Firmware Interfaces for Realm Management Extension

Document number	DEN0149
Document quality	ALP0
Document version	1.0
Document confidentiality	Non-confidential

Firmware Interfaces for Realm Management Extension

Release information

Date	Version	Changes
2025/Jul/09	ALP0	• Replace FIRME prefix with MFI (Monitor Firmware Interface) in ABI and register names to enable usage on systems that do no 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.L0GPTSZ 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 IMPDEF cookie parameters to IDE key configuration ABIs to facilitate matching of requests to responses • Miscellaneous fixes and language clarifications
2025/Apr/11	DEV0	• First DEV version that includes ABIs for MEC managemet, Granule management and IDE key management

Non-Confidential Proprietary Notice

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

PRE-20349

8 March 2024

Contents

Firmware Interfaces for Realm Management Extension

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_{iiiii}

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

Content item classes

Declaration

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

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

A Declaration does not describe behaviour.

A Declaration is rendered with the label *D*.

Rule

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

A Rule explains what happens in a particular situation.

A Rule does not define concepts or terminology.

A Rule is rendered with the label *R*.

Goal

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

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

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

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

A Goal is rendered with the label *G*.

Information

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

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

Rationale

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

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

Implementation note

An Implementation note provides guidance on implementation of the specification.

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

Software usage

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

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

Additional reading

This section lists publications by Arm and by third parties.

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

[1] *Arm® Architecture Reference Manual Armv8, for Armv8-A architecture profile*. See https://developer.arm.com/documentation/ddi0487/latest

[2] *Realm Management Monitor specification*. See https://developer.arm.com/documentation/den0137/latest

[3] *Arm® FF-A memory management protocol version 1.2*. See https://developer.arm.com/documentation/den0140/a

[4] *PCI Express 6.0 specification*. See https://pcisig.com/pci-express-6.0-specification

[5] *CXL 3.2 specification*. See https://computeexpresslink.org/cxl-specification/

[6] *RME system architecture spec*. See https://documentation-service.arm.com/static/60d3309b677cf7536a55bae0

[7] *Arm CCA Security Model*. See https://developer.arm.com/documentation/DEN0096/latest/

[8] *SMC Calling Convention v1.2*. See https://developer.arm.com/documentation/den0028/c

[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. See also:
  - Chapter 3 *Granule management*.
- IDE key management at PCIe and CXL Root ports. See also:
  - Chapter 4 *IDE key management*.
- Memory Encryption Context management. See also:
  - Chapter 5 *Memory Encryption Context management*.

Issue The goal of the FIRME specification is to specify a standard versioned interface between EL3 firmware and the software component in EL2 or EL1 in any non-Root Security state, for services related to FEAT_RME. In addition to the services listed above, future revisions of the FIRME spec will also cover the following services:

- ABIs to support PCIe TDISP for on-chip devices e.g. Root Complex integrated endpoints (RCiEPs).
- Boot protocol between EL3 firmware and RMM.

Arm requests feedback on the need to include other EL3 firmware services in the scope of the FIRME specification.

Chapter 2
Concepts

D0002 An implementation of FIRME at a valid Exception level boundary between EL3 and the highest Exception level in a different Security state is called an *MFI instance*.

D0003 An implementation of FIRME at the Exception level boundary between EL3 and the highest implemented Exception level in the Non-secure Security state is called the *NS MFI instance*.

D0004 An implementation of FIRME at the Exception level boundary between EL3 and the highest implemented Exception level in the Secure Security state is called the *Secure MFI instance*.

D0005 An implementation of FIRME at the Exception level boundary between EL3 and the highest implemented Exception level in the Realm Security state is called the *Realm MFI instance*.

R0006 A MFI instance is compliant with SMCCC version >= 1.2.

R0007 If FEAT_SVE is implemented, a MFI instance is compliant with SMCCC version >= 1.3.

R0008 If FEAT_SME is implemented, a MFI instance is compliant with SMCCC version >= 1.4.

R0009 A MFI instance uses the SMC64 calling convention.

I0010 The presence of a MFI instance, and its version, is discovered via the MFI_VERSION ABI.

R0011 If a MFI instance is present, the MFI_VERSION interface is implemented by that instance.

R0012 The following rules apply to the major version and minor version numbers returned by the MFI_VERSION interface at a MFI instance:

- 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 valueff 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.

R0013 An invocation of the MFI_VERSION ABI returns the version of the available MFI instance to the caller. If no instance is present, the ABI invocation completes with the NOT_SUPPORTED return status code.

I0014 EL3 firmware is permitted to implement different versions of FIRME at different MFI instances. Furthermore, presence of one MFI instance does not imply presence of other instances. The presence of each MFI instance, and its compatibility with another instance (if present) is IMPLEMENTATION DEFINED in EL3 firmware.

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

I0015 A MFI instance may support a different set of features on different Arm CCA platforms. Features include implementations of FIRME ABIs as well as other configuration options at a MFI instance. The features supported by a MFI instance are discovered by reading feature registers via the MFI_FEATURES ABI.

R0016 If a MFI instance is present, the MFI_FEATURES interface is implemented by that instance.

I0017 This version of the FIRME specification specifies the following feature registers:

- MFI feature register 0.
- MFI feature register 1.
- MFI feature register 2.

I0018 MFI feature register 0 reports the presence of a FIRME ABI at a MFI instance.

I0019 MFI feature register 1 reports architectural configuration options used by EL3 firmware as follows:

- Physical Granule size.
- Level 0 GPT entry size.
- Protected Physical Address Size.
- Common MECID width.

D0020 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*.

R0021 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.

I0022 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 achieve 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 MFI instance is IMPLEMENTATION DEFINED.
- 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.

In both usage models the FIRME ABI that uses the buffer specifies the physical base address, and the size of the buffer.

X0023 EL3 firmware could establish a shared buffer with the RMM via an IMPLEMENTATION DEFINED boot protocol. The first usage model enables a FIRME ABI to use this buffer for transmitting input parameters, and receiving output parameters.

I0024 MFI feature register 2 reports software configuration options used by EL3 firmware as follows:

- Minimum size and alignment boundary of a shared buffer.
- Maximum size of a shared buffer.
- Maximum size of the platform attestation token.
- Maximum size of the platform attestation token.
- Presence of support for retrieving the public portion of the Realm attestation key.
- Encoding format of the Realm attestation key.
- Presence of support for signing a Realm attestation token by the CCA HES.

See also:

- 6.3.2 *Platform attestation token retrieval*.
- 6.3.3 *Realm attestation key retrieval*.
- 6.3.4 *Realm attestation token signing*.

R0025 An invocation of the MFI_FEATURES ABI returns the MFI feature register requested by the caller, and completes with the SUCCESS return status code, if none of the following are true:

- There is no MFI instance available to the caller.
  - In this case, the ABI invocation completes with the NOT_SUPPORTED return status code.
- The value of Feature register index is reserved.
  - In this case, the ABI invocation completes with the INVALID_PARAMETERS return status code.

Chapter 3
Granule management

3.1 Overview

I0026 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 GPT Block, 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 Interface (GMI) is an ABI implemented by EL3 firmware to 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.

Figure 3.1 illustrates that GMI is implemented at the EL boundary between EL3 and the Non-secure, Realm and Secure security states.

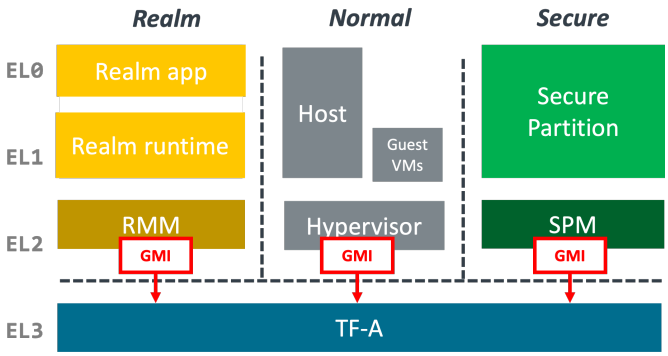

Figure 3.1: Granule management interface

3.2 Scope

I0027 This version of the specification 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

D0028 A *GPI transition* is a change of the GPI encoding of a granule from its current GPI encoding to a target GPI encoding.

I0029 The FIRME specification defines the granule security policy which specifies the GPI transitions that are permitted for a granule.

I0030 The MFI_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 *Current GPI encoding* and the *Target GPI encoding* fields of the *Attributes* input parameter.

D0031 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.

R0032 When FEAT_RME is implemented, the granule security policy permits a GPI transition if the caller's Security state, current GPI encoding of the granule and the target GPI encoding of the granule are as follows:

Table 3.1: Permitted GPI transitions for FEAT_RME

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

Otherwise, the GPI transition is not permitted.

R0033 When FEAT_RME_GPC2 is implemented, the granule security policy permits a GPI transition if any of the following is true:

- The GPI transition is permitted when FEAT_RME is implemented.
- The caller's Security state, current GPI encoding of the granule and the target GPI encoding of the granule are as follows:

Table 3.2: Permitted GPI transitions for FEAT_RME_GPC2

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

Otherwise, the GPI transition is not permitted.

S0034 The following pseudocode illustrates the GPI transition 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 transition_granule(uint64_t *addr, uint8_t target_gpi) {

    if ((target_gpi != NSO || (target_gpi != NS))
        return ERROR;

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

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

    return SUCCESS;
}
```

R0035 When FEAT_RME_GDI is implemented, the granule security policy permits a GPI transition if any of the following is true:

- The GPI transition is permitted when FEAT_RME_GPC2 is implemented.
- The caller's Security state, current GPI encoding of the granule and the target GPI encoding of the granule are as follows:

Table 3.3: Permitted GPI transitions for FEAT_RME_GDI

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

Otherwise, the GPI transition is not permitted.

S0036 The following pseudocode illustrates the GPI transition of a single granule from Non-secure to NSP.

```
int32_t granule_delegate_to_nsp(uint64_t *addr) {

    // Update the GPI encoding
    write_gpt(addr, NSP)
    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);
}
```

S0037 The following pseudocode illustrates the GPI transition of a single granule from NSP to Non-secure.

```
int32_t granule_undelegate_from_nsp(uint64_t *addr) {

    // 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);
}
```

R0038 The size of a granule that can undergo a GPI transition is equal to the PGS.

U0039 The PGS can be discovered via the *PGS* field in MFI feature register 1.

R0040 The physical base address of the physically contiguous granules that undergoes a GPI transition is aligned to the PGS.

D0041 The first granule at the *Base address* input parameter of the MFI_GM_GPI_SET ABI is called *Granule 0*.

R0042 An invocation of the MFI_GM_GPI_SET ABI successfully performs the GPI transition of one or more granules 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 and no granule undergoes a GPI transition.
- The base address of the physically contiguous granules is not aligned to the 4K boundary.
  - In this case, the ABI invocation completes with the INVALID_PARAMETERS return status code and no granule undergoes a GPI transition.
- The physically contiguous granules does not lie in the region in the physical address map protected by the GPC mechanism and specified in GPCCR_EL3.PPS.
  - In this case, the ABI invocation completes with the INVALID_PARAMETERS return status code and no granule undergoes a GPI transition.
- A field in the *Attributes* input parameter is incorrectly encoded.
  - In this case, the ABI invocation completes with the INVALID_PARAMETERS return status code and no granule undergoes a GPI transition.
- The transition from *Current GPI* encoding to *Target GPI* encoding specified in the *Attributes* input parameter is not permitted by the granule security policy.
  - In this case, the ABI invocation completes with the INVALID_PARAMETERS return status code and no granule undergoes a GPI transition.
- 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 and no granule undergoes a GPI transition.
- The GPI encoding of a granule does not match the *Current GPI* encoding specified in the *Attributes* input parameter.
  - In this case all of the following are true:
    - The ABI invocation completes with the DENIED return status code.
    - The *Granule count* output parameter specifies the count of granules (starting from Granule 0) that underwent a successful GPI transition.
    - If the granule is Granule 0, the *Granule count* output parameter is 0.

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

- The *Granule count* output parameter is >= 1.
- The count of granules starts from Granule 0.

I0043 When an invocation of the MFI_GM_GPI_SET ABI completes with the SUCCESS return status code, the *Granule count* output parameter specifies the count of granules that underwent a successful GPI transition.

If the output *Granule Count* is not equal to the input *Granule count*, the caller should invoke the MFI_GM_GPI_SET ABI again with the updated *Base address* and input *Granule count* that did not undergo the requested GPI transition.

R0044 When an invocation of the MFI_GM_GPI_SET ABI does not complete with the SUCCESS or the DENIED return status code, the value of the *Granule count* output parameter is UNKNOWN and is ignored by the caller.

U0045 The failure condition where the MFI_GM_GPI_SET ABI completes with the DENIED return status code, prevents the implementation from ensuring that the GPI encoding of all granules matches the *Current GPI* encoding before any GPI transition is performed. Instead, the implementation performs a successful GPI transition of all granules starting from Granule 0 until it encounters a granule that cannot undergo the requested transition.

If Granule 0 cannot undergo the requested transition, the implementation is unable to transition any granule, In this case, the MFI_GM_GPI_SET ABI completes with the DENIED return status code and the *Granule count* output parameter is 0.

Otherwise, the MFI_GM_GPI_SET ABI completes with the DENIED return status code, and the *Granule Count* output parameter specifies the count of granules (starting from Granule 0 that underwent a successful GPI transition.

I0046 The Host uses the RMI_GRANULE_DELEGATE ABI in [2] to request the RMM to change the GPI encoding of a granule from Non-secure to Realm.

The Host uses the RMI_GRANULE_UNDELEGATE ABI to request the RMM to change the GPI encoding of a granule from Realm to Non-secure.

The RMM uses the MFI_GM_GPI_SET ABI to perform the GPI transition requested by the Host.

I0047 The OS or Hypervisor uses the FFA_MEM_LEND ABI in [3] to request the SPM to change the GPI encoding of a granule from Non-secure to Secure.

The OS or Hypervisor uses the FFA_MEM_RECLAIM ABI to request the SPM to change the GPI encoding of a granule from Secure to Non-secure.

The SPM uses the MFI_GM_GPI_SET ABI to perform the GPI transition requested by the OS or Hypervisor.

I0048 The Host can use the MFI_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.

S0049 Figure 3.2 shows an example flow where the Host delegates a single 4K granule to the RMM via RMI_GRANULE_DELEGATE [2].

Figure 3.3 shows an example flow where the single 4K granule is undelegated by the RMM back to the Host in response to RMI_GRANULE_DELEGATE [2].

The RMM uses MFI_GM_GPI_SET to perform a GPI transition of the granule from Non-secure to Realm and vice-versa.

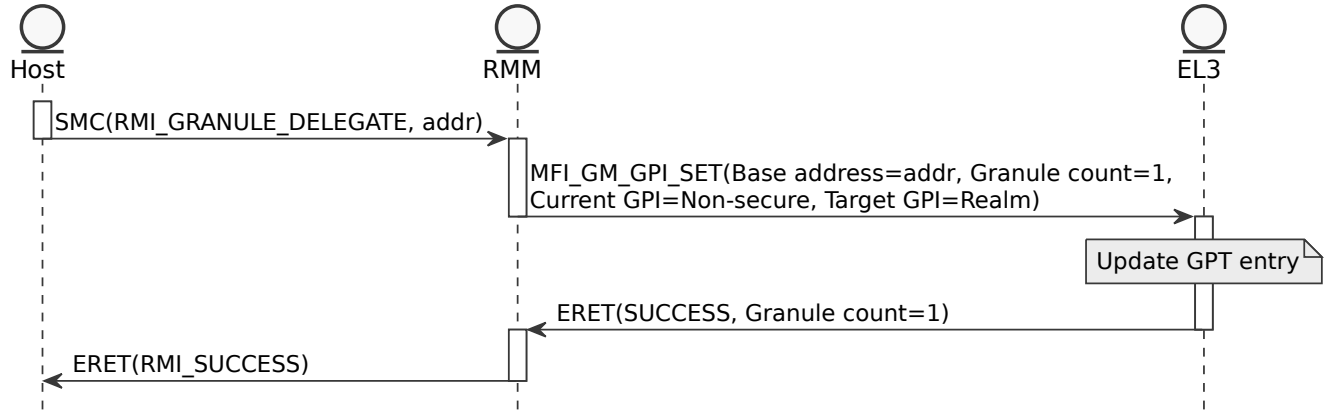

Figure 3.2: Example granule delegation flow with RMI_GRANULE_DELEGATE

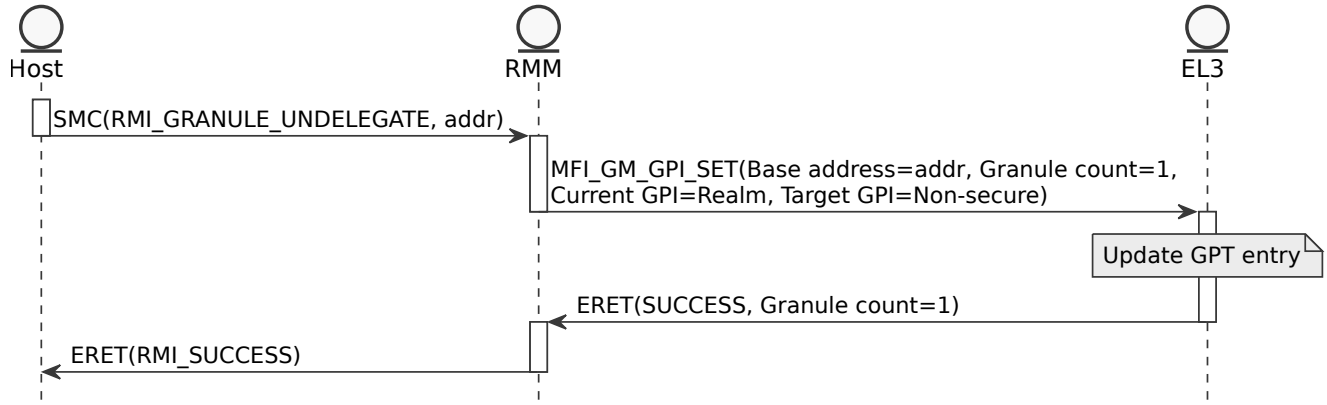

Figure 3.3: Example granule undelegation flow with RMI_GRANULE_UNDELEGATE

S0050 Figure 3.4 shows an example flow where the Host lends a list of physically discontiguous memory regions to an SP via FFA_MEM_LEND [3].

Figure 3.5 shows an example flow where the SP returns the memory regions RMM back to the Host via FFA_MEM_RECLAIM [3].

The SPMC uses MFI_GM_GPI_SET to perform a GPI transition of the granule from Non-secure to Secure and vice-versa.

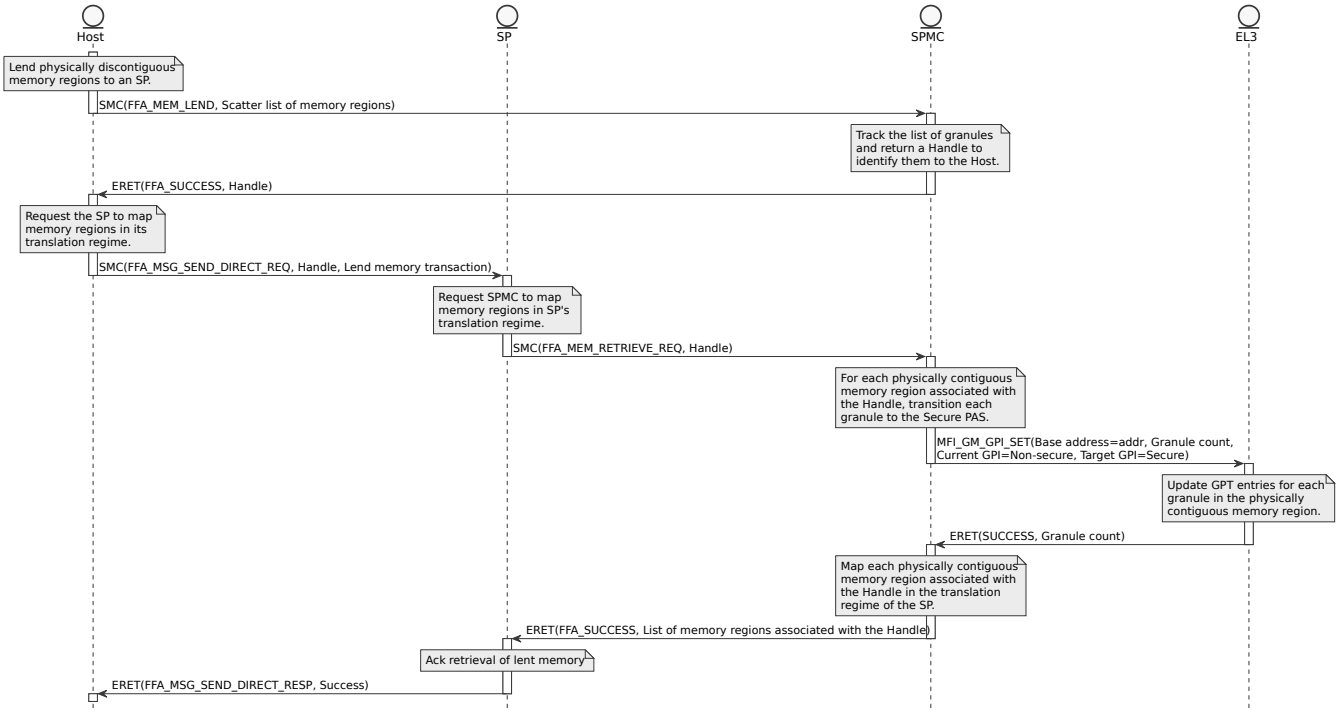

Figure 3.4: Example flow to lend memory from Non-secure to Secure with FFA_MEM_LEND

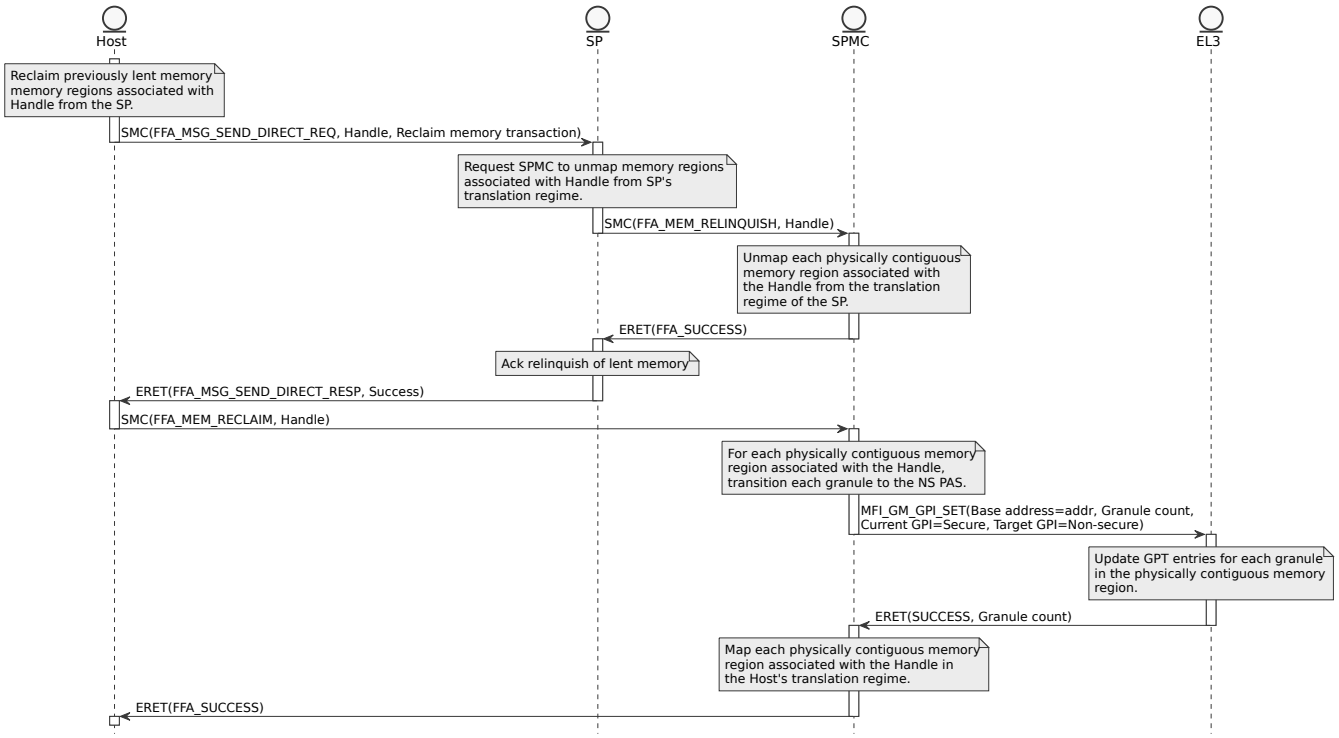

Figure 3.5: Example flow to return memory from Secure to Non-secure with FFA_MEM_RECLAIM

Chapter 4
IDE key management

4.1 Overview

I0051 The PCIe [4] and CXL [5] IDE Key programming mechanisms permit a Root Complex to use IMPLEMENTATION DEFINED key management. The RME System architecture specification [6] 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 configuration interface is a set of ABIs implemented by EL3 firmware i.e. MSD firmware to provide integrity and data encryption for TLPs transmitted through the link between a PCIe or CXL Root Port (RP) and an endpoint function.

4.2 Scope

I0052 The scope of the FIRME IDE key configuration interface is as follows:

- The FIRME IDE key configuration interface is inaccessible from the Secure world.
- On a system that implements FEAT_RME, the Host or OS in the Normal world uses the IDE key configuration ABIs exposed by the RMM 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 configuration interface to establish an IDE stream between the Root port and an Endpoint.

Issue Arm is not aware of use cases where software components in the Secure world require access to the FIRME IDE key configuration interface. Arm requests feedback on the validity of this constraint.

4.3 Concepts

I0053 The IDE key configuration 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.

D0054 An invocation of an IDE key configuration ABI with the *Input Flags.Request type* field set to `0b00` is a request to configure a PCIe or CXL.io Selective IDE stream.

D0055 An invocation of an IDE key configuration ABI with the *Input Flags.Request type* field set to `0b11` is a request to configure a CXL.cachemem Link IDE stream.

D0056 The target keyset in an IDE key configuration ABI is identified by the *keyset ID*.

R0057 *Keyset ID* is a 64-bit bitmap that is encoded as follows:

- Bits[63:30]: Reserved.
- Bits[29:14]: Root port ID.
- Bits[13:6]: Stream ID.
- Bits[5:2]: Substream ID.
- Bits[1]: Direction.
- Bits[0]: Key set.

I0058 The names of the fields in a keyset id are terms defined in [4].

R0059 The *Key Set* field in a keyset id is Reserved (MBZ) in a request to configure a CXL.cachemem Link IDE stream.

X0060 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.

R0061 The value of the *Stream ID* field in a keyset id is 0 in a request to configure a CXL.cachemem Link IDE stream.

R0062 The value of the *Substream ID* field in a keyset id is `0b1000` in a request to configure a CXL.cachemem Link IDE stream.

R0063 The ECAM address space of a Root port defines the namespace for keyset ids.

R0064 A keyset id is unique in the namespace defined by the ECAM address space of a Root port.

I0065 The MFI_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 MFI_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 [4], 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 [5], 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 [5]. 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.

R0066 An invocation of the MFI_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 *Request type* field in the *Input Flags* input parameter is set to a reserved 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 key configuration interface 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 pending operation for the specified keyset ID and the caller must retry the operation.
  - In this case, the ABI invocation completes with the RETRY return status code.
- 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:
    * MFI_IDE_KEYSET_POLL.

I0067 The MFI_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.

R0068 An invocation of the MFI_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 *Request type* field in the *Input Flags* input parameter is set to a reserved 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 key configuration interface 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 pending operation for the specified keyset ID and the caller must retry the operation.
  - In this case, the ABI invocation completes with the RETRY return status code.
- No key was programmed for the specified keyset ID.
  - In this case, the ABI invocation completes with the DENIED return status code.
- 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:
    * MFI_IDE_KEYSET_POLL.

I0069 The MFI_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.

R0070 An invocation of the MFI_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 *Request type* field in the *Input Flags* input parameter is set to a reserved 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 key configuration interface 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 pending operation for the specified keyset ID and the caller must retry the operation.
  - In this case, the ABI invocation completes with the RETRY return status code.
- No key was programmed for the specified keyset ID.
  - In this case, the ABI invocation completes with the DENIED return status code.
- 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:
    * MFI_IDE_KEYSET_POLL.

D0071 The FIRME IDE key configuration interface allows EL3 firmware to 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.

I0072 The MFI_IDE_KEYSET_POLL ABI is used to determine the status of a non-blocking IDE key configuration 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.

S0073 The FIRME IDE key configuration ABIs caters for a scenario where EL3 firmware delegates the responsibility for IDE key configuration to the 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 configuration ABI after delegating the operation to the Trusted subsystem. The caller polls the status of the operation via the MFI_IDE_KEYSET_POLL ABI until the operation completes.

I0074 EL3 firmware does not provide a mechanism to determine which IDE key configuration 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 configuration ABIs tracks the type of operation that was requested. Each IDE key configuration 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 configuration 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 MFI_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.

R0075 An invocation of the MFI_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 *Request type* field in the *Input Flags* input parameter is set to a reserved 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 key configuration interface 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.
- 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.
- 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 INVALID_REQUEST return status code.

I_{0076} Completion of an IDE key configuration ABI with the INCOMPLETE return status code does not imply that the operation will complete successfully. The caller uses the MFI_IDE_KEYSET_POLL ABI to check the status of the requested operation.

R_{0077} An invocation of any IDE key configuration ABI completes with the NOT_SUPPORTED return status code if the ABI is invoked from the Secure security state.

S_{0078} 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 [2]. RMM uses the FIRME IDE key configuration ABIs to enable IDE for the stream at the Root port.

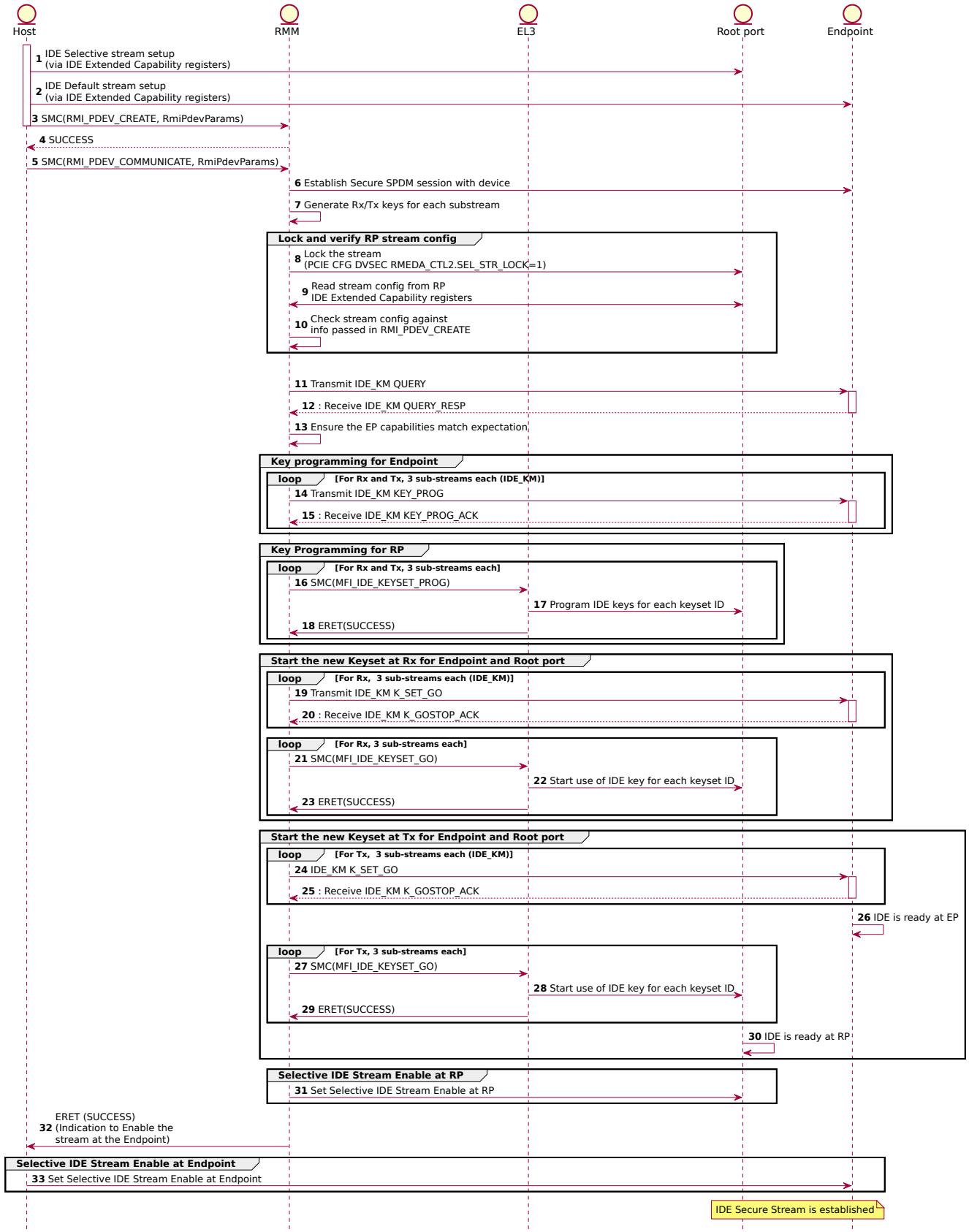

Figure 4.1: Example IDE key programming flow

S0079 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 [2]. RMM uses the FIRME IDE key programming ABIs to refresh the IDE key for the stream at the Root port.

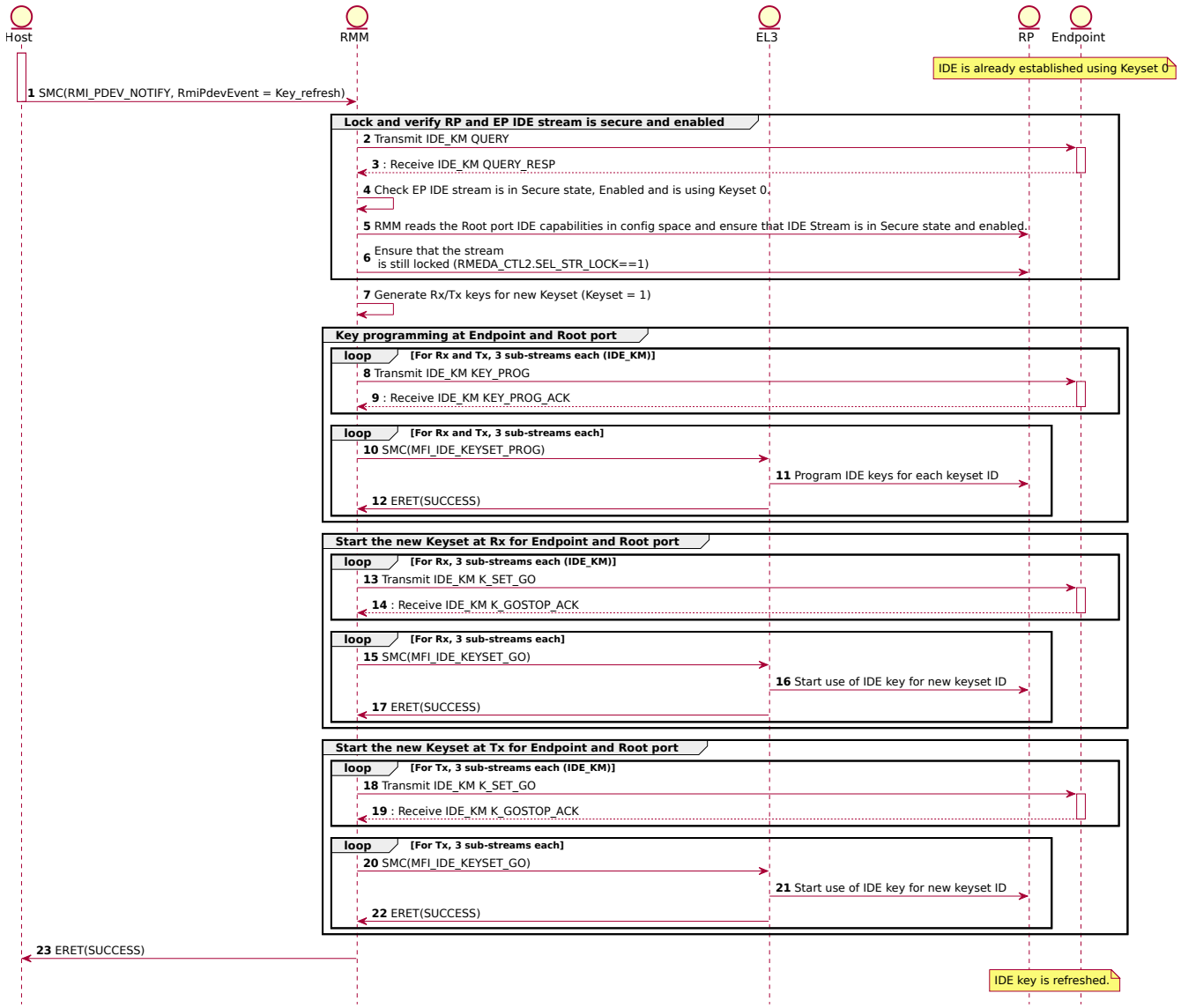

Figure 4.2: Example IDE key refresh flow

Chapter 5
Memory Encryption Context management

5.1 Overview

I$_{0080}$ 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 MEC management interface defines the ABI implemented by EL3 firmware, to refresh a MEC corresponding to a MECID.

5.2 Scope

I$_{0081}$ This version of the specification supports refresh of a MEC within the scope of features added by the following architecture extensions:

- FEAT_MEC

I$_{0082}$ 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

R$_{0083}$ A MEC, associated with a MECID, is refreshed when the existing encryption context is invalidated and a new encryption context is generated.

I$_{0084}$ 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 memory encryption system must ensure that input values cannot repeat during the uptime of a system.

R$_{0085}$ A FIRME MEC management interface implementation uses the common MECID width [1].

I$_{0086}$ The common MECID width is discoverable via the COMMON_MECID_WIDTH field in MFI feature register 1.

R$_{0087}$ An invocation of the MFI_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 MEC management interface is busy and the caller must retry the operation.
  - In this case, the ABI invocation completes with the RETRY return status code.

I$_{0088}$ 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.

X$_{0089}$ 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.

Chapter 6
Attestation token management

6.1 Overview

I0090 CCA attestation allows a user of a service provided by a Realm i.e. a reliant party, to determine the trustworthiness of the Realm, and of the implementation of the CCA platform [7]. In the delegated attestation model, the RMM relies on EL3 firmware to access the delegated 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.

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

6.2 Scope

I0091 This version of the specification only supports the Delegated attestation model. Other attestation models described in [7] are out of scope.

6.3 Concepts

I0092 The MFI_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*.

I0093 The MFI_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*.

I0094 The MFI_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*.

6.3.1 Shared buffer usage

I0095 A shared buffer is used by the following FIRME attestation token management ABIs to transmit input data to, and receive output data from EL3 firmware:

- MFI_ATTEST_PAT_GET ABI.
- MFI_ATTEST_RAK_GET ABI.
- MFI_ATTEST_RAT_SIGN ABI.

I0096 The layout of the shared buffer used by the FIRME attestation token management ABIs is described by the *Shared buffer base address* and the *Shared buffer size* input parameters.

R0097 When a FIRME attestation token management ABI is invoked, if the shared buffer is used, and is already mapped in the EL3 translation regime, it is not unmapped from the EL3 translation regime before the invocation of this ABI completes.

R0098 When a FIRME attestation token management ABI is invoked, if the shared buffer is used, and is not already mapped in the EL3 translation regime, then all of the following are true:

- The buffer is mapped in the EL3 translation regime.
- The buffer is unmapped from the EL3 translation regime before the invocation of this ABI completes.

R0099 The size of the shared buffer used by a FIRME attestation token management ABI fulfils the following constraints:

- The size is >= Minimum shared buffer size determined from the *MIN_SH_BUF_SZ* field in MFI feature register 2.
- The size is <= Maximum shared buffer size determined from the *MAX_SH_BUF_SZ* field in MFI feature register 2.

R0100 The physical base address of the shared buffer used by a FIRME attestation token management ABI is aligned to the minimum shared buffer size determined from the *MIN_SH_BUF_SZ* field in MFI feature register 2.

6.3.1.1 Data retrieval in chunks

I0101 It is possible that the size of the output data of a FIRME attestation token management 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.

I0102 Execution of a FIRME attestation token management ABI that uses the shared buffer 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 has completely written the output data to the shared buffer. In this case, the ABI is invoked again after the interrupt is handled to continue retrieval of the output data via the shared buffer.

D0103 A FIRME attestation token management ABI that uses a shared buffer to transmit output data in chunks supports *partial retrieval* of output data.

I0104 FIRME attestation token management ABIs that support partial retrieval include the following output parameters to let the caller track progress of this transmission:

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

D0105 Upon completion, a FIRME attestation token management 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.

D0106 Upon completion, a FIRME attestation token management 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.

D0107 Upon completion, a FIRME attestation token management ABI that supports partial retrieval, indicates that no chunk of output data was retrieved as follows:

- *Written size* is zero.
- *Remaining size* is zero.

I0108 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 start retrieval of output data to the CCA HES, 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.

R0109 When a FIRME attestation token management ABI that supports partial retrieval completes a request to *initiate* retrieval of output data with the SUCCESS return status code, if the *Written size* is zero, the *Remaining size* of the data cannot be non-zero.

X0110 This specification assumes that it is not possible for EL3 firmware to not have the output data but know its size. Hence, a scenario is not possible where EL3 firmware is unable to return a chunk of the output data to the caller but is able to return its size.

I0111 When a FIRME attestation token management ABI that supports partial retrieval completes a request to *continue* retrieval of 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 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

R0112 The MFI_ATTEST_PAT_GET ABI is implemented only at the NS MFI instance and the Realm MFI instance.

I0113 The MFI_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.

I0114 The *MAX_PAT_SZ* field in MFI feature register 2 specifies the maximum size of the platform attestation token. If this size is <= Maximum shared buffer size determined from the *MAX_SH_BUF_SZ* field in the same register, the caller of the MFI_ATTEST_PAT_GET ABI can dynamically allocate the exact amount of memory required to retrieve the complete platform attestation token.

I0115 The encoding of the Platform attestation token obtained from the CCA HES is specified in [2].

D0116 An invocation of the MFI_ATTEST_PAT_GET ABI with a non-zero *Platform challenge size* input parameter is a request to initiate retrieval of the platform attestation token.

D0117 An invocation of the MFI_ATTEST_PAT_GET ABI with a zero *Platform challenge size* input parameter is a request to continue retrieval of the platform attestation token.

R0118 If the MFI_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 MFI instance.

I0119 Once a caller has initiated retrieval of the platform attestation token, a subsequent invocation of the MFI_ATTEST_PAT_GET ABI on any PE at the same MFI 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).

I0120 Table 6.1 lists the valid combinations and description of the following parameters when an invocation of the MFI_ATTEST_PAT_GET ABI completes successfully.

- *Platform challenge size* input parameter.
- *Written size* output parameter.
- *Remaining size* output parameter.

Table 6.1: Description of parameters that govern progress of MFI_ATTEST_PAT_GET

Platform challenge size	Written size	Remaining size	Description
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 MFI_ATTEST_PAT_GET ABI must be invoked again to retrieve the remainder of the platform attestation token.
0	0	Non-zero	No chunk of the platform attestation token was written to the shared buffer. The MFI_ATTEST_PAT_GET ABI must be invoked again to retrieve the remainder of the platform attestation token.
0	Non-zero	0	Retrieval of the platform attestation token is complete and the last remaining chunk has been written to the shared buffer.
0	Non-zero	Non-zero	A chunk of the platform attestation token has been written to the shared buffer. The MFI_ATTEST_PAT_GET ABI must be invoked again to retrieve the remainder of the platform attestation token.
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 MFI_ATTEST_PAT_GET ABI must be invoked again to retrieve the remainder of the platform attestation token.
Non-zero	Non-zero	0	The complete platform attestation token has been written to the shared buffer.
Non-zero	Non-zero	Non-zero	A chunk of the platform attestation token has been written to the shared buffer. The MFI_ATTEST_PAT_GET ABI must be invoked again to retrieve the remainder of the platform attestation token.

R0121 An invocation of the MFI_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 MFI 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.
- 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 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.

S0122 If an invocation of the MFI_ATTEST_PAT_GET ABI completes with the ABORTED error status code, it is IMPLEMENTATION DEFINED whether the next request to initiate retrieval of the platform attestation token from the caller will succeed. EL3 firmware can return the ABORTED error status code in case it is unable to communicate with the CCA HES. This could be a temporary scenario and the next request could complete successfully. The caller can retry retrieval of the token an IMPLEMENTATION DEFINED number of times. If each try completes with the ABORTED error status code, it should consider the MFI_ATTEST_PAT_GET ABI unavailable.

S0123 The following pseudocode illustrates retrieval of the platform attestation token via the MFI_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 = MFI_FEATURES(Feature register 2, &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 = MFI_ATTEST_PAT_GET(shbuf, offset, alloc_sz, plat_challenge_sz, &
                ↪written_sz, &remaining_sz);
            plat_challenge_sz = 0; // Reset this value in case we need to retrieve
                ↪rest of the platform attestation token.
            offset += written_sz;
        } while (remaining_sz && offset < alloc_sz);

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

    return ret;
}
```

6.3.3 Realm attestation key retrieval

R0124 The MFI_ATTEST_RAK_GET ABI is implemented only at the Realm MFI instance.

I0125 The MFI_ATTEST_RAK_GET ABI uses a physically contiguous shared buffer for receiving the Realm attestation key.

I0126 The *RAK_FORMAT* field in MFI feature register 2 specifies the format in which the Realm attestation key is encoded in the shared buffer.

I0127 The MFI_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 MFI feature register 2.

D0128 An invocation of the MFI_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.

D0129 An invocation of the MFI_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.

R0130 If the MFI_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 MFI instance.

I0131 Once a caller has initiated retrieval of the specified portion of the Realm attestation key, a subsequent invocation of the MFI_ATTEST_RAK_GET ABI on any PE at the same MFI 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.

I0132 Table 6.2 lists the valid combinations and description of the following parameters when an invocation of the MFI_ATTEST_RAK_GET ABI completes successfully.

- *Request type* flag input parameter.
- *Written size* output parameter.
- *Remaining size* output parameter.

Table 6.2: Description of parameters that govern progress of MFI_ATTEST_RAK_GET

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 MFI_ATTEST_RAK_GET ABI must be invoked again to retrieve the remainder of the key.
0	0	Non-zero	No chunk of the specified portion of the Realm attestation key was written to the shared buffer. The MFI_ATTEST_RAK_GET ABI must be invoked again to retrieve the remainder of the key.
0	Non-zero	0	Retrieval of the specified portion of the Realm attestation key is complete and the last remaining chunk has been written to the shared buffer.
0	Non-zero	Non-zero	A chunk of the specified portion of the Realm attestation key has been written to the shared buffer. The MFI_ATTEST_RAK_GET ABI must be invoked again to retrieve the remainder of the key.
Non-zero	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 MFI_ATTEST_RAK_GET ABI must be invoked again to retrieve the remainder of the key.

Request type	Written size	Remaining size	Description
Non-zero	Non-zero	0	The specified portion of the Realm attestation key has been completely written to the shared buffer.
Non-zero	Non-zero	Non-zero	A chunk of the specified portion of the Realm attestation key has been written to the shared buffer. The MFI_ATTEST_RAK_GET ABI must be invoked again to retrieve the remainder of the key.

R0133 An invocation of the MFI_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 MFI 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 *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.
- The CCA HES has rejected the request to retrieve the specified portion of the Realm attestation key.
  - In this case, the ABI invocation completes with the INVALID_REQUEST 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 cannot complete retrieval the 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.

S0134 If an invocation of the MFI_ATTEST_RAK_GET ABI completes with the ABORTED error status code, it is IMPLEMENTATION DEFINED whether the next request to initiate retrieval of the specified portion of the Realm attestation key from the caller will succeed. EL3 firmware can return the ABORTED error status code in case it is unable to communicate with the CCA HES. This could be a temporary scenario and the next request could complete successfully. The caller can retry retrieval of the token an IMPLEMENTATION DEFINED number of times. If each try completes with the ABORTED error status code, it should consider the MFI_ATTEST_RAK_GET ABI as unavailable.

S0135 The following pseudocode illustrates retrieval of the Realm attestation key via the MFI_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 = 4KB;
```

```
    uint64_t shbuf, input_flags;

    // 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 = MFI_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 (remaining_sz && offset < alloc_sz);

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

    return ret;
}
```

6.3.4 Realm attestation token signing

R0136 The MFI_ATTEST_RAT_SIGN ABI is implemented only at the Realm MFI instance.

I0137 The MFI_ATTEST_RAT_SIGN ABI is optional. Its presence is determined by the value of the *RAT_SIGN* field in MFI feature register 2.

R0138 The *RAK_PUB_POR* field in MFI feature register 2 is 1 if the MFI_ATTEST_RAT_SIGN ABI is implemented.

X0139 The Realm attestation token includes a hash of 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.

I0140 The MFI_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.

I0141 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 MFI_FEATURES ABI.

D0142 An invocation of the MFI_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.

R0143 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.

D$_{0144}$ An invocation of the MFI_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.

R$_{0145}$ 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.

R$_{0146}$ 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.

X$_{0147}$ 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.

I$_{0148}$ 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.

S$_{0149}$ The RMM queues multiple requests for the CCA HES to sign Realm attestation tokens on behalf of various Realms via the MFI_ATTEST_RAT_SIGN ABI. Each request also includes information 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 MFI_ATTEST_RAT_SIGN ABI. Each response also includes the request identifier to match it to the corresponding request.

R$_{0150}$ An invocation of the MFI_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 MFI 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 size* 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.
- 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 MFI_ATTEST_RAK_GET ABI.
  - In this case, the ABI invocation completes with the DENIED return status code.

Chapter 7
Interface

R0151 ABIs defined in this version of the FIRME specification use the status codes defined in Table 7.1.

Table 7.1: FIRME error status codes

Status code	Description
0	SUCCESS
-1	NOT_SUPPORTED
-2	INVALID_PARAMETERS
-3	ABORTED
-4	INCOMPLETE
-5	DENIED
-6	RETRY
-7	INVALID_REQUEST

R0152 An invocation of any FIRME ABI completes successfully if the return status code is SUCCESS.

R0153 All ABIs in the FIRME specification are compliant with version 1.2 or later of the Arm SMC calling convention [8].

R0154 All ABIs in the FIRME specification use the SMC64 calling convention.

R_{0155} An unused input parameter register in a FIRME ABI is Reserved and SBZ.

R_{0156} An unused output parameter register in a FIRME ABI is Reserved and MBZ.

X_{0157} Unused input parameter registers and output parameter registers are reserved as they could be used in a future version of the ABI.

An unused input parameter register should be 0 as it is not guaranteed to be preserved when an invocation of the ABI completes. The SBZ behaviour is a strong recommendation to the caller to avoid the overhead in EL3 firmware of validating that each unused input parameter register is 0.

An unused output parameter register must be 0 to avoid leakage of information from EL3 into the caller's Exception level in a less privileged Security state. From an SMC calling convention standpoint, the value of 0 in an unused output parameter register is treated as a call result. This prevents the caller from assuming that contents of this register will be preserved across a FIRME ABI invocation and allows use of this register to return actual results in a future version of the ABI.

7.1 MFI_VERSION

R0158 The input parameters of the MFI_VERSION interface are listed in Table 7.2.

Table 7.2: MFI_VERSION input parameters

Parameter	Register	Value
uint32 Function ID	w0	• 0xC4000400.

R0159 The output parameters of the MFI_VERSION interface are listed in Table 7.3.

Table 7.3: MFI_VERSION output parameters

Parameter	Register	Value
int64 Return status	x0	• On success, the format of the version number is as follows: – Bits[63:31]: MBZ. – Bits[30:16]: Major version. ∗ Must be 1 for this revision of the FIRME specification. – Bits[15:0]: Minor version. ∗ Must be 0 for this revision of the FIRME specification. • NOT_SUPPORTED

7.2 MFI_FEATURES

R0160 The input parameters of the MFI_FEATURES interface are listed in Table 7.4.

Table 7.4: MFI_FEATURES input parameters

Parameter	Register	Value
uint32 Function ID	w0	• 0xC4000401.
uint32 Feature register index	w1	• Index of the feature register to be returned to the caller. – 0: MFI feature register 0. – 1: MFI feature register 1. – 2: MFI feature register 2. – All other values are reserved.

R0161 The output parameters of the MFI_FEATURES interface are listed in Table 7.5.

Table 7.5: MFI_FEATURES output parameters

Parameter	Register	Value
int64 Return status	x0	• SUCCESS • NOT_SUPPORTED • INVALID_PARAMETERS

R0162 The encoding of MFI_FEATURES feature register 0 is specified in Table 7.6.

Table 7.6: MFI_FEATURES feature register 0

Bits	Name	Description
0	MFI_GM_GPI_SET	Indicates presence of the MFI_GM_GPI_SET ABI. • 0b0: MFI_GM_GPI_SET is not implemented. • 0b1: MFI_GM_GPI_SET is implemented.
1	MFI_IDE_KEYSET_PROG	Indicates presence of the MFI_IDE_KEYSET_PROG ABI. • 0b0: MFI_IDE_KEYSET_PROG is not implemented. • 0b1: MFI_IDE_KEYSET_PROG is implemented. The value of this field is 0b0 if MFI_FEATURES is called from the Secure security state.
2	MFI_IDE_KEYSET_GO	Indicates presence of the MFI_IDE_KEYSET_GO ABI. • 0b0: MFI_IDE_KEYSET_GO is not implemented. • 0b1: MFI_IDE_KEYSET_GO is implemented. The value of this field is 0b0 if MFI_FEATURES is called from the Secure security state.

Bits	Name	Description
3	MFI_IDE_KEYSET_STOP	Indicates presence of the MFI_IDE_KEYSET_STOP ABI. • 0b0: MFI_IDE_KEYSET_STOP is not implemented. • 0b1: MFI_IDE_KEYSET_STOP is implemented. The value of this field is 0b0 if MFI_FEATURES is called from the Secure security state.
4	MFI_IDE_KEYSET_POLL	Indicates presence of the MFI_IDE_KEYSET_POLL ABI. • 0b0: MFI_IDE_KEYSET_POLL is not implemented. • 0b1: MFI_IDE_KEYSET_POLL is implemented. The value of this field is 0b0 if MFI_FEATURES is called from the Secure security state.
5	MFI_MEC_REFRESH	Indicates presence of the MFI_MEC_REFRESH ABI. • 0b0: MFI_MEC_REFRESH is not implemented. • 0b1: MFI_MEC_REFRESH is implemented. The value of this field is 0b0 if MFI_FEATURES is called from the Secure security state or the Non-secure security state.
6	MFI_ATTEST_PAT_GET	Indicates presence of the MFI_ATTEST_PAT_GET ABI. • 0b0: MFI_ATTEST_PAT_GET is not implemented. • 0b1: MFI_ATTEST_PAT_GET is implemented. The value of this field is 0b0 if MFI_FEATURES is called from the Secure security state.
7	MFI_ATTEST_RAK_GET	Indicates presence of the MFI_ATTEST_RAK_GET ABI. • 0b0: MFI_ATTEST_RAK_GET is not implemented. • 0b1: MFI_ATTEST_RAK_GET is implemented. The value of this field is 0b0 if MFI_FEATURES is called from the Secure security state or the Non-secure security state.
8	MFI_ATTEST_TOKEN_SIGN	Indicates presence of the MFI_ATTEST_TOKEN_SIGN ABI. • 0b0: MFI_ATTEST_TOKEN_SIGN is not implemented. • 0b1: MFI_ATTEST_TOKEN_SIGN is implemented. The value of this field is 0b0 if MFI_FEATURES is called from the Secure security state or the Non-secure security state.
63:9	Reserved	MBZ.

R0163 The encoding of MFI_FEATURES feature register 1 is specified in Table 7.7.

Table 7.7: MFI_FEATURES feature register 1

Bits	Name	Description
1:0	PGS	Value of Physical Granule size as encoded in GPCCR_EL3.PGS in [1].
5:2	L0GPTSZ	Value of Level 0 GPT entry size as encoded in GPCCR_EL3.L0GPTSZ in [1].
8:6	PPS	Value of Protected Physical Address Size as encoded in GPCCR_EL3.PPS in [1].
12:9	COMMON_MECID_WIDTH	Value of this field plus 1 is the Common MECID width as specified in [1].
63:13	Reserved	MBZ.

R0164 The encoding of MFI_FEATURES feature register 2 is specified in Table 7.8.

Table 7.8: MFI_FEATURES feature register 2

Bits	Name	Description
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 reserved.
15:2	MAX_SH_BUF_SZ	Value of this field is used to determine the maximum size of a shared buffer as follows: Maximum size of shared buffer (in KB)= (MAX_SH_BUF_SZ + 1)* Minimum size of a shared buffer (in KB). The minimum size of a shared buffer is determined from the MIN_SH_BUF_SZ field.
23:16	MAX_PAT_SZ	Value of this field is used to determine the maximum size of the platform attestation token as follows: Maximum size of the platform attestation token (in KB)= (MAX_PAT_SZ + 1)* Minimum size of a shared buffer (in KB). The minimum size of a shared buffer is determined from the MIN_SH_BUF_SZ field.

Bits	Name	Description
24	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.
27:25	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 MFI_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 reserved.
28	RAT_SIGN	Value of this field is used to determine whether EL3 firmware implements the MFI_ATTEST_RAT_SIGN ABI at this MFI instance as follows: • `0b0`: MFI_ATTEST_RAT_SIGN is not supported. • `0b1`: MFI_ATTEST_RAT_SIGN is supported. All other values are reserved.
63:29	Reserved	MBZ.

7.3 MFI_GM_GPI_SET

R0165 The input parameters of the MFI_GM_GPI_SET interface are listed in Table 7.9.

Table 7.9: MFI_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 *Base Address*.
uint32 Attributes	x3	• Attributes of this request. – Bits[63:8]: Reserved. – Bits[7:4]: Current GPI encoding of the specified granules. – Bits[3:0]: Target GPI encoding of the specified granules.

R0166 The output parameters of the MFI_GM_GPI_SET interface are listed in Table 7.10.

Table 7.10: MFI_GM_GPI_SET output parameters

Parameter	Register	Value
int64 Return status	x0	• SUCCESS • NOT_SUPPORTED • INVALID_PARAMETERS • DENIED • RETRY
uint64 Granule count	x1	• Count of granules starting from the first granule at the *Base Address* whose GPI encoding was changed.

7.4 MFI_IDE_KEYSET_PROG

R0167 The input parameters of the MFI_IDE_KEYSET_PROG interface are listed in Table 7.11.

Table 7.11: MFI_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]: Request 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 reserved (MBZ). • Bits[63:2]: Reserved (MBZ).
uint64 Keyset ID	x3	• Keyset ID of the keyset in the Root port.
uint64 KeyQW0	x4	• Quad word[63:0] of AES-GCM 256 bit key.
uint64 KeyQW1	x5	• Quad word[127:64] of AES-GCM 256 bit key.
uint64 KeyQW2	x6	• Quad word[191:128] of AES-GCM 256 bit key.
uint64 KeyQW3	x7	• Quad word[255:192] of AES-GCM 256 bit key.
uint64 Cookie1	x8	• Implementation defined value specified by the caller to track a non-blocking IDE key configuration operation. • MBZ if this parameter is not used by the caller.
uint64 Cookie2	x9	• Implementation defined value specified by the caller to track a non-blocking IDE key configuration operation. • MBZ if this parameter is not used by the caller.

R0168 The output parameters of the MFI_IDE_KEYSET_PROG interface are listed in Table 7.12.

Table 7.12: MFI_IDE_KEYSET_PROG output parameters

Parameter	Register	Value
int64 Return status	x0	• SUCCESS • NOT_SUPPORTED • INVALID_PARAMETERS • RETRY • INCOMPLETE

7.5 MFI_IDE_KEYSET_GO

R0169 The input parameters of the MFI_IDE_KEYSET_GO interface are listed in Table 7.13.

Table 7.13: MFI_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]: Request 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 reserved (MBZ). • Bits[63:2]: Reserved (MBZ).
uint64 Keyset ID	x3	• Keyset ID of the keyset in the Root port.
uint64 Cookie1	x4	• Implementation defined value specified by the caller to track a non-blocking IDE key configuration operation. • MBZ if this parameter is not used by the caller.
uint64 Cookie2	x5	• Implementation defined value specified by the caller to track a non-blocking IDE key configuration operation. • MBZ if this parameter is not used by the caller.

R0170 The output parameters of the MFI_IDE_KEYSET_GO interface are listed in Table 7.14.

Table 7.14: MFI_IDE_KEYSET_GO output parameters

Parameter	Register	Value
int64 Return status	x0	• SUCCESS • NOT_SUPPORTED • INVALID_PARAMETERS • RETRY • DENIED • INCOMPLETE

7.6 MFI_IDE_KEYSET_STOP

R0171 The input parameters of the MFI_IDE_KEYSET_STOP interface are listed in Table 7.15.

Table 7.15: MFI_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]: Request 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 reserved (MBZ). • Bits[63:2]: Reserved (MBZ).
uint64 Keyset ID	x3	• Keyset ID of the keyset in the Root port.
uint64 Cookie1	x4	• Implementation defined value specified by the caller to track a non-blocking IDE key configuration operation. • MBZ if this parameter is not used by the caller.
uint64 Cookie2	x5	• Implementation defined value specified by the caller to track a non-blocking IDE key configuration operation. • MBZ if this parameter is not used by the caller.

R0172 The output parameters of the MFI_IDE_KEYSET_STOP interface are listed in Table 7.16.

Table 7.16: MFI_IDE_KEYSET_STOP output parameters

Parameter	Register	Value
int64 Return status	x0	• SUCCESS • NOT_SUPPORTED • INVALID_PARAMETERS • RETRY • DENIED • INCOMPLETE

7.7 MFI_IDE_KEYSET_POLL

R0173 The input parameters of the MFI_IDE_KEYSET_POLL interface are listed in Table 7.17.

Table 7.17: MFI_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]: Request 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 reserved (MBZ). • Bits[2]: Pending response type. – `0b0`: Return a pending response corresponding to the *Keyset ID* parameter. – `0b1`: Return any pending response and ignore the *Keyset ID* parameter. • Bits[63:3]: Reserved (MBZ).
uint64 Keyset ID	x3	• Keyset ID of the keyset in the Root port.

R0174 The output parameters of the MFI_IDE_KEYSET_POLL interface are listed in Table 7.18.

Table 7.18: MFI_IDE_KEYSET_POLL output parameters

Parameter	Register	Value
int64 Return status	x0	• SUCCESS • NOT_SUPPORTED • INVALID_PARAMETERS • RETRY • INCOMPLETE • INVALID_REQUEST • DENIED
uint64 Cookie1	x4	• Implementation defined value that was specified by the caller to track a non-blocking IDE key configuration operation. • MBZ if this parameter was not used by the caller or if *Return status* is not SUCCESS.
uint64 Cookie2	x5	• Implementation defined value that was specified by the caller to track a non-blocking IDE key configuration operation. • MBZ if this parameter was not used by the caller or if *Return status* is not SUCCESS.

7.8 MFI_MEC_REFRESH

R0175 The input parameters of the MFI_MEC_REFRESH interface are listed in Table 7.19.

Table 7.19: MFI_MEC_REFRESH input parameters

Parameter	Register	Value
uint32 Function ID	w0	• 0xC4000407.
uint64 MEC params	x1	• Bits[63:48]: Reserved (SBZ). • Bits[47:32]: MECID. • Bits[31:1]: Reserved (SBZ). • Bits[0]: MEC refresh reason. – 0b'0: Realm creation. – 0b'1: Realm destruction.

R0176 The output parameters of the MFI_MEC_REFRESH interface are listed in Table 7.20.

Table 7.20: MFI_MEC_REFRESH output parameters

Parameter	Register	Value
int64 Return status	x0	• SUCCESS • NOT_SUPPORTED • INVALID_PARAMETERS • RETRY

7.9 MFI_ATTEST_PAT_GET

R0177 The input parameters of the MFI_ATTEST_PAT_GET interface are listed in Table 7.21.

Table 7.21: MFI_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 *Shared buffer base address* at which the platform attestation token must be written by EL3 firmware.
uint64 Shared buffer size	x3	• Bits[13:0]: Value of this field is used to determine the size of the buffer as follows: – `Size of buffer (in KB)= (Shared buffer size + 1)* Minimum size of a shared buffer (in KB).` – The minimum size of a shared buffer is determined from the `MIN_SH_BUF_SZ` field in MFI feature register 2. • Bits[63:14]: Reserved (MBZ).
uint64 Platform challenge size	x4	• Bits[31:0]: Size (in bytes) of the challenge populated at offset 0x0 from *Shared buffer base address*. • Bits[63:32]: Reserved (MBZ).

R0178 The output parameters of the MFI_ATTEST_PAT_GET interface are listed in Table 7.22.

Table 7.22: MFI_ATTEST_PAT_GET output parameters

Parameter	Register	Value
int64 Return status	x0	• SUCCESS • NOT_SUPPORTED • INVALID_PARAMETERS • RETRY • ABORTED
uint64 Written size	x1	• If *Return status* is SUCCESS, this is the size (in bytes) of the chunk of the platform attestation token starting from the *Write offset* input parameter. • Otherwise, this register is Reserved (MBZ)
uint64 Remaining size	x2	• If *Return status* 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 Reserved (MBZ)

7.10 MFI_ATTEST_RAK_GET

R0179 The input parameters of the MFI_ATTEST_RAK_GET interface are listed in Table 7.23.

Table 7.23: MFI_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 *Shared buffer base address* at which the Realm attestation key must be written by EL3 firmware.
uint64 Shared buffer size	x3	• Bits[13:0]: Value of this field is used to determine the size of the buffer as follows: – `Size of buffer (in KB)= (Shared buffer size + 1)* Minimum size of a shared buffer (in KB).` – The minimum size of a shared buffer is determined from the `MIN_SH_BUF_SZ` field in MFI feature register 2. • Bits[63:14]: Reserved (MBZ).
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. – Reserved (MBZ) if the `Request type` 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. – Reserved (MBZ) if the `Request type` field is 1. • Bits[7:3]: Reserved (MBZ). • Bits[15:8]: Type of elliptic curve. • Bits[63:16]: Reserved (MBZ).

R0180 The output parameters of the MFI_ATTEST_RAK_GET interface are listed in Table 7.24.

Table 7.24: MFI_ATTEST_RAK_GET output parameters

Parameter	Register	Value
int64 Return status	x0	• SUCCESS • NOT_SUPPORTED • INVALID_PARAMETERS • RETRY • ABORTED
uint64 Written size	x1	• If *Return status* is SUCCESS, this is the size (in bytes) of the chunk of the Realm attestation key starting from the *Write offset* input parameter. • Otherwise, this register is Reserved (MBZ)
uint64 Remaining size	x2	• If *Return status* 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 Reserved (MBZ)

7.11 MFI_ATTEST_RAT_SIGN

R0181 The input parameters of the MFI_ATTEST_RAT_SIGN interface are listed in Table 7.25.

Table 7.25: MFI_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 size	x2	• Bits[13:0]: Value of this field is used to determine the size of the buffer as follows: – `Size of buffer (in KB)= (Shared buffer size + 1)* Minimum size of a shared buffer (in KB).` – The minimum size of a shared buffer is determined from the `MIN_SH_BUF_SZ` field in MFI feature register 2. • Bits[63:14]: Reserved (MBZ).
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]: Reserved (MBZ). • 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 Reserved (MBZ).

R0182 The output parameters of the MFI_ATTEST_RAT_SIGN interface are listed in Table 7.26.

Table 7.26: MFI_ATTEST_RAT_SIGN output parameters

Parameter	Register	Value
int64 Return status	x0	• SUCCESS • NOT_SUPPORTED • INVALID_PARAMETERS • RETRY

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 *Return status* is SUCCESS, this is the size (in bytes) of the payload containing the signature. – If *Return status* 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 Reserved (MBZ).

Glossary

ABI

Application Binary Interface

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

MFI

Monitor Firmware Interface

MSD

Monitor Security Domain

OS

Operating System

PE

Processing Element

PGS

Protected Granule Size

RME

Realm Management Extension

RMI

Realm Management Interface

RMM

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

SPM

Secure Partition Manager

TDI

TEE Device Interface

VM

Virtual Machine