* mechanism, EL3 firmware preserves the parameter values that were specified by the caller.
- When the **FIRME_IDE_KEYSET_POLL** ABI completes to acknowledge successful completion of a previously requested operation, EL3 firmware returns the preserved parameter values as return parameters, so that the caller can match a request to the response.

R₀₁₉₀ An invocation of the **FIRME_IDE_KEYSET_POLL** ABI completes with the **SUCCESS** return status code, to indicate that a *non-blocking* operation for the **keyset ID** specified by the caller completed successfully, if none of the following are true:

- The ABI is either not implemented by EL3 firmware or the caller is not allowed to invoke the ABI.
 - In this case, the ABI invocation completes with the **NOT_SUPPORTED** return status code.
- The base address of the ECAM address space of the target Root port is invalid.
 - In this case, the ABI invocation completes with the **INVALID_PARAMETERS** return status code.
- The *IDE type* field in the *Input Flags* input parameter is set to an unused value.
 - In this case, the ABI invocation completes with the **INVALID_PARAMETERS** return status code.
- The **keyset ID** is invalid.
 - In this case, the ABI invocation completes with the **INVALID_PARAMETERS** return status code.
- The FIRME IDE key management service at the Root port is busy and the caller must retry the operation.
 - In this case, the ABI invocation completes with the **RETRY** return status code.

- There is no **non-blocking** operation in progress for the specified **keyset ID** and the *Pending response type* field in the *Input flags* input parameter field was set to 0.
 - In this case, the ABI invocation completes with the **DENIED** return status code and has no effect.
- There is no **non-blocking** operation in progress for any **keyset ID** and the *Pending response type* field in the *Input flags* input parameter field was set to 1.
 - In this case, the ABI invocation completes with the **DENIED** return status code and has no effect.
- The **non-blocking** operation for the specified **keyset ID** is not complete.
 - In this case, the ABI invocation completes with the **INCOMPLETE** return status code.
- The **non-blocking** operation for the specified **keyset ID** could not complete because of an error.
 - In this case, the ABI invocation completes with the **ABORTED** return status code.

I₀₁₉₁ Completion of an IDE key management ABI with the **INCOMPLETE** return status code does not imply that the operation will complete successfully. The caller uses the **FIRME_IDE_KEYSET_POLL** ABI to check the status of the requested operation.

R₀₁₉₂ An invocation of any IDE key management ABI completes with the **NOT_SUPPORTED** return status code if the ABI is invoked from the Secure security state.

S₀₁₉₃ **Figure 4.1** shows an example flow where the Host enables PCIe IDE for a Selective stream between a TDI and a Root port, via the RMM. Communication between the Host and RMM takes place via **RMI_PDEV_COMMUNICATE** [3]. RMM uses the **FIRME IDE** key management ABIs to enable IDE for the stream at the Root port.

Chapter 4. IDE key management service

4.3. Concepts

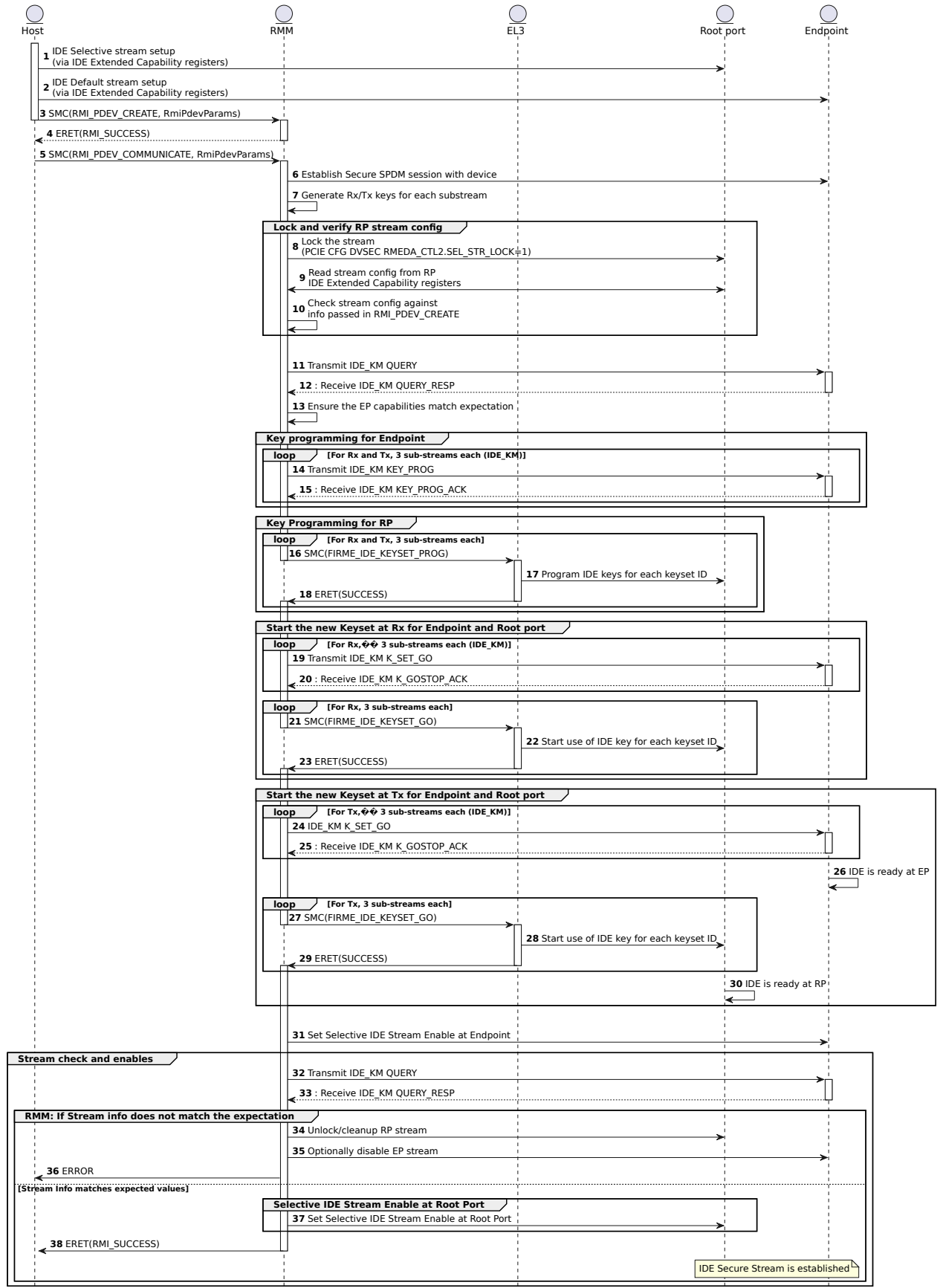


Figure 4.1: Example IDE key programming flow

S0194

Figure 4.2 shows an example flow where the Host refreshes the IDE key for a Selective stream between a TDI and a Root port, via the RMM. Communication between the Host and RMM takes place via RMI_PDEV_NOTIFY [3]. RMM uses the FIRME IDE key programming ABIs to refresh the IDE key for the stream at the Root port.

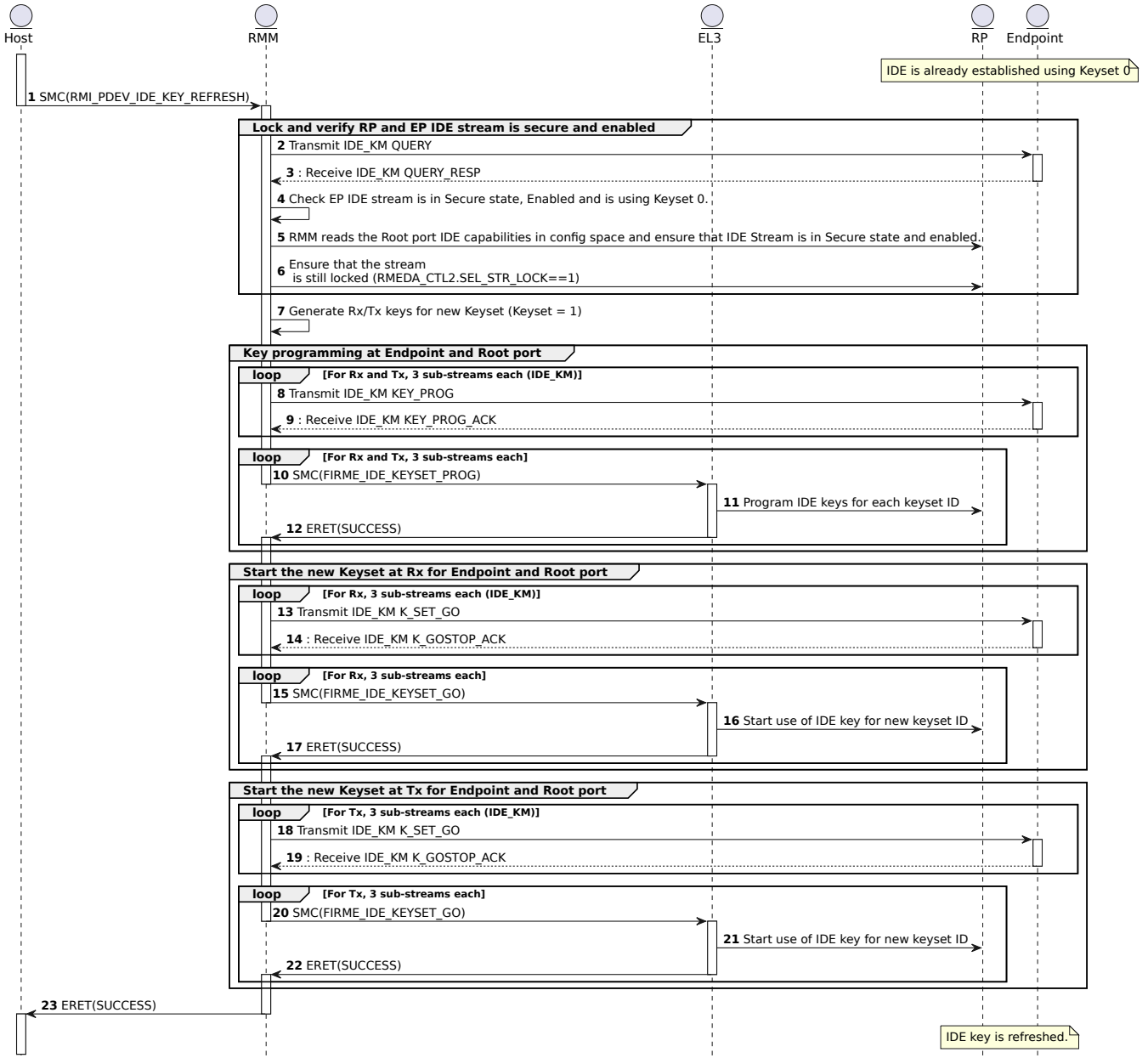


Figure 4.2: Example IDE key refresh flow

4.3.1 Feature discovery

I0195

This version of the FIRME IDE key management service specifies the following feature registers:

- Feature register 0.

See also:

- [2.5 Feature discovery](#).

R₀₁₉₆ The size of each feature register, its index, and its encoding is specified in [Table 4.1](#). All unallocated values are unused.

Table 4.1: FIRME IDE key management service feature registers

Feature register name	Index	Size	Encoding
Feature register 0	0x1	64-bit	• Bits[63:4]: Unused. See also 2.2 SMC interface • Bits[3]: Indicates presence of the FIRME_IDE_KEYSET_POLL ABI. • Bits[2]: Indicates presence of the FIRME_IDE_KEYSET_STOP ABI. • Bits[1]: Indicates presence of the FIRME_IDE_KEYSET_GO ABI. • Bits[0]: Indicates presence of the FIRME_IDE_KEYSET_PROG ABI. • Value of each bit is determined from Table 4.2.

4.3.2 Compliance

R₀₁₉₇ The compliance rules for ABIs specified in this version of the FIRME IDE key management service are listed in [Table 4.2](#). See also:

- [2.6 Compliance](#).

Table 4.2: FIRME IDE key management service ABIs compliance rules

ABI name	NS FIRME instance	Secure FIRME instance	Realm FIRME instance
FIRME_IDE_KEYSET_PROG	Mandatory	Not supported ¹	Mandatory
FIRME_IDE_KEYSET_GO	Mandatory	Not supported ¹	Mandatory
FIRME_IDE_KEYSET_STOP	Mandatory	Not supported ¹	Mandatory
FIRME_IDE_KEYSET_POLL	Mandatory	Not supported ¹	Mandatory

¹: The [Secure FIRME instance](#) does not implement the IDE key management service. See also [Table 2.3](#).

Chapter 5

MECID management service

5.1 Overview

I₀₁₉₈ FEAT_MEC is an extension to Arm's Realm Management Extension (RME), enhancing the security and isolation of protected execution environments known as Realms. While RME utilizes isolation and external memory protection to secure the Realm's state, FEAT_MEC introduces additional mechanisms to further strengthen these protections. Specifically, FEAT_MEC assigns unique memory encryption contexts to different memory regions, such as those assigned to a Realm, ensuring that each region's data is encrypted distinctly.

The MECID is a numeric ID that identifies a specific instance of a MEC. The MECID width is 1 to 16 bits. A MECID is expected to be reused multiple times until the system is reset. The cryptographic parameters in a MEC associated with a MECID are periodically refreshed so that there is no correlation between the contents of memory encrypted by this MEC before and after the refresh. For example, a MEC associated with a MECID is updated when a Realm is created or destroyed. This helps to reduce the risk of attacks that compromise the confidentiality of encrypted data in memory. For example, pattern-based or cryptanalytic attacks, physical attacks, attacks due to data leakage across Realms.

The FIRME MECID management service defines the ABI implemented by EL3 firmware, to refresh a MEC corresponding to a MECID.

5.2 Scope

I₀₁₉₉ This version of the FIRME MECID management service supports refresh of a MEC within the scope of features added by the following architecture extensions:

- FEAT_MEC

I₀₂₀₀ The FIRME MEC management interface is inaccessible from the Normal world and the Secure world. This is because the Non-secure and Secure address spaces each support only one MEC, which is associated with the default MECID of zero.

5.3 Concepts

R₀₂₀₁ A MEC, associated with a MECID, is refreshed when the existing encryption context is invalidated and a new encryption context is generated.

I₀₂₀₂ Refreshing a MEC could be performed by replacing an encryption key associated with the MECID, or more likely by updating a contributing input value, e.g., a tweak, into the memory encryption cipher. The design of the memory encryption system must ensure that input values cannot repeat during the uptime of a system.

R₀₂₀₃ When a MEC, associated with a MECID, is refreshed, for a given plaintext, the ciphertext generated by the MEC after refresh is not the same as any ciphertext for the same plaintext before the refresh.

R₀₂₀₄ A FIRME MECID management service implementation uses the common MECID width [1].

I₀₂₀₅ The common MECID width is discoverable via the COMMON_MECID_WIDTH field in feature register 1. See also:

- Table 5.1.

R₀₂₀₆ An invocation of the [FIRME_MEC_REFRESH](#) ABI successfully refreshes the MEC identified by the MECID specified by the caller, and completes with the [SUCCESS](#) return status code, if none of the following are true:

- The ABI is either not implemented by EL3 firmware or the caller is not allowed to invoke the ABI.
 - In this case, the ABI invocation completes with the [NOT_SUPPORTED](#) return status code.
- The MECID exceeds the common MECID width.
 - In this case, the ABI invocation completes with the [INVALID_PARAMETERS](#) return status code.
- A field in the MEC params input parameter is incorrectly encoded.
 - In this case, the ABI invocation completes with the [INVALID_PARAMETERS](#) return status code.
- The FIRME MECID management service is busy and the caller must retry the operation.
 - In this case, the ABI invocation completes with the [RETRY](#) return status code.

I₀₂₀₇ The RMM can specify the reason for refreshing a MEC to EL3 firmware. If the MEC refresh is requested when a Realm is created, the RMM sets the *MEC refresh reason* field in the *MEC params* input parameter to 0. If the MEC refresh is requested when a Realm is destroyed, the RMM sets the *MEC refresh reason* in the *MEC params* input parameter to 1.

X₀₂₀₈ When a Realm is created, a MECID is assigned to it by the RMM. The RMM requests a refresh of the MEC associated with the MECID to ensure that when the Realm's code and data is encrypted via this MEC, it has no correlation with memory encrypted via this MEC previously.

When the Realm is destroyed, its memory is scrubbed to prevent leakage of its data or code into another Realm. The RMM can instead refresh the MEC associated with the MECID assigned to the destroyed Realm. Memory scrubbing is not required, as the memory contents are guaranteed to be accessed via a different MEC, even if the physical memory is allocated to another Realm that is assigned the same MECID. Without memory scrubbing, the RMM is required to perform data cache clean and invalidate on all Realm memory before refreshing the MEC.

The *MEC refresh reason* field in the *MEC params* input parameter is a hint to the callee that enables it to apply IMPLEMENTATION DEFINED optimizations and mitigations. E.g., the callee could disable the specified MECID when the value of this parameter is 1. The callee could enable the specified MECID when the value of this parameter is 0.

5.3.1 Feature discovery

I₀₂₀₉ This version of the FIRME MECID management service specifies the following feature registers:

- Feature register 0.

See also:

- [2.5 Feature discovery](#).

R0210

The size of each feature register, its index, and its encoding is specified in [Table 5.1](#). All unallocated values are unused.

Table 5.1: FIRME MECID management service feature registers

Feature register name	Index	Size	Encoding
Feature register 0	0x1	64-bit	• Bits[63:1]: Unused. See also 2.2 SMC interface. • Bits[0]: Indicates presence of the FIRME_MEC_REFRESH ABI. • Value of each bit is determined from Table 5.2.

5.3.2 Compliance

R0211

The compliance rules for ABIs specified in this version of the FIRME MECID management service are listed in [Table 5.2](#). See also:

- [2.6 Compliance](#).

Table 5.2: FIRME MECID management service ABI compliance rules

ABI name	NS FIRME instance	Secure FIRME instance	Realm FIRME instance
FIRME_MEC_REFRESH	Not supported ¹	Not supported ¹	Mandatory

¹: The [Secure FIRME instance](#) and the [NS FIRME instance](#) do not implement the MECID management service. See also [Table 2.3](#).

Chapter 6

Attestation service

6.1 Overview

I₀₂₁₂ CCA attestation allows a user of a service provided by a Realm i.e. a relying party, to determine the trustworthiness of the Realm, and of the implementation of the CCA platform [8]. The RMM relies on EL3 firmware to access the attestation service implemented by the CCA HES for the following purposes:

- Obtain the Realm attestation key (RAK).
- Obtain the Platform attestation token (PAT).
- Issue Realm attestation token signing requests when the RMM does not have access to the private portion of the RAK.
- Extend the Platform attestation token (PAT) with CCA platform extension claims for components which have been added to the CCA platform at runtime.

The FIRME attestation service is a set of ABIs implemented by EL3 firmware to enable access to the attestation service implemented by the CCA HES.

6.2 Scope

I₀₂₁₃ This version of the FIRME attestation service only supports the attestation models described in [8]. Other attestation models are out of scope.

6.3 Concepts

I₀₂₁₄ The [FIRME_ATTEST_PAT_GET](#) ABI is used to obtain the platform attestation token from the CCA HES. See also:

- [6.3.2 Platform attestation token retrieval.](#)

I₀₂₁₅ The [FIRME_ATTEST_RAK_GET](#) ABI is used to obtain the public portion or the private portion of the Realm attestation key from the CCA HES. See also:

- [6.3.3 Realm attestation key retrieval.](#)

I₀₂₁₆ The [FIRME_ATTEST_RAT_SIGN](#) ABI is used to request the CCA HES to sign a Realm attestation token when the RMM does not have access to the private portion of the Realm attestation key. See also:

- [6.3.4 Realm attestation token signing.](#)

I₀₂₁₇ The FIRME attestation service specifies a set of ABIs to manage extension of the platform attestation token with a set of CCA platform extension claims as follows:

- The [FIRME_ATTEST_PAT_EXT_CLAIMS_STAGE](#) ABI is used to transmit a set of CCA platform extension claims to the CCA HES. The claims are staged by the CCA HES in a local buffer.
- The [FIRME_ATTEST_PAT_EXT_CLAIMS_FINALISE](#) ABI is used to validate that the combination of staged CCA platform extension claims are well formed CBOR before they can be added to the platform attestation token.
- The [FIRME_ATTEST_PAT_EXT_CLAIMS_CLEAR](#) ABI is used to remove all the CCA platform extension claims from the platform attestation token in the CCA HES. The staged claims are also removed from the local buffer.

The updated platform attestation token is retrieved via the [FIRME_ATTEST_PAT_GET](#) ABI.

See also:

- [6.3.5 Platform attestation token extension.](#)

I₀₂₁₈ An implementation of the FIRME attestation service at the [Realm FIRME instance](#) either implements all platform attestation token extension ABIs or none at all. See also:

- [Table 6.4.](#)

6.3.1 Shared buffer usage

I₀₂₁₉ A [shared buffer](#) is used by the following FIRME attestation service ABIs to transmit input data to, and receive output data from EL3 firmware:

- [FIRME_ATTEST_PAT_GET](#) ABI.
- [FIRME_ATTEST_RAK_GET](#) ABI.
- [FIRME_ATTEST_RAT_SIGN](#) ABI.
- [FIRME_ATTEST_PAT_EXT_CLAIMS_STAGE](#) ABI.

Additionally, these ABIs support [partial retrieval](#) to receive output data in chunks.

See also:

- [2.8 Shared buffer.](#)
- [2.8.1 Data retrieval in chunks.](#)

I₀₂₂₀ EL3 firmware retrieves output data from the CCA HES. The communication interface between these two components could be such that while EL3 firmware is able to send a request to the CCA HES to start retrieval of output data, it does not obtain an immediate response from the CCA HES. In this scenario, EL3 firmware returns a zero *Written size* and a zero *Remaining size* of the output data. The caller invokes the ABI as many times as it is required by EL3 firmware to obtain the entire output data from the CCA HES and return it to the caller.

I₀₂₂₁ When a FIRME attestation service ABI that supports [partial retrieval](#) completes a request to retrieve output data with the [INCOMPLETE](#) return status code, if the *Written size* is zero, the *Remaining size* of the output data can be non-zero. This could happen in a scenario where EL3 firmware is unable to write a new chunk of the output data due to an IMPLEMENTATION DEFINED reason.

6.3.2 Platform attestation token retrieval

- I₀₂₂₂ The [FIRME_ATTEST_PAT_GET](#) ABI is implemented only at the [NS FIRME instance](#) and the [Realm FIRME instance](#). See also:
- [Table 6.4](#).
- I₀₂₂₃ The [FIRME_ATTEST_PAT_GET](#) ABI uses a physically contiguous [shared buffer](#) for transmitting the platform challenge as input data, and receiving the platform attestation token as output data.
- I₀₂₂₄ The [MAX_PAT_PG_CNT](#) field in feature register 1 specifies the maximum size of the platform attestation token. If this size is $\leq$ Maximum shared buffer size determined from the [MAX_SH_BUF_PG_CNT](#) field in the same register, the caller of the [FIRME_ATTEST_PAT_GET](#) ABI can dynamically allocate the exact amount of memory required to retrieve the complete platform attestation token. See also:
- [Table 6.3](#).
- I₀₂₂₅ The encoding of the platform attestation token obtained from the CCA HES is specified in [3].
- D₀₂₂₆ An invocation of the [FIRME_ATTEST_PAT_GET](#) ABI with a non-zero *Platform challenge size* input parameter is a request to initiate retrieval of the platform attestation token.
- D₀₂₂₇ An invocation of the [FIRME_ATTEST_PAT_GET](#) ABI with a zero *Platform challenge size* input parameter is a request to continue retrieval of the platform attestation token.
- I₀₂₂₈ Once a caller has initiated retrieval of the platform attestation token, a subsequent invocation of the [FIRME_ATTEST_PAT_GET](#) ABI on any PE at the same [FIRME instance](#) either continues retrieval of the platform attestation token (if the *Platform challenge size* is zero), or restarts retrieval of the platform attestation token (if the *Platform challenge size* is non-zero).
- R₀₂₂₉ If the [FIRME_ATTEST_PAT_GET](#) ABI must be invoked multiple time to retrieve the complete platform attestation token, there can be a single in-progress retrieval of the platform attestation token at a [FIRME instance](#).
- R₀₂₃₀ When an invocation of the [FIRME_ATTEST_PAT_GET](#) ABI completes with the [SUCCESS](#) return status code, all of the following are true:
- *Written size* > 0.
 - *Remaining size* == 0.
- I₀₂₃₁ When an invocation of the [FIRME_ATTEST_PAT_GET](#) ABI completes with the [SUCCESS](#) return status code, retrieval of the platform attestation token is complete.
- I₀₂₃₂ When an invocation of the [FIRME_ATTEST_PAT_GET](#) ABI completes with the [INCOMPLETE](#) return status code, [Table 6.1](#) describes the valid combinations of the following parameters:
- *Platform challenge size* input parameter.
 - *Written size* output parameter.
 - *Remaining size* output parameter.

Table 6.1: Parameter description when [FIRME_ATTEST_PAT_GET](#) returns [INCOMPLETE](#)

Platform challenge size	Written size	Remaining size	Description
Non-zero	0	0	EL3 firmware has not obtained the platform attestation token from the CCA HES. No chunk of the token was written to the shared buffer. The FIRME_ATTEST_PAT_GET ABI must be invoked again to retrieve the platform attestation token.

Platform challenge size	Written size	Remaining size	Description
Non-zero	0	Non-zero	No chunk of the platform attestation token was written to the shared buffer. The FIRME_ATTEST_PAT_GET ABI must be invoked again to retrieve the platform attestation token.
Non-zero	Non-zero	Non-zero	A chunk of the platform attestation token has been written to the shared buffer. The FIRME_ATTEST_PAT_GET ABI must be invoked again to retrieve the platform attestation token.
0	0	0	EL3 firmware has not obtained the platform attestation token from the CCA HES. No chunk of the token was written to the shared buffer. The FIRME_ATTEST_PAT_GET ABI must be invoked again to retrieve the platform attestation token.
0	0	Non-zero	No chunk of the platform attestation token was written to the shared buffer. The FIRME_ATTEST_PAT_GET ABI must be invoked again to retrieve the platform attestation token.
0	Non-zero	Non-zero	A chunk of the platform attestation token has been written to the shared buffer. The FIRME_ATTEST_PAT_GET ABI must be invoked again to retrieve the platform attestation token.

R0233

An invocation of the [FIRME_ATTEST_PAT_GET](#) ABI successfully completes with the [SUCCESS](#) return status code, if none of the following are true:

- The ABI is not implemented at this [FIRME instance](#).
 - In this case, the ABI invocation completes with the [NOT_SUPPORTED](#) return status code.
- The physical base address of the [shared buffer](#) is invalid.
 - In this case, the ABI invocation completes with the [INVALID_PARAMETERS](#) return status code.
- The *Write offset* from the base of the [shared buffer](#) is invalid.
 - In this case, the ABI invocation completes with the [INVALID_PARAMETERS](#) return status code.
- The *Shared buffer size* is invalid.
 - In this case, the ABI invocation completes with the [INVALID_PARAMETERS](#) return status code.
- The *Platform challenge size* in a request to initiate retrieval of the platform attestation token is invalid.
 - In this case, the ABI invocation completes with the [INVALID_PARAMETERS](#) return status code.
- This is a request to continue retrieval of the platform attestation token when retrieval of the token was not previously initiated.
 - In this case, the ABI invocation completes with the [INVALID_PARAMETERS](#) return status code.
- EL3 firmware could not allocate translation tables to map the [Shared buffer](#) in its translation regime.
 - In this case, the ABI invocation completes with the [NO_MEMORY](#) return status code.
- The CCA HES interface to request the platform attestation token is busy and the caller must retry the operation.
 - In this case, the ABI invocation completes with the [RETRY](#) return status code.
- EL3 firmware has started a [stateful long-running](#) operation as the retrieval of the platform attestation token is incomplete.
 - In this case, the ABI invocation completes with the [INCOMPLETE](#) return status code.
- EL3 firmware cannot complete retrieval the platform attestation token due to an IMPLEMENTATION DEFINED reason.
 - In this case, the ABI invocation completes with the [ABORTED](#) return status code and the caller must restart the operation. See also:
 - * [6.3.6 Aborted operations](#).

S0234

The following pseudocode illustrates retrieval of the platform attestation token via the [FIRME_ATTEST_PAT_GET](#) ABI. It assumes that the [shared buffer](#) is dynamically allocated and it is not already mapped in the EL3 translation

regime prior to the invocation of this ABI.

```
int get_platform_attestation_token(uchar *plat_challenge, uint64_t
    ↪ plat_challenge_sz)
{
    int ret;
    uint64_t alloc_sz;
    uint64_t shbuf;

    // Determine the minimum shared buffer size.
    ret = FIRME_SERVICE_FEATURES(Attestation service ID, Feature register 1, &
    ↪ alloc_sz);
    if (ret) {
        return ret;
    }

    // Allocate the shared buffer in bytes.
    shbuf = alloc(alloc_sz * 1024);

    // Copy the platform challenge into the allocated buffer at offset 0x0.
    memcpy(shbuf, plat_challenge, plat_challenge_sz);

    do {
        uint64_t offset = 0, written_sz, remaining_sz;

        do {
            ret = FIRME_ATTEST_PAT_GET(shbuf, offset, alloc_sz, plat_challenge_sz,
            ↪ &written_sz, &remaining_sz);
            plat_challenge_sz = 0; // Reset this value since we need to retrieve
            ↪ rest of the platform attestation token.
            offset += written_sz;
        } while (ret == INCOMPLETE && remaining_sz && offset < alloc_sz);

    } while (ret == INCOMPLETE && remaining_sz && (offset >= alloc_sz));

    return ret;
}
```

6.3.3 Realm attestation key retrieval

- I₀₂₃₅ The [FIRME_ATTEST_RAK_GET](#) ABI is implemented only at the [Realm FIRME instance](#). See also:
- [Table 6.4](#).
- I₀₂₃₆ The [FIRME_ATTEST_RAK_GET](#) ABI uses a physically contiguous [shared buffer](#) for receiving the Realm attestation key.
- I₀₂₃₇ The [RAK_FORMAT](#) field in feature register 1 specifies the format in which the Realm attestation key is encoded in the [shared buffer](#). See also:
- [Table 6.3](#).
- I₀₂₃₈ The [FIRME_ATTEST_RAK_GET](#) ABI is used to obtain either the private portion or the public portion of the Realm attestation key from the CCA HES. In this version of the specification, it cannot be used to obtain both portions of the Realm attestation key. Also, support for retrieval of the public portion of the Realm attestation key is optional and determined by the value of the [RAK_PUB_POR](#) field in feature register 1. See also:
- [Table 6.3](#).
- X₀₂₃₉ Retrieval of the private portion and the public portion of the Realm attestation key is mutually exclusive for the following reasons:

- An implementation chooses to protect the private portion of the key in the CCA HES. The RMM retrieves the public portion of the key, and includes it in the Realm attestation token, before using the [FIRME_ATTEST_RAT_SIGN](#) ABI to request the CCA HES to sign the token with the private portion of the key. See also:
 - [6.3.4 Realm attestation token signing](#).
- An implementation allows RMM to access the private portion of the key. The RMM retrieves the private portion of the key via the [FIRME_ATTEST_RAK_GET](#) ABI, and uses the key to sign a Realm attestation token. The RMM uses the private portion of the key to derive the public portion of the key. The hash of the public portion is used as the platform challenge in an invocation of the [FIRME_ATTEST_PAT_GET](#) ABI to obtain the platform attestation token.

- D₀₂₄₀ An invocation of the [FIRME_ATTEST_RAK_GET](#) ABI with the *Request type* field in the *Input flags* input parameter set to 0 is a request to initiate retrieval of the specified portion of the Realm attestation key as output data.
- D₀₂₄₁ An invocation of the [FIRME_ATTEST_RAK_GET](#) ABI with the *Request type* field in the *Input flags* input parameter set to 1 is a request to continue retrieval of the specified portion of the Realm attestation key as output data.
- R₀₂₄₂ If the [FIRME_ATTEST_RAK_GET](#) ABI is invoked multiple times to retrieve the specified portion of the Realm attestation key, there is a single in-progress retrieval of this output data at a [FIRME instance](#).
- I₀₂₄₃ Once a caller has initiated retrieval of the specified portion of the Realm attestation key, a subsequent invocation of the [FIRME_ATTEST_RAK_GET](#) ABI on any PE at the same [FIRME instance](#) results in one of the following outcomes:
- If the *Request type* flag is 1, retrieval of the specified portion of the Realm attestation key continues.
 - If the *Request type* flag is 0, a new request to retrieve a portion of the Realm attestation key is started.
- R₀₂₄₄ When an invocation of the [FIRME_ATTEST_RAK_GET](#) ABI completes with the [SUCCESS](#) return status code, all of the following are true:
- *Written size* > 0.
 - *Remaining size* == 0.
- I₀₂₄₅ When an invocation of the [FIRME_ATTEST_RAK_GET](#) ABI completes with the [SUCCESS](#) return status code, retrieval of the specified portion of the Realm attestation key is complete.
- I₀₂₄₆ When an invocation of the [FIRME_ATTEST_RAK_GET](#) ABI completes with the [INCOMPLETE](#) return status code, [Table 6.2](#) describes the valid combinations of the following parameters:
- *Request type* flag input parameter.
 - *Written size* output parameter.
 - *Remaining size* output parameter.

Table 6.2: Parameter description when [FIRME_ATTEST_RAK_GET](#) returns [INCOMPLETE](#)

Request type	Written size	Remaining size	Description
0	0	0	EL3 firmware has not obtained the specified portion of the Realm attestation key from the CCA HES. No chunk of the key was written to the shared buffer. The FIRME_ATTEST_RAK_GET ABI must be invoked again to retrieve the key.
0	0	Non-zero	No chunk of the specified portion of the Realm attestation key was written to the shared buffer. The FIRME_ATTEST_RAK_GET ABI must be invoked again to retrieve the key.

Request type	Written size	Remaining size	Description
0	Non-zero	Non-zero	A chunk of the specified portion of the Realm attestation key has been written to the shared buffer. The FIRME_ATTEST_RAK_GET ABI must be invoked again to retrieve the key.
1	0	0	EL3 firmware has not obtained the specified portion of the Realm attestation key from the CCA HES. No chunk of the key was written to the shared buffer. The FIRME_ATTEST_RAK_GET ABI must be invoked again to retrieve the key.
1	0	Non-zero	No chunk of the specified portion of the Realm attestation key was written to the shared buffer. The FIRME_ATTEST_RAK_GET ABI must be invoked again to retrieve the key.
1	Non-zero	Non-zero	A chunk of the specified portion of the Realm attestation key has been written to the shared buffer. The FIRME_ATTEST_RAK_GET ABI must be invoked again to retrieve the key.

R0247

An invocation of the [FIRME_ATTEST_RAK_GET](#) ABI successfully completes with the [SUCCESS](#) return status code, if none of the following are true:

- The ABI is not implemented at this [FIRME instance](#).
 - In this case, the ABI invocation completes with the [NOT_SUPPORTED](#) return status code.
- The physical base address of the [shared buffer](#) is invalid.
 - In this case, the ABI invocation completes with the [INVALID_PARAMETERS](#) return status code.
- The *Write offset* from the base of the [shared buffer](#) is invalid.
 - In this case, the ABI invocation completes with the [INVALID_PARAMETERS](#) return status code.
- The *Shared buffer page count* is invalid.
 - In this case, the ABI invocation completes with the [INVALID_PARAMETERS](#) return status code.
- The *Request type* flag in the *Input flags* input parameter is 1, and flags to retrieve the public portion, and the private portion of the Realm attestation key are both set.
 - In this case, the ABI invocation completes with the [INVALID_PARAMETERS](#) return status code.
- The flag to retrieve the public portion of the Realm attestation key is set but retrieval of this portion of the key is not supported by EL3 firmware.
 - In this case, the ABI invocation completes with the [INVALID_PARAMETERS](#) return status code.
- The *Request type* flag in the *Input flags* input parameter is 1, and at least one of the flags to retrieve the public portion, and the private portion of the Realm attestation key is set.
 - In this case, the ABI invocation completes with the [INVALID_PARAMETERS](#) return status code.
- The type of elliptic curve is not supported by the CCA HES.
 - In this case, the ABI invocation completes with the [INVALID_PARAMETERS](#) return status code.
- This is a request to continue retrieval of the Realm attestation key when retrieval of the key was not previously initiated.
 - In this case, the ABI invocation completes with the [INVALID_PARAMETERS](#) return status code.
- EL3 firmware could not allocate translation tables to map the [Shared buffer](#) in its translation regime.
 - In this case, the ABI invocation completes with the [NO_MEMORY](#) return status code.
- The CCA HES interface to request the specified portion of the Realm attestation key is busy and the caller must retry the operation.
 - In this case, the ABI invocation completes with the [RETRY](#) return status code.
- EL3 firmware has started a [stateful long-running](#) operation as the retrieval of the specified portion of the Realm attestation key is incomplete.
 - In this case, the ABI invocation completes with the [INCOMPLETE](#) return status code.
- EL3 firmware cannot complete retrieval of the specified portion of the Realm attestation key due to an IMPLEMENTATION DEFINED reason.
 - In this case, the ABI invocation completes with the [ABORTED](#) return status code and the caller must

restart the operation. See also:
* [6.3.6 Aborted operations](#).

S₀₂₄₈

The following pseudocode illustrates retrieval of the Realm attestation key via the [FIRME_ATTEST_RAK_GET](#) ABI. It assumes that the [shared buffer](#) is dynamically allocated and it is not already mapped in the EL3 translation regime prior to the invocation of this ABI.

```
int get_realm_attestation_key(bool portion, uint8_t elliptic_curve)
{
    int ret;
    uint64_t alloc_sz;
    uint64_t shbuf, input_flags;

    // Determine the minimum shared buffer size.
    ret = FIRME_SERVICE_FEATURES(Attestation service ID, Feature register 1, &
        ↪alloc_sz);
    if (ret) {
        return ret;
    }

    // Allocate the shared buffer in bytes.
    shbuf = alloc(alloc_sz * 1024);

    // Specify which portion of the key to return.
    portion == public ? input_flags.Bits[1] = 1: input_flags.Bits[2] = 1;

    // Set the request type.
    input_flags.Bits[0] = 0;

    do {
        uint64_t offset = 0, written_sz, remaining_sz;

        do {
            ret = FIRME_ATTEST_RAK_GET(shbuf, offset, alloc_sz, input_flags, &
                ↪written_sz, &remaining_sz);
            input_flags.Bits[0] = 1; // Reset this value in case we need to
                ↪retrieve rest of the key.
            offset += written_sz;
        } while (ret == INCOMPLETE && remaining_sz && offset < alloc_sz);

    } while (ret == INCOMPLETE && remaining_sz && (offset >= alloc_sz));

    return ret;
}
```

6.3.4 Realm attestation token signing

I₀₂₄₉

The [FIRME_ATTEST_RAT_SIGN](#) ABI is optional and can be implemented only at the [Realm FIRME instance](#). See also:

- [Table 6.4](#).

R₀₂₅₀

The [RAK_PUB_POR](#) field in feature register 1 is 1 if the [FIRME_ATTEST_RAT_SIGN](#) ABI is implemented. See also:

- [Table 6.3](#).

X₀₂₅₁

The Realm attestation token includes the public portion of the Realm attestation key. The RMM must be able to retrieve the public portion of the Realm attestation key in order to generate a Realm attestation token signing request.

- I₀₂₅₂ The [FIRME_ATTEST_RAT_SIGN](#) ABI uses a physically contiguous [shared buffer](#) as follows:
- For transmitting the Realm attestation token to be signed to the CCA HES.
 - For receiving the signature of the Realm attestation token from the CCA HES.
- I₀₂₅₃ The encoding of the request to sign a Realm attestation token and the encoding of the response that contains the signature is IMPLEMENTATION DEFINED.
- Issue** This version of the [FIRME](#) specification assumes that the RMM is aware of the encodings used by the CCA HES via an IMPLEMENTATION DEFINED mechanism. The role of EL3 firmware is to transmit the request to sign a Realm attestation token from the RMM to the CCA HES, and transmit the response that contains the signature of the token from the CCA HES to the RMM. Hence, it does not need to interpret the contents of the request and the response. Arm requests feedback whether a standard encoding of the request and response is required, and whether this should be discoverable via the [FIRME_SERVICE_FEATURES](#) ABI.
- D₀₂₅₄ An invocation of the [FIRME_ATTEST_RAT_SIGN](#) ABI with the *Request type* flag set to 0 in the *Request attributes* input parameter is a request to sign a Realm attestation token.
- R₀₂₅₅ In a request to sign a Realm attestation token, all of the following are true:
- The payload containing the Realm attestation token is populated at offset 0x0 of the [shared buffer](#).
 - The size of the payload is specified in the *Input payload size* field of the *Request attributes* input parameter.
- D₀₂₅₆ An invocation of the [FIRME_ATTEST_RAT_SIGN](#) ABI with the *Request type* flag set to 1 in the *Request attributes* input parameter is a request to retrieve the signature of a Realm attestation token.
- R₀₂₅₇ If a request to retrieve the signature of a Realm attestation token completes with the [SUCCESS](#) return status code, all of the following are true:
- The payload containing the signature of a Realm attestation token is populated at offset 0x0 of the [shared buffer](#).
 - The size of the payload is specified in the *Output payload size* parameter.
- If no signature of a Realm attestation token is available for retrieval, the size of the payload in the *Output payload size* parameter is 0.
- R₀₂₅₈ If a request to retrieve the signature of a Realm attestation token does not complete with the [SUCCESS](#) return status code, the value of the *Output payload size* parameter indicates if there is a response payload in addition to the error code returned by EL3 firmware.
- If the value of this parameter is 0, there is no additional response payload. Otherwise, there is an additional response payload and all of the following are true:
- The response payload is populated at offset 0x0 of the [shared buffer](#).
 - The size of the payload is specified in the *Output payload size* parameter.
- X₀₂₅₉ There can be multiple outstanding requests to sign a Realm attestation token. A request to retrieve the signature of a Realm attestation token returns any available response to a previous signing request.
- It is assumed that the RMM encodes information in the payload that contains the Realm attestation token, that it can use to match a request to a response from the CCA HES. The CCA HES encodes the same information in the payload that contains the signature of the Realm attestation token. This mechanism enables the RMM to match a successful response to a signing request.
- It is possible that the CCA HES is unable to fulfil a signing request. E.g., after the CCA HES has refreshed the Realm attestation key, it denies any new signing requests until the RMM retrieves the refreshed public portion of the Realm attestation key. In this case, the CCA HES populates a payload in addition to returning the [DENIED](#) error code back to the RMM. The additional payload contains information that allows the RMM to determine which signing request was denied. This payload can be used by the CCA HES to signal other errors too.
- I₀₂₆₀ The number of outstanding requests for signing a Realm attestation token, and the number of outstanding responses with signatures of Realm attestation tokens is IMPLEMENTATION DEFINED. An IMPLEMENTATION DEFINED mechanism is used between the RMM and the CCA HES to match requests to responses.

- S₀₂₆₁** The RMM queues multiple requests for the CCA HES to sign Realm attestation tokens on behalf of various Realms via the [FIRME_ATTEST_RAT_SIGN](#) ABI. Each request also encodes information in the payload to uniquely identify it e.g., a request identifier. The RMM dequeues responses from the CCA HES by retrieving signatures of the Realm attestation tokens via the [FIRME_ATTEST_RAT_SIGN](#) ABI. Each response also includes the request identifier to match it to the corresponding request.
- R₀₂₆₂** An invocation of the [FIRME_ATTEST_RAT_SIGN](#) ABI successfully completes with the [SUCCESS](#) return status code, if none of the following are true:
- The ABI is not implemented at this [FIRME instance](#).
 - In this case, the ABI invocation completes with the [NOT_SUPPORTED](#) return status code.
 - The physical base address of the [shared buffer](#) is invalid.
 - In this case, the ABI invocation completes with the [INVALID_PARAMETERS](#) return status code.
 - The [Shared buffer page count](#) is invalid.
 - In this case, the ABI invocation completes with the [INVALID_PARAMETERS](#) return status code.
 - The *Input payload size* field in the *Request attributes* input parameter is invalid.
 - In this case, the ABI invocation completes with the [INVALID_PARAMETERS](#) return status code.
 - EL3 firmware could not allocate translation tables to map the [Shared buffer](#) in its translation regime.
 - In this case, the ABI invocation completes with the [NO_MEMORY](#) return status code.
 - The CCA HES interface that accepts a Realm attestation token signing request, and returns the signature of a Realm attestation token is busy and the caller must retry the operation.
 - In this case, the ABI invocation completes with the [RETRY](#) return status code.
 - The CCA HES has refreshed the Realm attestation key but the RMM has not retrieved the refreshed public portion of the Realm attestation key via the [FIRME_ATTEST_RAK_GET](#) ABI.
 - In this case, the ABI invocation completes with the [DENIED](#) return status code and has no effect.
 - EL3 firmware cannot complete the request for signing the Realm attestation token due to an IMPLEMENTATION DEFINED reason.
 - In this case, the ABI invocation completes with the [ABORTED](#) return status code. See also:
 - * [6.3.6 Aborted operations](#).

6.3.5 Platform attestation token extension

- I₀₂₆₃** The [FIRME_ATTEST_PAT_EXT_CLAIMS_STAGE](#) ABI is implemented only at the [Realm FIRME instance](#). See also:
- [Table 6.4](#).
- I₀₂₆₄** The [FIRME_ATTEST_PAT_EXT_CLAIMS_STAGE](#) ABI uses a physically contiguous [shared buffer](#) for transmitting a set of CCA platform extension claims as input data.
- I₀₂₆₅** The size of a set of CCA platform extension claims could exceed the size of the [shared buffer](#). In this case, a chunk of the set of claims is transmitted in each invocation of the [FIRME_ATTEST_PAT_EXT_CLAIMS_STAGE](#) ABI.
- R₀₂₆₆** If a set of CCA platform extension claims is transmitted in more than one chunk, the CCA HES populates its local buffer with the claims as follows:
- If the local buffer is empty, the chunk is written at offset 0x0 in the buffer.
 - Otherwise, the chunk is written at an offset that is equal to the size of all previously received chunks.
- I₀₂₆₇** The [MAX_PAT_EXT_BUF_PG_CNT](#) field in feature register 1 specifies the maximum size of the local buffer allocated in the CCA HES for staging CCA platform extension claims. The caller of the [FIRME_ATTEST_PAT_EXT_CLAIMS_STAGE](#) ABI can use this field to determine the maximum number of CCA platform extension claims that can be staged in the CCA HES before they can be added to the platform attestation token. See also:
- [Table 6.3](#).
- I₀₂₆₈** The value of the [MAX_PAT_EXT_BUF_PG_CNT](#) field in feature register 1 is 0 if any of the following is true:
- The [FIRME_SERVICE_FEATURES](#) ABI is not invoked at the [Realm FIRME instance](#).

- The `FIRME_ATTEST_PAT_EXT_CLAIMS_STAGE` ABI is not implemented at the [Realm FIRME instance](#).

See also:

- [Table 6.3](#).

I₀₂₆₉ The encoding of a CCA platform extension claim is specified in [3].

R₀₂₇₀ In a request to transmit a set of CCA platform extension claims, all of the following are true:

- The payload containing the set of CCA platform extension claims is populated at offset 0x0 of the [shared buffer](#).
- The size of the payload is specified in the *Input payload size* field of the *Request attributes* input parameter.

R₀₂₇₁ An invocation of the `FIRME_ATTEST_PAT_EXT_CLAIMS_STAGE` ABI successfully completes with the `SUCCESS` return status code, if none of the following are true:

- The ABI is not implemented at this [FIRME instance](#).
 - In this case, the ABI invocation completes with the `NOT_SUPPORTED` return status code.
- The physical base address of the [shared buffer](#) is invalid.
 - In this case, the ABI invocation completes with the `INVALID_PARAMETERS` return status code.
- The *Shared buffer page count* is invalid.
 - In this case, the ABI invocation completes with the `INVALID_PARAMETERS` return status code.
- The *Input payload size* field in the *Request attributes* input parameter is invalid.
 - In this case, the ABI invocation completes with the `INVALID_PARAMETERS` return status code.
- EL3 firmware could not allocate translation tables to map the [Shared buffer](#) in its translation regime.
 - In this case, the ABI invocation completes with the `NO_MEMORY` return status code.
- The CCA HES interface that accepts CCA platform extension claims is busy and the caller must retry the operation.
 - In this case, the ABI invocation completes with the `RETRY` return status code.
- The size of the last chunk of CCA platform attestation claims exceeds the size of the local buffer in the CCA HES to stage the claims.
 - In this case, the ABI invocation completes with the `NO_MEMORY` return status code, and any previously staged CCA platform attestation claims are cleared from the CCA HES local buffer.
- EL3 firmware cannot complete transmission of CCA platform extension claims due to an `IMPLEMENTATION_DEFINED` reason.
 - In this case, the ABI invocation completes with the `ABORTED` return status code, and the state of any claims that were previously staged is `UNKNOWN`. See also:
 - * [6.3.6 Aborted operations](#).

When an invocation of the `FIRME_ATTEST_PAT_EXT_CLAIMS_STAGE` ABI completes with the `SUCCESS` return status code, the CCA platform extension claims are successfully staged for addition to the CCA platform attestation token.

I₀₂₇₂ The `FIRME_ATTEST_PAT_EXT_CLAIMS_FINALISE` ABI is implemented only at the [Realm FIRME instance](#).
See also:

- [Table 6.4](#).

R₀₂₇₃ An invocation of the `FIRME_ATTEST_PAT_EXT_CLAIMS_FINALISE` ABI successfully completes with the `SUCCESS` return status code, if none of the following are true:

- The ABI is not implemented at this [FIRME instance](#).
 - In this case, the ABI invocation completes with the `NOT_SUPPORTED` return status code.
- The CCA HES interface that adds CCA platform extension claims is busy and the caller must retry the operation.
 - In this case, the ABI invocation completes with the `RETRY` return status code.
- The combination of all staged CCA platform extension claims are not well formed CBOR.
 - In this case, the ABI invocation completes with the `BAD_DATA` return status code, and the staged CCA platform extension claims are cleared from the CCA HES local buffer.

- EL3 firmware cannot validate that the combination of all staged CCA platform extension claims are well formed CBOR due to an IMPLEMENTATION DEFINED reason.
 - In this case, the ABI invocation completes with the **ABORTED** return status code, and the state of any claims that were previously staged is UNKNOWN. See also:
 - * [6.3.6 Aborted operations](#).

When an invocation of the **FIRME_ATTEST_PAT_EXT_CLAIMS_FINALISE** ABI completes with the **SUCCESS** return status code, the CCA HES has validated that the combination of all staged CCA platform extension claims are well formed CBOR, and can be added to the platform attestation token.

I₀₂₇₄

The **FIRME_ATTEST_PAT_EXT_CLAIMS_CLEAR** ABI is implemented only at the **Realm FIRME instance**. See also:

- [Table 6.4](#).

R₀₂₇₅

An invocation of the **FIRME_ATTEST_PAT_EXT_CLAIMS_CLEAR** ABI successfully completes with the **SUCCESS** return status code, if none of the following are true:

- The ABI is not implemented at this **FIRME instance**.
 - In this case, the ABI invocation completes with the **NOT_SUPPORTED** return status code.
- The CCA HES interface that clears CCA platform extension claims is busy and the caller must retry the operation.
 - In this case, the ABI invocation completes with the **RETRY** return status code.
- EL3 firmware cannot complete removal of CCA platform extension claims due to an IMPLEMENTATION DEFINED reason.
 - In this case, the ABI invocation completes with the **ABORTED** return status code, and the state of any claims that were previously added to the platform attestation token is UNKNOWN. See also:
 - * [6.3.6 Aborted operations](#).

When an invocation of the **FIRME_ATTEST_PAT_EXT_CLAIMS_CLEAR** ABI completes with the **SUCCESS** return status code, all of the following are true:

- CCA platform extension claims are removed from the platform attestation token.
- Any CCA platform extension claims that were staged previously are cleared from the CCA HES local buffer.

S₀₂₇₆

[Figure 6.1](#) shows an example flow in which CCA platform extension claims are added to the platform attestation token. This happens in response to a request from the Host to the RMM to update the platform attestation token after all CMEM devices have been added. The RMM uses the FIRME attestation token extension ABIs to extend the platform attestation token in the CCA HES.

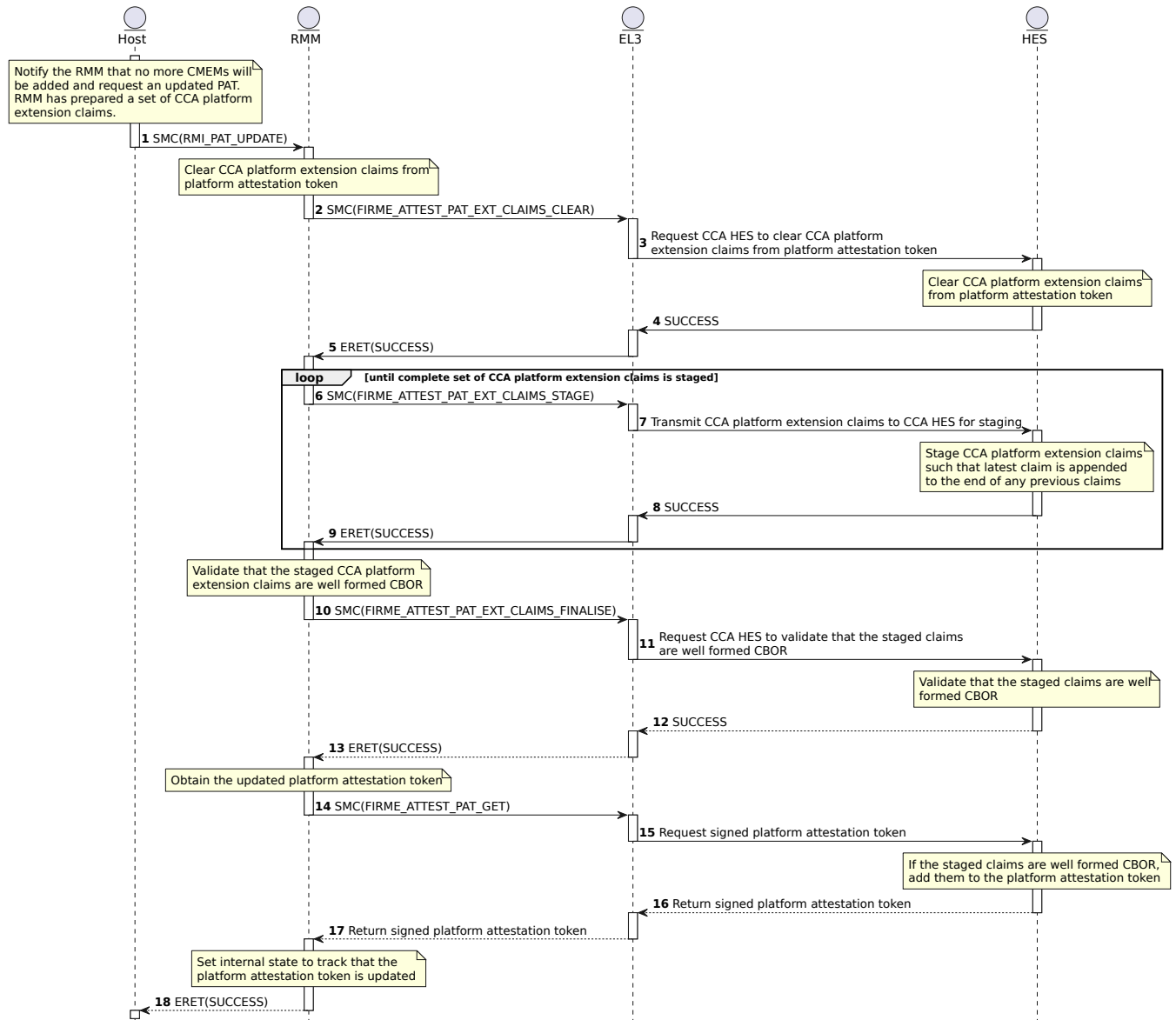


Figure 6.1: Example programming flow to extend the platform attestation token

6.3.6 Aborted operations

I₀₂₇₇

If an invocation of a FIRME attestation service ABI completes with the **ABORTED** error status code, it is IMPLEMENTATION DEFINED whether the next invocation of this ABI will succeed. EL3 firmware can return the **ABORTED** error status code in the following non-exhaustive list of scenarios:

- It is unable to communicate with the CCA HES.
- The CCA HES indicates to EL3 firmware that it cannot complete the requested operation due to an IMPLEMENTATION DEFINED reason.

This could be a temporary scenario and the next request could complete successfully. The caller can try restarting the operation an IMPLEMENTATION DEFINED number of times. If each try completes with the **ABORTED** error status code, it should consider the ABI unavailable.

6.3.7 Feature discovery

I₀₂₇₈

This version of the FIRME attestation service specifies the following feature registers:

- Feature register 0.
- Feature register 1.

See also:

- [2.5 Feature discovery](#).

R₀₂₇₉

The size of each feature register, its index, and its encoding is specified in [Table 6.3](#). All unallocated values are unused.

Table 6.3: FIRME attestation service feature registers

Feature register name	Index	Size	Encoding
Feature register 0	0x1	64-bit	• Bits[63:6]: Unused. See also 2.2 SMC interface. • Bits[5]: Indicates presence of the FIRME_ATTEST_PAT_EXT_CLAIMS_CLEAR ABI. • Bits[4]: Indicates presence of the FIRME_ATTEST_PAT_EXT_CLAIMS_FINALISE ABI. • Bits[3]: Indicates presence of the FIRME_ATTEST_PAT_EXT_CLAIMS_STAGE ABI. • Bits[2]: Indicates presence of the FIRME_ATTEST_RAT_SIGN ABI. • Bits[1]: Indicates presence of the FIRME_ATTEST_RAK_GET ABI. • Bits[0]: Indicates presence of the FIRME_ATTEST_PAT_GET ABI. • Value of each bit is determined from Table 6.4.
Feature register 1	0x0	64-bit	• Bits[63:13]: Unused. See also 2.2 SMC interface. • Bits[12]: RAK_PUB_POR. – Value of this field is used to determine whether retrieval of the public portion of the Realm attestation key is supported by EL3 firmware as follows: * 0b0: Retrieval of the public key portion is not supported. * 0b1: Retrieval of the public key portion is supported.

Feature register name	Index	Size	Encoding
			• Bits[11:10]: RAK_FORMAT. – Value of this field is used to determine how a portion of the Realm attestation key is encoded in the shared buffer in an invocation of the FIRME_ATTEST_RAK_GET ABI as follows: * 0b000: Key portion is encoded in an IMPLEMENTATION DEFINED format. * 0b001: Key portion is encoded as a COSE Key structure [9]. – All other values are unused. See also 2.2 SMC interface. • Bits[9:8]: MAX_PAT_EXT_BUF_PG_CNT. – If platform attestation token extension ABIs are implemented, value of this field is used to determine the maximum size of the local buffer in the CCA HES to stage CCA platform extension claims as follows : * If P = Maximum size of the local buffer in the CCA HES (in KB). * If Q = Minimum size of shared buffer (in KB). · This is derived from the value of the MIN_SH_BUF_SZ field. * Then $P = (MAX_PAT_EXT_BUF_PG_CNT + 1) * Q$. – Otherwise, value of this field is 0 and must be ignored by the caller. • Bits[7:0]: MAX_PAT_PG_CNT. – Value of this field is used to determine the maximum size of the platform attestation token as follows: * If P = Maximum size of the platform attestation token (in KB). * If Q = Minimum size of shared buffer (in KB). · This is derived from the value of the MIN_SH_BUF_SZ field. * Then $P = (MAX_PAT_PG_CNT + 1) * Q$.

6.3.8 Compliance

R0280

The compliance rules for ABIs specified in this version of the FIRME attestation service are listed in [Table 6.4](#). See also:

- [2.6 Compliance](#).

Table 6.4: FIRME attestation service ABIs compliance rules

ABI name	NS FIRME instance	Secure FIRME instance	Realm FIRME instance
FIRME_ATTEST_PAT_GET	Mandatory	Not supported ²	Mandatory
FIRME_ATTEST_RAK_GET	Not supported	Not supported ²	Mandatory
FIRME_ATTEST_RAT_SIGN	Not supported	Not supported ²	Optional
FIRME_ATTEST_PAT_EXT_CLAIMS_STAGE	Not supported	Not supported ²	Optional ¹
FIRME_ATTEST_PAT_EXT_CLAIMS_FINALISE	Not supported	Not supported ²	Optional ¹
FIRME_ATTEST_PAT_EXT_CLAIMS_CLEAR	Not supported	Not supported ²	Optional ¹

¹: Either all platform attestation token extension ABIs are implemented or none are implemented.

²: The [Secure FIRME instance](#) does not implement the FIRME attestation service. See also [Table 2.3](#).

Chapter 7

Integrated device management service

7.1 Overview

I₀₂₈₁

The PCIe [5] specification defines the TEE Device Interface (TDI) Security Protocol (TDISP) that enables a TVM to establish trust in a device, and use TDIs implemented by the device securely. For discrete devices, TDISP messages are exchanged between the TSM in the RMM, and the DSM in the device as Secured CMA-SPDM payloads over DOE. The interconnect between the Host and the device is secured via PCIe IDE.

As per the RME System architecture specification [7], an RCiEP is not required to implement IDE. As per the PCIe specification, an RCiEP is not required to implement Secure CMA-SPDM. The DSM in an RCiEP can be implemented in EL3 firmware. The FIRME integrated device management service is a set of ABIs implemented by EL3 firmware to transport the following messages between the TSM in the RMM, and an RCiEP DSM in EL3 firmware:

- TDISP messages for TDI management.
- IMPLEMENTATION DEFINED messages to obtain device measurements.

7.2 Scope

I₀₂₈₂

In this version of the FIRME specification, the scope of the FIRME integrated device management service is restricted to providing a transport for TDISP messages between the RMM and the DSM for an RCiEP. The service does not interpret the messages and only provides a mechanism for transmitting them via shared buffers and general purpose registers with non-blocking semantics.

7.3 Concepts

I₀₂₈₃ The FIRME integrated device management service is implemented only at the [Realm FIRME instance](#). See also:

- [Table 7.7](#).

I₀₂₈₄ The FIRME integrated device management service specifies the [FIRME_IDEV_OP_START](#) ABI to start a TDISP operation, or an operation to obtain device measurements. The [FIRME_IDEV_OP_CONTINUE](#) ABI is used to resume an incomplete TDISP operation, or an incomplete operation to obtain device measurements.

I₀₂₈₅ In an invocation of the [FIRME_IDEV_OP_START](#) ABI, if the *Protocol type* field in the *Flags* input parameter is set to 0b01, the encoding of the request and response message to obtain device measurements is IMPLEMENTATION DEFINED.

I₀₂₈₆ Protocol messages of ≤ 120 bytes in size can be encoded either in a [shared buffer](#), or in input or output parameter registers from x3-x17. If the size of a message is > 120 bytes, it is encoded in a [shared buffer](#).

The choice of storage is specified via the *Storage type* flag in the *Flags* input parameter of a FIRME integrated device management ABI as follows:

- If *Storage type* == 0 in an invocation of the [FIRME_IDEV_OP_START](#) ABI, the protocol request and response messages are encoded in a [shared buffer](#).
- If *Storage type* == 1 in an invocation of the [FIRME_IDEV_OP_START](#) ABI, the protocol request and response messages are encoded in input and output parameter registers respectively.
- If *Storage type* == 0 in an invocation of the [FIRME_IDEV_OP_CONTINUE](#) ABI, the protocol response messages are encoded in a [shared buffer](#).
- If *Storage type* == 1 in an invocation of the [FIRME_IDEV_OP_CONTINUE](#) ABI, the protocol response messages are encoded in output parameter registers.

Additionally, if a shared buffer is used, these ABIs support [partial retrieval](#) to receive output data in chunks.

See also:

- [2.8 Shared buffer](#).
- [2.8.1 Data retrieval in chunks](#).

R₀₂₈₇ When *Storage type* = 1 in the *Flags* input parameter of a FIRME integrated device management service ABI, Byte[N] of a protocol message is populated in register x(P) where all of the following are true:

- $N = 0 - 119$.
- $P = (N \div 8) + 3$.

In register x(P), Byte[Q] = Byte[N] of the protocol message where $Q = N \bmod 8$.

S₀₂₈₈ A protocol message of size 28 bytes is encoded in parameter registers as follows:

- x3
 - RegisterByte[0] = MessageByte[0]
 - RegisterByte[1] = MessageByte[1]
 - RegisterByte[2] = MessageByte[2]
 - RegisterByte[3] = MessageByte[3]
 - RegisterByte[4] = MessageByte[4]
 - RegisterByte[5] = MessageByte[5]
 - RegisterByte[6] = MessageByte[6]
 - RegisterByte[7] = MessageByte[7]
- x4
 - RegisterByte[0] = MessageByte[8]
 - RegisterByte[1] = MessageByte[9]
 - RegisterByte[2] = MessageByte[10]
 - RegisterByte[3] = MessageByte[11]
 - RegisterByte[4] = MessageByte[12]
 - RegisterByte[5] = MessageByte[13]
 - RegisterByte[6] = MessageByte[14]
 - RegisterByte[7] = MessageByte[15]

- x5
 - RegisterByte[0] = MessageByte[16]
 - RegisterByte[1] = MessageByte[17]
 - RegisterByte[2] = MessageByte[18]
 - RegisterByte[3] = MessageByte[19]
 - RegisterByte[4] = MessageByte[20]
 - RegisterByte[5] = MessageByte[21]
 - RegisterByte[6] = MessageByte[22]
 - RegisterByte[7] = MessageByte[23]
- x6
 - RegisterByte[0] = MessageByte[24]
 - RegisterByte[1] = MessageByte[25]
 - RegisterByte[2] = MessageByte[26]
 - RegisterByte[3] = MessageByte[27]

R₀₂₈₉ When *Storage type* = 1 in the *Flags* input parameter of a FIRME integrated device management service ABI, [Table 7.1](#) describes encoding of protocol messages in parameter registers.

Table 7.1: Parameter description when *Storage type* = 1 in the *Flags* input parameter

Parameter register	Parameter name	Parameter description
x3-x17	uint64 Protocol message.	• Encoding of protocol request and response messages in 64-bit registers.

R₀₂₉₀ If the [shared buffer](#) is used in the [FIRME_IDEV_OP_START](#) ABI, the protocol request message is populated at offset 0x0 in the buffer. The size of the message (in bytes) is populated in the *Input message size* field of the *Request attributes* input parameter.

R₀₂₉₁ When an invocation of a FIRME integrated device management ABI that uses a [shared buffer](#) completes with the [SUCCESS](#) return status code, all of the following are true:

- *Written size* > 0.
- *Remaining size* == 0.

R₀₂₉₂ When an invocation of a FIRME integrated device management ABI that uses a [shared buffer](#) completes with the [INCOMPLETE](#) return status code, [Table 7.2](#) lists all valid combinations of the following output parameters.

- *Written size*.
- *Remaining size*.

Table 7.2: Parameter description when INCOMPLETE is returned

Written size	Remaining size	Description
0	0	The DSM has not generated a protocol response message. No chunk of the message was written to the shared buffer. The FIRME_IDEV_OP_CONTINUE ABI must be invoked to retrieve the response message.
0	Non-zero	No chunk of the response message was written to the shared buffer. The FIRME_IDEV_OP_CONTINUE ABI must be invoked again to retrieve the response message.

Written size	Remaining size	Description
Non-zero	Non-zero	A chunk of the protocol response message has been written to the shared buffer. The FIRME_IDEV_OP_CONTINUE ABI must be invoked again to retrieve the response message.

- I₀₂₉₃ If a [shared buffer](#) is used in a FIRME integrated device management ABI, it is possible that the DSM is able to receive a protocol request message to start retrieval of output data, but the DSM is unable to send a protocol response message immediately. In this scenario, EL3 firmware returns a zero *Written size* and a zero *Remaining size* of the output data. The caller invokes the ABI as many times as it is required by EL3 firmware to obtain the entire output data from the DSM and return it to the caller.
- I₀₂₉₄ When a FIRME integrated device management ABI that supports [partial retrieval](#) via a [shared buffer](#), completes a request to retrieve output data with the [SUCCESS](#) return status code, if the *Written size* is zero, the *Remaining size* of the output data can be non-zero. This could happen in a scenario where the DSM is unable to write a new chunk of the output data due to an IMPLEMENTATION DEFINED reason.
- R₀₂₉₅ An invocation of the [FIRME_IDEV_OP_START](#) ABI, or the [FIRME_IDEV_OP_CONTINUE](#) ABI, successfully completes with the [SUCCESS](#) return status code, if none of the following are true:
- The ABI is not implemented at this [FIRME instance](#).
 - In this case, the ABI invocation completes with the [NOT_SUPPORTED](#) return status code.
 - The physical base address of the [shared buffer](#) is invalid.
 - In this case, the ABI invocation completes with the [INVALID_PARAMETERS](#) return status code.
 - The *Write offset* from the base of the [shared buffer](#) is invalid.
 - In this case, the ABI invocation completes with the [INVALID_PARAMETERS](#) return status code.
 - The *Shared buffer size* is invalid.
 - In this case, the ABI invocation completes with the [INVALID_PARAMETERS](#) return status code.
 - The *Flags* input parameter is invalid.
 - In this case, the ABI invocation completes with the [INVALID_PARAMETERS](#) return status code.
 - If the [FIRME_IDEV_OP_START](#) ABI is invoked, the *Input message size* field in the *Request attributes* input parameter is invalid.
 - In this case, the ABI invocation completes with the [INVALID_PARAMETERS](#) return status code.
 - If the [FIRME_IDEV_OP_CONTINUE](#) ABI is invoked, the *Cookie* input parameter is invalid.
 - In this case, the ABI invocation completes with the [INVALID_PARAMETERS](#) return status code.
 - *Storage type* = 0 in the *Flags* input parameter, and EL3 firmware could not allocate translation tables to map the [Shared buffer](#) in its translation regime.
 - In this case, the ABI invocation completes with the [NO_MEMORY](#) return status code.
 - The FIRME integrated device management service is busy and the caller must retry the operation.
 - In this case, the ABI invocation completes with the [RETRY](#) return status code.
 - The caller is not allowed to invoke the requested operation.
 - In this case, the ABI invocation completes with the [DENIED](#) return status code and has no effect.
 - The requested operation was paused due to an IMPLEMENTATION DEFINED reason and the caller must resume the operation.
 - In this case, the ABI invocation completes with the [INCOMPLETE](#) return status code.
- R₀₂₉₆ The output parameters of the [FIRME_IDEV_OP_START](#) and [FIRME_IDEV_OP_CONTINUE](#) interfaces are listed in [Table 7.3](#).

Table 7.3: FIRME_IDEV_OP_START and FIRME_IDEV_OP_CONTINUE output parameters

Parameter	Register	Value
int32 Return status	w0	• SUCCESS • NOT_SUPPORTED • INVALID_PARAMETERS • RETRY • DENIED
Output parameter register	x1	• Unused. See also 2.2 SMC interface.
uint64 Cookie	x2	• If <i>Return status</i> is INCOMPLETE, this is a cookie returned by the callee. The caller passes the cookie to the callee in an invocation of the FIRME_IDEV_OP_CONTINUE ABI. • Otherwise, this register is Unused. See also 2.2 SMC interface.
Output parameter register	x3-x17	• If <i>Storage type</i> is 0, see Table 7.5 for a description of these parameters. • Otherwise, see Table 7.1 for a description of these parameters.

R0297

Table 7.4: Input parameter description when *Storage type* = 0 in the *Flags* input parameter

Input parameter register	Parameter name	Parameter description
x3	uint64 Shared buffer base address.	• 64-bit physical base address of the physically contiguous buffer for transmitting a protocol request and response messages.
x4	uint64 Write offset.	• Offset (in bytes) from the base of <i>Shared buffer base address</i> at which the protocol response message must be written by EL3 firmware.
x5	uint64 Shared buffer page count.	• Bits[13:0]: Value of this field is used to determine the size of the buffer as follows: – If P = Size of buffer (in KB). – If Q = Minimum size of a shared buffer (in KB). * This is derived from the value of the <code>MIN_SH_BUF_SZ</code> field in feature register 1 of the FIRME base service. – Then $P = (\text{Shared buffer page count} + 1) * Q.$ • Bits[63:14]: Unused. See also 2.2 SMC interface.
x6	uint64 Request attributes	• In an invocation of the FIRME_IDEV_OP_START ABI, this parameter is encoded as follows: – Bits[31:0]: Input Message Size * Size (in bytes) of the request message. – Bits[63:32]: Unused. • Otherwise, this parameter is unused. See also 2.2 SMC interface.

Input parameter register	Parameter name	Parameter description
x7-x17	Unused. See also 2.2 SMC interface .	• NA

R₀₂₉₈

Table 7.5: Output parameter description when *Storage type* = 0 in the *Flags* input parameter

Output parameter register	Parameter name	Parameter description
x3	uint64 Written size	• If <i>Return status</i> is SUCCESS, this is the size (in bytes) of the chunk of the protocol response message starting from the <i>Write offset</i> input parameter. • Otherwise, this register is unused. See also 2.2 SMC interface.
x4	uint64 Remaining size	• If <i>Return status</i> is SUCCESS, this is the remaining size (in bytes) of the protocol response message that could not be written to the buffer. • Otherwise, this register is unused. See also 2.2 SMC interface.
x5-x17	Unused. See also 2.2 SMC interface .	• NA

7.3.1 Feature discovery

I₀₂₉₉

This version of the FIRME integrated device management service specifies the following feature registers:

- Feature register 0.

See also:

- [2.5 Feature discovery](#).

R₀₃₀₀

The size of each feature register, its index, and its encoding is specified in [Table 7.6](#). All unallocated values are unused.

Table 7.6: FIRME IDE key management service feature registers

Feature register name	Index	Size	Encoding
Feature register 0	0x1	64-bit	• Bits[63:2]: Unused. See also 2.2 SMC interface. • Bits[1]: Indicates presence of the FIRME_IDEV_OP_CONTINUE ABI. • Bits[0]: Indicates presence of the FIRME_IDEV_OP_START ABI. • Value of each bit is determined from Table 7.7.

7.3.2 Compliance

R0301

The compliance rules for ABIs specified in this version of the FIRME IDE key management service are listed in [Table 7.7](#). See also:

- [2.6 Compliance](#).

Table 7.7: FIRME Integrated device management service ABIs compliance rules

ABI name	NS FIRME instance	Secure FIRME instance	Realm FIRME instance
FIRME_IDEV_OP_START	Not supported ¹	Not supported ¹	Mandatory
FIRME_IDEV_OP_CONTINUE	Not supported ¹	Not supported ¹	Mandatory

¹: The [Secure FIRME instance](#) and the [NS FIRME instance](#) do not implement the FIRME integrated device management service. See also [Table 2.3](#).

Chapter 8

Interface

8.1 FIRME_SERVICE_VERSION

R₀₃₀₂ The input parameters of the FIRME_SERVICE_VERSION interface are listed in [Table 8.1](#).

Table 8.1: FIRME_SERVICE_VERSION input parameters

Parameter	Register	Value
uint32 Function ID	w0	0xC4000400.
uint64 Service ID	x1	- Bits[63:8]: Unused. See also 2.2 SMC interface. - Bits[7:0]: ID of the FIRME service whose version must be returned to the caller. See also Table 2.2.

R₀₃₀₃ The output parameters of the FIRME_SERVICE_VERSION interface are listed in [Table 8.2](#).

Table 8.2: FIRME_SERVICE_VERSION output parameters

Parameter	Register	Value
int32 Return status	w0	- A positive value encodes the FIRME service version number as follows: - Bits[31]: Used to differentiate between a positive and a negative integer. - Bits[30:16]: Major version. - Bits[15:0]: Minor version. - See also Table 2.3. - A negative value encodes one of the following return status codes: NOT_SUPPORTED

8.2 FIRME_SERVICE_FEATURES

R₀₃₀₄ The input parameters of the FIRME_SERVICE_FEATURES interface are listed in [Table 8.3](#).

Table 8.3: FIRME_SERVICE_FEATURES input parameters

Parameter	Register	Value
uint32 Function ID	w0	0xC4000401.
uint64 Service ID	x1	- Bits[7:0]: ID of the FIRME service whose features must be returned to the caller. See also Table 2.2. - Bits[63:8]: Unused. See also 2.2 SMC interface.
uint64 Feature register index	x2	- Bits[7:0]: Index of the feature register to be returned to the caller. See also 2.5 Feature discovery. - Bits[63:1]: Unused. See also 2.2 SMC interface.

R₀₃₀₅ The output parameters of the FIRME_SERVICE_FEATURES interface are listed in [Table 8.4](#).

Table 8.4: FIRME_SERVICE_FEATURES output parameters

Parameter	Register	Value
int32 Return status	w0	SUCCESS NOT_SUPPORTED
uint64 Feature register value	x1	- If the <i>Return status</i> is SUCCESS, this is the value of the feature register corresponding to the <i>FIRME service ID</i> and <i>Feature register index</i> input parameters. - Otherwise, this register is unused. See also 2.2 SMC interface.

8.3 FIRME_GM_GPI_SET

R₀₃₀₆ The input parameters of the FIRME_GM_GPI_SET interface are listed in [Table 8.5](#).

Table 8.5: FIRME_GM_GPI_SET input parameters

Parameter	Register	Value
uint32 Function ID	w0	0xC4000402.
uint64 Base address	x1	64-bit physical base address of the physically contiguous granules whose GPI encoding must be changed.
uint64 Granule count	x2	Number of physically contiguous granules starting from the <i>Base Address</i>.
uint32 Attributes	x3	- Attributes of this request. - Bits[63:4]: Unused. See also 2.2 SMC interface. - Bits[3:0]: Target GPI encoding of the specified granules.

R₀₃₀₇ The output parameters of the FIRME_GM_GPI_SET interface are listed in [Table 8.6](#).

Table 8.6: FIRME_GM_GPI_SET output parameters

Parameter	Register	Value
int32 Return status	w0	SUCCESS NOT_SUPPORTED INVALID_PARAMETERS DENIED RETRY INCOMPLETE IN_PROGRESS
uint64 Granule count	x1	Count of granules starting from the first granule at the <i>Base Address</i> whose GPI encoding was changed.
uint64 Cookie	x2	- If <i>Return status</i> is INCOMPLETE, this is a cookie returned by the callee. The caller passes the cookie to the callee in an invocation of the FIRME_GM_GPI_OP_CONTINUE ABI. - Otherwise, this register is unused. See also 2.2 SMC interface.

8.4 FIRME_GM_GPI_OP_CONTINUE

R0308 The input parameters of the FIRME_GM_GPI_OP_CONTINUE interface are listed in [Table 8.7](#).

Table 8.7: FIRME_GM_GPI_OP_CONTINUE input parameters

Parameter	Register	Value
uint32 Function ID	w0	0xC4000412.
uint64 Cookie	x1	An IMPLEMENTATION DEFINED value returned by the callee in a previous invocation of a FIRME Granule management interface to identify, and track progress of an incomplete operation. The caller passes this value to the callee to resume the incomplete operation.

R0309 The output parameters of the FIRME_GM_GPI_OP_CONTINUE interface are listed in [Table 8.8](#).

Table 8.8: FIRME_GM_GPI_OP_CONTINUE output parameters

Parameter	Register	Value
int32 Return status	w0	SUCCESS NOT_SUPPORTED INVALID_PARAMETERS DENIED RETRY INCOMPLETE IN_PROGRESS
uint64 Granule count	x1	Count of granules whose GPI encoding was changed in this invocation of this ABI.
uint64 Cookie	x2	- If <i>Return status</i> is INCOMPLETE, this is a cookie returned by the callee. The caller passes the cookie to the callee in an invocation of the FIRME_GM_GPI_OP_CONTINUE ABI. - Otherwise, this register is unused. See also 2.2 SMC interface.

8.5 FIRME_GM_L1_GPT_CREATE

R₀₃₁₀ The input parameters of the FIRME_GM_L1_GPT_CREATE interface are listed in [Table 8.9](#).

Table 8.9: FIRME_GM_L1_GPT_CREATE input parameters

Parameter	Register	Value
uint32 Function ID	w0	0xC400040E.
uint64 PA range base	x1	64-bit physical base address of a protected address range.
uint64 L1 GPT base	x2	64-bit physical base address of the L1 GPT for the specified protected address range.

R₀₃₁₁ The output parameters of the FIRME_GM_L1_GPT_CREATE interface are listed in [Table 8.10](#).

Table 8.10: FIRME_GM_L1_GPT_CREATE output parameters

Parameter	Register	Value
int32 Return status	w0	SUCCESS NOT_SUPPORTED INVALID_PARAMETERS DENIED RETRY INCOMPLETE IN_PROGRESS NO_ENTRY EXISTS

8.6 FIRME_GM_L1_GPT_DESTROY

R₀₃₁₂ The input parameters of the FIRME_GM_L1_GPT_DESTROY interface are listed in [Table 8.11](#).

Table 8.11: FIRME_GM_L1_GPT_DESTROY input parameters

Parameter	Register	Value
uint32 Function ID	w0	0xC400040F.
uint64 PA range base	x1	64-bit physical base address of a protected address range.

R₀₃₁₃ The output parameters of the FIRME_GM_L1_GPT_DESTROY interface are listed in [Table 8.12](#).

Table 8.12: FIRME_GM_L1_GPT_DESTROY output parameters

Parameter	Register	Value
int32 Return status	w0	SUCCESS NOT_SUPPORTED INVALID_PARAMETERS DENIED RETRY INCOMPLETE IN_PROGRESS NO_ENTRY
uint64 L1 GPT base	x2	64-bit physical base address of the L1 GPT for the specified protected address range.

8.7 FIRME_IDE_KEYSET_PROG

R0314 The input parameters of the FIRME_IDE_KEYSET_PROG interface are listed in [Table 8.13](#).

Table 8.13: FIRME_IDE_KEYSET_PROG input parameters

Parameter	Register	Value
uint32 Function ID	w0	0xC4000403.
uint64 ECAM address	x1	64-bit physical base address of the ECAM address space of the target Root port.
uint64 Input flags	x2	- Bits[1:0]: IDE type. 0b00: Request to configure a PCIe or CXL.io Selective IDE stream. 0b11: Request to configure a CXL.cachemem Link IDE stream. - All other values are unused. - Bits[63:2]: Unused. See also 2.2 SMC interface.
uint64 Keyset ID	x3	Keyset ID of the keyset in the Root port.
uint64 KeyQW0 - KeyQW3	x4 - x7	- For $m = 0 - 3$, <i>Quad word</i>[64m+63:64m] of the AES-GCM 256 bit key is encoded as follows: $KeyQWm[8n+7:8n] = Quad\ word[(8n+64m)+7:8n+64m]$, where $n = 0 - 7$.
uint64 Cookie1	x8	- Implementation defined non-zero value specified by the caller to track a non-blocking IDE key management operation. - A value of 0 indicates this parameter is not used by the caller.
uint64 Cookie2	x9	- Implementation defined non-zero value specified by the caller to track a non-blocking IDE key management operation. - A value of 0 indicates this parameter is not used by the caller.

R0315 The output parameters of the FIRME_IDE_KEYSET_PROG interface are listed in [Table 8.14](#).

Table 8.14: FIRME_IDE_KEYSET_PROG output parameters

Parameter	Register	Value
int32 Return status	w0	- SUCCESS - NOT_SUPPORTED - INVALID_PARAMETERS - RETRY - IN_PROGRESS - INCOMPLETE

8.8 FIRME_IDE_KEYSET_GO

R₀₃₁₆ The input parameters of the FIRME_IDE_KEYSET_GO interface are listed in [Table 8.15](#).

Table 8.15: FIRME_IDE_KEYSET_GO input parameters

Parameter	Register	Value
uint32 Function ID	w0	0xC4000404.
uint64 ECAM address	x1	64-bit physical base address of the ECAM address space of the target Root port.
uint64 Input flags	x2	- Bits[1:0]: IDE type. 0b00: Request to configure a PCIe or CXL.io Selective IDE stream. 0b11: Request to configure a CXL.cachemem Link IDE stream. - All other values are unused. - Bits[63:2]: Unused. See also 2.2 SMC interface.
uint64 Keyset ID	x3	Keyset ID of the keyset in the Root port.
uint64 Cookie1	x4	- Implementation defined non-zero value specified by the caller to track a non-blocking IDE key configuration operation. - A value of 0 indicates this parameter is not used by the caller.
uint64 Cookie2	x5	- Implementation defined non-zero value specified by the caller to track a non-blocking IDE key configuration operation. - A value of 0 indicates this parameter is not used by the caller.

R₀₃₁₇ The output parameters of the FIRME_IDE_KEYSET_GO interface are listed in [Table 8.16](#).

Table 8.16: FIRME_IDE_KEYSET_GO output parameters

Parameter	Register	Value
int32 Return status	w0	SUCCESS NOT_SUPPORTED INVALID_PARAMETERS RETRY IN_PROGRESS DENIED INCOMPLETE

8.9 FIRME_IDE_KEYSET_STOP

R₀₃₁₈ The input parameters of the FIRME_IDE_KEYSET_STOP interface are listed in [Table 8.17](#).

Table 8.17: FIRME_IDE_KEYSET_STOP input parameters

Parameter	Register	Value
uint32 Function ID	w0	0xC4000405.
uint64 ECAM address	x1	64-bit physical base address of the ECAM address space of the target Root port.
uint64 Input flags	x2	- Bits[1:0]: IDE type. 0b00: Request to configure a PCIe or CXL.io Selective IDE stream. 0b11: Request to configure a CXL.cachemem Link IDE stream. - All other values are unused. - Bits[63:2]: Unused. See also 2.2 SMC interface.
uint64 Keyset ID	x3	Keyset ID of the keyset in the Root port.
uint64 Cookie1	x4	- Implementation defined non-zero value specified by the caller to track a non-blocking IDE key configuration operation. - A value of 0 indicates this parameter is not used by the caller.
uint64 Cookie2	x5	- Implementation defined non-zero value specified by the caller to track a non-blocking IDE key configuration operation. - A value of 0 indicates this parameter is not used by the caller.

R₀₃₁₉ The output parameters of the FIRME_IDE_KEYSET_STOP interface are listed in [Table 8.18](#).

Table 8.18: FIRME_IDE_KEYSET_STOP output parameters

Parameter	Register	Value
int32 Return status	w0	- SUCCESS - NOT_SUPPORTED - INVALID_PARAMETERS - RETRY - IN_PROGRESS - DENIED - INCOMPLETE

8.10 FIRME_IDE_KEYSET_POLL

R₀₃₂₀ The input parameters of the FIRME_IDE_KEYSET_POLL interface are listed in [Table 8.19](#).

Table 8.19: FIRME_IDE_KEYSET_POLL input parameters

Parameter	Register	Value
uint32 Function ID	w0	0xC4000406.
uint64 ECAM address	x1	64-bit physical base address of the ECAM address space of the target Root port.
uint64 Input flags	x2	- Bits[1:0]: IDE type. 0b00: Request to configure a PCIe or CXL.io Selective IDE stream. 0b11: Request to configure a CXL.cachemem Link IDE stream. - All other values are unused. - Bits[2]: Pending response type. 0b0: Return a pending response corresponding to the <i>Keyset ID</i> parameter. 0b1: Return any pending response and ignore the <i>Keyset ID</i> parameter. - Bits[63:3]: Unused. See also 2.2 SMC interface.
uint64 Keyset ID	x3	Keyset ID of the keyset in the Root port.

R₀₃₂₁ The output parameters of the FIRME_IDE_KEYSET_POLL interface are listed in [Table 8.20](#).

Table 8.20: FIRME_IDE_KEYSET_POLL output parameters

Parameter	Register	Value
int32 Return status	w0	SUCCESS NOT_SUPPORTED INVALID_PARAMETERS RETRY INCOMPLETE ABORTED DENIED
uint64 Cookie1	x1	- Implementation defined non-zero value that was specified by the caller to track a non-blocking IDE key configuration operation. - The value 0 is returned if this input parameter was not used by the caller. - This register is unused if <i>Return status</i> is not SUCCESS. See also 2.2 SMC interface.

Parameter	Register	Value
uint64 Cookie2	x2	• Implementation defined non-zero value that was specified by the caller to track a non-blocking IDE key configuration operation. • The value 0 is returned if this input parameter was not used by the caller. • This register is unused if <i>Return status</i> is not SUCCESS. See also 2.2 SMC interface.

8.11 FIRME_MEC_REFRESH

R₀₃₂₂ The input parameters of the FIRME_MEC_REFRESH interface are listed in [Table 8.21](#).

Table 8.21: FIRME_MEC_REFRESH input parameters

Parameter	Register	Value
uint32 Function ID	w0	• 0xC4000407.
uint64 MEC params	x1	• Bits[63:48]: Unused. See also 2.2 SMC interface. • Bits[47:32]: MECID. • Bits[31:1]: Unused. See also 2.2 SMC interface. • Bits[0]: MEC refresh reason. – 0b'0: Realm creation. – 0b'1: Realm destruction.

R₀₃₂₃ The output parameters of the FIRME_MEC_REFRESH interface are listed in [Table 8.22](#).

Table 8.22: FIRME_MEC_REFRESH output parameters

Parameter	Register	Value
int32 Return status	w0	• SUCCESS • NOT_SUPPORTED • INVALID_PARAMETERS • RETRY

8.12 FIRME_ATTEST_PAT_GET

R₀₃₂₄

The input parameters of the FIRME_ATTEST_PAT_GET interface are listed in [Table 8.23](#).

Table 8.23: FIRME_ATTEST_PAT_GET input parameters

Parameter	Register	Value
uint32 Function ID	w0	0xC4000408.
uint64 Shared buffer base address	x1	64-bit physical base address of the physically contiguous buffer for transmitting input and output parameters.
uint64 Write offset	x2	Offset (in bytes) from the base of <i>Shared buffer base address</i> at which the platform attestation token must be written by EL3 firmware.
uint64 Shared buffer page count	x3	- Bits[13:0]: Value of this field is used to determine the size of the buffer as follows: - If P = Size of buffer (in KB). - If Q = Minimum size of a shared buffer (in KB). * This is derived from the value of the MIN_SH_BUF_SZ field in FIRME feature register 1. - Then $P = (\text{Shared buffer page count} + 1) * Q$. - Bits[63:14]: Unused. See also 2.2 SMC interface.
uint64 Platform challenge size	x4	- Bits[31:0]: Size (in bytes) of the challenge populated at offset 0x0 from <i>Shared buffer base address</i>. - Bits[63:32]: Unused. See also 2.2 SMC interface.

R₀₃₂₅

The output parameters of the FIRME_ATTEST_PAT_GET interface are listed in [Table 8.24](#).

Table 8.24: FIRME_ATTEST_PAT_GET output parameters

Parameter	Register	Value
int32 Return status	w0	SUCCESS NOT_SUPPORTED INVALID_PARAMETERS RETRY ABORTED INCOMPLETE
uint64 Written size	x1	If <i>Return status</i> is SUCCESS, this is the size (in bytes) of the chunk of the platform attestation token starting from the <i>Write offset</i> input parameter.
uint64 Remaining size	x2	- If <i>Return status</i> is SUCCESS, this is the remaining size (in bytes) of the platform attestation token that could not be written to the buffer. - Otherwise, this register is unused. See also 2.2 SMC interface.

8.13 FIRME_ATTEST_RAK_GET

R0326 The input parameters of the FIRME_ATTEST_RAK_GET interface are listed in [Table 8.25](#).

Table 8.25: FIRME_ATTEST_RAK_GET input parameters

Parameter	Register	Value
uint32 Function ID	w0	0xC4000409.
uint64 Shared buffer base address	x1	64-bit physical base address of the physically contiguous buffer for transmitting input and output parameters.
uint64 Write offset	x2	Offset (in bytes) from the base of <i>Shared buffer base address</i> at which the Realm attestation key must be written by EL3 firmware.
uint64 Shared buffer page count	x3	- Bits[13:0]: Value of this field is used to determine the size of the buffer as follows: - If P = Size of buffer (in KB). - If Q = Minimum size of a shared buffer (in KB). * This is derived from the value of the <code>MIN_SH_BUF_SZ</code> field in FIRME feature register 1. - Then $P = (\text{Shared buffer page count} + 1) * Q$. - Bits[63:14]: Unused. See also 2.2 SMC interface.
uint64 Input flags	x4	- Bits[0]: Request type. 0b0: This is a request to initiate retrieval of the RAK. 0b1: This is a request to continue retrieval of the RAK. - Bits[1]: Retrieve the public key portion of the Realm attestation key. 0b0: Do not retrieve the public key portion. 0b1: Retrieve the public key portion. - Unused if the <code>Request type</code> field is 1. - Bits[2]: Retrieve the private key portion of the Realm attestation key. 0b0: Do not retrieve the private key portion. 0b1: Retrieve the private key portion. - Unused if the <code>Request type</code> field is 1. - Bits[7:3]: Unused. - Bits[15:8]: Type of elliptic curve. 0x0: ECC SECP384R1. - All other values are unused. - Bits[63:16]: Unused. See also 2.2 SMC interface.

R0327 The output parameters of the FIRME_ATTEST_RAK_GET interface are listed in [Table 8.26](#).

Table 8.26: FIRME_ATTEST_RAK_GET output parameters

Parameter	Register	Value
int32 Return status	w0	• SUCCESS • NOT_SUPPORTED • INVALID_PARAMETERS • RETRY • ABORTED • INCOMPLETE
uint64 Written size	x1	• If <i>Return status</i> is SUCCESS, this is the size (in bytes) of the chunk of the Realm attestation key starting from the <i>Write offset</i> input parameter. • Otherwise, this register is unused. See also 2.2 SMC interface.
uint64 Remaining size	x2	• If <i>Return status</i> is SUCCESS, this is the remaining size (in bytes) of the Realm attestation key that could not be written to the buffer. • Otherwise, this register is unused. See also 2.2 SMC interface.

8.14 FIRME_ATTEST_RAT_SIGN

R₀₃₂₈ The input parameters of the FIRME_ATTEST_RAT_SIGN interface are listed in [Table 8.27](#).

Table 8.27: FIRME_ATTEST_RAT_SIGN input parameters

Parameter	Register	Value
uint32 Function ID	w0	0xC400040A.
uint64 Shared buffer base address	x1	64-bit physical base address of the physically contiguous buffer for transmitting input and output parameters.
uint64 Shared buffer page count	x2	- Bits[13:0]: Value of this field is used to determine the size of the buffer as follows: - If P = Size of buffer (in KB). - If Q = Minimum size of a shared buffer (in KB). * This is derived from the value of the MIN_SH_BUF_SZ field in FIRME feature register 1. - Then $P = (\text{Shared buffer page count} + 1) * Q$. - Bits[63:14]: Unused. See also 2.2 SMC interface.
uint64 Request attributes	x3	- Bits[0]: Request type. 0b0: This is a request to sign a Realm attestation token. 0b1: This is a request to retrieve the signature of a Realm attestation token. - Bits[31:1]: Unused. See also 2.2 SMC interface. - Bits[63:32]: Input Payload size. - If this is a request to sign a Realm attestation token, this is the size (in bytes) of the payload containing the Realm attestation token. - Otherwise, this register is unused. See also 2.2 SMC interface.

R₀₃₂₉ The output parameters of the FIRME_ATTEST_RAT_SIGN interface are listed in [Table 8.28](#).

Table 8.28: FIRME_ATTEST_RAT_SIGN output parameters

Parameter	Register	Value
int32 Return status	w0	SUCCESS NOT_SUPPORTED INVALID_PARAMETERS RETRY ABORTED DENIED

Parameter	Register	Value
uint64 Output payload size	x1	- If this was a request to retrieve the signature of a Realm attestation token, a value of 0 indicates there is no payload in the shared buffer. Otherwise, one of the following is true: - If <i>Return status</i> is SUCCESS, this is the size (in bytes) of the payload containing the signature. - If <i>Return status</i> is not SUCCESS, and there is an additional response payload for the caller, this is the size (in bytes) of the payload. - Otherwise, this register is unused. See also 2.2 SMC interface.

8.15 FIRME_ATTEST_PAT_EXT_CLAIMS_STAGE

R0330

The input parameters of the FIRME_ATTEST_PAT_EXT_CLAIMS_STAGE interface are listed in [Table 8.29](#).

Table 8.29: FIRME_ATTEST_PAT_EXT_CLAIMS_STAGE input parameters

Parameter	Register	Value
uint32 Function ID	w0	0xC400040B.
uint64 Shared buffer base address	x1	64-bit physical base address of the physically contiguous buffer for transmitting input and output parameters.
uint64 Shared buffer page count	x2	- Bits[13:0]: Value of this field is used to determine the size of the buffer as follows: - If P = Size of buffer (in KB). - If Q = Minimum size of a shared buffer (in KB). * This is derived from the value of the MIN_SH_BUF_SZ field in FIRME feature register 1. - Then $P = (\text{Shared buffer page count} + 1) * Q$. - Bits[63:14]: Unused. See also 2.2 SMC interface.
uint64 Request attributes	x3	- Bits[31:0]: Unused. See also 2.2 SMC interface. - Bits[63:32]: Input Payload size. - Size (in bytes) of the payload containing a set of CCA platform extension claims.

R0331

The output parameters of the FIRME_ATTEST_PAT_EXT_CLAIMS_STAGE interface are listed in [Table 8.30](#).

Table 8.30: FIRME_ATTEST_PAT_EXT_CLAIMS_STAGE output parameters

Parameter	Register	Value
int32 Return status	w0	SUCCESS NOT_SUPPORTED INVALID_PARAMETERS RETRY ABORTED

8.16 FIRME_ATTEST_PAT_EXT_CLAIMS_CLEAR

R₀₃₃₂

The input parameters of the FIRME_ATTEST_PAT_EXT_CLAIMS_CLEAR interface are listed in [Table 8.31](#).

Table 8.31: FIRME_ATTEST_PAT_EXT_CLAIMS_CLEAR input parameters

Parameter	Register	Value
uint32 Function ID	w0	• 0xC400040C.

R₀₃₃₃

The output parameters of the FIRME_ATTEST_PAT_EXT_CLAIMS_CLEAR interface are listed in [Table 8.32](#).

Table 8.32: FIRME_ATTEST_PAT_EXT_CLAIMS_CLEAR output parameters

Parameter	Register	Value
int32 Return status	w0	• SUCCESS • NOT_SUPPORTED • RETRY • ABORTED

8.17 FIRME_ATTEST_PAT_EXT_CLAIMS_FINALISE

R0334

The input parameters of the FIRME_ATTEST_PAT_EXT_CLAIMS_FINALISE interface are listed in [Table 8.33](#).

Table 8.33: FIRME_ATTEST_PAT_EXT_CLAIMS_FINALISE input parameters

Parameter	Register	Value
uint32 Function ID	w0	0xC400040D.

R0335

The output parameters of the FIRME_ATTEST_PAT_EXT_CLAIMS_FINALISE interface are listed in [Table 8.34](#).

Table 8.34: FIRME_ATTEST_PAT_EXT_CLAIMS_FINALISE output parameters

Parameter	Register	Value
int32 Return status	w0	SUCCESS NOT_SUPPORTED RETRY ABORTED BAD_DATA

8.18 FIRME_IDEV_OP_START

R₀₃₃₆ The input parameters of the FIRME_IDEV_OP_START interface are listed in [Table 8.35](#).

Table 8.35: FIRME_IDEV_OP_START input parameters

Parameter	Register	Value
uint32 Function ID	w0	0xC4000410.
uint64 Flags	x1	- Bits[0]: Storage type. 0b0: The protocol request and response messages are populated in the shared buffer. 0b1: The protocol request and response messages are populated in input parameter registers. - Bits[2:1]: Protocol type. 0b00: TDISP protocol message. 0b01: IMPLEMENTATION DEFINED protocol message to request device measurements. - All other values are unused. - Bits[63:3]: Unused. See also 2.2 SMC interface.
Input parameter register	x2	Unused. See also 2.2 SMC interface.
Input parameter register	x3-x17	- If <i>Storage type</i> is 0, see Table 7.4 for a description of these parameters. - Otherwise, see Table 7.1 for a description of these parameters.

I₀₃₃₇ The output parameters of the FIRME_IDEV_OP_START interface are listed in [Table 7.3](#).

8.19 FIRME_IDEV_OP_CONTINUE

R₀₃₃₈ The input parameters of the FIRME_IDEV_OP_CONTINUE interface are listed in [Table 8.36](#).

Table 8.36: FIRME_IDEV_OP_CONTINUE input parameters

Parameter	Register	Value
uint32 Function ID	w0	0xC4000411.
uint64 Flags	x1	- Bits[0]: Storage type. 0b0: The protocol response message is populated in the shared buffer. 0b1: The protocol response message is populated in input parameter registers. - Bits[63:1]: Unused. See also 2.2 SMC interface.
uint64 Cookie	x2	- Cookie returned by the callee in a previous invocation of FIRME_IDEV_OP_START. - Otherwise, this register is unused. See also 2.2 SMC interface.
Input parameter register	x3-x17	- If <i>Storage type</i> is 0, see Table 7.4 for a description of these parameters. - Otherwise, these registers are unused. See also 2.2 SMC interface.

I₀₃₃₉ The output parameters of the FIRME_IDEV_OP_CONTINUE interface are listed in [Table 7.3](#).

Glossary

ABI

Application Binary Interface

CMEM

Coherent Memory Device

DOE

Data Object Exchange

DSM

Device Security Manager

Host

Software executing in Non-secure Security state which manages resources used by Realms

MBP

Must be preserved

MBZ

Must be zero

MEC

Memory Encryption Context

MSD

Monitor Security Domain

OS

Operating System

PAS

Physical Address Space

PE

Processing Element

PGS

Protected Granule Size

RCiEP

Root Complex integrated Endpoint

RME

Realm Management Extension

RMI

Realm Management Interface

RMM

Glossary

	Realm Management Monitor
RMSD	
	Realm Management Security Domain
SBZ	
	Should be zero
SMC	
	Secure Monitor Call
SMCCC	
	SMC Calling Convention
SMMU	
	System Memory Management Unit
SPDM	
	Security Protocol and Data Model
SPM	
	Secure Partition Manager
TDI	
	TEE Device Interface
TSM	
	TEE Security Manager
TVM	
	Trusted Execution Environment VM
VM	
	Virtual Machine