Realm Host Interface specification

Document number	DEN0148
Document quality	BET
Document version	1.0-bet1
Document confidentiality	Non-confidential
Document build information	19ee47ac doctool 0.56.1

Realm Host Interface specification

Release information

1.0-bet1 (02-02-2026)

New features

- Added context handles for use with RHI_DA_VDEV_CONTINUE (FENIMORE-1280)
- Added RHI_DA_DEV_EXTENSION_EVIDENCE to RHIDADevCommObject
- Always request signed measurements when supported + always cache a transcript (FENIMORE-1342)
- Remove measurement indices and always request all device measurements (FENIMORE-1367)

Clarifications

- Added definitions for RHI DA ABI Status Codes (FENIMORE-1273)
- Updated definition of RHIVdevMeasurementFlags to match RMM changes (FENIMORE-1276)
- Clarify usage text for RHI_DA_VDEV_GET_MEASUREMENTS (FENIMORE-1274)
- Update definition of RHI_DA_VDEV_P2P_UNBIND to work with two VDEVs
- Add informative Device Attestation appendix (FENIMORE-1332)

Defects

None

Relaxations

None

1.0-bet0 (02-07-2025)

New features

- Updated set of Device Assignment ABIs to reflect new design
- Added Host Configuration chapter (FENIMORE-1054)
- Added RMI_DEV_VCA object type support to the RHI_DA calls (FENIMORE-1029)

Clarifications

- Added clarifications to the threat model for the Boot Injection protocol

Defects

- Modified definition of object types used in RHI Device Assignment calls (FENIMORE-951)
- Removed the RHI_IMPLEMENTATION_FEATURES ABI. Protocol implementation status is now determined using the RHI…VERSION ABIs within each protocol.
- Reordered parameters of RHI_DA_OBJECT_READ for consistency with other ABIs
- Improved consistency for naming of RHI_DA_ERRORs

Relaxations

None

1.0-alp2 (31-10-2024)

New features

- Addition of RHI_FAL_CLOSE ABI

Clarifications

- Alter wording of parameter blocks to description, [value] form.
- Move to consistent 'sync' terminology in Appendix A (previously a mix of 'sync' and 'injection').
- Typo corrections

Defects

- Allocate FIDs within the range reserved for RHI in SMCCC.
- Swap SessionID and connectionType parameters for RHI_SESSION_OPEN to be consistent with other calls in protocol.
- Types within Appendix A renamed to use `BSB` naming instead of `BIB` naming.

Relaxations

None

- Added Device Assignment ABI chapter

1.0-alp1 (19-09-2024)

- Added Device Assignment ABI chapter

Arm Non-Confidential Document License ("License")

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

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

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

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

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

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

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

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

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

The content of this document is informational only. Any solutions presented herein are subject to changing conditions, information, scope, and data. This document was produced using reasonable efforts based on information available as of the date of issue of this document. The scope of information in this document may exceed that which Arm is required to provide, and such additional information is merely intended to further assist the recipient and does not represent Arm's view of the scope of its obligations. You acknowledge and agree that you possess the necessary expertise in system security and functional safety and that you shall be solely responsible for compliance with all legal, regulatory, safety and security related requirements concerning your products, notwithstanding any information or support that may be provided by Arm herein. In addition, you are responsible for any applications which are used in conjunction with any Arm technology described in this document, and to minimize risks, adequate design and operating safeguards should be provided for by you.

Reference by Arm to any third party's products or services within this document is not an express or implied approval or endorsement of the use thereof.

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

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

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

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

Arm document reference: PRE-21585

version 5.0, March 2024

Contents
Realm Host Interface specification

Preface

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] *Realm Management Monitor specification*. (ARM DEN 0137) Arm Limited.

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

[3] *Live Firmware Activation SMC Interface*. (ARM DEN 0147) Arm Ltd.

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

[5] *NIST Special Publication 800-56A. Recommendation for Pair-Wise Key-Establishment Schemes Using Discrete Logarithm Cryptography*. See https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-56Ar3.pdf

[6] *An EAT Profile for Device Attestation*. See https://rats-device-attestation.github.io/draft-poirier-rats-eat-da/draft-poirier-rats-eat-da.html

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

[8] *IANA Hash Function Textual Names*. See https://www.iana.org/assignments/hash-function-text-names/hash-function-text-names.xhtml

[9] *RME system architecture specification*. (ARM DEN 0129) Arm Ltd.

Feedback

Arm welcomes feedback on its documentation.

Feedback on this book

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

- The title (Realm Host Interface specification).
- The number (DEN0148 1.0-bet1).
- 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
Realm Host Interface (RHI)

1.1 Overview

I RHI enables software running within Realms to request specific services or data items from the non-secure host. This is in contrast to the Realm Services Interface (RSI) which provides access to services or data provided by the Realm Management Monitor (RMM) [1].

RHI supports sets of commands dedicated to specific access cases. This specification covers the usage and call patterns for these commands.

1.2 Use cases

I There are use cases where it may be necessary for Realm software to communicate with services executing within the context of the non-secure host responsible for the management of Realms. Use cases may fall into the following categories:

- To access data relevant to the state of the host platform
- To access a service that maps some state or resource controlled by the host to be accessed by the specific realm.

The inclusion of different sets of commands may depend upon the use cases addressed by a deployment. Some command sets have specific uses tied into the programming model required to access deployed features and support use of the Realm Services Interface. Others are more open ended and will require more usage explanation within the context of a given deployment.

Note

Data received from such host calls is provided from outside of the Trusted Computing Base for Realms. As such, it is necessary for a use case specific security mechanism to be available for any data received.

1.3 Transport

R All RHI calls use the RSI_HOST_CALL command as a transport.

I Within the RsiHostCall data structure, the gprs[0] value is an SMCCC Function Identifier (FID) which specifies the function to be called.

See also:

- Realm Management Monitor (RMM) architecture specification [1]
- Arm SMCCC calling convention [2]

I In the description below of the RHI commands, parameters are listed corresponding to members of the RsiHostCall data structure.

Chapter 2
Host Session

G The following collection of calls are used to establish and use a communication channel between the Realm and an external entity. This is a general mechanism, which is specialised for a specific purpose by the data passed through the channel.

I The communication channel is created via a service running in the non-secure host. This service will determine the relevant external entity, for example a client of the host service managing the Realm.

Note that the channel does not provide any security mechanism. Security must be provided within any exchange of messages through the channel.

I For an example of a specific protocol using this communication channel, see Appendix A: Boot Injection

I The communication channel can be created in one of two modes:

- BLOCKING: ABI calls do not return until communications are complete
- NON-BLOCKING: ABI calls return immediately and may need to be repeated until communications are complete.

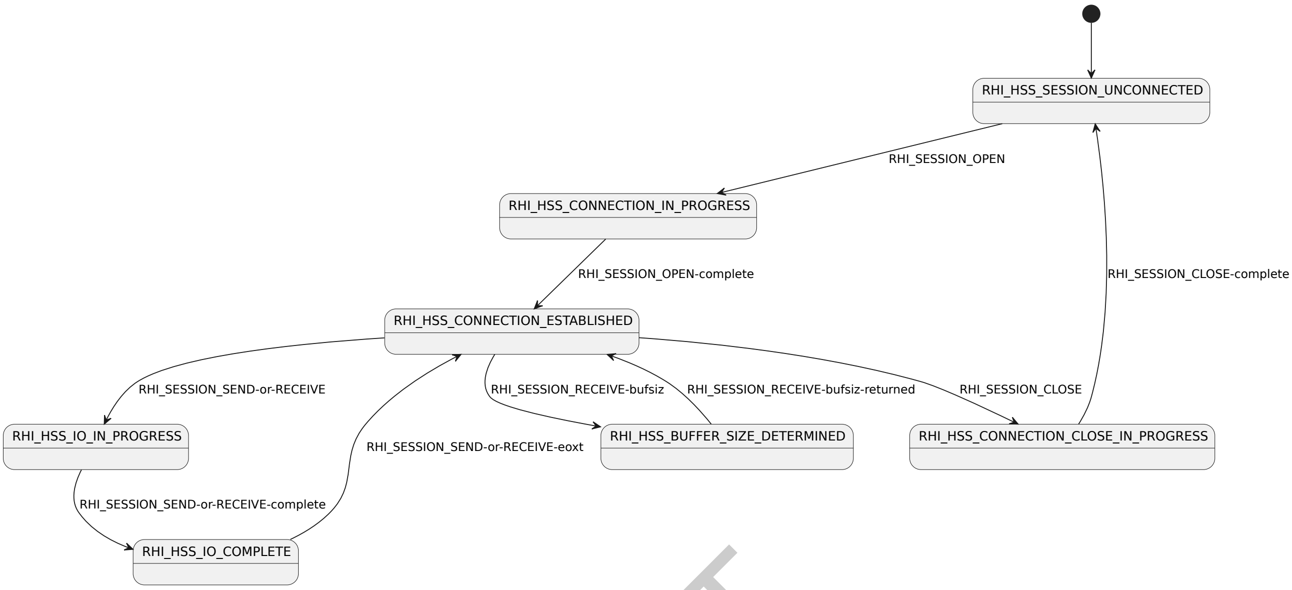
It is IMPLEMENTATION DEFINED whether either communication type is supported

I The Host Session ABIs cause an internal Host Session State (HSS) to be maintained for a connection. This state which can be used to determine when communications are complete. The current state of the connection is held at the realm and user context endpoints of the connection. The states that a connection can be in are:

- RHI_HSS_SESSION_UNCONNECTED
- RHI_HSS_CONNECTION_IN_PROGRESS
- RHI_HSS_CONNECTION_ESTABLISHED
- RHI_HSS_IO_IN_PROGRESS
- RHI_HSS_IO_COMPLETE
- RHI_HSS_BUFFER_SIZE_DETERMINED

- RHI_HSS_CONNECTION_CLOSE_IN_PROGRESS

State Diagram:



2.1 RHI_SESSION Types

2.1.1 RHI Host Session States

State	Value
RHI_HSS_SESSION_UNCONNECTED	0
RHI_HSS_CONNECTION_IN_PROGRESS	1
RHI_HSS_CONNECTION_ESTABLISHED	2
RHI_HSS_IO_IN_PROGRESS	3
RHI_HSS_IO_IN_COMPLETE	4
RHI_HSS_BUFFER_SIZE_DETERMINED	5
RHI_HSS_CONNECTION_CLOSE_IN_PROGRESS	6

2.1.2 RHI Session Connection Type

I Modes which a session can be opened in.

State	Value
BLOCKING	0
NON-BLOCKING	1

2.1.3 RHI Session ABI Status Codes

I The status codes that may be generated by the RHI Session calls are as follows:

Error Code	Value	Notes
RHI_SESS_SUCCESS	0	Call Succeeded
RHI_SESS_PEER_NOT_AVAILABLE	1	communication error to external entity
RHI_SESS_INVALID_STATE_FOR_OPERATION	2	invalid state
RHI_SESS_INVALID_SESSION_ID	3	unknown SessionID parameter
RHI_SESS_ACCESS_FAILED	4	buffer not read/writeable or not granule aligned

2.2 RHI_SESSION_VERSION

The RHI_SESSION_VERSION ABI returns the implemented numeric version of the RHI_SESSION calls within this protocol. For this version of the specification, the return value must be 0x10000, which corresponds to version 1.0.

R The RHI_SESSION_VERSION ABI must be implemented, even if the rest of the RHI_SESSION protocol is not.

2.2.1 Parameters

Argument	Type	Value
Imm	UInt16	0
gprs[0]	UInt64	0xC500_0040

2.2.2 Return values

Argument	Type	Value
gprs[0]	UInt64	Protocol Set version

R If the returned value is SMC_NOT_SUPPORTED (-1), then the whole RHI_SESSION protocol is not supported on this host implementation.

2.3 RHI_SESSION_FEATURES

The features ABI provides implementation details for the RHI_SESSION calls within this protocol.

R If the protocol set is reported as supported, via an appropriate return value from the RHI_SESSION_VERSION ABI, the RHI_SESSION_FEATURES ABI must be implemented.

2.3.1 Parameters

Argument	Type	Value
Imm	UInt16	0
gprs[0]	UInt64	0xC500_0041

2.3.2 Return values

Argument	Type	Value
gprs[0]	UInt64	ABI calls supported bitmap
gprs[1]	UInt64	Connection modes supported bitmap

- ABI calls supported: each bit set in this bitmap indicates that the corresponding call is supported by the host implementation.
  - Bit 0: RHI_SESSION_OPEN supported
  - Bit 1: RHI_SESSION_CLOSE supported
  - Bit 2: RHI_SESSION_SEND supported
  - Bit 3: RHI_SESSION_RECEIVE supported
- Connection Modes supported: indicates which Connection Modes are supported by the host implementation.
  - Bit 0: BLOCKING connection mode supported
  - Bit 1: NON-BLOCKING connection mode supported

2.4 RHI_SESSION_OPEN

Call initiates a communication channel between the Realm and the host determined external entity

2.4.1 Parameters

If the connectionType is NON-BLOCKING, this behaviour also applies to RHI_SESSION_OPEN operations and the call will return immediately before the host has established any connection with the external entity. The protocol state will change to `RHI_HSS_CONNECTION_IN_PROGRESS` to indicate this state. RHI_SESSION_OPEN can then be called again until the protocol state changes to `RHI_HSS_CONNECTION_ESTABLISHED` (or error). For these subsequent calls, the SessionID returned from the first call must be specified. If the protocol state is 'RHI_HSS_SESSION_UNCONNECTED' when the RHI_SESSION_OPEN call is made, then the SessionID parameter is ignored.

If the connectionType is BLOCKING, the call will not return until the host has establised the connection with the external entity (or error).

Argument	Type	Value
Imm	UInt16	0
gprs[0]	UInt64	0xC500_0042
gprs[1]	UInt64	SessionID
gprs[2]	connectionType	BLOCKING or NON-BLOCKING

2.4.2 Return values

Argument	Type	Value
gprs[0]	UInt64	Return Code |
gprs[1]	UInt64	SessionID for the new channel
gprs[2]	UInt64	Protocol state for the session |

2.4.3 Return conditions:

- RHI_SESS_SUCCESS
- RHI_SESS_PEER_NOT_AVAILABLE: host could not establish a connection to external entity
- RHI_SESS_INVALID_STATE_FOR_OPERATION: state is not RHI_HSS_SESSION_UNCONNECTED
- RHI_SESS_INVALID_SESSION_ID: 0 not passed on initial call or unknown SessionID passed (NON_BLOCKING)
- RHI_SESS_CONNECTION_TYPE_NOT_SUPPORTED: the implementation does not support this connection type

2.4.4 Protocol state on return:

- BLOCKING: RHI_HSS_CONNECTION_ESTABLISHED or connection_unconnected (on error)

- NON-BLOCKING: RHI_HSS_CONNECTION_IN_PROGRESS on initial call, this state remains on subsequent calls until RHI_HSS_CONNECTION_ESTABLISHED or connection_unconnected (error)

2.5 RHI_SESSION_CLOSE

Call closes a previously opened communication channel.

2.5.1 Parameters

Argument	Type	Value
Imm	UInt16	0
gprs[0]	UInt64	0xC500_0043
gprs[1]	UInt64	Channel SessionID

2.5.2 Return values

Argument	Type	Value
gprs[0]	UInt64	Return Code
gprs[1]	UInt64	SessionID
gprs[2]	UInt64	Protocol state for the session

2.5.3 Return Conditions:

- RHI_SESS_SUCCESS
- RHI_SESS_INVALID_STATE_FOR_OPERATION: state is RHI_HSS_SESSION_UNCONNECTED
- RHI_SESS_PEER_NOT_AVAILABLE: host could not gracefully close session
- RHI_SESS_INVALID_SESSION_ID: unknown SessionID parameter

2.5.4 Protocol State on return:

- BLOCKING: connection_unconnected (RHI_SESS_SUCCESS) or RHI_HSS_CONNECTION_ESTABLISHED (on error)

- NON-BLOCKING: call will return immediately and state will change to RHI_HSS_CONNECTION_CLOSE_IN_PRO this state remains on subsequent calls until state becomes RHI_HSS_SESSION_UNCONNECTED or an error is returned.

2.6 RHI_SESSION_SEND

Call transmits data on previously opened communication channel. Overlapped calls are not supported.

2.6.0.1 Parameters

Argument	Type	Value
Imm	UInt16	0
gprs[0]	UInt64	0xC500_0044
gprs[1]	UInt64	SessionID of channel
gprs[2]	Address	Realm IPA for buffer containing data, granule aligned
gprs[3]	UInt64	Length of data to send in bytes
gprs[4]	Offset	Offset in buffer from which to send data

2.6.1 Return values

Argument	Type	Value
gprs[0]	UInt64	Return Code |
gprs[1]	UInt64	SessionID |
gprs[2]	UInt64	Protocol state for the session |
gprs[3]	UInt64	Length of data transmitted in bytes

2.6.2 Return Conditions:

- RHI_SESS_SUCCESS: if BLOCKING, operation ended. If NON=BLOCKING, see Protocol State
- RHI_SESS_INVALID_STATE_FOR_OPERATION: must be in session_established_state or RHI_HSS_IO_COMPLET for first call or in RHI_HSS_IO_IN_PROGRESS state (NON-BLOCKING)
- RHI_SESS_INVALID_SESSION_ID: unknown SessionID parameter
- RHI_SESS_PEER_NOT_AVAILABLE: communication error to external entity
- RHI_SESS_ACCESS_FAILED: buffer is not readable or is not granule aligned

2.6.3 Protocol State on return:

- BLOCKING: RHI_HSS_IO_COMPLETE (RHI_SESS_SUCCESS) or RHI_HSS_CONNECTION_ESTABLISHED (on error)

- NON-BLOCKING: RHI_HSS_IO_IN_PROGRESS on initial call, this state remains on subsequent calls until RHI_HSS_IO_COMPLETE (whether RHI_SESS_SUCCESS or error condition)

2.7 RHI_SESSION_RECEIVE

Call reads data from Communication channel.

2.7.1 Parameters

Argument	Type	Value
Imm	UInt16	0x0
gprs[0]	UInt64	0xC500_0045
gprs[1]	UInt64	SessionID of target channel
gprs[2]	Address	Realm IPA for buffer used to receive data, granule aligned
gprs[3]	UInt64	Size of receiving data buffer
gprs[4]	Offset	Offset in buffer where received data is to be written

If gprs[3] (buffer size) and gprs[2] (buffer address) parameters are both set to 0, then the call is intended to determine the size of data buffer required, which will be returned in gprs[2].

If only gprs[3] (buffer size) is 0, then no data will be read.

2.7.2 Return values

Argument	Type	Value
gprs[0]	UInt64	Return Code |
gprs[0]	UInt64	SessionID |
gprs[1]	UInt64	Protocol state for the session |
gprs[2]	UInt64	Data length received (or buffer size required)

2.7.3 Return Conditions:

- RHI_SESS_SUCCESS: if BLOCKING, operation ended. If NON=BLOCKING, see Protocol State
- RHI_SESS_INVALID_STATE_FOR_OPERATION: must be in session_established_state or RHI_HSS_IO_COMPLE for first call or in RHI_HSS_IO_IN_PROGRESS state (NON-BLOCKING)
- RHI_SESS_INVALID_SESSION_ID: unknown SessionID parameter
- RHI_SESS_PEER_NOT_AVAILABLE: communication error to external entity
- RHI_SESS_ACCESS_FAILED: buffer is not writeable or is not granule aligned

2.7.4 Protocol State on return:

- RHI_HSS_BUFFER_SIZE_DETERMINED if 0 values passed for buffer address and size
- BLOCKING: RHI_HSS_IO_COMPLETE (RHI_SESS_SUCCESS) or RHI_HSS_CONNECTION_ESTABLISHED (on error)

- NON-BLOCKING: RHI_HSS_IO_IN_PROGRESS on initial call, this state remains on subsequent calls until RHI_HSS_IO_COMPLETE (whether RHI_SESS_SUCCESS or error condition)

Chapter 3

Firmware Activity Log

3.1 Introduction

G These interfaces are used to obtain the Firmware Activity Log (FAL). This log reflects the Firmware changes made to the CCA Platform. For further details see the Live Firmware Activation specification [3]

I The Firmware Activity Log contains a series of entries that describe changes applied to the firmware that makes up the CCA Platform. The initial state of the firmware deployed on the CCA Platform is captured at boot time. Updates to individual components may be made post boot, which will be captured as additional entries in the Firmware Activity Log. The Firmware Activity Log is maintained within the non-secure host system. The format and contents of the log is IMPLEMENTATION DEFINED and is intended to be passed to an enlightened verifier component to support the trustworthiness appraisal of the CCA Platform token. A typical entry within the log would include at least the following information about a firmware component:

- Component Identity
- Cryptographic measurement of the component in memory
- Identity of the signing authority public key
- The Security Version of the component maintained by Live Firmware Activation

3.1.1 Security Considerations

I The Firmware Activity Log is an important part of the evidence used by a Relying Party to established the trustworthiness of a CCA Platform instance. The RHI interface does not provide any security guarantees for integrity of the data read from the non-secure Host. Data integrity for the log is established by the verifier, using a measurement entry within the CCA Platform attestation token [1]. This measurement is a compound value, established by extending a hash measurement of each entry within the Log. The verifier can recompute this value from the Log entries it receives, and match the attestation report entry to ensure that the log contents are as expected.

3.2 RHI_FAL Types

3.2.1 RHI FAL ABI Status Codes

I The status codes that may be generated by the RHI Firmware Activity Log calls are as follows:

Error Code	Value	Notes
RHI_FAL_SUCCESS	0	Call Succeeded
RHI_FAL_ACCESS_FAILED	1	buffer not writeable or not granule aligned

3.3 RHI_FAL_VERSION

The RHI_FAL_VERSION ABI returns the implemented numeric version of the RHI_FAL calls within this protocol. For this version of the specification, the return value must be 0x10000, which corresponds to version 1.0.

R The RHI_FAL_VERSION ABI must be implemented, even if the rest of the RHI_FAL protocol is not.

3.3.1 Parameters

Argument	Type	Value
Imm	UInt16	0
gprs[0]	UInt64	0xC500_0046

3.3.2 Return values

Argument	Type	Value
gprs[0]	UInt64	Protocol Set version

R If the returned value is SMC_NOT_SUPPORTED (-1), then the whole RHI_FAL protocol is not supported on this host implementation.

3.4 RHI_FAL_FEATURES

The features ABI provides implementation details for the FAL calls within this protocol.

R If the protocol set is reported as supported, via an appropriate return value from the RHI_FAL_VERSION ABI, the RHI_FAL_FEATURES ABI must be implemented.

3.4.1 Parameters

Argument	Type	Value
Imm	UInt16	0
gprs[0]	UInt64	0xC500_0047

3.4.2 Return values

Argument	Type	Value
gprs[0]	UInt64	ABI calls supported bitmap

- ABI calls supported: each bit set in this bitmap indicates that the corresponding call is supported by the host implementation.
  - Bit 0: RHI_FAL_GET_SIZE supported
  - Bit 1: RHI_FAL_READ supported

3.5 RHI_FAL_GET_SIZE

The RHI_FAL_GET_SIZE returns the overall size of the FW Activity Log.

3.5.1 Parameters

Argument	Type	Value
Imm	UInt16	0x0
gprs[0]	UInt64	0xC500_0048

3.5.2 Return values

Argument	Type	Value
gprs[0]	UInt64	Size in bytes of the FW Activity Log

3.6 RHI_FAL_READ

The RHI_FAL_READ fetches the contents of the Firmare Activity Log

3.6.1 Parameters

Argument	Type	Value
Imm	UInt16	0x0
gprs[0]	UInt64	0xC500_0049
gprs[1]	Address	IPA for Receiving Buffer, granule aligned
gprs[2]	UInt64	Size of Receiving Buffer

3.6.2 Return values

Argument	Type	Value
gprs[0]	UInt64	Return code
gprs[1]	UInt64	Size in bytes written to buffer
gprs[2]	UInt64	Remaining size of FAL in bytes to be read

- Return Code:
  - RHI_FAL_SUCCESS
  - RHI_FAL_ACCESS_FAILED: buffer is not writable or not granule aligned

The ABI returns the number of bytes written to the buffer in gprs[1]. This may be fewer than the buffer size if the write is the last portion of the log to be transferred. If the number of bytes written is 0, this indicates that the full log has been transferred.

Chapter 4
Device Assignment

4.1 Introduction

G Device Assignment (DA) This protocol set supplies ABI calls used to retrieve Device evidence cached within the Non Secure Host.

I To ensure that data is available in the host to be retrieved by these calls, consult the 'Realm Device Assignment' chapter of [1] which documents the relevant interactions.

4.2 DA Types

4.2.1 RHIDADevCommObject

The RHI Device Assignment ABIs are used to access multiple object types. The supported types of object are:

Object	Value
RHI_DA_DEV_VCA	0
RHI_DA_DEV_CERTIFICATE	1
RHI_DA_DEV_MEASUREMENTS	2
RHI_DA_DEV_INTERFACE_REPORT	3
RHI_DA_DEV_EXTENSION_EVIDENCE	4

RHI_DA_DEV_VCA objects have PDEV scope.

RHI_DA_DEV_CERTIFICATE objects have PDEV scope.

RHI_DA_DEV_MEASUREMENTS objects have VDEV scope.

RHI_DA_DEV_INTERFACE_REPORT objects have VDEV scope.

4.2.2 RHIDAVDevTDIState

The RHIDAVDevTDIState states correspond to the TDISP states as defined in the PCIe Specification [4]

State	Value
RHI_DA_TDI_CONFIG_UNLOCKED	0
RHI_DA_TDI_CONFIG_LOCKED	1
RHI_DA_TDI_CONFIG_RUN	2

4.2.3 RHIVdevMeasurementFlags

The RHIVdevMeasurementFlags type is a 64 bit bitfield with the following fields defined.

Member	Bit Position	Description (if bit set)
raw	0	Request a raw bitstream in the response
	1:63	Reserved (SBZ)

4.2.4 RHIVdevGetMeasurementParams

The members of the RmiVdevMeasureParams structure are shown in the following table.

Member	Byte Offset	Type	Descripton
flags	0	RHIVdevMeasurementFlags	Measurement protocol choices
nonce	0x100	256 bit field	nonce value

The `nonce` field provides the nonce value that is used in a signed protocol exchange with the device.

4.2.5 RHI DA ABI Status Codes

I The status codes that may be generated by the RHI DA calls are as follows:

Error Code	Value	Notes
RHI_DA_SUCCESS	0	Call Succeeded
RHI_DA_ERROR_INCOMPLETE	1	Call sequence not yet finished
RHI_DA_ERROR_DATA_NOT_AVAILABLE	2	Data not yet present
RHI_DA_ERROR_INVALID_VDEV_ID	3	VDEV_ID not found
RHI_DA_ERROR_INVALID_OBJECT	4	Invalid Object value
RHI_DA_ERROR_INPUT	5	Invalid Parameter
RHI_DA_ERROR_DEVICE	6	Error in device state
RHI_DA_ERROR_INVALID_OFFSET	7	Offset invalid for buffer size
RHI_DA_ERROR_ACCESS_FAILED	8	Invalid address for IPA
RHI_DA_ERROR_BUSY	9	Device cannot accept command
RHI_DA_ABORTED_OPERATION_HAD_COMPLETED	10	Aborted command had completed
RHI_DA_ERROR_OBJECT_UNSUPPORTED	11	Object type not implemented

4.3 RHI_DA_VERSION

The RHI_DA_VERSION ABI returns the implemented numeric version of the RHI_DA calls within this protocol. For this version of the specification, the return value must be 0x10000, which corresponds to version 1.0.

R The RHI_DA_VERSION ABI must be implemented, even if the rest of the RHI_DA protocol is not.

4.3.1 Parameters

Argument	Type	Value
Imm	UInt16	0
gprs[0]	UInt64	0xC500_004A

4.3.2 Return values

Argument	Type	Value
gprs[0]	UInt64	Protocol Set version

R If the returned value is SMC_NOT_SUPPORTED (-1), then the whole RHI_DA protocol is not supported on this host implementation.

4.4 RHI_DA_FEATURES

The features ABI provides implementation details for the DA calls within this protocol.

R If the protocol set is reported as supported, via an appropriate return value from the RHI_DA_VERSION ABI, the RHI_DA_FEATURES ABI must be implemented.

4.4.1 Parameters

Argument	Type	Value
Imm	UInt16	0
gprs[0]	UInt64	0xC500_004B

4.4.2 Return values

Argument	Type	Value
gprs[0]	UInt64	ABI calls supported bitmap

- ABI calls supported: each bit set in this bitmap indicates that the corresponding call is supported by the host implementation.
  - Bit 0: RHI_DA_OBJECT_SIZE supported.
  - Bit 1: RHI_DA_OBJECT_READ supported.
  - Bit 2: RHI_DA_VDEV_CONTINUE supported.
  - Bit 3: RHI_DA_VDEV_GET_MEASUREMENTS supported.
  - Bit 4: RHI_DA_VDEV_GET_INTERFACE_REPORT supported.
  - Bit 5: RHI_DA_VDEV_SET_TDI_STATE supported.
  - Bit 6: RHI_DA_VDEV_P2P_UNBIND supported.
  - Bit 7: RHI_DA_VDEV_ABORT supported.

4.5 RHI_DA_VDEV_CONTINUE

Many of the commands in this RHI_DA set are long running. In these cases, the host may return an RHI_DA_ERROR_INCOMPLETE status in response to the initial RHI call made. When this happens, the realm code should enter a loop calling RHI_DA_VDEV_CONTINUE until a statue of RHI_DA_SUCCESS is received, indicating the operation is complete.

Pseudo code for such operations:

```
ret, handle = RHI_DA_xxx(vdev_id, ...);
if (ret == RHI_DA_ERROR_INVALID_VDEV_ID || ret == ...other error states )
{
    // bad parameters, error exit
}
do {
    ret = RHI_DA_VDEV_CONTINUE(vdev_id, handle);
} while (ret == RHI_DA_ERROR_INCOMPLETE);

if (ret != RHI_DA_SUCCESS)
{
    // error condition experienced during communications with device
}
```

4.5.1 Parameters

Argument	Type	Value
Imm	UInt16	0x0
gprs[0]	UInt64	0xC500_0051
gprs[1]	UInt64	VDEV_ID
gprs[2]	UInt64	handle from initiating call

VDEV_ID is the ID the host used for VDEV_CREATE.

The handle parameter is received from the call that initiated the continue loop. The value of this parameter is opaque to the caller.

4.5.2 Return values

Argument	Type	Value
gprs[0]	UInt64	Return code

- Return Code:
  - RHI_DA_SUCCESS: command has completed successfully
  - RHI_DA_ERROR_INCOMPLETE: command not yet complete, a further RHI_DA_VDEV_CONTINUE command is required
  - RHI_DA_ERROR_INVALID_VDEV_ID: VDEV_ID parameter not found
  - RHI_DA_ERROR_DEVICE: error in device state

4.6 RHI_DA_VDEV_GET_MEASUREMENTS

The RHI_DA_VDEV_GET_MEASUREMENTS is sent as a request to the host to initiate a command sequence that will result in measurement data being requested from the device. This data will be cached in the host for later retrieval via the RHI_DA_OBJECT_READ ABI. This may be a long running command and must follow the device communication loop, see RHI_DA_VDEV_CONTINUE.

4.6.1 Parameters

Argument	Type	Value
Imm	UInt16	0x0
gprs[0]	UInt64	0xC500_0052
gprs[1]	UInt64	VDEV_ID
gprs[2]	UInt64	IPA of RHIVdevGetMeasurementParams

4.6.2 Return values

Argument	Type	Value
gprs[0]	UInt64	Return code
gprs[1]	UInt64	caller handle for continue loop

- Return Code:
  - RHI_DA_ERROR_INCOMPLETE: command is accepted, Realm to follow the device communication loop, see RHI_DA_VDEV_CONTINUE
  - RHI_DA_ERROR_INVALID_VDEV_ID: VDEV_ID parameter not found.
  - RHI_DA_ERROR_ACCESS_FAILED: invalid address for IPA
  - RHI_DA_ERROR_INPUT: invalid parameter
  - RHI_DA_ERROR_DEVICE: error in device state
  - RHI_DA_ERROR_BUSY: device cannot accept command at this time, retry the call

See also: RHIVdevGetMeasurementParams

4.7 RHI_DA_VDEV_GET_INTERFACE_REPORT

The RHI_DA_VDEV_GET_INTERFACE_REPORT is sent as a request to the host to initiate a command sequence that will result in the TDSIP Device Interface Report being requested from the device. This data will be cached in the host for later retrieval via the RHI_DA_OBJECT_READ ABI. This may be a long running command and must follow the device communication loop, see RHI_DA_VDEV_CONTINUE.

4.7.1 Parameters

Argument	Type	Value
Imm	UInt16	0x0
gprs[0]	UInt64	0xC500_0053
gprs[1]	UInt64	VDEV_ID

4.7.2 Return values

Argument	Type	Value
gprs[0]	UInt64	Return code
gprs[1]	UInt64	caller handle for continue loop

- Return Code:
  - RHI_DA_ERROR_INCOMPLETE: command is accepted, Realm to follow the device communication loop, see RHI_DA_VDEV_CONTINUE
  - RHI_DA_ERROR_INVALID_VDEV_ID: VDEV_ID parameter not found.
  - RHI_DA_ERROR_BUSY: device cannot accept command at this time, retry the call

4.8 RHI_DA_VDEV_SET_TDI_STATE

The RHI_DA_VDEV_SET_TDI_STATE ABI requests a TDI state change for the specified VDEV_ID.

VDEV_ID is the ID the host used for VDEV_CREATE.

4.8.1 Parameters

Argument	Type	Value
Imm	UInt16	0x0
gprs[0]	UInt64	0xC500_0054
gprs[1]	UInt64	VDEV_ID
gprs[2]	RHIDAVDevTDIState	Target state

4.8.2 Return values

Argument	Type	Value
gprs[0]	UInt64	Return code
gprs[1]	UInt64	caller handle for continue loop

- Return Code:
  - RHI_DA_SUCCESS: command has completed successfully, in this case the Realm does not need to follow the device communication loop
  - RHI_DA_ERROR_INCOMPLETE: command is accepted, Realm to follow the device communication loop, see RHI_DA_VDEV_CONTINUE
  - RHI_DA_ERROR_INVALID_VDEV_ID: VDEV_ID parameter not found
  - RHI_DA_ERROR_INVALID_OBJECT: invalid RHIDAVDevTDIState type
  - RHI_DA_ERROR_BUSY: device cannot accept command at this time, retry the call

See also: RHIDAVDevTDIState

4.9 RHI_DA_VDEV_P2P_UNBIND

The RHI_DA_VDEV_P2P_UNBIND ABI initiates a request to unbinds a P2P consent from the specified VDEV_IDs. This may be a long running command and must follow the device communication loop, see RHI_DA_VDEV_CONTINUE. The device communication loop must be completed for both passed VDEV_ID parameters before the command is completed.

4.9.1 Parameters

Argument	Type	Value
Imm	UInt16	0x0
gprs[0]	UInt64	0xC500_0055
gprs[1]	UInt64	VDEV_ID_1
gprs[2]	UInt64	VDEV_ID_2

4.9.2 Return values

Argument	Type	Value
gprs[0]	UInt64	Return code
gprs[1]	UInt64	caller handle for continue loop

- Return Code:
  - RHI_DA_ERROR_INCOMPLETE: command is accepted, Realm to follow the device communication loop, see RHI_DA_VDEV_CONTINUE
  - RHI_DA_ERROR_INVALID_VDEV_ID: VDEV_ID parameter not found
  - RHI_DA_ERROR_BUSY: device cannot accept command at this time, retry the call

4.10 RHI_DA_VDEV_ABORT

The RHI_DA_VDEV_ABORT can be used to terminate the device communication loop used for long running commands.

VDEV_ID is the ID the host used for VDEV_CREATE.

4.10.1 Parameters

Argument	Type	Value
Imm	UInt16	0x0
gprs[0]	UInt64	0xC500_0056
gprs[1]	UInt64	VDEV_ID

4.10.2 Return values

Argument	Type	Value
gprs[0]	UInt64	Return code

- Return Code:
  - RHI_DA_SUCCESS: the host terminated the current command for the vdev
  - RHI_DA_ABORTED_OPERATION_HAD_COMPLETED: where the command that was requested to abort would result in a state change on the device, this error code notifies the caller that the command had completed before the operation was aborted and the state change will have occured on the device
  - RHI_DA_ERROR_INVALID_VDEV_ID: VDEV_ID parameter not found

4.11 RHI_DA_OBJECT_SIZE

The RHI_DA_OBJECT_SIZE ABI returns the size of the relevant object for the requested VDEV_ID. VDEV_ID is the ID the host used for VDEV_CREATE.

4.11.1 Parameters

Argument	Type	Value
Imm	UInt16	0x0
gprs[0]	UInt64	0xC500_004C
gprs[1]	UInt64	VDEV_ID
gprs[2]	RHIDADevCommObject	Type of the target object

4.11.2 Return values

Argument	Type	Value
gprs[0]	UInt64	Return code
gprs[1]	UInt64	Size in bytes of the object

- Return Code:
  - RHI_DA_SUCCESS.
  - RHI_DA_ERROR_INVALID_VDEV_ID: VDEV_ID parameter not found.
  - RHI_DA_ERROR_INVALID_OBJECT: invalid RHIDADevCommObject type.
  - RHI_DA_ERROR_OBJECT_UNSUPPORTED: the supplied RHIDADevCommObject type is not supported on this implementation.
  - RHI_DA_ERROR_DATA_NOT_AVAILABLE: the data for this object is not available, the operation may need to be retried.

4.12 RHI_DA_OBJECT_READ

The RHI_DA_OBJECT_READ ABI fetches a relevant object for a requested VDEV_ID and writes it to an offset within a buffer in NS PAS. VDEV_ID is the ID the host used for VDEV_CREATE.

4.12.1 Parameters

VDEV_ID is the ID the host used for VDEV_CREATE.

Argument	Type	Value
Imm	UInt16	0x0
gprs[0]	UInt64	0xC500_004D
gprs[1]	UInt64	VDEV_ID
gprs[2]	RHIDADevCommObject	Type of the target object
gprs[3]	UInt64	IPA of buffer for read object
gprs[4]	UInt64	Max length of buffer memory
gprs[5]	UInt64	Offset within buffer where data copy starts

4.12.2 Return values

Argument	Type	Value
gprs[0]	UInt64	Return code
gprs[1]	UInt64	Size in bytes of data read

- Return Code:
  - RHI_DA_SUCCESS.
  - RHI_DA_ERROR_INVALID_VDEV_ID: VDEV_ID parameter not found.
  - RHI_DA_ERROR_INVALID_OBJECT: invalid RHIDADevCommObject type.
  - RHI_DA_ERROR_OBJECT_UNSUPPORTED: the supplied RHIDADevCommObject type is not supported on this implementation.
  - RHI_DA_ERROR_DATA_NOT_AVAILABLE: the data for this object is not available, the operation may need to be retried.
  - RHI_DA_ERROR_INVALID_OFFSET: invalid offset for object size. Either offset is beyond buffer size or object will not fit into remaining buffer space.
  - RHI_DA_ERROR_ACCESS_FAILED: invalid address for IPA.

Chapter 5
Host Configuration

5.1 Introduction

G These interfaces are used to obtain the information about the configuration of the host, relevant to Realm code activity.

5.2 RHI_HOSTCONF_VERSION

The RHI_HOSTCONF_VERSION ABI returns the implemented numeric version of the RHI_HOSTCONF calls within this protocol. For this version of the specification, the return value must be 0x10000, which corresponds to version 1.0.

R The RHI_HOSTCONF_VERSION ABI must be implemented, even if the rest of the RHI_HOSTCONF protocol is not.

5.2.1 Parameters

Argument	Type	Value
Imm	UInt16	0
gprs[0]	UInt64	0xC500_004E

5.2.2 Return values

Argument	Type	Value
gprs[0]	UInt64	Protocol Set version

R If the returned value is SMC_NOT_SUPPORTED (-1), then the whole RHI_HOSTCONF protocol is not supported on this host implementation.

5.3 RHI_HOSTCONF_FEATURES

The features ABI provides implementation details for the HOSTCONF calls within this protocol.

R If the protocol set is reported as supported, via an appropriate return value from the RHI_HOSTCONF_VERSION ABI, the RHI_HOSTCONF_FEATURES ABI must be implemented.

5.3.1 Parameters

Argument	Type	Value
Imm	UInt16	0
gprs[0]	UInt64	0xC500_004F

5.3.2 Return values

Argument	Type	Value
gprs[0]	UInt64	ABI calls supported bitmap

- ABI calls supported: each bit set in this bitmap indicates that the corresponding call is supported by the host implementation.
  - Bit 0: RHI_HOSTCONF_GET_IPA_CHANGE_ALIGNMENT supported

5.4 RHI_HOSTCONF_GET_IPA_CHANGE_ALIGNMENT

The RHI_HOSTCONF_GET_IPA_CHANGE_ALIGNMENT call discovers the required alignment for IPA range requests for changes to the properties of its IPA space. These include changes to Realm IPA State (RIPAS) and Stage 2 Access Permissions (S2AP).

5.4.1 Parameters

Argument	Type	Value
Imm	UInt16	0x0
gprs[0]	UInt64	0xC500_0050

5.4.2 Return values

Argument	Type	Value
gprs[0]	UInt64	Required alignment. This must be a multiple of 4KB

Part A
Appendix A

Boot Injection

Introduction

This section describes a protocol for injecting parameters or secret blocks during Realm boot. This 'BIB Protocol' uses the RHI_SESSION_* calls as a transport.

The use case for this BIB protocol is to inject values used in the early boot sequence of Realm firmware. Such values could be UEFI parameters or secrets used to protect user data. Such a BIB protocol can be required as:

- there is no network stack available in the early stages of boot
- in order to maintain confidential computing guarantees, these values need to be provided by the User Context of the Realm initiator, which is known by the host service context, but difficult to establish for the Realm code at this early stage of boot

In this BIB protocol, messages are exchanged between Realm code and a User Context using a transport provided by the Host Session RHI calls. The messages have a common encoding, which includes an identifier for the type of message defined within the BIB protocol and a message specific data block. The BIB protocol does not identify or restrict the types of values transferred, this can be use case specific. As the Host Interface provides no security guarantees with respect to any data passed, much of the BIB protocol described here bootstraps an encrypted channel between the endpoints to protect the confidentiality and integrity of data transferred through the BIB protocol.

System Overview

There are multiple phases to the communication through the BIB protocol.

- In the first phase, the two communicating parties calculate the secrets required to create a Diffie-Hellman (DH) [5] key exchange and share them with each other.
- In the second phase, once shared keys have been calculated, the Realm sends the User Context an Attestation Report that the User Context can validate to confirm that communication is really to a Realm.
- After this point, the channel can be used to supply the relevant data required to specialise the boot sequence, using the shared key to protect the data.

Phase 1 - protected channel establishment

Within the Realm code, appropriate parameters are chosen for the Diffie-Hellman exchange. The Realm then generates a random value and uses the chosen DH parameters to create a public value `R` The Realm code uses the RHI_SESSION_OPEN ABI to open a Host controlled session to the User Context. The host service is responsible for routing the message connection to a relevant service.

In the communication flow description below '*Sent*' refers to the the Realm code using the RHI_SESSION_SEND ABI and '*Receive state*' refers to the Realm code using the RHI_SESSION_RECEIVE ABI, both referencing the SessionID received from the RHI_SESSION_OPEN. All messages within the BIB protocol use data structures with a common header to encode the message type.

The initial message *Sent* uses a KEY_XCHG_RESP data structure. This contains the DH parameters and the value of `R`. The User Context service uses the key exchange request parameters provided, along with another generated random value, to create a public value `U`. The public values `U` and `R` are used to compute a shared secret `K`. That `K` is used to derive an encryption key, `KE` and a binding key, `KB`. The User Context prepares a KEY_XCHG_RESP data structure containing the value of `U`, and responds with this data to the non-secure service. The non-secure service forwards on this data to the Realm. In the meantime, the Realm is in *Receive state* until the KEY_XCHG_RESP data structure is delivered. Once available, `U` is extracted and the Realm performs the same cryptographic operations as the User Context (using `U` and `R` to compute `K`, then deriving `KE` and `KB`). Both end parties in the communication should now have identical keys which can be used to protect communication in further phases.

Phase 2 - establishment of trust

Once the communication pathway is established, the User Context can establish the trustworthiness of the Realm.

The Realm code uses the RSI_ATTESTATION_TOKEN_INIT and RSI_ATTESTATION_TOKEN_CONTINUE ABIs to obtain an Attestation Report (`AR`). The RSI_ATTESTATION_TOKEN_INIT ABI requires a `Challenge` ↪ parameter that establishes the context for the Report request. In this BIB protocol the `Challenge` used is a hash of the computed binding key (`KB`). The hash algorithm used here is fixed for the overall BIB protocol as it will also be used within the User Context. The resultant report is encrypted using the computed encryption key (`KE`) and a fixed algorithm for the BIB protocol. The encrypted data is packaged into a BOOT_SYNC_ATTESTATION_REQUEST data structure and *Sent* to the User Context. The User Context extracts the data from the BOOT_SYNC_ATTESTATION_REQUEST data structure and uses its computed encryption key, `KE`, to decrypt `AR`. `AR` is then verified for correctness using an Attestation Service. As well as overall correctness of `AR` the verification operation should check:

- that `AR` was produced by a CCA Platform determined to be trustworthy by an applicable policy
- that code measurements within `AR` match those for the expected (guest firmware) code in the Realm
- that `Challenge` claim within `AR` matches a hash of the binding key, `KB`, that the User Context computed.

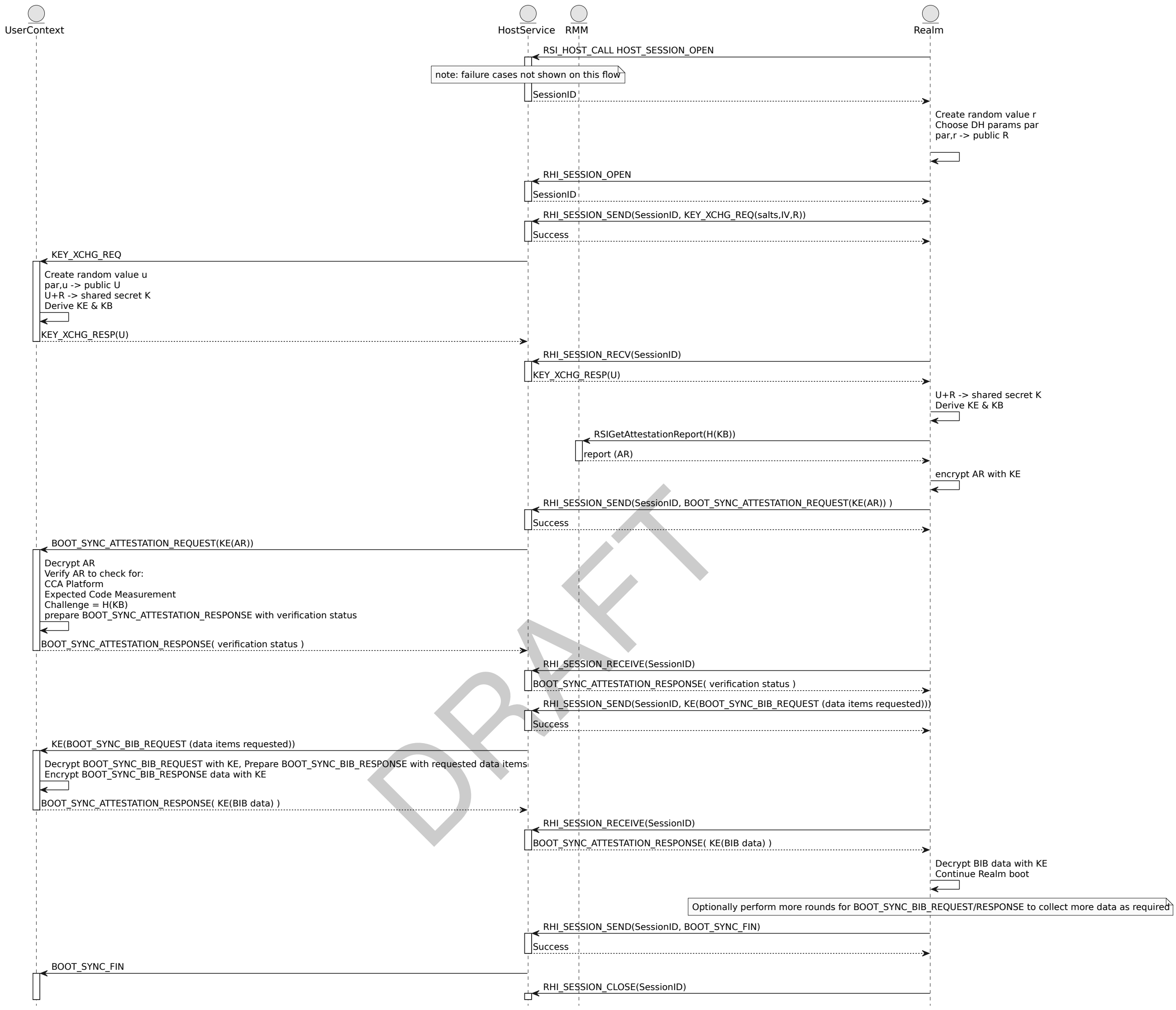
If all of the above checks pass, the User Context can determine that the Realm is trustworthy to received the (confidential) Boot Data. The verification status, and optionally data from the verification result are packaged in a BOOT_SYNC_ATTESTATION_RESPONSE data structure. The Realm code waits in *Receive State* for the confirmation that the trust exchange has been approved.

Phase 3 - transfer of boot information

The Realm can now make requests for the data required to continue the boot process (e.g UEFI variable data or secrets required to access encrypted content). Data requests are made by the Realm having *Sent* a BOOT_SYNC_BIB_REQUEST data structure and waiting in *Receive State* for a BOOT_SYNC_BIB_RESPONSE data structure to be sent from the User Context. The BOOT_SYNC_BIB_REQUEST data structure can have a variable number of elements, depending upon the data required to continue the Realm boot. This data can be supplied in a single communication round or in multiple communication rounds.

Once all communications are complete, a BOOT_SYNC_FIN data structure can be *Sent* by the Realm to signal that no more data is required. No reply is expected to this message.

Flow Diagram



Security Considerations

The data path established by the RHI HOST_SESSION… ABI calls provides no security guarantees. Any message on this path could be inspected or modified by the host infrastructure that creates the communication channel. To mitigate this, the first phase of the BIB protocol establishes a secure channel, using a Diffie-Hellman exchange such that the two ends of the channel both compute protection keys without the intermediate party knowing these.

Diffie-Hellman exchanges are susceptible to person-in-the-middle attacks, where a component between the endpoints establishes separate secure connections with each party but then has access to all the communications traffic. To mitigate this possibility, the Realm supplies an Attestation Report, with bound data obtained from a binding key computed from the DH shared secret. Verification of the Attestation Report will confirm that it has not been tampered with in transmission and the bound data can be checked to ensure it matches the User Context value for the binding key. This mitigation allows detection of the above attack as it would result in different binding keys for each of the end parties.

The threat model for the BIB protocol may include data injection from an in-the-middle attacker that could insert malfaesant code in order to reveal later supplied data. To mitigate this, the BIB protocol can work in multiple rounds. In this mode, the Realm state is updated with measurements of data supplied (apparently) from the User Context and further attestation reports are supplied in sequence. This allows the User Context to further validate the state of the Realm before supplying sensitive data.

Verification of the Attestation Report also allows the User Context to confirm that the other end of the connection is running under a CCA Platform and with expected software content in the Realm.

Note that the threat model for this protocol does not mitigate against running on a deployment where an attacker has full control of attestation.

System Message Data Structures

Cryptography Algorithm considerations

The data structures below are structured to support a BIB protocol using the following cryptographic algorithms. If other algorithms are used to construct a similar BIB protocol then it is likely that some of the fields would change. The likely candidates for alteration have the variable names below annotated with (*). Array sizes are as appropriate for the algorithms used.

Algorithms:

- ECDH key using the ECC Curve-P384 for key exchange
- AEAD AES-GCM for encryption
- SHA512 HMAC-based Extract-and-Expand Key Derivation Function (HKDF) for key Derivation
- SHA256 for IV rolling hash

EFI_GUID

The BIB protocol reuses the EFI_GUID structure to define a 128-bit unique identifier value. EFI_GUID is commonly defined in several code bases.

Type	Variable
UINT32	Data1
UINT16	Data2
UINT16	Data3
UINT8[8]	Data4

BOOT_SYNC_GUID_BLOB

Common data structure prefix for all BIB protocol messages

Type	Variable	Notes
EFI_GUID	Name	GUID identifying message type
UINT32	Length	Length in bytes of full data message

GUID definitions for BIB protocol messages

Implementation GUIDs used to for message identities within the following data structures.

GUID Name	Value
gArmBootSyncKeyEncData	
gArmBootSyncKeyXchgReqGuid	BBD5E1D8-C8E9-48CB-A850-A30B15D08A22 |
gArmBootSyncKeyXchgRespGuid	D83BF2F8-6B49-4238-859D-7E4C29150995 |
gArmBootSyncAttReqGuid	A31E8A1B-5D80-4336-8C0D-6F653B0CC8D1 |
gArmBootSyncAttRespGuid	1B3C3C27-51E1-4D1B-9C44-189EAEA48263 |
gArmBootSyncAttReport	59FC4FCE-B2B2-4DB6-A0CD-3053D9F49738 |
gArmBootSyncAttResult	FE4A5C90-FEC5-4029-B515-699A872E3B01 |
gArmBootSyncBibReqGuid	60E65392-591A-43A4-98E8-257985B9FEB0 |
gArmBootSyncBibRespGuid	0DA1DE44-D38D-40E2-9D0B-A4BAD90B1A5A |
gArmBootSyncRequestOptions	54E1D918-311F-4F3F-B775-9A74A039C438 |
gArmBootSyncVarData	1DB974DF-3F49-44EB-B324-3CB7BA00F589 |
gArmBootSyncSecretData	9755286D-E064-41B4-8FC2-54101280525C |
gArmBootSyncFinGuid	AD96854E-794C-43CA-B91D-61DE98313E45 |
gArmBootSyncNackGuid	7731492A-093E-49ED-8A54-8AD6B8CEC450 |

BOOT_SYNC_ENCRYPTED_DATA

Common data structure prefix for all BIB protocol messages that contain an encrypted payload. This data structure will be followed by EncDataLength bytes of encrypted data that must be decoded with the BIB protocol `KE` before being interpreted as the relevant payload data structure.

Type	Variable	Notes
GUID_BLOB		GUID: gArmBootSyncKeyEncData
UINT8[TAG_SIZE]	Tag*	AES GCM tag |
UINT32	EncDataLength	(encrypted) data size beyond this header

BOOT_SYNC_BSB_HEADER

Common data structure used to define the structure of a multi element data block used in the BIB protocol. The data structure identifies the type and the number of elements contained within the message. This data structure will be followed by ElementCount number of data structures which will need interpreting per identified type.

Type	Variable	Notes
GUID_BLOB		GUID: identifies request or response message
UINT32	ElementCount	Number of following BOOT_SYNC_BSB_ELEMENT structures

BOOT_SYNC_BSB_ELEMENT

Common data structure used to identify and package a data packet within a BIB protocol message This data structure will be followed by the data for the payload (length of message is within GUID_BLOB).

Type	Variable	Notes
GUID_BLOB		GUID identifies the relevant payload

BOOT_SYNC_KEY_XCHG_REQ

Datastructure sent from Realm to User Context to initiate the BIB protocol Key Exchange

Type	Variable	Notes
GUID_BLOB		GUID: gArmBootSyncKeyXchgReqGuid
UINT32	Version	Protocol version in Realm
UINT8[SALT_SIZE]	SaltKeyBinding*	Salt for Binding Key
UINT8[SALT_SIZE]	SaltKeyEncryption*	Salt for Encryption Key
UINT8[IV_SIZE]	IV*	Initialisation vector
UINT32	PEMDataLen	Size of Realm DH Public value (PEM format)
UINT8[]	PEMData	The fixed fields of this structure are followed by PEMDataLen bytes of PEM data

BOOT_SYNC_KEY_XCHG_RESP

Data structure sent from User Context to Realm to finalise the BIB protocol Key Exchange

Type	Variable	Notes
GUID_BLOB		GUID: gArmBootSyncKeyXchgRespGuid
UINT32	Version	Protocol version in User Context
UINT32	PEMDataLen	Size of User Context DH Public value (PEM format)
UINT8[]	PEMData	The fixed fields of this structure are followed by PEMDataLen bytes of PEM data

BOOT_SYNC_ATTESTATION_REQUEST

Datastructure sent from Realm to User Context, containing the Attestation Report that can be verified to perform the trustworthiness step.

Type	Variable	Notes
BOOT_SYNC_ENCRYPTED_DATA	EncDataHeader	
BOOT_SYNC_BSB_HEADER	BSBHeader	GUID: gArmBootSyncAttReqGuid

Type	Variable	Notes
BOOT_SYNC_BSB_ELEMENT	Element[0]	GUID: gArmBootSyncAttReport

Note: additional elements would be supplied if extra information is requires to supplement the Attestation Report

BOOT_SYNC_ATTESTATION_RESPONSE

Datastructure sent from User Context to Realm, containing the results of verifying the Attestation Report supplied in the corresponding BOOT_SYNC_ATTESTATION_REQUEST structure. This response could just be a Boolean SUCCESS / FAILURE or may contain a more detailed attestation result analysis.

Type	Variable	Notes
BOOT_SYNC_ENCRYPTED_DATA	EncDataHeader	
BOOT_SYNC_BSB_HEADER	BSBHeader	GUID: gArmBootSyncAttRespGuid
BOOT_SYNC_BSB_ELEMENT	Element[0]	GUID: gArmBootSyncAttResult

BOOT_SYNC_BIB_REQUEST

Datastructure sent from Realm to User Context to request the config data items required to continue boot.

Type	Variable	Notes
BOOT_SYNC_ENCRYPTED_DATA	EncDataHeader	
BOOT_SYNC_BSB_HEADER	BSBHeader	GUID: gArmBootSyncBibReqGuid
BOOT_SYNC_BSB_ELEMENT	Element[0]	GUID: gArmBootSyncRequestOptions
UNIT64	Options	Identifies the data items required

BOOT_SYNC_BIB_RESPONSE

Datastructure sent from User Context to Realm in response to the BOOT_SYNC_BIB_REQUEST message and containing the requested items. (Typical element content shown as example).

Type	Variable	Notes
BOOT_SYNC_ENCRYPTED_DATA	EncDataHeader	
BOOT_SYNC_BSB_HEADER	BSBHeader	GUID: gArmBootSyncBibRespGuid
BOOT_SYNC_BSB_ELEMENT	Element[0]	GUID: gArmBootSyncVarData
UINT8[Element[0].DataLength]	data	UEFI Variables data
BOOT_SYNC_BSB_ELEMENT	Element[1]	GUID: gArmBootSyncSecretData
UINT8[Element[1].DataLength]	data	Secrets data

BOOT_SYNC_FIN

Data structure sent from either party in the BIB protocol requesting termination of communications.

Type	Variable	Notes
GUID_BLOB		gArmBootSyncFinGuid
UINT64 | Reason		Connection termination reason

Expected termination values for the Reason variable: SUCCESS - the protocol is terminating gracefully.

BOOT_SYNC_NACK

Data structure sent in response to any Request message indicating that the protocol state is incorrect. For example, if a BOOT_SYNC_BIB_REQUEST is sent by the Realm prior to trust being established, then a BOOT_SYNC_NACK would be returned.

Type	Variable	Notes
GUID_BLOB		gArmBootSyncNackGuid
UINT64 | Reason		NACK reason |

Expected termination values are: + ECONNABORTED - the protocol is in an invalid state e.g. BIB_REQUEST sent before ATTESTATION_RESP received + ENOTCONN - the connection is unavailable/

Part B
Appendix B

Device Attestation

Introduction

An essential part of Realm usage is remote attestation of the TCB that the Realm user must trust to determine that they are working within an execution environment that supports the guarantees of the Arm Confidential Computing Architecture (CCA). The evidence provided for Realm attestation allows assessment of the CCA platform and Realm workload. The TCB of the Realm can be extended by hardware devices available to the host platform. These devices can either be provided to support functions in all Realms (`Comprehensive Trust Devices`) or can be assigned on a demand basis to an individual Realm (`Selective Trust Devices`). These devices provide assistance services to the software within the Realm and can have access to Realm memory for data I/O. To ensure that a device can be trusted to operate as expected, it is necessary to appraise attestation evidence provided by that device to ensure trustworthiness. `Comprehensive Trust Devices` are made available by the Platform to all Realms and the evidence to support their usage is created independently of the Realm workload. Assessment of the trustworthiness of such devices is normally conducted as part of the Realm remote attestation process.

`Selective Trust Devices` are assigned by a process executed within Realm software. As part of the assignment process, evidence of the device identity and configuration are collected. This evidence is appraised for trustworthiness within the Realm workload and only devices assessed as trustworthy by the Realm code are given access to the Realm. For some Realm use cases, it can be desirable to also assess the evidence for Selective Trust devices as part of the Realm remote attestation process. This is considered below.

For more details about the process of device usage, see the `Realm device assignment` chapter of [1].

Device Evidence

The exact nature of the device evidence will depend upon the underlying protocol required to assign the device to the Realm.

In general, the evidence will consist of:

- An authoritative device identity provided by the device manufacturer. This is normally in the form of an X.509 certificate.
- Digest(s) of the firmware and configuration active on the device.

Appraisal of this evidence will typically confirm that the manufacturer is a trusted supplier, that the device is an appropriate part to provide the expected service and that firmware and / or configuration of the device is up to date.

EAT-DA format

It is recommended that the EAT-DA token format is based upon the following IETF draft [6]. This draft defines an EAT Profile for device evidence that is generic, extensible, and architecture agnostic. Usage of this profile takes a sympathetic approach to the EAT profiles used in the Realm and CCA Platform aspects of remote attestation.

Comprehensive Trust Devices

Comprehensive Trust Devices are platform-provided devices that are made available to all Realms. Use of such a device implies that all Realms implicitly trust the entire device, including its firmware and configuration, and that the device forms part of the platform trust boundary.

Developing trust in a Comprehensive Trust Device does not involve a per-Realm assignment flow or per-Realm appraisal performed by Realm code. The host is responsible for device integration and for creating the device representations that are exposed for Realm use. The RMM acts as the TDISP TEE Security Manager (TSM) and interacts with the Device Security Manager (DSM) within the device to measure and lock down the trusted device interfaces. Realm software may use the device, but does not determine whether the device is trustworthy for inclusion in the Realm TCB.

For some platforms, a Realm's code and data may be placed in memory provided by a Comprehensive Trust Device (for example, CXL-T3) prior to any remote attestation. In such cases, the device is already part of the Realm's

TCB regardless of any subsequent appraisal outcome. Attestation evidence for Comprehensive Trust Devices is therefore provided to allow an external relying party to determine whether to accept the resulting TCB (including the device) for the intended purpose (for example, granting access to services or relying on the Realm's outputs).

Evidence

As part of the device assignment process, the RMM will collect evidence from the device. This evidence includes:

- SPDM VCA (negotiated SPDM version, capabilities, and algorithms)
- A certificate chain that provides the device identity and the organisation which created the device.
- Measurements of firmware deployed on the device.

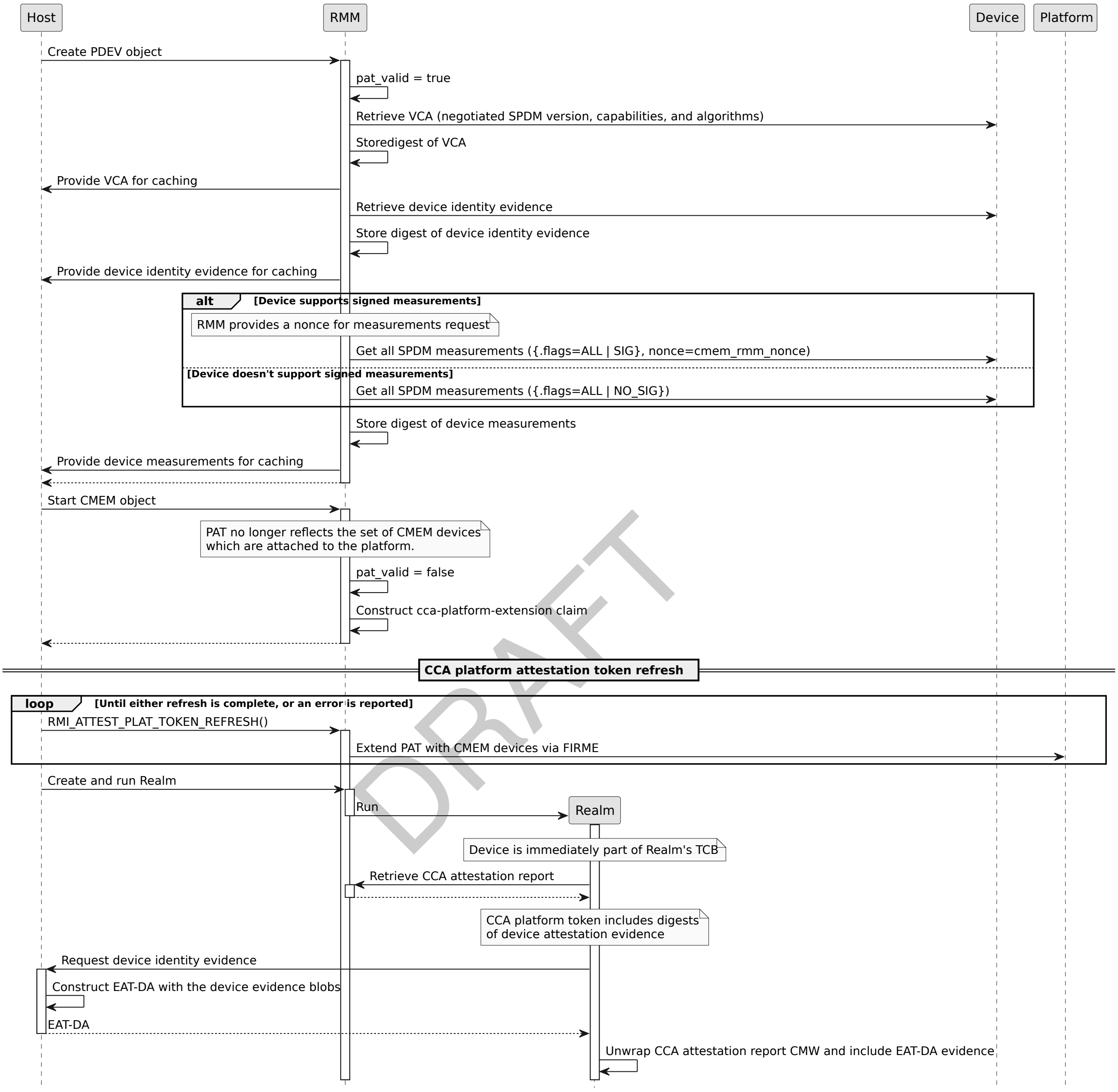
The RMM encapsulates a digest of this evidence in a cca-platform-extension claim, as specified in [1].

Evidence retrieval

The overall flow for comprehensive trust device evidence retrieval is as follows:

1. The Host executes an RMI command that causes the RMM to create a PDEV object for the device. At this point, the PAT reflects the available devices, the RMM tracks this by setting pat_valid = true.
2. The RMM performs a device transaction to retrieve VCA (negotiated SPDM version, capabilities, and algorithms), device identity from the Device.
3. The RMM stores a digest of the VCA and device identity.
4. The RMM returns the VCA and device identity separately to the Host for caching.
5. The RMM performs a device transaction to retrieve all SPDM measurements from the Device, as follows:
   - If the Device supports signed measurements, the RMM supplies a nonce for the measurements request and requests measurements with flags {ALL | SIG} and nonce = cmem_rmm_nonce.
   - Otherwise, the RMM requests measurements with flags {ALL | NO_SIG}.
6. The RMM stores a digest of the device measurements.
7. The RMM returns the device measurements to the Host for caching.
8. The Host executes an RMI command that causes the RMM to start a CMEM object.
9. The RMM sets pat_valid = false, indicating that the PAT no longer reflects the set of CMEM devices attached to the platform.
10. The RMM constructs the CCA platform-extension claim.
11. The Host refreshes the CCA platform attestation token by repeatedly invoking RMI_ATTEST_PLAT_TOKEN_REFRES until either: refresh completes successfully, or an error is reported.
12. For each RMI_ATTEST_PLAT_TOKEN_REFRESH() invocation, the RMM requests the Platform to extend the PAT with the CMEM devices via FIRME.
13. After PAT refresh completes, Realm is created.
14. The Realm executes an RSI request to retrieve a CCA attestation report from the RMM.
15. The Realm requests device identity evidence from the Host via an RHI call.
16. The Host constructs an EAT-DA token that contains the cached device identity evidence blobs and returns it to the Realm.
17. The Realm unwraps the CCA attestation report CMW and incorporates the EAT-DA evidence.
18. The resulting attestation evidence set that is sent to the verifier includes a CCA platform token that contains digests of device attestation evidence, with the corresponding device evidence blobs supplied to the Realm via EAT-DA.

The following sequence illustrates retrieval of comprehensive trust device evidence.



Selective Trust Devices

Developing trust in a device that is selectively assigned to a Realm requires an interaction between the host, the Realm, the RMM and the device. The host is responsible for initial device integration and the creation of PDEVs and VDEVs. The RMM acts as the TDISP TSM. The TSM is responsible for interacting with the DSM within the device to measure and lock down the trusted device interfaces. Realm code performs the act of assigning the device by collecting and validating device evidence and then informing the TSM that trust has been established.

Evidence

As part of the device assignment process, the RMM will collect evidence from the device. This evidence includes:

- A certificate chain that provides the device identity and the organisation which created the device.
- Measurements of firmware deployed on the device.

- Details about the configured device interface.

The evidence collected may be the specific artefact. Alternatively, if the device supports signed measurements, the retrieved device measurements by the TSM will be signed. In this case, the evidence collected will be the full protocol transactions, such that the signed interactions can be verified.

Evidence retrieval

Prior to any Realm interaction, the host will interact with the RMM at the point of PDEV creation to obtain the device certificate chain. The certificate chain will be stored in the host and the RMM will store a hash of the chain and the device public key extracted from certificate.

At the point of assignment to the Realm, the host will have created a VDEV. The Realm uses the VDEV Id to call to the host via RHI_DA_VDEV_GET_MEASUREMENTS. This call will cause the host to interact with the RMM and for the TSM in the RMM to interact with the device DSM to obtain the device measurements. The details of the TSM to DSM interaction will be affected by the parameters passed to RHI_DA_VDEV_GET_MEASUREMENTS and will involve the Realm specifying a nonce if signed interactions are supported. The Realm code may have to call RHI_DA_VDEV_CONTINUE for the communication process to be completed successfully. The measurements obtained from the device are stored in the host. The RMM stores a hash of the measurements.

Once the measurements are successfully stored in the host, the Realm can call RHI_DA_VDEV_GET_INTERFACE_REPOR in a similar interaction pattern as above. Once this sequence is successfully concluded, the device Interface Report will be stored in the host and a hash of the report will be stored in the RMM.

To verify the evidence (see process below) the Realm code must first obtain the evidence from the host. The Realm code uses the RHI_DA_OBJECT_READ ABI to obtain the various evidence artifacts from the host. Once these are read into the Realm, they can be validated to confirm they correspond to the data received by the TSM from the device. This is achieved by using the RSI_VDEV_GET_INFO ABI to obtain a set of hash data for the artifacts from the RMM.

When the Realm code has successfully validated the evidence for a given VDEV, it can enable use of the device within the Realm by calling the RSI_VDEV_DMA_ENABLE ABI to allow usage of the mapped memory in the Protected IPA.

The Realm code can also choose to construct a cca-platform-selective-device-evidence token from the evidence collected from all assigned VDEVs. This token can be packaged together with the evidence provided for Realm remote attestation to allow a relying party to perform additional appraisal of the assigned devices in the context of Realm usage.

Note that it will be the responsibility of the Realm code to coordinate the process of device evidence collection with any requests for Realm attestation. Should a request for Realm attestation be made while the device evidence collection is incomplete then that will need to be handled as an error condition or the indeterminate status noted for the device evidence.

Evidence verification

The (minimal) verification steps that Realm code must take to validate the evidence for a given VDEV are:

- Determine the appropriate hashing algorithm used by the RMM. This will be in the data returned from RSI_VDEV_GET_INFO.
- Extract the public key of the device certificate, hash that public key and confirm that it matches the data received from RSI_VDEV_GET_INFO.
- Perform any required policy acceptance of the device identity and certificate issuer organisation obtained from the device certificate chain.
- Create hash values for the additional evidence artifacts and confirm that these match the relevant hash values received from RSI_VDEV_GET_INFO.
- Confirm that the memory maps used for device data access provided by the host match the mappings in the device interface report.
- Where signed interactions have been requested between the TSM and the DSM, the artifacts will be transcripts of the protocol interactions.

- Confirm that the nonce values supplied to the data requests match the relevant values received from RSI_VDEV_GET_INFO.
- Validate the signing state of the transcripts

Any additional policy appraisal of the evidence is custom to the requirements of the Realm code.

An external relying party can choose to perform policy determined appraisal of the device evidence by examining data supplied to it in a cca-platform-selective-device-evidence token.