



Arm® Firmware Framework for M 1.1 Extension

Document number: AES 0039
Release Quality: Beta
Issue Number: 0
Confidentiality: Non-confidential
Date of Issue: 18/01/2023

Copyright © 2020, 2023 Arm Limited or its affiliates. All rights reserved

BETA RELEASE

This is a proposed set of updates to the *DEN 0063 Arm® Platform Security Architecture Firmware Framework 1.0* specification.

This is a BETA release in order to enable wider review and feedback on the changes proposed to be included in the v1.1 specification.

At this quality level, the proposed changes and interfaces are complete, and suitable for initial product development. However, the specification is still subject to change.

Abstract

This manual is part of the Arm Platform Security Architecture family of specifications. It defines a standard programming environment and firmware interfaces for implementing and accessing security services within a device's Root of Trust.

Contents

About this document	vi
Release information	vi
Arm Non-Confidential Document Licence ("Licence")	vii
References	ix
Terms and abbreviations	ix
Conventions	xiv
Typographical conventions	xiv
Numbers	xiv
Current status and anticipated changes	xiv
Feedback	xiv
Feedback on this book	xv
Inclusive language commitment	xv
1 Introduction	16
1.1 Objectives for version 1.1	16
1.2 Compatibility	17
1.3 Overview of new features	17
1.3.1 Secure Functions	17
1.3.2 Stateless RoT Services	18
1.3.3 Memory-mapped IOVECS	18
1.3.4 Support for peripheral drivers	18
1.3.5 Feature deprecation	19
1.3.6 Miscellaneous improvements	19
2 Framework features and permitted configurations	20
2.1 Changes to the Programming API	20
2.1.1 Firmware framework version	20
2.1.2 Discovering framework feature availability	21
2.2 Permitted configurations of FF-M version 1.1	22
3 Secure Functions	24
3.1 Background & rationale	24

3.2	The Secure Function model	24
3.2.1	Overview of the SFN model	24
3.2.2	Secure Partition execution	25
3.2.3	Scheduling Secure Partitions	26
3.2.4	Processing RoT Service messages	27
3.2.5	Interrupts	28
3.2.6	Doorbell	29
3.3	Implementation options	29
3.4	Selecting a Secure Partition model	29
3.5	Changes to the Programming API	29
3.5.1	Manifest changes	30
3.5.2	Secure Partition API changes	31
4	Stateless Root of Trust services	33
4.1	Background and rationale	33
4.2	Programming model	34
4.2.1	Overview of stateless RoT Services	34
4.2.2	RoT Service identification	34
4.2.3	RoT Service versioning	35
4.2.4	Requesting stateless RoT Services	35
4.2.5	Processing RoT Service messages	35
4.2.6	Programmer Error	36
4.2.7	Comparison of service types	36
4.3	Implementation options	37
4.4	Selecting the RoT Service type	38
4.5	Changes to the Programming API	38
4.5.1	Manifest changes	38
4.5.2	Client API changes	39
4.5.3	Secure Partition API changes	40
5	Memory-mapped IOVECs	41
5.1	Background and rationale	41
5.2	Programming model	42
5.2.1	Implementation flexibility	42
5.2.2	Typical deployment scenarios	43
5.2.3	RoT Service configuration	43
5.2.4	Accessing client input and output vectors	43
5.2.5	Interaction with the isolation model	44
5.3	Changes to the Programming API	44
5.3.1	Discovering MM-IOVEC availability	44
5.3.2	Enabling the MM-IOVEC API	45

5.3.3	Mapping RoT Service IO vectors	45
5.3.4	Changes to existing Secure Partition APIs	50
6	Enhancements for Secure Partition peripheral drivers	51
6.1	Background and rationale	51
6.1.1	Bounded interrupt response time	51
6.1.2	Managing interrupts	52
6.1.3	Accessing MMIO registers	52
6.2	Programming model	52
6.2.1	Definitions	52
6.2.2	Impact of Isolation	53
6.2.3	Impact of Concurrency	53
6.2.4	Interrupt model	54
6.2.5	FLIH Execution model	56
6.2.6	Secure Partition execution model	57
6.3	Changes to the Programming API	58
6.3.1	Manifest changes	58
6.3.2	Secure Partition API changes for FLIH	59
6.3.3	Secure Partition API changes for interrupt control	61
6.3.4	Register access functions for MMIO	63
6.4	Writing Secure Partition peripheral drivers	66
6.4.1	Programming patterns using FLIH	66
7	Deprecated features	69
7.1	Background and rationale	69
7.2	Changes to the Programming API	69
8	Miscellaneous changes	70
8.1	RoT Service terminology and requirements	70
8.1.1	The meaning of 'Root of Trust Service'	70
8.1.2	PSA RoT Services and Secure Partitions	71
8.2	Use of the manifest type attribute in isolation level 1	73
8.2.1	Changes to the specification	74
8.3	Availability of the PSA Lifecycle API in NSPE	74
8.4	Relaxation of memory access rules for Constant data	74
8.4.1	Changes to the specification	75
8.5	Reworking the optional isolation rules	75
8.5.1	Changes to the specification	76
8.6	Replace the term 'reverse handle' with 'rhandle'	79

8.7	Symbolic definition of Secure Partition resources	79
8.7.1	stack_size (attribute)	79
8.7.2	heap_size (attribute)	80
8.8	Implementation defined maximum call type value	80
8.8.1	Changes to the specification	81
8.9	Secure Partition RoT Service status codes	81
8.9.1	Changes to the specification	82
A	Reference header files	84
B	Summary of manifest attributes	86
B.1	Secure Partition object	86
B.1.1	Required attributes	86
B.1.2	Optional attributes	87
B.1.3	Example	87
B.2	Service object	88
B.2.1	Required attributes	88
B.2.2	Optional attributes	89
B.2.3	Example	89
B.3	Named Region object	89
B.3.1	Required attributes	89
B.3.2	Example	90
B.4	Numbered Region object	90
B.4.1	Required attributes	90
B.4.2	Example	90
B.5	IRQ object	90
B.5.1	Required attributes	90
B.5.2	Optional attributes	91
B.5.3	Example	91
B.6	Typed string attributes	91
B.6.1	c_macro	91
B.6.2	c_symbol	91
B.6.3	hex_string	92
C	Migrating Secure Partitions to version 1.1	93
C.1	Using an unmodified version 1.0 Secure Partition	93
C.2	Update the manifest to version 1.1	93
C.2.1	Manifest changes	93
C.2.2	Source code changes	94
C.3	Using version 1.1 features	94
C.3.1	Using the SFN model	94

C.3.2	Using a stateless RoT Service	95
C.3.3	Using MM-IOVEC	96
C.3.4	Using FLIH	97
D	Comparison between FF-M and TF-M frameworks	99
D.1	Background	99
D.1.1	The IPC model	99
D.1.2	The Library model	100
D.2	Analysis	100
D.2.1	One or two architectures?	100
D.2.2	Scaling and Flexibility	101
E	Implementing session-less RoT Services	103
E.1	Background	103
E.2	Analysis	105
E.3	Framework options	107
F	Change history	109
F.1	Changes between <i>Alpha (Issue 0)</i> and <i>Beta (Issue 0)</i>	109

About this document

Release information

The change history table lists the changes that have been made to this document.

Table 1 Document revision history

Date	Version	Confidentiality	Change
December 2020	1.1 Alpha 0	Non-confidential	Initial release of the 1.1 Extensions specification
January 2023	1.1 Beta 0	Non-confidential	Update for Beta release, incorporating feedback on new features and original specification.

For a detailed list of changes, see [Change history on page 109](#).

Arm® Firmware Framework for M

Copyright © 2020, 2023 Arm Limited or its affiliates. All rights reserved. The copyright statement reflects the fact that some draft issues of this document have been released, to a limited circulation.

Arm Non-Confidential Document Licence (“Licence”)

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

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

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

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

- (i) use and copy the Document for the purpose of designing and having designed products that comply with the Document;
- (ii) manufacture and have manufactured products which have been created under the licence granted in (i) above; and
- (iii) sell, supply and distribute products which have been created under the licence granted in (i) above.

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

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

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

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

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

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

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

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

The Arm corporate logo and words marked with ® or ™ are registered trademarks or trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. All rights reserved. Other brands and names mentioned in this document may be the

trademarks of their respective owners. No licence, express, implied or otherwise, is granted to Licensee under this Licence, to use the Arm trade marks in connection with the Document or any products based thereon. Visit Arm's website at www.arm.com/company/policies/trademarks for more information about Arm's trademarks.

The validity, construction and performance of this Licence shall be governed by English Law.

Copyright © 2020, 2023 Arm Limited or its affiliates. All rights reserved.

Arm Limited. Company 02557590 registered in England.
110 Fulbourn Road, Cambridge, England CB1 9NJ.

Arm document reference: LES-PRE-21585 version 4.0

References

This document refers to the following documents.

Table 2 Documents referenced by this document

Ref	Document Number	Title
[FF-M]	DEN 0063	Arm® Platform Security Architecture Firmware Framework. https://pages.arm.com/psa-apis.html
[PSA-SM]	DEN 0128	PSA Security Model. https://pages.arm.com/psa-apis.html
[PSA-TB]	DEN 0072	PSA Trusted Boot and Firmware Update. https://pages.arm.com/psa-apis.html
[TF-M]		trustedfirmware.org, Trusted Firmware-M. https://git.trustedfirmware.org/trusted-firmware-m.git/about/
[GP-ROT]		GlobalPlatform, Root of Trust Definitions and Requirements, v1.1, June 2018. https://globalplatform.org/wp-content/uploads/2018/07/GP_RoT_Definitions_and_Requirements_v1.1_PublicRelease-2018-06-28.pdf
[PSA-ITS]	IHI 0087	PSA Storage API. https://pages.arm.com/psa-apis.html
[PSA-CRYPT]	IHI 0086	PSA Cryptography API. https://pages.arm.com/psa-apis.html

Terms and abbreviations

This document uses the following terms and abbreviations.

Table 3 Terms and abbreviations

Term	Meaning
Application firmware	The main application firmware for the platform, typically comprising a Real-Time OS (RTOS) and application tasks.
Application Root of Trust	This is the security domain in which additional security services are implemented. See <i>PSA Security Model</i> [PSA-SM] for details.
Application RoT Service	This is a <i>Secure Partition RoT Service</i> within the <i>Application Root of Trust</i> domain.
Confused deputy attack	This is a specific type of privilege escalation exploit, in which a privileged component acts on behalf of an unprivileged attacker without correctly validating the request.

Table 3 (continued)

Term	Meaning
connection handle	<p>A handle that is used to make requests to a connection-based RoT Service. The handle value is returned by a successful connection to a connection-based RoT Service.</p> <p>See also stateless handle.</p>
Connection-based RoT Service	<p>This is the type of an Secure Partition RoT Service which uses a connection when making requests. These provide a set of operations that have some shared resources or state managed by the RoT Service. See Stateless Root of Trust services on page 33.</p> <p>In version 1.0, all Secure Partition RoT Services are connection-based.</p> <p>See also stateless RoT Service.</p>
First-level interrupt handling (FLIH)	<p>This is a form of interrupt handling that is carried out immediately when the interrupt exception takes place.</p> <p>This can be a traditional, privileged interrupt handler, or a deprived interrupt handler.</p>
FLIH	See First-level interrupt handling .
FLIH context	The execution context within the Secure Partition that is used to run a FLIH Functions . See also FLIH Execution model on page 56 .
FLIH function	A function that is used to perform First-level interrupt handling for an interrupt. An FLIH function runs in FLIH context .
I/O vector (iovec)	An object that holds a memory reference. I/O vectors are used to pass parameters from a client to a Secure Partition RoT Service . Input parameters are passed as a <code>psa_invec</code> , and output parameters as a <code>psa_outvec</code> .
IMPLEMENTATION DEFINED	<p>Behavior that is not defined by the this specification, but is defined and documented by individual implementations.</p> <p>Firmware developers can choose to depend on IMPLEMENTATION DEFINED behavior, but must be aware that their code might not be portable to another implementation.</p>
InterProcess Communication (IPC)	The Firmware Framework for M specifies an IPC mechanism to provide a communication channel for requests between isolated firmware partitions.
iovec	See I/O vector .
IPC	See InterProcess Communication .
IPC model	<p>The programming model and communication framework for Secure Partitions that is defined in version 1.0 of <i>Arm® Platform Security Architecture Firmware Framework [FF-M]</i>. Each Secure Partition is programmed like a C program to poll for messages and other events, and respond to them.</p> <p>See also Secure Function model and Library model.</p>
JOP	See Jump-oriented programming .

Table 3 (continued)

Term	Meaning
Jump-oriented programming (JOP)	This is a computer security exploit technique that allows an attacker to execute code in the presence of security defenses, for example executable space protection and code signing.
Library model	<p>This is a simple programming model and communication framework for security services that is implemented in version 1.0 of the <i>Trusted Firmware-M</i> [TF-M] project.</p> <p>The Library model is not compliant with <i>Arm® Platform Security Architecture Firmware Framework</i> [FF-M], but it has informed the extensions that are proposed in this document. See Comparison between FF-M and TF-M frameworks on page 99.</p> <p>See also IPC model and Secure Function model.</p>
MPU	Memory Protection Unit
Non-secure Processing Environment (NSPE)	This is the security domain outside of the Secure Processing Environment . It is the Application domain, typically containing the application firmware and hardware.
NSPE	See Non-secure Processing Environment .
Panic	An abnormal termination of an execution context in response to the invalid use of a programming interface.
Partition manifest	Metadata about a Partition describing the runtime resources and any assignment of privilege.
PROGRAMMER ERROR	<p>An error that is caused by the misuse of a programming interface.</p> <p>A PROGRAMMER ERROR is in the caller of the interface, but it is detected by the implementer of the interface.</p>
PSA	Platform Security Architecture
PSA Immutable Root of Trust	The hardware, code and data that cannot be modified following manufacturing. See <i>PSA Security Model</i> [PSA-SM] for details.
PSA Root of Trust	This defines the most trusted security domain within a PSA system. See <i>PSA Security Model</i> [PSA-SM] for details.
PSA RoT Service	<p>This is an RoT Service within the PSA Root of Trust domain.</p> <p>It is IMPLEMENTATION DEFINED whether a PSA RoT Service is a Secure Partition RoT Service.</p>
PSA Updatable Root of Trust	The Root of Trust firmware that can be updated following manufacturing. See <i>PSA Security Model</i> [PSA-SM] for details.
Return-oriented programming (ROP)	This is a computer security exploit technique that allows an attacker to execute code in the presence of security defenses, for example executable space protection and code signing.

Table 3 (continued)

Term	Meaning
rhandle	A value that is associated with a specific Secure Partition RoT Service connection by the RoT Service implementation. This value can be used to link a client connection with data or resources managed by the Secure Partition. In version 1.0, this was referred to as a <i>reverse handle</i> . See Replace the term 'reverse handle' with 'rhandle' on page 79 .
Root of Trust (RoT)	This is the minimal set of software, hardware and data that is implicitly trusted in the platform – there is no software or hardware at a deeper level that can verify that the Root of Trust is authentic and unmodified. See <i>Root of Trust Definitions and Requirements</i> [GP-ROT].
Root of Trust Service (RoT Service)	A set of related security operations that are provided and protected within a Root of Trust . See also Secure Partition Root of Trust Service .
ROP	See Return-oriented programming .
RoT	See Root of Trust .
RoT Service	See Root of Trust Service .
Second-level interrupt handling (SLIH)	This is a form of interrupt handling that is deferred until after the interrupt exception. This handling occurs within a thread context, and is subject to normal scheduling. See First-level interrupt handling .
Secure Function (SFN)	A callback function in a Secure Partition that handles requests for a single Root of Trust Service . SFNs are used to implement RoT Services for Secure Partitions that are using the SFN model .
Secure Function model (SFN model)	The programming model and communication framework for Secure Partitions that is defined in version 1.1 of <i>Arm® Platform Security Architecture Firmware Framework</i> [FF-M]. Each security service is a C function that is invoked as a callback from the framework, in response to a call from a client function. See Secure Functions on page 24 . See also IPC model and Library model .
Secure Partition	An execution environment with protected runtime state within the Secure Processing Environment . A Secure Partition is a container for the implementation of one or more Secure Partition RoT Services or one or more secure peripheral drivers. A Secure Partition must either use the IPC model or the SFN model for implementation of RoT Services. Multiple Secure Partitions are allowed in a platform.
Secure Partition Manager (SPM)	Part of the Firmware Framework that is responsible for isolating software in Partitions, managing the execution of software within Partitions, and providing IPC between Partitions.

Table 3 (continued)

Term	Meaning
Secure Partition Root of Trust Service (Secure Partition RoT Service)	<p>A Root of Trust Service that is implemented in a Secure Partition. Multiple RoT Services can coexist in a single Secure Partition.</p> <p>A Secure Partition RoT Service uses either the SFN model or IPC model communication framework to receive service requests from clients.</p>
Secure Partition RoT Service	See Secure Partition Root of Trust Service .
Secure Partition thread context	<p>The main execution context within the Secure Partition.</p> <ul style="list-style-type: none"> For a Secure Partition that is using the IPC model, this is the Secure Partition thread. For a Secure Partition that is using the SFN model, this is any Secure Function within the Secure Partition.
Secure Processing Environment (SPE)	This is the security domain that includes the PSA Root of Trust and the Application Root of Trust domains.
Service ID (SID)	Service Identification. The identifier used for a PSA RoT Service or an Application RoT Service .
SFN	See Secure Function .
SFN model	See Secure Function model .
SID	See Service ID .
SLIH	See Second-level interrupt handling .
SPE	See Secure Processing Environment .
SPM	See Secure Partition Manager .
stateless handle	<p>A handle that is used to make requests to a stateless RoT Service.</p> <p>The handle value is a compile-time constant which is defined by the framework implementation.</p> <p>See also connection handle.</p>
Stateless RoT Service	<p>This a type of Secure Partition RoT Service which does not need connections. These provide a set of standalone operations, and do not need a connection to manage resources or state between separate RoT Service requests. See Stateless Root of Trust services on page 33.</p> <p>See also connection-based RoT Service.</p>
Trusted Boot	Trusted Boot is technology to provide a chain of trust for all the components during boot. See PSA Trusted Boot and Firmware Update [PSA-TB] .

Conventions

Typographical conventions

The typographical conventions are:

<i>italic</i>	Introduces special terminology, and denotes citations.
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 <i>Terms and abbreviations</i> .
Red text	Indicates an open issue.
Blue text	Indicates a link. This can be <ul style="list-style-type: none">• A cross-reference to another location within the document• A URL, for example http://infocenter.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.

Current status and anticipated changes

This document is at Beta quality status which has a particular meaning to Arm of which the recipient must be aware. A Beta quality specification will be sufficiently stable & committed for initial product development, however all aspects of the architecture described herein remain SUBJECT TO CHANGE. Please ensure that you have the latest revision.

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 arm.psa-feedback@arm.com. Give:

- The title (Arm® Firmware Framework for M).
- The number and issue (AES 0039 1.1 Extension Beta (Issue 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.

Inclusive language commitment

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

To report offensive language in this document, email terms@arm.com.

1 Introduction

This document introduces a set of updates and extensions to the *Arm® Platform Security Architecture Firmware Framework* [FF-M] specification, designed to build on the capabilities provided in version 1.0.

When the proposed extensions are fully stable, they will be integrated into the FF-M version 1.1 specification.

These extensions have been developed in conjunction with the *Trusted Firmware-M* [TF-M] project, which is developing a reference implementation of the PSA firmware specifications.

A version 1.1 compliant implementation must include all features defined in FF-M version 1.1 that are not described as being optional. See *Framework features and permitted configurations* on page 20.

Note

This version of the document includes *Rationale* commentary that provides background information relating to the design decisions that led to the current set of proposals. This enables the reader to understand the wider context and alternative approaches that have been considered.

The rationale is presented in green boxes, as this note is.

1.1 Objectives for version 1.1

There are three primary drivers for providing an update to version 1.0 of [FF-M]:

- A need for a security framework specification that can target a smaller, simpler system architecture. This kind of framework is demonstrated by the *Library model* in the *Trusted Firmware-M* [TF-M] project.
Ideally, there is continuity between this light-weight framework and the version 1.0 Firmware Framework, making it easier to migrate RoT Service code between the different types of framework.
- The development of *Secure Partition RoT Services* using the version 1.0 Framework has highlighted some optimization challenges for individual services or entire systems.
Improving efficiency for these use cases involves some reduction in flexibility or security mitigation. The extensions to the Framework defined in this proposal allow these trade-offs to be made at the appropriate level by the service developers and system integrators.
- The mechanism for handling secure interrupts within a Secure Partition in [FF-M] version 1.0 does not support low latency and bounded interrupt response time. Secure peripherals requiring this behavior can only be implemented within the SPM using an *IMPLEMENTATION DEFINED* mechanism in version 1.0.
System robustness and security can be improved if low latency interrupt handling for these peripherals can execute within the Secure Partition context.

1.2 Compatibility

We have considered adopting the existing [\[TF-M\] Library model](#) as the light-weight framework for this specification. However, the design of the Library model in TF-M is not compatible with the [\[FF-M\]](#) interfaces, and is not compliant with all of the requirements for the Firmware Framework. A more detailed analysis of these two frameworks is provided in [Comparison between FF-M and TF-M frameworks on page 99](#).

Instead, this proposal provides a suite of extensions to [\[FF-M\]](#) version 1.0 that together enable the creation of a framework implementation which shares the design and efficiency characteristics of the TF-M Library model. Some of these extensions also provide opportunities for improving the efficiency of Secure Partition RoT Services that use the version 1.0 framework interfaces.

The C language interfaces defined by version 1.1 of [\[FF-M\]](#) are backward compatible with version 1.0. Secure Partition source code that is written for version 1.0 will work on a version 1.1 framework that provides support for the [IPC model](#) defined in version 1.0. See also [Framework features and permitted configurations on page 20](#).

The JSON schema for the manifest source files for version 1.1 is not directly compatible with version 1.0. To use any of the version 1.1 features, a Secure Partition manifest file will need to declare "psa_framework_version": 1.1, and make the necessary changes required by the version 1.1 manifest definition. See [Migrating Secure Partitions to version 1.1 on page 93](#).

Framework implementations that support Secure Partitions using the IPC model must support Secure Partition manifest files that declare "psa_framework_version": 1.0, and implement these Secure Partitions as defined in version 1.0 of [\[FF-M\]](#).

1.3 Overview of new features

The definition of the new and updated features for the Firmware Framework assumes familiarity with version 1.0 of [\[FF-M\]](#).

1.3.1 Secure Functions

This extension introduces the [SFN model](#), which is an additional programming model for Secure Partitions. The existing programming model in version 1.0 is now referred to as the [IPC model](#).

When the SFN model is selected for a Secure Partition, each of the RoT Services within the Secure Partition provides a handler function, referred to as a [Secure Function](#) (SFN). The SFN is invoked by the framework to process messages for the RoT Service.

For this kind of Secure Partition, the developer does not provide a signal handling loop. Instead, the SFNs are called directly by the framework, with the message as a parameter. The message is completed using the return value from the SFN.

The framework does not have to provide a dedicated execution context i.e (thread/stack) for a partition using the SFN model, as long as the isolation rules are satisfied for the implemented isolation level.

See [Secure Functions on page 24](#).

1.3.2 Stateless RoT Services

This extension introduces the option for a [Secure Partition RoT Service](#) to be a [stateless RoT Service](#), as an alternative to a [connection-based RoT Service](#) which is defined in version 1.0. This enables a much more efficient implementation of RoT Services that do not make use of the connection-related features of the version 1.0 interface.

When a Secure Partition RoT service is declared to be stateless in a Secure Partition manifest, the RoT Service does not use connections:

- Clients cannot connect to the service using `psa_connect()`.
- The service does not receive any *connection* or *disconnection messages*.
- Client's invoke the service using a special handle value in a call to `psa_call()`, which is received by the service as a *request message*.

Both connection-based RoT Services and stateless RoT Services can be used with either the IPC model or the SFN model.

See [Stateless Root of Trust services on page 33](#).

1.3.3 Memory-mapped IOVECS

This extension introduces the ability for Secure Partition RoT Service code to map client input and output buffer parameters into the Secure Partition, enabling direct access to the client memory. This is of particular efficiency concern in small systems where the following are true:

- The memory protection implemented in the framework already permits the SPE to access all of the client memory.
- The footprint/runtime cost of transferring the client data to the SPE is prohibitive for the use case. This might be because the RoT Service can operate on the data “in place” in the NSPE, or is implemented using hardware that can directly address the client memory.

However, direct access to client memory introduces security risks which the version 1.0 API prevents by design, and direct access to client memory may be complex on some systems, and impossible on others.

Support for this feature is optional, allowing each implementation to select a balance between efficiency, complexity and security that is appropriate for the system and use case.

See [Memory-mapped IOVECs on page 41](#).

1.3.4 Support for peripheral drivers

This extension adds a number of improvements for implementing secure peripheral drivers in Secure Partitions:

- [First-level interrupt handling](#) (FLIH) is a de-privileged, low-latency, interrupt handling capability for Secure Partitions. This enables Secure Partitions to be used for peripheral drivers that require secure interrupts to be handled within a bounded time.
- An API for managing interrupts supports FLIH and fills a gap in the version 1.0 API.

- Accessors for MMIO registers ensure more portability between implementations and system architectures.

See [Enhancements for Secure Partition peripheral drivers on page 51](#).

1.3.5 Feature deprecation

This extension marks the doorbell notification feature as deprecated. See [Deprecated features on page 69](#).

1.3.6 Miscellaneous improvements

This extension includes clarifications and relaxations to terminology and APIs in the version 1.0 specification as described in [Miscellaneous changes on page 70](#).

2 Framework features and permitted configurations

The version 1.1 extensions defines some features that are optional for implementations of the framework, and also permits some of the original version 1.0 features to be optional.

To enable the development of portable code that uses this framework, some feature discovery APIs are defined. These make it possible for a Secure Partition RoT Service to select the appropriate API for the framework that it is being built for. See [Discovering framework feature availability on page 21](#).

Optional features and APIs provide flexibility, enabling a framework implementor to optimize the framework for the specific system and use case. However, a flexible feature-set increases the complexity of Secure Partition RoT Service code that is portable between different framework implementations.

This document defines a permitted set of framework configurations, to balance these competing requirements. A framework must match one of the permitted configurations to be compliant with version 1.1 of FF-M. See [Permitted configurations of FF-M version 1.1 on page 22](#).

2.1 Changes to the Programming API

2.1.1 Firmware framework version

PSA_FRAMEWORK_VERSION (macro)

```
#define PSA_FRAMEWORK_VERSION (0x0101u)
```

This existing API is modified for version 1.1

An implementation of version 1.1 must report the value 0x0101.

psa_framework_version (function)

```
uint32_t psa_framework_version(void);
```

Returns: uint32_t

Description

This existing API is modified for version 1.1

An implementation of version 1.1 must return the value 0x0101.

2.1.2 Discovering framework feature availability

Discovery mechanisms are introduced for optional features and features with varying levels of support. The discovery APIs are provided in a new header file `psa/framework_feature.h`.

An implementation of version 1.1 must provide an instance of `psa/framework_feature.h`, and include all required API elements.

The following pre-processor symbols are defined, to enable compile-time code variation:

- `PSA_FRAMEWORK_ISOLATION_LEVEL`
- `PSA_FRAMEWORK_HAS_MM_IOVEC`

To use these symbols in Secure Partition source code, include the `psa/framework_feature.h` header file.

Note:

Pre-processor symbols are also defined for each Secure Partition, which indicate the communication model specified in the Secure Partition manifest file. These value of these macros depends on the `model` attribute in the Secure Partition manifest. See [Secure Functions on page 24](#).

PSA_FRAMEWORK_ISOLATION_LEVEL (macro)

A pre-processor symbol that declares the isolation level implemented by the framework.

```
#define PSA_FRAMEWORK_ISOLATION_LEVEL /* implementation-defined value */
```

This macro must be defined by a version 1.1 implementation.

The `IMPLEMENTATION_DEFINED` value of this macro indicates the isolation level, which is a value between 1 and 3.

See *Arm® Platform Security Architecture Firmware Framework [FF-M]* §3.1.3 *Protection Domains* for the definition of the three isolation levels.

PSA_FRAMEWORK_HAS_MM_IOVEC (macro)

A pre-processor symbol that declares the compile-time availability of the MM-IOVEC API.

```
#define PSA_FRAMEWORK_HAS_MM_IOVEC /* implementation-defined status */
```

This macro must be defined by a version 1.1 implementation that provides MM-IOVEC functionality.

We recommend that this macro is defined as 0 by a version 1.1 implementation that does not provide MM-IOVEC functionality. This improves portability of Secure Partition RoT Service code which uses the feature.

The `IMPLEMENTATION_DEFINED` value of this macro indicates the availability of the MM-IOVEC feature:

0	The MM-IOVEC API is not provided by the implementation
1	The MM-IOVEC API is provided by the implementation

See [Memory-mapped IOVECs on page 41](#).

Usage

In portable code, `PSA_FRAMEWORK_HAS_MM_IOVEC` is used to select code that uses the MM-IOVEC feature for parameter access, instead of the `psa_read()` and `psa_write()` functions. For example:

```
#include "psa/framework_feature.h"

...

#if PSA_FRAMEWORK_HAS_MM_IOVEC
    // use psa_map_invec() and psa_map_outvec() to access parameters
#else
    // use psa_read() and psa_write() to access parameters
#endif
```

2.2 Permitted configurations of FF-M version 1.1

Version 1.1 introduces some optional features, and also makes some features of version 1.0 optional for new implementations.

The following features are optional for a version 1.1 implementation:

- Support for the IPC model.
- Support for the SFN model.
- Support for connection-based RoT Services.
- Support for MM-IOVEC.
- Support for Secure Partitions within the PSA Root of Trust.

See the following sections for more details on these features:

- [Secure Functions on page 24](#)
- [Stateless Root of Trust services on page 33](#)
- [Memory-mapped IOVECs on page 41](#)
- [PSA RoT Services and Secure Partitions on page 71](#)

An implementation is compliant with version 1.1 of FF-M if it implements all of the required features and APIs, and meets the following constraints on the optional features and APIs:

- At least one of the SFN model and IPC model must be supported.
- If both of the SFN model and the IPC model are supported by the implementation, then the framework must support a hybrid system which includes Secure Partitions with different models.
- The framework support for connection-based RoT Services is not constrained by this specification. Each implementation makes an appropriate decision based on technical and security criteria.
- The framework support for MM-IOVEC is not constrained by this specification. Each implementation makes an appropriate decision based on technical and security criteria.

- The framework support for PSA RoT Secure Partitions is not constrained by this specification. Each implementation makes an appropriate decision based on technical and security criteria.

3 Secure Functions

This extension introduces the [SFN model](#), which is an additional programming model for Secure Partitions. The existing programming model in version 1.0 is now referred to as the [IPC model](#).

3.1 Background & rationale

The programming model in version 1.0 provides a significant level of control for the Secure Partition developer. The Secure Partition can process signals in any order, and can defer responding to a message while continuing to process other signals. To provide this control, the framework has to maintain a dedicated execution context for each Secure Partition.

Many RoT Services do not require this level of execution control. However, the programming model in version 1.0 prevents an implementation from reducing these overheads, even in constrained systems that only require limited isolation.

The *Trusted Firmware-M* [\[TF-M\]](#) project provides an implementation of [\[FF-M\]](#), and refers to this programming model as the *IPC model*.

TF-M also provides a much simpler abstraction for secure services. The programming model is based around a set of secure service functions, each of which handles requests from a corresponding client-side function. TF-M refers to this service programming model as the [Library model](#). This programming model enables a very efficient framework implementation when the services all run within a single SPE protection domain.

The Secure Function model reuses the concept of RoT Service callback functions from the TF-M Library model, and integrates this into the FF-M Secure Partition architecture.

If all Secure Partitions use the SFN model, then the implementation is able to significantly reduce the framework overhead for systems that do not require high levels of isolation.

3.2 The Secure Function model

The [Secure Function model](#) (SFN model) is introduced as an alternative programming model for code within a Secure Partition. Each Secure Partition must either use the [IPC model](#), which has the APIs and programming model defined in [\[FF-M\]](#) version 1.0, or the Secure Function model, which uses a modified API and manifest definition as defined in version 1.1.

3.2.1 Overview of the SFN model

The SFN model is a simpler programming model for developing Secure Partition RoT Services, in comparison with the IPC model. However, it is still suitable for many types of RoT Service.

The SFN model impacts some of the interfaces used to implement the service within the Secure Partition, while the Client API is identical for both models.

In a Secure Partition using the SFN model:

- Secure services are implemented as [Secure Functions](#) (SFN) that are called by the framework when the client makes a request to the service.
- Each RoT Service within the Secure Partition has its own SFN.
- The framework calls an SFN in response to a client call to `psa_connect()`, `psa_call()`, or `psa_close()` for that RoT Service.
- The SFN is called with a `psa_msg_t` object that describes the client request, including the same information as this object provides in the IPC model.
- The SFN accesses client parameters in the same way as the IPC model, using the `psa_read()` and `psa_write()` APIs.
- The SFN return value is used as the response to the client message, instead of using `psa_reply()`.

The following sections provide more specific changes to the programming model that is described in [\[FF-M\]](#) version 1.0 §3 *Secure Processing Environment programming model*.

3.2.2 Secure Partition execution

Execution of Secure Partitions for the IPC model is described in [\[FF-M\]](#) §3.2.3 *Secure Partition execution*.

In the SFN model, the Secure Partition is made up of a collection of callback functions:

- One optional initialization function, which is declared as the `entry_init` symbol in the Secure Partition manifest. See [Secure Partition initialization function on page 26](#).
- A set of Secure Functions (SFN), one for every RoT Service that is defined in the Secure Partition manifest. The name of each SFN is based on the RoT Service `name` attribute provided in the manifest.

The framework implementation is responsible for invoking these callback functions in response to system startup or client calls to `psa_connect()`, `psa_call()` and `psa_close()` for an RoT Service within the Secure Partition.

An SFN becomes *active* when the framework calls it in response to a client request, and remains active until the SFN returns. At most one SFN within a Secure Partition can be active at the same time. That is, while an SFN is active, no other SFN in the same Secure Partition will execute. This results in single-threaded behavior within the Secure Partition, which is also provided by the IPC model.

The SFN model still uses Secure Partition signals for interrupts and the Secure Partition doorbell feature. To query the signal state or block until a specific signal is asserted, an SFN calls `psa_wait()`. If the SFN blocks while waiting, this prevents all other SFNs in that Secure Partition from running: see [Scheduling Secure Partitions on page 26](#). If there is no active SFN, an asserted interrupt or doorbell signal does not trigger code execution within the Secure Partition. The [FLIH](#) extension provides an alternative mechanism for an interrupt to be handled within a Secure Partition using the SFN model, see [Interrupts on page 28](#).

In the SFN model, the framework determines the order in which SFNs are called, when more than one has an outstanding message.

Implementation note

The framework does not have to provide a dedicated execution context i.e (thread/stack) for a partition using the SFN model, as long as isolation rules are satisfied.

Secure Partition initialization function

A Secure Partition using the SFN model can optionally provide an initialization function that is declared in the manifest file using the [entry_init](#) attribute.

The initialization function has the following signature:

```
psa_status_t «entry_init»(void);
```

where «entry_init» is the value of the [entry_init](#) attribute.

This function is called prior to any other function in the Secure Partition, and can be used to perform initialization that is required before requests are made to the Secure Partition RoT Services. The Secure Partition initialization function is permitted to use the client API to call RoT Services in other Secure Partitions.

If the initialization function returns `PSA_SUCCESS`, the framework will enable all of the RoT Services for the Secure Partition.

If the initialization function returns any other status code, the framework will not call any Secure Function within the Secure Partition.

The effect on the system of an initialization function reporting an error is [IMPLEMENTATION DEFINED](#). If the framework does not halt or restart the system, a client that attempts to connect or call to any RoT Service in the Secure Partition will receive the error `PSA_ERROR_CONNECTION_REFUSED`.

Note:

The initialization function must return, unlike [entry_point](#) in a Secure Partition using the IPC model. If the Secure Partition initialization is partially successful, then the recommended approach is as follows:

- The initialization function returns `PSA_SUCCESS`, which enables the RoT Service SFNs.
 - Individual RoT Services that cannot operate can respond with an appropriate error status when clients try to connect to or use the RoT Service.
-

Rationale

The choice to have the initialization function return an error status, rather than panic on a critical failure, allows the framework implementation more flexibility in how to handle Secure Partition initialization errors.

3.2.3 Scheduling Secure Partitions

The framework can allow different Secure Partitions to interleave execution. In particular, an SFN from one Secure Partition can run concurrently with an SFN from a different Secure Partition. This capability is [IMPLEMENTATION DEFINED](#), as it is in version 1.0.

The definitions in [\[FF-M\]](#) version 1.0 §3.2.4 *Scheduling Secure Partitions* are updated or extended as follows:

- An SFN is *pending* if a client has made a request to the RoT Service, but the framework has not yet called the SFN.

- An SFN is *active* if the framework has called the SFN, but the SFN has not yet returned from the call.
- A Secure Partition using the SFN model is *idle* after the Secure Partition initialization function has been run and are no *active* or *pending* SFNs.
- A *running* Secure Partition using the SFN model becomes *idle* if the SFN returns and there are no *pending* SFNs.
- A *running* Secure Partition using the SFN model becomes *ready-to-run* if the SFN returns and there is at least one *pending* SFN. At this point the framework can select this Secure Partition to continue running or schedule a different *ready-to-run* Secure Partition.
- An *idle* Secure Partition becomes *ready-to-run* if one of the SFNs in the Secure Partition becomes *pending*.
- When a *ready-to-run* Secure Partition that is using the SFN model is selected for execution, it will either resume execution of an *active* SFN, if there is one, or the framework will select one of the *pending* SFNs to make *active* and start execution of that SFN.

3.2.4 Processing RoT Service messages

When a Secure Partition is using the SFN model, the *connection*, *disconnection* and *request messages* do not cause a Secure Partition signal to be asserted. Instead, the Secure Function (SFN) for the RoT Service is invoked by the framework, with the message details provided as a parameter to the SFN.

The SFN must have the following signature:

```
psa_status_t «name»_sfn(const psa_msg_t* msg);
```

where «name» is the name of the RoT Service defined in the Secure Partition manifest.

A SFN processes the delivered message using the `psa_read()`, `psa_write()`, `psa_skip()`, and `psa_set_rhandle()` functions. These functions operate the same way in both the SFN and IPC models.

When processing is complete, the return value from the SFN is used as the reply status for the message.

A SFN cannot use the `psa_get()` or `psa_reply()` functions, as this functionality is performed by the framework.

The behavior of the framework is otherwise identical for both IPC and SFN models:

- The RoT Service will receive *connection*, *disconnection* and *request messages*. See [Stateless Root of Trust services on page 33](#) for an extension that eliminates *connection* and *disconnection messages* for services that do not need them.
- The handling of error responses, particularly `PSA_ERROR_PROGRAMMER_ERROR`.
- The use of `psa_read()`, `psa_write()`, `psa_skip()`, and `psa_set_rhandle()` functions to process messages.
- An SFN can use `psa_wait()` to wait for interrupt signals that are defined in the manifest, or the Secure Partition doorbell signal.

Implementation note

Conceptually, for a single service named `SERVICE1` in a Secure Partition manifest, the framework behaves as if it was the following IPC model entry point:

```

void sp_main(void)
{
    psa_msg_t msg;

    for (;;)
    {
        psa_wait(SERVICE1_SIGNAL, PSA_BLOCK);
        if (psa_get(SERVICE1_SIGNAL, &msg) == PSA_SUCCESS)
            psa_reply(msg.handle, service1_sfn(&msg));
    }
}

```

This example can be extended in the obvious way for Secure Partitions with more than one RoT Service.

In practice, the framework can choose to implement this very differently. For example, by running `service1_sfn()` directly on a common execution stack.

Warning: It is not recommended that an implementation copy this approach directly:

- It requires the RoT Services to have signals allocated that are visible to the Secure Partition.
- It requires that the Secure Partition is able to invoke the `psa_get()` and `psa_reply()` functions, which are defined to respond with a [PROGRAMMER ERROR](#) when used in a Secure Partition that is using the SFN model.

3.2.5 Interrupts

The definition of interrupts is the same for Secure Partitions using either the [IPC model](#) or the [SFN model](#).

Interrupts in the SFN model behave the same way as interrupts sources in the IPC model, by allocating a signal value and causing a Secure Partition signal to be set when the interrupt occurs. To respond to an interrupt in the SFN model, an SFN must call `psa_wait()` to query or block for the interrupt signal. The SFN must call `psa_eoi()` to clear the signal when received, after interrupt processing is complete.

This permits an SFN to initiate a hardware operation and then wait for it to complete, without blocking the CPU.

Note:

Unlike the IPC model, a Secure Partition using the SFN model cannot respond to an interrupt signal unless it has an active SFN which calls `psa_wait()`.

The [Enhancements for Secure Partition peripheral drivers on page 51](#) extension in this update provides an additional mechanism for responding to interrupts.

3.2.6 Doorbell

The Secure Partition doorbell behaves the same way as in the IPC model, by causing the doorbell Secure Partition signal to be set when the notification is sent. The only way to respond to a doorbell in the SFN model is for an SFN to call `psa_wait()` to poll or block for the doorbell signal. The SFN must call `psa_clear()` to clear the signal.

3.3 Implementation options

Both the SFN model and the IPC model are optional in an implementation of version 1.1 of the Firmware Framework, but at least one of them must be supported.

See [Permitted configurations of FF-M version 1.1 on page 22](#) for the list of compliant configurations of all of the optional features in version 1.1.

If the framework supports Secure Partitions within the PSA Root of Trust, then the framework is permitted to only support a subset of the communication models for this type of Secure Partition. See also [PSA RoT Services and Secure Partitions on page 71](#).

The framework defines pre-processor symbols that enable code to determine the model that is used for a specific Secure Partition. This makes it simpler to develop RoT Service code that can be built into a Secure Partition using either the IPC model or the SFN model. See the Secure Partition manifest `model` attribute.

3.4 Selecting a Secure Partition model

The SFN model is recommended for a Secure Partition unless you have any of the following requirements:

1. The ability to defer completion of an RoT Service message, while continuing to service other messages.
2. The ability to respond asynchronously to a doorbell or interrupt signal. This is often required in conjunction with requirement #1.
3. Control over the order in which RoT Service signals are processed.

Note that requirement 2 could be met instead using the [Enhancements for Secure Partition peripheral drivers on page 51](#) extension.

3.5 Changes to the Programming API

The changes to [\[FF-M\] §4 Programming API](#) are described in the following sections:

- [Manifest changes on page 30](#)
- [Secure Partition API changes on page 31](#)

3.5.1 Manifest changes

model (attribute)

New attribute at the top level of a Secure Partition manifest.

Properties: Required, Unique.

This attribute declares which programming model is used in this Secure Partition. This attribute must take one of the following values:

"IPC"	The Secure Partition uses the IPC model
"SFN"	The Secure Partition uses the SFN model

A manifest file that defines "psa_framework_version": 1.0 is implicitly using the [IPC model](#).

The implementation reports an error if the selected model is not supported by the framework.

Rationale

Defining this as an optional attribute which defaults to "IPC" is a possible alternative. Although this would make migration of a manifest from v1.0 to v1.1 easier, it is better for the manifest to be explicit about the selection of programming model, and not depend on the reader knowing the default behavior.

The Secure Partition header file, `psa_manifest/«manifest-filename».h`, must include the definitions of `«name»_MODEL_IPC` and `«name»_MODEL_SFN`, where «name» is the value of the name attribute in the Secure Partition manifest.

If the Secure Partition uses the IPC model, the following definitions are used:

```
#define «name»_MODEL_IPC 1
#define «name»_MODEL_SFN 0
```

If the Secure Partition uses the SFN model, the following definitions are used:

```
#define «name»_MODEL_IPC 0
#define «name»_MODEL_SFN 1
```

entry_point (attribute)

This existing attribute is only required for a Secure Partition using the [IPC model](#). It must not be present in a Secure Partition using the [SFN model](#), which uses the optional [entry_init](#) attribute instead.

entry_init (attribute)

New attribute at the top level of a Secure Partition manifest.

Properties: Optional, Unique.

This attribute indicates an optional entry point for an initialization function within a Secure Partition that is using the [SFN model](#). This attribute must not be present in a Secure Partition using the [IPC model](#), see [entry_point](#).

If defined, the value of `entry_init` must be the C identifier of a function with the signature:

```
psa_status_t «entry_init»(void);
```

C++ source files must use the `extern "C"` keyword if necessary.

Rationale

Reusing the existing [entry_point](#) attribute is not ideal, because these functions have different properties: [entry_init](#) is **optional**, and it **must return**.

services (attribute)

The definition of RoT Services in the manifest is modified for the [SFN model](#).

Service signals

In the SFN model, the implementation **does not** allocate a Secure Partition signal for each service, and **does not** define the identifier `«name»_SIGNAL` in the `psa_manifest/«manifest-filename».h` header file.

Service handlers

For a Secure Partition that uses the SFN model, each RoT Service has a Secure Function that implements the service. This function has the following prototype:

```
psa_status_t «name»_sfn(const psa_msg_t* msg);
```

The C identifier used for an SFN is constructed by adding the suffix `_sfn` to a lowercase version of the RoT Service's name attribute.

The Secure Partition header file, `psa_manifest/«manifest-filename».h`, must include the prototype definition of each RoT Service SFN.

The service developer defines the SFN using this identifier in their source code.

3.5.2 Secure Partition API changes

In a Secure Partition using the [SFN model](#), there are no RoT Service signal identifiers defined by the framework in the Secure Partition header file.

psa_get (function)

This existing function is constrained to be used only in Secure Partitions using the IPC model.

It is a [PROGRAMMER ERROR](#) to call `psa_get()` in a Secure Partition that is using the SFN model.

Implementation note

This behavior naturally results from the existing requirement that the caller provides the signal value for exactly one RoT Service, and signal values are not defined for RoT Services in a Secure Partition using the SFN model.

psa_reply (function)

This existing function is constrained to be used only in Secure Partitions using the IPC model.

It is a [PROGRAMMER ERROR](#) to call `psa_reply()` in a Secure Partition that is using the SFN model.

Instead, the message response status is provided to the framework as the return value from the Secure Function.

4 Stateless Root of Trust services

This extension introduces the option for a [Secure Partition RoT Service](#) to be a [stateless RoT Service](#), as an alternative to a [connection-based RoT Service](#) which is defined in version 1.0. This enables much more efficient implementation of Secure Partition RoT Services that do not make use of the connection-related features of the version 1.0 interface.

Stateless RoT Services are a required feature in an implementation of version 1.1.

Connection-based RoT Services are an optional feature in an implementation of version 1.1.

Change for 1.1 Beta

Permitting an implementation to not provide support for connection-based service makes it possible to create a highly-optimized, static framework implementation that has no dynamic resource requirements, and associated runtime failure risk.

4.1 Background and rationale

Many RoT Service APIs provide standalone operations that do not require any non-volatile state or resources to be maintained by the RoT Service itself, or do not expose any kind of context or session to the caller of the API. For example, each function in the *PSA Storage API* [\[PSA-ITS\]](#) works atomically on the stored data.

To implement these functions as a Secure Partition RoT Service using the [\[FF-M\]](#) version 1.0 API, the client side implementation of the service must use one of the following techniques:

1. In every function, use `psa_connect()` to connect to the RoT Service, use `psa_call()` to request the operation, and then use `psa_close()` to release the connection handle.
2. Use `psa_connect()` to create a connection to the RoT Service once, and then store the connection handle for reuse by all the other service functions.

The first technique has a significant runtime overhead as it requires three calls to the SPM and the RoT Service for every operation.

The second technique removes that overhead, but requires a reliable way for the client code to use a global or static variable to hold the connection handle.

The client code for an RoT Service might be used in an NSPE application, or in a Secure Partition. In many instances, the RoT Service will have multiple clients within a single system. Separate clients should not share a single connection, as this conflicts with client identification and with client isolation requirements.

The result is that using a shared connection variable will not always have the right isolation and security properties for the service, depending on the framework implementation.

This analysis indicates that the version 1.0 framework is not sufficient to enable portable and efficient implementation of standalone RoT Service operations.

The appendix, [Implementing session-less RoT Services on page 103](#), provides a more detailed analysis of this challenge, and the approach to solving it for version 1.1.

4.2 Programming model

In version 1.1, each Secure Partition RoT Service is either *connection-based* or *stateless*.

All Secure Partition RoT Services in version 1.0 were connection-based, the description in this chapter focusses on the definition and behavior of stateless RoT Services.

4.2.1 Overview of stateless RoT Services

The service type is defined in the Secure Partition manifest file when defining the RoT Service. A single Secure Partition can contain both types of RoT Service.

Stateless RoT Services do not use connections:

- There is no call to `psa_connect()` or `psa_close()` by the client.
- There is no corresponding *connection* and *disconnection message* delivered to the RoT Service.
- The RoT Service cannot use the *handle* functionality.

Requests to the service are made by calling `psa_call()` using a fixed handle value for the RoT Service. The identifier name for the stateless RoT Service handle is defined by this specification, but the value of that handle is [IMPLEMENTATION DEFINED](#).

4.2.2 RoT Service identification

A connection-based RoT Service defines an RoT *Service ID* (SID). A client of the service uses this SID in a call to `psa_connect()`, before issuing requests to the service using the handle returned by `psa_connect()`. See the `services` attribute in [\[FF-M\] §4.1.1](#).

A stateless RoT Service does not require a client to call to `psa_connect()`, and the client does not use the SID to identify the service. Instead the client uses a special *stateless handle* for the RoT Service in the call to `psa_call()`.

The stateless handle is declared by the framework in the `psa_manifest/sid.h` manifest header file, alongside the SID. The stateless handle value is constructed by the framework in an [IMPLEMENTATION DEFINED](#) way.

The SID must still be defined for a stateless RoT Service. The SID can be used in a call to `psa_version()`.

Rationale

The use of an [IMPLEMENTATION DEFINED](#) value for the stateless handle permits the framework to optimize the routing of the request to the Rot Service, and also incorporate information to validate the version of the RoT Service.

The alternative approach would be to define a different function that includes the SID, RoT Service version and all of the request parameters. This cannot be optimized as effectively, because the SID value is defined by the RoT Service developer.

See [stateless_handle](#) for details on the specification and definition of the stateless handle for a stateless RoT Service.

4.2.3 RoT Service versioning

The version policy must still be enforced by the implementation when clients use a stateless handle to send a request to a stateless RoT Service.

The mechanism used to validate the version of a stateless RoT Service is [IMPLEMENTATION DEFINED](#).

Implementation note

For example, the following techniques can be used, depending on the implementation design:

- If the implementation builds all of the clients, services and frameworks together, then the version is assumed to match between the RoT Service client code and the RoT Service implementation code.
- The framework encodes the service version into the stateless handle value, along with the stateless handle index.
This enables the version expected by the client, which is compiled into the stateless handle value when the client was built, to be checked against the version running in the implementation when `psa_call()` is invoked. The implementation might cache the last stateless handle value used in order to elide the version checking on every call using a stateless handle.

4.2.4 Requesting stateless RoT Services

A client sends a request to a stateless RoT Service by using the [stateless handle](#), defined in the `psa_manifest/sid.h` manifest header file, with the `psa_call()` function.

All other parameters to a stateless RoT Service request are identical to the connection-based RoT Services defined in version 1.0. See *Requesting Services* in [\[FF-M\]](#) §3.3.2. *Using RoT Services*.

4.2.5 Processing RoT Service messages

Processing messages for connection-based RoT Services is described in [\[FF-M\]](#) §3.3.3. *Processing RoT Service messages*.

A stateless RoT Service does not receive a *connection* or *disconnection message* for any client.

Note:

A stateless RoT Service has no direct means to detect that a client has terminated or restarted.

In comparison, a connection-based RoT Service will receive a *disconnection message* from the framework if a client exits without explicitly closing the connection, or if the framework terminates the connection due to [PROGRAMMER ERROR](#).

A stateless RoT Service only receives *request messages* from the framework, that correspond to a client calling `psa_call()`. The message `type` value is always ≥ 0 .

A stateless RoT Service cannot use the [rhandle](#) functionality that is available to connection-based RoT Services. The `rhandle` value in the message is always `NULL`, and the RoT Service must not call `psa_set_rhandle()`.

Except for the unavailable `rhandle` functionality, a stateless RoT Service processes the *request message* in the same way as a connection-based RoT Service.

Rationale

The main use case for a stateless RoT Service is to replace the use of temporary connections in every request to the RoT Service.

One side effect of using a temporary connection for these use cases, is that the RoT Service cannot effectively use the connection's `rhandle`, because the connection itself is transient.

This makes the `rhandle` functionality redundant for this type of RoT Service.

4.2.6 Programmer Error

If a [PROGRAMMER ERROR](#) occurs when a request is sent to a stateless RoT Service, or during processing of the request, there is no connection to terminate.

However, the response to the caller is the same as defined in [\[FF-M\]](#) §3.5.2. *Programmer error*:

- If the source of the `PROGRAMMER ERROR` is a Secure Partition, the SPM must panic the Secure Partition in response to a `PROGRAMMER ERROR`.
- If the source of the `PROGRAMMER ERROR` is in the NSPE, the NSPE implementation of the Client API must implement one of the following behaviors:
 - Terminate the NSPE task or execution context that is the source of the `PROGRAMMER ERROR`.
 - Return `PSA_ERROR_PROGRAMMER_ERROR` to the NSPE task that called `psa_call()`.

If client execution continues after a `PROGRAMMER ERROR`, the client can make another call to the same stateless RoT Service using the stateless handle.

4.2.7 Comparison of service types

Table 5 Comparison of connection-based and stateless services

Action/item	Connection-based RoT Service	Stateless RoT Service
Connection	Explicit call to <code>psa_connect()</code> with the service SID and version from <code>psa_manifest/sid.h</code> .	Implicit. Calling <code>psa_connect()</code> with a stateless service SID is a PROGRAMMER ERROR .
Connection message	Delivered to the service for each call to <code>psa_connect()</code> . The service can accept or refuse the connection.	Not used.

Table 5 (continued)

Action/item	Connection-based RoT Service	Stateless RoT Service
rhandle	Calling <code>psa_set_rhandle()</code> on a <i>connection</i> or <i>request message</i> will set the rhandle value, which will be returned in the rhandle member of any future messages received on that connection.	Calling <code>psa_set_rhandle()</code> is a PROGRAMMER ERROR . The rhandle value in a received message is always NULL.
Making requests	Call <code>psa_call()</code> using a <i>connection handle</i> that was returned by a successful call to <code>psa_connect()</code> .	Call <code>psa_call()</code> using a <i>stateless handle</i> that is defined by the framework in the generated <code>psa_manifest/sid.h</code> header file.
Request messages	Delivered to service with <code>type</code> and <code>iovecs</code> from client, and rhandle value from a call to <code>psa_set_rhandle()</code> on a prior message.	Delivered to service with <code>type</code> and <code>iovecs</code> from client, and rhandle is NULL.
PROGRAMMER ERROR	Replying to a message with <code>PSA_ERROR_PROGRAMMER_ERROR</code> will terminate the connection, causing a <i>disconnection message</i> to be received. The client call might not return, or might return the error code leaving the connection in an error state.	A client call which is replied with <code>PSA_ERROR_PROGRAMMER_ERROR</code> might not return, or might return the error code.
Disconnection	Calling <code>psa_close()</code> with a <i>connection handle</i> will explicitly disconnect the connection. Connections can be disconnected by the service by responding to a message with <code>PSA_ERROR_PROGRAMMER_ERROR</code> . Connections might be disconnected by the framework when the client terminates.	Calling <code>psa_close()</code> with a <i>stateless handle</i> is a PROGRAMMER ERROR . The service is not informed if a client terminates.
Disconnection messages	Delivered to the service when the connection is closed for any reason.	Not used.

4.3 Implementation options

In an implementation of version 1.1 of the Firmware Framework:

- Support for connection-based RoT Services is optional
- Support for stateless RoT Services is mandatory

See [Permitted configurations of FF-M version 1.1 on page 22](#) for the list of compliant configurations of all of the optional features in version 1.1.

If the framework supports Secure Partitions within the PSA Root of Trust, then the framework is permitted to only support stateless RoT Services for this type of Secure Partition, even if it supports

connection-based RoT Services in an Application RoT Secure Partition. See also [PSA RoT Services and Secure Partitions on page 71](#).

4.4 Selecting the RoT Service type

Both connection-based RoT Services and stateless RoT Services can be used with either the [IPC model](#) or the [SFN model](#). See [Secure Functions on page 24](#).

It is recommended to define a Secure Partition RoT Service as *stateless*, if it consists entirely of stand-alone functions. This avoids the need for transient connections, and the performance overhead that these incur.

A framework implementation might not support connection-based RoT Services. Refer to the implementation documentation for details.

If the API exposes some form of context to the client, and this can be used to manage a connection handle, it is recommended that the RoT Service is *connection-based*. Using a connection does not require one of the limited number of *stateless handle indexes* in the framework.

If the RoT Service manages volatile state for the client, it is recommended that the RoT Service is *connection-based*. This allows the RoT Service implementation to utilise the [rhandle](#) functionality to manage resources for the client.

4.5 Changes to the Programming API

The changes to [\[FF-M\] §4 Programming API](#) are described in the following sections:

- [Manifest changes](#)
- [Client API changes on page 39](#)
- [Secure Partition API changes on page 40](#)

4.5.1 Manifest changes

connection_based (attribute)

This is a required attribute for service definitions in a Secure Partition manifest that is using "psa_framework_version": 1.1.

connection_based is a boolean attribute, and can take the value true or false:

true	The service is a <i>connection-based</i> service
false	The service is <i>stateless</i> service

A manifest file that defines "psa_framework_version": 1.0 is implicitly defining all services as connection-based.

The implementation reports an error if the selected RoT Service type is not supported by the framework.

stateless_handle (attribute)

This is an optional attribute for service definitions in the Secure Partition manifest which define a stateless RoT Service. A connection-based RoT Service must not have a `stateless_handle` attribute.

If specified, `stateless_handle` must either be "auto" or a small positive number between 1 and an [IMPLEMENTATION DEFINED](#) maximum value.

The `stateless_handle` specifies a *stateless handle index*, which is used by the implementation to construct the *stateless handle value* for this RoT Service.

If the manifest defines a `stateless_handle` to be "auto", the implementation allocates a stateless handle index for this service. This is also the default behavior if there is no `stateless_handle` attribute specified for a stateless RoT Service.

The implementation must support at least 32 stateless handle indexes.

The stateless handle index for a stateless RoT Service must be unique within the system, whether the index is defined in the manifest or allocated by the implementation.

The implementation defines a macro for the stateless handle in `psa_manifest/sid.h`, of the following form:

```
#define «name»_HANDLE ((psa_handle_t) «stateless_handle_value» )
```

where «name» is the name of the service and the «stateless_handle_value» is constructed by the implementation, using the stateless handle index.

This stateless handle is used by a client when making requests to the service.

Implementation note

The `stateless_handle` attribute has the following JSON definition:

```
"stateless_handle": {
  "description": "Optional: The index for a stateless handle for this service.",
  "anyOf": [
    { "$ref": "#/definitions/positive_integer_or_hex_string" },
    { "const": "auto" }
  ]
}
```

4.5.2 Client API changes

psa_connect (function)

This existing function is constrained to be used only for connection-based RoT Services.

The `sid` passed to `psa_connect()` must be the SID of a connection-based RoT Service.

Calling `psa_connect()` with the SID of a stateless RoT Service is a [PROGRAMMER ERROR](#).

The handle returned by a successful call to `psa_connect()` is a *connection handle*.

psa_call (function)

This existing function works with both stateless and connection-based RoT Services.

The type of `handle` passed to `psa_call()` depends on the type of RoT Service being requested:

- For a stateless RoT Service, `handle` must be a *stateless handle* that is defined in the `psa_manifest/sid.h` file.
- For a connection-based RoT Service, `handle` must be a *connection handle* that was returned by a previous call to `psa_connect()`.

psa_close (function)

This existing function is constrained to be used only for connection-based RoT Services.

The `handle` passed to `psa_close()` must either be the *null handle*, or a *connection handle* returned by a previous call to `psa_connect()`.

Passing a *stateless handle* to `psa_close()` is a [PROGRAMMER ERROR](#).

4.5.3 Secure Partition API changes

psa_msg_t (type)

This existing type behaves differently for stateless RoT Services.

The `rhandle` member of the `psa_msg_t` object received by a stateless RoT Service is always `NULL`.

psa_set_rhandle (function)

This existing function is constrained to be used only for connection-based RoT Services.

Calling `psa_set_rhandle()` on a message for a stateless RoT Service is a [PROGRAMMER ERROR](#) and will not return.

Replying to a request message with `PSA_ERROR_PROGRAMMER_ERROR`

Replying to a message for a stateless RoT Service with `PSA_ERROR_PROGRAMMER_ERROR` has no effect on the service, as there is no connection to terminate abnormally. This has the same effect on the client as for a connection-based service.

Note:

The API for replying to a message depends on the Secure Partition model:

- In a Secure Partition that is using the [IPC model](#), call `psa_reply()` with the message status.
- In a Secure Partition that is using the [SFN model](#), return from the [Secure Function](#) with the message status.

See [Secure Functions on page 24](#).

5 Memory-mapped IOVECs

This extension introduces the ability for [Secure Partition RoT Service](#) code to map the client input and output buffer parameters into the Secure Partition, enabling direct access to the buffer memory in the client.

Direct access to parameter data in the client is of value, especially in small systems, where the overhead of copying buffers of data into the Secure Processing Environment could be prohibitive.

This extension is called *Memory mapped iovecs* (MM-IOVEC), in reference to the input and output vectors used to pass parameters to an RoT Service.

The new APIs defined in this section are optional in an implementation of version 1.1. The new feature discovery APIs enable portable Secure Partition RoT Service code to use the MM-IOVEC functionality in systems that support MM-IOVEC. See [Discovering framework feature availability on page 21](#).

5.1 Background and rationale

The Secure Partition API in *Arm® Platform Security Architecture Firmware Framework [FF-M]* provides no direct access from a Secure Partition RoT Service to the client input and output vectors. Instead, RoT Services have to read from an input vector into their own memory, and write to an output vector from their own memory, using the provided Secure Partition API.

This approach is aligned with the design goals for version 1.0:

- It enables the framework to be implemented on systems where the SPE cannot directly access some or all of the NSPE memory. For example, in a System-on-Chip where the NSPE is using a 40-bit physical address space and the SPE is running on a 32-bit CPU that can only address a 32-bit address space.
- It prevents many common errors in secure service implementation, that frequently lead to exploitable vulnerabilities. The API design prevents services from introducing double-fetch, buffer overflow, alignment and access validation failure vulnerabilities — mitigating these is done by the framework implementation.
- It discourages passing a pointer within the input vector, which then points to other data in the client. This technique will not work if the SPE cannot access NSPE memory, or if the client and the RoT Service are configured with different memory address translation.

The [Library model](#) in the *Trusted Firmware-M [TF-M]* project provides a much simpler framework for developing security services, in which a security service is provided with pointers to the client parameters. This is better suited to constrained devices, as directly accessing client memory does not require the service to copy an input vector into the security service memory before using it.

The threat model that is used by FF-M assumes that the attacker can execute arbitrary code in the NSPE. This requires that the service developer consider the threats that are posed by the attacker being in control of the location and content of the client input and output vectors passed to the service:

- The memory location of the vector might not be within the client's accessible memory.

- The memory location of the vector might not be accessible by the RoT Service, or have the correct access permissions.
- Reading the same memory address twice might not produce the same result.
- The pointer might not be aligned in the expected way, that is, the natural structure alignment provided by the C compiler.
- Incorrect service code can cause a read or write outside of the supplied buffer.

In practice, writing services securely in the Library model requires that the developers copy most input vectors from the client into their own memory to mitigate these risks – eliminating the apparent benefit of direct access to the memory.

However, The following use cases can benefit significantly from being able to directly read input vectors and write output vectors in the client memory:

- Processing of large data buffers, for example, by cryptographic algorithms.
- Transferring data through multiple Secure Partitions, potentially processed at each step. Direct client memory access would avoid the need to have additional copies of data at each stage in the chain.

5.2 Programming model

The version 1.0 APIs for accessing client input and output vectors, `psa_read()` and `psa_write()`, are required for a version 1.1 implementation.

MM-IOVEC is an additional mechanism for accessing the content of client input and output vectors to the Secure Partition RoT Service. Direct mapping of client input and output vectors into the Secure Partition provides a memory and runtime optimization for larger buffers, but reduces mitigation for common security vulnerabilities, and can reduce the effective isolation provided by the framework.

A Secure Partition RoT Service cannot mix the use of the existing read and write functions with the MM-IOVEC functions when accessing an input or output vector. This avoids complex interactions between these different access mechanisms, and simplifies the implementation.

5.2.1 Implementation flexibility

Support for MM-IOVEC is optional in an implementation of the framework. There are various reasons for an implementation to exclude support for MM-IOVEC, for example:

- Direct mapping of client memory can be inefficient, or even impossible, for otherwise compliant implementations of [\[FF-M\]](#).
- Direct access to client memory might be denied by the security requirements for the system.

The MM-IOVEC discovery API provides a mechanism for Secure Partition RoT Service developers to determine if the implementation supports MM-IOVEC. This enables compile-time variation of RoT Service code to select the best access mechanism. See [Discovering framework feature availability on page 21](#).

5.2.2 Typical deployment scenarios

The MM-IOVEC API and the discovery API is designed to support the following framework implementation scenarios:

1. A system that cannot or must not permit security services to have direct access to client memory. This can be a result of technical limitations of the platform, or security requirements for the product. The framework implementation reports that the MM-IOVEC functionality is not present, and the MM-IOVEC APIs are not provided by the implementation.
2. A constrained system that uses isolation level 1, where all code within the SPE can access data in the SPE and NSPE. The framework implementation provides an implementation of the MM-IOVEC functionality. There is little cost to the framework to provide direct access to client input and output vectors, because the memory access is already permitted. The MM-IOVEC interface is simple to implement and has no runtime failure modes.
3. A more complex system that can dynamically create mappings for client input and output vectors when requested through the MM-IOVEC API. The framework implementation can provide an implementation of the MM-IOVEC functionality, if it guarantees that any mapping request will succeed. If it is not possible to guarantee that the mapping can be created for a valid MM-IOVEC request, then the implementation does not provide MM-IOVEC functionality. For example, if the mapping would require dynamic allocation of memory, or the use of another limited, shared resource.

5.2.3 RoT Service configuration

Direct access to client memory can provide a powerful way to escalate an attack against an RoT Service. To reduce the attack surface for Secure Partition RoT Services that do not use MM-IOVEC, a Secure Partition RoT Service must explicitly enable the functionality using the `mm_iovec` attribute within the service specification in the Secure Partition manifest file. See [Enabling the MM-IOVEC API on page 45](#).

The `mm_iovec` attribute has no effect if the framework does not support MM-IOVEC.

5.2.4 Accessing client input and output vectors

If the implementation supports MM-IOVEC, a Secure Partition RoT Service which has enabled MM-IOVEC can use either MM-IOVEC or the existing `psa_read()` and `psa_write()` functions to access each input or output vector.

Once an input or output vector has been accessed using one of the `psa_read()`, `psa_skip()`, or `psa_write()` functions, the vector cannot be mapped using MM-IOVEC functions.

Similarly, once an input or output vector has been mapped using an MM-IOVEC function, the vector cannot be accessed using any of the `psa_read()`, `psa_skip()`, or `psa_write()` functions.

It is a **PROGRAMMER ERROR** to try and map a zero-length input or output vector.

The framework unmaps an input or output vector either in response to an unmapping call from the RoT Service, or automatically when message processing is completed.

An explicit unmapping call is required for output vectors when the RoT Service has written data into the output vector, to report the number of bytes written to the client. If no unmapping call is made, output vectors will report that no data has been written to the client.

Explicitly unmapping other input and output vectors is optional. In some framework implementations, this can release framework resources that are required to create the mapping.

5.2.5 Interaction with the isolation model

In an implementation that provides a high level of isolation, MM-IOVEC provides a mechanism that can conflict with the isolation rules. For example:

- In a system using isolation level 3, a Secure Partition is not permitted to access another Secure Partition's Private data. MM-IOVEC can provide a mechanism for one Secure Partition to access the other's Private data.
- In a system that implements isolation rule **I6** (see [FF-M] §3.1.5), only the SPM is permitted to access memory in another protection domain when required. MM-IOVEC can provide access from a Secure Partition directly to client memory.

Access to an input or output vector's buffer for the duration of the call is expected by the client, so this does not itself present a new attack surface. However, the mechanisms that an implementation uses to map input and output vectors can be imprecise. That is, the mapping mechanism can provide more access than is strictly required. Hardware limitations that can impact mapping precision include the following examples:

- The Memory Protection Unit (MPU) used to control access has size and alignment constraints, so that all regions must start and end on 32-byte boundaries. Other memory mapping techniques can have much larger alignment requirements, such as the 4096-byte pages in the Armv7-A virtual memory architecture.
- The MPU used to control access cannot provide write-only permission, so any existing data in an output vector can be read by the RoT Service.

It is **IMPLEMENTATION DEFINED** how such imprecise mappings are handled by the implementation. The decision to permit imprecise mappings depends on the security requirements for the system, and the nature of the additional access.

An implementation that provides MM-IOVEC functionality must document its behavior when an input or output vector mapping is imprecise.

5.3 Changes to the Programming API

5.3.1 Discovering MM-IOVEC availability

See [Discovering framework feature availability on page 21](#).

5.3.2 Enabling the MM-IOVEC API

A new manifest attribute is introduced in version 1.1 manifest files. The implementation must accept this attribute in a manifest file, whether or not the framework implements MM-IOVEC.

To use this API in a Secure Partition RoT Service, the service definition in the Secure Partition manifest file must include the attribute `mm_iovec`, with the value "enable".

`mm_iovec` (attribute)

This is an optional attribute for service definitions in a Secure Partition manifest that is using "psa_framework_version": 1.1.

`mm_iovec` takes one of the following values:

"enable"	If the framework supports MM-IOVEC, then the MM-IOVEC APIs are enabled for messages to the RoT Service.
"disable"	If the framework supports MM-IOVEC, then using the MM-IOVEC APIs for messages to the RoT Service is a PROGRAMMER ERROR . This is the default value if the <code>mm_iovec</code> attributes is not specified.

5.3.3 Mapping RoT Service IO vectors

The MM-IOVEC Secure Partition API, for mapping and unmapping Secure Partition RoT Service input and output vectors. The following API elements are added to `psa/service.h`:

- `psa_map_invec()`
- `psa_unmap_invec()`
- `psa_map_outvec()`
- `psa_unmap_outvec()`

`psa_map_invec` (function)

Map a client input vector for direct access by a Secure Partition RoT Service.

```
const void * psa\_map\_invec(psa_handle_t msg_handle,  
                           uint32_t invec_idx);
```

Parameters

<code>msg_handle</code>	Handle for the client's message.
<code>invec_idx</code>	Index of the input vector to map. Must be less than <code>PSA_MAX_IOVEC</code> .

Returns: `const void *`

A pointer to the input vector data.

Programmer Error

The call is a [PROGRAMMER ERROR](#) if one or more of the following are true:

- MM-IOVEC has not been enabled for the RoT Service that received the message.
- `msg_handle` is invalid.
- `msg_handle` does not refer to a *request message*.
- `invec_idx >= PSA_MAX_IOVEC`
- The input vector has length zero.
- The input vector has already been mapped using `psa_map_invec()`.
- The input vector has already been accessed using `psa_read()` or `psa_skip()`.

A PROGRAMMER ERROR will panic the caller.

Availability

The API is optional in version 1.1. Use [PSA_FRAMEWORK_HAS_MM_IOVEC](#) to determine the availability of this function.

To call this function, the MM-IOVEC functionality must be enabled for the RoT Service using the `mm_iovec` attribute in the Secure Partition manifest file.

Description

This function will provide a mapping of a non-zero length client input vector in the RoT Service address context, allowing the service to read the vector data directly.

Warning: Using this API exposes the RoT Service to vulnerabilities caused by invalid assumptions about the input vector data, or errors in the RoT Service code. For example:

- Reading the same memory address twice can produce different results.
- The pointer can be incorrectly aligned for the data type being accessed.
- Incorrect RoT Service code can cause an undetected read outside of the client input vector.

It is a PROGRAMMER ERROR to call this function if the length of the input vector is zero.

The RoT Service must not read more data than specified by the input vector size. The input vector size is provided in the corresponding `in_size[]` element in the request's `psa_msg_t` object.

When the RoT Service has finished processing the input vector, it can remove the mapping by calling `psa_unmap_invec()` with the same message handle and input vector index.

When the message processing is completed, the framework removes all input vector mappings for that message.

psa_unmap_invec (function)

Unmap a previously mapped client input vector from a Secure Partition RoT Service.

```
void psa_unmap_invec(psa_handle_t msg_handle,  
                    uint32_t invec_idx);
```

Parameters

msg_handle	Handle for the client's message.
invec_idx	Index of the input vector to unmap. Must be less than PSA_MAX_IOVEC.

Returns: void

Programmer Error

The call is a [PROGRAMMER ERROR](#) if one or more of the following are true:

- msg_handle is invalid.
- msg_handle does not refer to a *request message*.
- invec_idx >= PSA_MAX_IOVEC
- The input vector has not been mapped by a call to [psa_map_invec\(\)](#).
- The input vector has already been unmapped by a call to [psa_unmap_invec\(\)](#).

A PROGRAMMER ERROR will panic the caller.

Availability

The API is optional in version 1.1. Use [PSA_FRAMEWORK_HAS_MM_IOVEC](#) to determine the availability of this function.

To call this function, the MM-IOVEC functionality must be enabled for the RoT Service using the [mm_iovec](#) attribute in the Secure Partition manifest file.

Description

This function will remove a previously successful mapping of a client input vector from the RoT Service address context.

Following this call, the RoT Service must not read from the input vector memory.

If [psa_unmap_invec\(\)](#) is not called for an input vector that has been mapped, the framework will remove the mapping automatically when the message is completed.

psa_map_outvec (function)

Map a client output vector for direct access by a Secure Partition RoT Service.

```
void * psa_map_outvec(psa_handle_t msg_handle,  
                    uint32_t outvec_idx);
```


Parameters

<code>msg_handle</code>	Handle for the client's message.
<code>outvec_idx</code>	Index of the output vector to map. Must be less than <code>PSA_MAX_IOVEC</code> .

Returns: `void *`

A pointer to the output vector data.

Programmer Error

The call is a [PROGRAMMER ERROR](#) if one or more of the following are true:

- MM-IOVEC has not been enabled for the RoT Service that received the message.
- `msg_handle` is invalid.
- `msg_handle` does not refer to a *request message*.
- `outvec_idx >= PSA_MAX_IOVEC`
- The output vector has length zero.
- The output vector has already been mapped using `psa_map_outvec()`.
- The output vector has already been accessed using `psa_write()`.

A PROGRAMMER ERROR will panic the caller.

Availability

The API is optional in version 1.1. Use [PSA_FRAMEWORK_HAS_MM_IOVEC](#) to determine the availability of this function.

To call this function, the MM-IOVEC functionality must be enabled for the RoT Service using the `mm_iovec` attribute in the Secure Partition manifest file.

Description

This function will provide a mapping of a non-zero length client output vector in the RoT Service address context, allowing the service to write the output vector data directly.

Warning: Using this API exposes the RoT Service to vulnerabilities caused by invalid assumptions about the output vector data, or errors in the RoT Service code. For example:

- Reading the same memory address twice can produce different results.
- The pointer can be incorrectly aligned for the data type being accessed.
- Incorrect RoT Service code can cause an undetected read or write outside of the client output vector.

It is a PROGRAMMER ERROR to call this function if the length of the output vector is zero.

The RoT Service must not write more data than specified by the output vector size. The output vector size is provided in the corresponding `out_size[]` element in the request's `psa_msg_t` object.

When the RoT Service has finished processing the output vector, it can remove the mapping and report the number of bytes written by calling `psa_unmap_outvec()` with the same message handle, output vector index, and the number of bytes written.

When the message processing is completed, the framework removes all output vector mappings for that message. Any output vectors that are still mapped will report that zero bytes have been written.

psa_unmap_outvec (function)

Unmap a previously mapped client output vector from a Secure Partition RoT Service.

```
void psa_unmap_outvec(psa_handle_t msg_handle,
                     uint32_t outvec_idx,
                     size_t len);
```

Parameters

<code>msg_handle</code>	Handle for the client's message.
<code>outvec_idx</code>	Index of the output vector to unmap. Must be less than <code>PSA_MAX_IOVEC</code> .
<code>len</code>	The number of bytes written to the output vector. This must be less than or equal to the size of the output vector.

Returns: void

Programmer Error

The call is a **PROGRAMMER ERROR** if one or more of the following are true:

- `msg_handle` is invalid.
- `msg_handle` does not refer to a *request message*.
- `outvec_idx >= PSA_MAX_IOVEC`
- `len` is greater than the output vector size.
- The output vector has not been mapped by a successful call to `psa_map_outvec()`.
- The output vector has already been unmapped by a call to `psa_unmap_outvec()`.

A **PROGRAMMER ERROR** will panic the caller.

Availability

The API is optional in version 1.1. Use `PSA_FRAMEWORK_HAS_MM_IOVEC` to determine the availability of this function.

To call this function, the MM-IOVEC functionality must be enabled for the RoT Service using the `mm_iovec` attribute in the Secure Partition manifest file.

Description

This function will remove a previously successful mapping of a client output vector from the RoT Service address context, and update the caller's `psa_outvec` structure with the number of bytes written to the output vector.

Following this call, the service must not write to the output vector memory.

If `psa_unmap_outvec()` is not called for an output vector that has been mapped, the framework will remove the mapping automatically when the message is completed. In this situation, the caller's `psa_outvec` structure is updated to state that zero bytes have been written to the output vector.

Note:

The API makes it possible for the RoT Service to write more bytes to the mapped output vector than it claims in the call to `psa_unmap_outvec()`, or to write less bytes than reported. When using `psa_write()`, the bytes in the buffer up to the number reported in the caller's `psa_outvec` are written by the RoT Service, and those after that point are typically unmodified.

However, [FF-M] makes no guarantee that buffer contents after `psa_outvec::len` are unmodified by `psa_call()`. *PSA Cryptography API [PSA-CRYPT]* §5.2.3 specifically says this region content is “unspecified”.

5.3.4 Changes to existing Secure Partition APIs

`psa_read` (function)

This existing function cannot be used after an input vector has been mapped using MM-IOVEC.

It is a **PROGRAMMER ERROR** to call `psa_read()` for an input vector that has been mapped using `psa_map_invec()`.

`psa_skip` (function)

This existing function cannot be used after an input vector has been mapped using MM-IOVEC.

It is a **PROGRAMMER ERROR** to call `psa_skip()` for an input vector that has been mapped using `psa_map_invec()`.

`psa_write` (function)

This existing function cannot be used after an output vector has been mapped using MM-IOVEC.

It is a **PROGRAMMER ERROR** to call `psa_write()` for an output vector that has been mapped using `psa_map_outvec()`.

6 Enhancements for Secure Partition peripheral drivers

This extension adds the following support for implementing secure peripheral drivers in Secure Partitions:

- [First-level interrupt handling](#) (FLIH) is a de-privileged, low-latency, interrupt handling capability for Secure Partitions. This enables Secure Partitions to be used for peripheral drivers that require secure interrupts to be handled within a bounded time.
- An API for managing interrupts supports FLIH and fills a gap in the version 1.0 API.
- Accessors for MMIO registers ensure more portability between implementations and system architectures.

These features are required in an implementation of version 1.1 of FF-M.

Note:

This extension does not provide a standard framework for running secure interrupt handlers in privileged modes - this remains an implementation-specific option, and is not recommended in general for systems that provide high levels of isolation.

6.1 Background and rationale

Arm® Platform Security Architecture Firmware Framework [\[FF-M\]](#) version 1.0 provides some support for handling interrupts within a Secure Partition.

However, the API presents two significant issues for implementing peripheral drivers in many use cases:

- The signal-based mechanism in version 1.0 makes it difficult to write drivers that need interrupts to be handled in a bounded time. Interrupt handling code runs within a Secure Partition thread, which is subject to delays due to scheduling of other threads and due to completion of current activity within the Secure Partition thread.
- The simple interface in v1.0 assumes that the Secure Partition will not need to manage the interrupt, except via the peripheral's own memory-mapped register interface. This assumption is not always valid, and requires the framework implementation to provide an implementation specific API for this.

6.1.1 Bounded interrupt response time

The [\[FF-M\]](#) version 1.0 API was designed to be simple and easy to use securely. It avoided concurrent execution within the Secure Partition by requiring the interrupt handling to run within the execution thread of the Secure Partition.

Working around the limitations of this API for lower-latency and bounded response time interrupt requirements is painful, complex, and error-prone.

Framework support for handling interrupts within a bounded response time is necessary to enable drivers for such peripherals to be written as Secure Partitions. This would enable more peripheral driver code to be run in a Secure Partition, reducing the risk of vulnerabilities in the PSA Root of Trust.

6.1.2 Managing interrupts

The [FF-M] version 1.0 API made assumptions about how peripherals behaved with respect to generation of interrupts. Specifically, for each interrupt that can be generated by the peripheral:

- There is a mechanism which can be used to disable and enable the interrupt at source. Typically, this is a control register provided by the peripheral.
- The peripheral resets with the interrupt disabled.

Together these requirements would allow a Secure Partition to manage an interrupt source with just the MMIO register interface to the device, without needing any way to control the interrupt's handling at the Interrupt Controller.

In reality, not all peripherals meet these requirements at design time, and sometimes peripheral hardware has implementation defects that can generate spurious interrupts. FF-M needs to accommodate real peripherals - this requires changes to the interrupt model, and the addition of interfaces to support managing the interrupts within the framework.

6.1.3 Accessing MMIO registers

In some systems, accessing a memory-mapped peripheral register is not possible using normal memory read and write operations. For example:

- The alignment requirements for access to a memory-mapped peripheral register can be stricter than those for access to normal memory locations.
- The system cannot provide direct access to all requested MMIO regions due to limitations of the memory protection hardware.

To ensure correctness and code portability for drivers that access MMIO registers, the framework needs to provide access functions that implement any system-specific requirements related to the register access.

6.2 Programming model

6.2.1 Definitions

To distinguish between different approaches to handling interrupts, the following terms are used within this document:

First-level interrupt handling (FLIH):

A type of interrupt handling which is carried immediately when an interrupt exception takes place. The handling occurs within an exception context, and might be privileged or deprived.

Second-level interrupt handling (SLIH):

A type of interrupt handling that is deferred until after the interrupt exception. This handling occurs within a thread context, and is subject to normal scheduling.

FLIH function:

A function that is used to perform *First-level interrupt handling* for a specific interrupt within a Secure Partition. An FLIH function runs in an *FLIH context*.

Secure Partition thread context:

The main execution context within a Secure Partition.

- For a Secure Partition that is using the *IPC model*, this is the Secure Partition thread.
- For a Secure Partition that is using the *SFN model*, this is any *Secure Function* within the Secure Partition.

FLIH context:

The execution context within a Secure Partition that is used to run a *FLIH Functions*. See also *FLIH Execution model on page 56*.

The interrupt model in [FF-M] version 1.0 is equivalent to Second-level interrupt handling as defined in the version 1.1 Extensions.

This extension to the framework enables a Secure Partition developer to provide First-level interrupt handling code within the Secure Partition.

6.2.2 Impact of Isolation

If a Secure Partition developer is able to provide code that runs as part of FLIH, then the isolation principles of *PSA Security Model [PSA-SM]* and [FF-M] require that this code runs with the same access to resources as the other code within the Secure Partition. That is, the FLIH function must execute within the same *protection domain* as the Secure Partition to which it belongs.

At isolation levels 2 or 3, this can require a context switch to run the FLIH function in a deprivileged state, outside of the protection domain containing the SPM.

Although bounded, the latency of a deprivileged FLIH function can still be inadequate for some use cases, which require that the interrupt is handled inside the SPM to meet very strict latency requirements. A vulnerability in this type of interrupt handler would put the entire PSA Root of Trust at risk.

This document does not define a standard framework for interrupt handling within the SPM. Provision of support for this is *IMPLEMENTATION DEFINED*.

6.2.3 Impact of Concurrency

FLIH functions, even when run within the Secure Partition protection domain, can run concurrently or interleaved with the execution of the *Secure Partition thread context*. This introduces the possibility of a data-race when two execution contexts access the same Secure Partition private data. For example, these two contexts can be the Secure Partition thread context and an FLIH function, or two FLIH functions with different priorities.

FLIH functions must be written to be data-race free in relation to other FLIH functions and code running in the Secure Partition thread context. Preventing data races requires support from the framework, either in the form of atomic data access functions or a mechanism that enables the developer to construct critical sections within the code.

6.2.4 Interrupt model

This section defines the interrupt model that is provided by the framework for Secure Partition device drivers. The model supports the provision of both FLIH and SLIH within a Secure Partition.

Rationale

This interrupt model is defined so that framework implementers and Secure Partition driver developers have a common understanding of the interfaces that support interrupt management and handling. This platform-independent model can ensure that the behavior is consistent across different hardware systems and implemented isolation levels.

Note:

This interrupt model borrows terminology from real system architectures. However, this is an abstract model, and the elements, states and transitions described with these terms will not necessarily align with the same concepts as used in a real system implementation.

Each interrupt that might be handled by a Secure Partition is identified by a *source* - which is specified in the Secure Partition manifest. This identifies the logical origin of the interrupt. The interrupt source is identified by a number or a name, and this is resolved in an [IMPLEMENTATION DEFINED](#) manner.

The interrupt source can generate an interrupt at any time, usually subject to configuration and operational programming of the source peripheral. An interrupt is *asserted* if its source is generating the interrupt.

Interrupts arrive at the Interrupt Controller (IC), typically attached to (or part of) the CPU. The IC is responsible for filtering the interrupt inputs, determining if the CPU should be interrupted, triggering an appropriate exception and identifying which interrupt the CPU should process.

Each interrupt has an associated *enabled* status, maintained by the IC. This enabled status is independent of any configuration control on the source peripheral. An asserted interrupt will have no effect if it is not also enabled. The Secure Partition API provides functions for setting and clearing the enabled status of an interrupt, see [Secure Partition API changes for interrupt control on page 61](#).

When an enabled and asserted interrupt is selected for handling by the CPU following an interrupt exception, the interrupt becomes *active* and interrupt handling begins immediately. An active interrupt cannot cause another interrupt exception until the interrupt has been deactivated by *end of interrupt* (EOI) processing, which also marks the end of interrupt handling.

After system reset, on initial entry into [Secure Partition thread context](#), the Secure Partition interrupts are all inactive. The initial interrupt enabled status is determined by the `psa_framework_version` attribute in the Secure Partition manifest file. See [Secure Partition execution model on page 57](#).

The two mechanisms available for handling an interrupt in a Secure Partition are [First-level interrupt handling](#) (FLIH) or [Second-level interrupt handling](#) (SLIH).

First-level interrupt handling

This type of interrupt handling is suitable for use cases where the response to the interrupt has a latency requirement that cannot be met using SLIH.

FLIH uses a callback function in the Secure Partition for the FLIH. The [FLIH function](#) runs in a special execution context:

- An FLIH function can interrupt the [Secure Partition thread context](#).
- Higher priority interrupts can preempt an FLIH function.
- An FLIH function can only use a small subset of the Secure Partition APIs.

See [FLIH Execution model on page 56](#).

On return from the FLIH function, the interrupt handling is finished and the interrupt is deactivated.

The framework will also set the Secure Partition interrupt signal, depending on the value returned by the FLIH function. The Secure Partition signal will then result in the scheduling of the Secure Partition thread context. When the Secure Partition thread context runs in response to the signal, it must clear the signal by calling `psa_reset_signal()`. Some example usage patterns for FLIH are described in [Programming patterns using FLIH on page 66](#).

Rationale

A distinct API is proposed for clearing the signal when using FLIH for the following reasons:

- This is not end-of-interrupt processing, as that already occurred at the end of the FLIH function.
- The programming pattern for clearing the signal when using FLIH is different from the pattern when using SLIH. Using a distinct API name makes it easier to review and maintain the code correctly.
- The required behavior is most like clearing the doorbell signal, which currently uses the API `psa_clear()`.

Second-level interrupt handling

This type of interrupt handling is suitable for use cases where there is no time limit for both responding to the interrupt, and for processing the related data.

SLIH defers all interrupt handling to the [Secure Partition thread context](#). This is achieved by the framework setting the Secure Partition interrupt signal during the interrupt exception and then returning to normal scheduling operation.

The end-of-interrupt occurs when the Secure Partition thread context calls `psa_eoi()`, which clears the interrupt signal and deactivates the interrupt.

The SLIH runs within the Secure Partition thread context:

- The SLIH can use all Secure Partition APIs, including calls that can block or complete messages.
- The SLIH is sequential with respect to other code within the same Secure Partition.
- The SLIH runs at the Secure Partition execution priority, and can be interrupted or preempted.

Second-level interrupt handling is the only mechanism available in the [\[FF-M\]](#) version 1.0 API.

Implementation note

In this model, the interrupt remains *active* until the Secure Partition calls `psa_eoi()`. In many systems, this does not map directly onto the underlying interrupt state:

- The SLIH runs within the Secure Partition thread context at a normal scheduling priority, rather than in an interrupt context at elevated execution priority.
- Interrupt exceptions are not masked when executing in Secure Partition thread context. However, until the SLIH runs, the source peripheral can continue to assert the interrupt. The framework must ensure that this does not result in continuous interrupt exceptions.

In some systems, the behavior required by SLIH might be supported by the interrupt controller. In other systems, the behavior can be implemented by the framework disabling the interrupt before exiting the interrupt exception, and then enabling the interrupt again when `psa_eoi()` is called.

6.2.5 FLIH Execution model

The *FLIH function* is executed as soon as possible after the interrupt occurs, only being delayed by higher priority exceptions and interrupts. This contrasts with the detection of an interrupt signal in Secure Partition thread context when using SLIH.

The FLIH function preempts any existing *Secure Partition thread context*, as well as any lower priority FLIH functions.

An FLIH function can be preempted by a higher priority exception, including other Secure Partition interrupts and FLIH functions. FLIH functions can nest, with handling completed in LIFO order, but not interleave in an arbitrary manner. An FLIH function cannot be preempted by a Secure Partition thread context. Interrupt priorities are not defined in this specification, but can be provided by an *IMPLEMENTATION DEFINED* interface.

The FLIH function executes within the target Secure Partition *protection domain* and memory context. That is, the FLIH function has the same access to memory resources as the other code in its Secure Partition.

The execution stack and context on which the FLIH function runs is *IMPLEMENTATION DEFINED*.

Implementation note

The possible implementation options for the FLIH execution context depend on the isolation level and Secure Partition model:

- If the Secure Partition is in the same protection domain as the SPM, then the FLIH function can run directly on the processor's interrupt stack, as it is not necessary to deprive the FLIH.
 - If the FLIH function cannot run on the SPM stack, the implementation can run the FLIH function on the Secure Partition's main stack or on a dedicated FLIH stack.
-

The FLIH function must follow the appropriate procedure call standard (ABI) for the system's architecture. This permits the framework to invoke the FLIH function as a standard C function without special register state management.

An FLIH function is not permitted to access most of the Secure Partition API. It is *IMPLEMENTATION DEFINED* whether the framework permits an FLIH function to call the following functions:

- The MMIO accessor functions, see [Register access functions for MMIO on page 63](#).
- The interrupt control functions, see [Secure Partition API changes for interrupt control on page 61](#).

Attempting to call any other Secure Partition function from within an FLIH function is a **PROGRAMMER ERROR**. Responding to request messages can only happen in the Secure Partition thread context. End-of-interrupt processing occurs on return from the FLIH function, prior to resuming the interrupted execution.

The framework can provide additional, implementation-specific interfaces for FLIH functions.

Change for 1.1 Beta

Relaxing the framework requirements for the FLIH function interface enables framework optimization.

FLIH functions are specific to the peripheral that is the interrupt source. As a result, making FLIH function code also dependent on the framework is not a major issue.

An FLIH function uses its return value to indicate to the SPM how to complete the FLIH processing:

PSA_FLIH_NO_SIGNAL	The framework does not set the interrupt signal
PSA_FLIH_SIGNAL	The framework sets the interrupt signal
PSA_FLIH_DISABLE	The framework disables the interrupt
PSA_FLIH_SIGNAL PSA_FLIH_DISABLE	The framework sets the interrupt signal and disables the interrupt
PSA_FLIH_PANIC	The framework panics the Secure Partition

6.2.6 Secure Partition execution model

In [\[FF-M\]](#) version 1.0, the framework enables all Secure Partition interrupts before first entry to the Secure Partition code.

For version 1.1, the initial state of interrupts depends on the framework version that is specified in the Secure Partition manifest file:

Framework version	Initial interrupt state
"psa_framework_version": 1.0	The framework enables all interrupts in the Secure Partition
"psa_framework_version": 1.1	The framework disables all interrupts in the Secure Partition

For an interrupt that is initially disabled, the Secure Partition enables the interrupt explicitly with a call to [psa_irq_enable\(\)](#). For example, this can be done in the Secure Partition `entry_point` or `entry_init` function after initializing the interrupt's source peripheral.

The `PSA_FRAMEWORK_VERSION` pre-processor macro can be used to identify the framework version, if the Secure Partition code can be built for different versions of the framework.

6.3 Changes to the Programming API

The changes to [FF-M] §4 *Programming API* are described in the following sections:

- [Manifest changes](#)
- [Secure Partition API changes for FLIH on page 59](#)
- [Secure Partition API changes for interrupt control on page 61](#)

6.3.1 Manifest changes

The `irq` Secure Partition manifest attribute is redefined to align its approach to naming with the service attributes, and to specify the type of interrupt handler.

name (attribute)

This is a required attribute for `irq` definitions in a Secure Partition manifest that is using "psa_framework_version": 1.1.

This attribute replaces the use of the `signal` attribute.

The `name` attribute is used to construct the C identifiers for API elements used in managing the interrupt. The implementation defines a macro for the interrupt signal in `psa_manifest/«manifest-filename».h`, of the following form:

```
#define «name»_SIGNAL /* implementation-defined value */
```

For an interrupt that specifies the `handling` attribute as "FLIH", a [FLIH function](#) must be provided in the Secure Partition. This function has the following prototype:

```
psa_flih_result_t «name»_flih(void);
```

The C identifier used for an FLIH function is constructed by adding the suffix `_flih` to a lowercase version of the interrupt's `name` attribute.

The Secure Partition header file, `psa_manifest/«manifest-filename».h`, must include the prototype definition of each FLIH function.

handling (attribute)

This is a required attribute for `irq` definitions in a Secure Partition manifest that is using "psa_framework_version": 1.1.

This attribute specifies the interrupt handling mechanism that is used for the interrupt. It must have one of the following values:

"SLIH"	Select the SLIH mechanism. This is the v1.0 interrupt handling model.
"FLIH"	Select the FLIH mechanism.

Rationale

Making the selection of interrupt handling mechanism explicit in the manifest source file is beneficial for long term understanding and maintenance of the Secure Partition source.

The proposed definition is preferred over one that supports a default value which is compatible with version 1.0, or using a boolean attribute to specify the FLIH mechanism.

6.3.2 Secure Partition API changes for FLIH

The following API elements are added to `psa/service.h`:

- `psa_flih_result_t`
- `PSA_FLIH_NO_SIGNAL`
- `PSA_FLIH_SIGNAL`
- `PSA_FLIH_DISABLE`
- `PSA_FLIH_PANIC`
- `psa_reset_signal()`

`psa_flih_result_t` (typedef)

The type of the return value from an *FLIH function*.

```
typedef uint32_t psa_flih_result_t;
```

The return value is constructed from one or more of the `PSA_FLIH_XXX` values.

`PSA_FLIH_NO_SIGNAL` (macro)

Following an *FLIH function*, do not set the interrupt signal.

```
#define PSA_FLIH_NO_SIGNAL ((psa_flih_result_t) 0U)
```

This is the default behavior if the FLIH function does not specify `PSA_FLIH_NO_SIGNAL` or `PSA_FLIH_SIGNAL`.

`PSA_FLIH_SIGNAL` (macro)

Following an *FLIH function*, set the interrupt signal.

```
#define PSA_FLIH_SIGNAL ((psa_flih_result_t) 1U)
```

This return value can be combined with `PSA_FLIH_DISABLE`, to both set the interrupt signal, and disable the interrupt on return from the FLIH function.

PSA_FLIH_DISABLE (macro)

Following an [FLIH function](#), disable the interrupt.

```
#define PSA_FLIH_DISABLE ((psa_flih_result_t) 2U)
```

This return value can be combined with [PSA_FLIH_SIGNAL](#), to both set the interrupt signal, and disable the interrupt on return from the FLIH function.

Change for 1.1 Beta

Some framework implementations do not permit calls to the SPM from an FLIH function. Using the return value to disable the interrupt, instead of calling `psa_irq_disable()`, supports the second programming pattern: [Hand-off between FLIH and Secure Partition thread context on page 68](#).

PSA_FLIH_PANIC (macro)

Following an [FLIH function](#), panic the Secure Partition.

```
#define PSA_FLIH_PANIC ((psa_flih_result_t) ~0U)
```

When an unrecoverable fault is detected, an FLIH function must return with [PSA_FLIH_PANIC](#), instead of calling `psa_panic()`.

This return value must not be combined with any other FLIH function return code.

Change for 1.1 Beta

Some framework implementations do not permit calls to the SPM from an FLIH function.

psa_reset_signal (function)

Reset the signal for an interrupt that is using [FLIH](#) handling.

```
void psa_reset_signal(psa_signal_t irq_signal);
```

Parameters

<code>irq_signal</code>	The interrupt signal to be reset. This must have a single bit set, corresponding to a currently asserted signal for an interrupt that is defined to use FLIH handling.
-------------------------	---

Returns: void

Programmer Error

The call is a [PROGRAMMER ERROR](#) if one or more of the following are true:

- `irq_signal` is not a signal for an interrupt that is specified with FLIH handling in the Secure Partition manifest.
- `irq_signal` indicates more than one signal.
- `irq_signal` is not currently asserted.

A PROGRAMMER ERROR will panic the caller.

Description

For an interrupt that is using the FLIH mechanism, the *Secure Partition thread context* calls `psa_reset_signal()` to clear a signal that was set by an *FLIH Function* returning `PSA_FLIH_SIGNAL`.

To avoid the risk of missing a subsequent signal from the FLIH, the Secure Partition thread context should call `psa_reset_signal()` before processing the data provided by the FLIH. See *Programming patterns using FLIH on page 66* for examples of how this function can be used.

`psa_eoi` (function)

This existing function is constrained to be used only for interrupts that are using *SLIH* handling.

The `irq_signal` passed to `psa_eoi()` must be the signal for an interrupt that was specified with SLIH handling in the Secure Partition manifest.

Calling `psa_eoi()` with the signal of an interrupt with FLIH handling is a [PROGRAMMER ERROR](#).

6.3.3 Secure Partition API changes for interrupt control

The following API elements are added to `psa/service.h`:

- `psa_irq_enable()`
- `psa_irq_disable()`

There is no API to query the status of an interrupt.

Rationale

After consideration of real-time requirements, and the need to avoid hidden race conditions, the interface proposed in the Alpha specification has been substantially simplified.

- An interface that provides the current state of an interrupt is subject to race conditions with interrupting, or interleaving, code that modifies the state of the interrupt. Without masking all interrupts, or auditing all code that might interleave with a specific code sequence, it is not generally possible to resolve the race condition.

Instead, it is recommended that Secure Partition which depends on knowing the status of an interrupt source, should maintain a record of the state.

- The previous issue of this specification provided a test-and-set variant of the `psa_irq_disable()` function. This was provided to assist in scenarios where the Secure Partition code has nested functions that need to mask an interrupt, but not know the current interrupt state. However, this required that the interrupt state immediately prior to being disabled is returned. In practice, implementations had to disable all interrupts between testing and disabling the interrupt, as this is not typically an atomic operation. Instead, it is recommended that Secure Partition code is structured to ensure that the interrupt state is known at any point where it is essential that the interrupt is masked.

psa_irq_enable (function)

Enable an interrupt.

```
void psa_irq_enable(psa_signal_t irq_signal);
```

Parameters

<code>irq_signal</code>	The signal for the interrupt to be enabled. This must have a single bit set, which must be the signal value for an interrupt in the calling Secure Partition.
-------------------------	--

Returns: void

Programmer Error

The call is a [PROGRAMMER ERROR](#) if one or more of the following are true:

- `irq_signal` is not an interrupt signal.
- `irq_signal` indicates more than one signal.

A PROGRAMMER ERROR will panic the caller.

Description

Enabling an interrupt that is already enabled has no effect.

Note:

Each interrupt must be explicitly enabled in a Secure Partition that specifies "psa_framework_version": 1.1 in the manifest file. See [Secure Partition execution model on page 57](#).

psa_irq_disable (function)

Disable an interrupt.

```
void psa_irq_disable(psa_signal_t irq_signal);
```

Parameters

<code>irq_signal</code>	The signal for the interrupt to be disabled. This must have a single bit set, which must be the signal value for an interrupt in the calling Secure Partition.
-------------------------	---

Returns: void

Programmer Error

The call is a :PROGRAMMER ERROR if one or more of the following are true:

- `irq_signal` is not an interrupt signal.
- `irq_signal` indicates more than one signal.

A PROGRAMMER ERROR will panic the caller.

Description

Disabling an interrupt that is already disabled has no effect.

6.3.4 Register access functions for MMIO

The following API elements are added to `psa/service.h`:

- `psa_mmio_read8()`
- `psa_mmio_read16()`
- `psa_mmio_read32()`
- `psa_mmio_write8()`
- `psa_mmio_write16()`
- `psa_mmio_write32()`

These functions can all be used in *Secure Partition thread context*. It is IMPLEMENTATION DEFINED whether they are available in *FLIH context*.

Implementation note

On many systems, these functions can be implemented efficiently and effectively as an inline function or a function-like macro. Where possible, this is the recommended implementation approach.

psa_mmio_read8 (function)

Read an 8-bit memory-mapped peripheral register.

```
uint8_t psa_mmio_read8(uintptr_t addr);
```


Parameters

addr	The memory address of the MMIO register to read.
------	--

Returns: uint8_t

The value of the register at addr.

Programmer Error

It is a **PROGRAMMER ERROR** to call this API if `addr` refers to a memory location that is not accessible to the Secure Partition.

If the framework implementation detects a `PROGRAMMER ERROR`, this function will not return.

psa_mmio_read16 (function)

Read a 16-bit memory-mapped peripheral register.

```
uint16_t psa_mmio_read16(uintptr_t addr);
```

Parameters

addr	The memory address of the MMIO register to read.
------	--

Returns: uint16_t

The value of the register at addr.

Programmer Error

It is a **PROGRAMMER ERROR** to call this API if `addr` refers to a memory location that is not accessible to the Secure Partition.

If the framework implementation detects a `PROGRAMMER ERROR`, this function will not return.

psa_mmio_read32 (function)

Read a 32-bit memory-mapped peripheral register.

```
uint32_t psa_mmio_read32(uintptr_t addr);
```

Parameters

addr	The memory address of the MMIO register to read.
------	--

Returns: uint32_t

The value of the register at addr.

Programmer Error

It is a **PROGRAMMER ERROR** to call this API if `addr` refers to a memory location that is not accessible to the Secure Partition.

If the framework implementation detects a `PROGRAMMER ERROR`, this function will not return.

psa_mmio_write8 (function)

Write to an 8-bit memory-mapped peripheral register.

```
void psa_mmio_write8(uintptr_t addr,  
                     uint8_t value);
```

Parameters

addr	The memory address of the MMIO register to write.
value	The value to write to the register at addr.

Returns: void

Programmer Error

It is a [PROGRAMMER ERROR](#) to call this API if `addr` refers to a memory location that is not accessible to the Secure Partition.

If the framework implementation detects a `PROGRAMMER ERROR`, this function will not return.

psa_mmio_write16 (function)

Write to a 16-bit memory-mapped peripheral register.

```
void psa_mmio_write16(uintptr_t addr,  
                      uint16_t value);
```

Parameters

addr	The memory address of the MMIO register to write.
value	The value to write to the register at addr.

Returns: void

Programmer Error

It is a [PROGRAMMER ERROR](#) to call this API if `addr` refers to a memory location that is not accessible to the Secure Partition.

If the framework implementation detects a `PROGRAMMER ERROR`, this function will not return.

psa_mmio_write32 (function)

Write to a 32-bit memory-mapped peripheral register.

```
void psa_mmio_write32(uintptr_t addr,  
                      uint32_t value);
```

Parameters

addr	The memory address of the MMIO register to write.
value	The value to write to the register at addr.

Returns: void

Programmer Error

It is a [PROGRAMMER ERROR](#) to call this API if `addr` refers to a memory location that is not accessible to the Secure Partition.

If the framework implementation detects a `PROGRAMMER ERROR`, this function will not return.

6.4 Writing Secure Partition peripheral drivers

It is recommended that SLIH is used for handling interrupts, if there is no time bound for both responding to the interrupt, and for processing the related data.

This is for the following reasons:

1. FLIH adds concurrent execution to the Secure Partition. Concurrent execution introduces a category of programming risks that are otherwise absent, and this demands more effort from the developer to ensure correctness.
2. The FLIH function cannot use most of the Secure Partition API. To manage a request message following an FLIH function, the interrupt signal must be used to hand-over execution to the [Secure Partition thread context](#). This splitting of the interrupt response adds further complexity to the Secure Partition.

FLIH can be used when there is a requirement for interrupt response that cannot be met using SLIH. In this case, the developer has to mitigate the additional risks that arise from concurrent execution of the FLIH function and the Secure Partition thread context.

There are two distinct patterns that can be used when an FLIH function sets the interrupt signal to hand over to the Secure Partition thread context, depending on whether the FLIH function must be available to run again immediately or not.

6.4.1 Programming patterns using FLIH

FLIH supports two use case scenarios:

- [Continuous FLIH execution on page 67](#)
- [Hand-off between FLIH and Secure Partition thread context on page 68](#)

Continuous FLIH execution

The FLIH function needs to be able to run continuously, even after signalling the Secure Partition thread context to process some data. For example, a peripheral that is producing a continuous stream of output.

In this case, the interrupt must remain enabled when the FLIH function returns which allows the FLIH to run again immediately. In particular, the FLIH can interrupt the Secure Partition thread context which is responding to the signal. The interaction between the FLIH and Secure Partition thread contexts requires careful design, as these contexts can run concurrently and there must be no race conditions on shared data.

To demonstrate this pattern, here is an example of the FLIH function and associated Secure Partition thread code, for an interrupt which is defined with "name": "IRQ2". Shared data is handled carefully to avoid race conditions. In this simple example, only the FLIH writes to the shared `irq2_status` variable, which is sized and aligned to ensure atomic reads and writes.

```
static uint32_t irq2_status;

psa_flih_result_t irq2_flih(void)
{
    // Process the interrupt
    // and update the status value
    irq2_status = ...;
    return PSA_FLIH_SIGNAL;
}
```

The Secure Partition thread context responds to the signal and can report the current counter value:

```
...

psa_signal_t s = psa_wait(PSA_WAIT_ANY, PSA_BLOCK);
if (s & IRQ2_SIGNAL)
{
    // Reset the signal before reading the shared data to avoid
    // missing a status update
    psa_reset_signal(IRQ2_SIGNAL);
    // Read the FLIH result data
    handle_status_change(irq2_status);
}

...
```

In this pattern, the signal is reset before the shared data is processed. This prevents the following race condition:

1. The FLIH runs, updates the shared data, and sets the signal.
2. The Secure Partition thread context detects the signal, and reads the shared data.
3. The FLIH interrupts the thread context, updates the shared data, and sets the signal.
4. The thread context resumes, and resets the signal.

At this point, the signal is clear, but the last update of the shared data #3 by the FLIH function has not been read by the Secure Partition thread context because the signal was cleared at #4.

Hand-off between FLIH and Secure Partition thread context

A low latency interrupt response is not required once the Secure Partition thread context has been signalled. For example, at the end of a sequence of write operations to a peripheral.

In this case, the interrupt can be disabled, either at source or at the end of the FLIH function. This allows the Secure Partition thread context to process the result of the FLIH function without being interrupted by the FLIH, avoiding data races. The Secure Partition thread context must enable the interrupt again to allow the FLIH function to run again.

To demonstrate this pattern, here is an example of the FLIH function and associated Secure Partition thread code, for an interrupt which is defined with "name": "IRQ1". The FLIH function should end by disabling the interrupt and signalling the Secure Partition:

```
static uint32_t irq1_status;

psa_flih_result_t irq1_flih(void)
{
    // Process the interrupt
    // and write the result to Secure Partition memory
    irq1_status = ...;
    // request that the framework signal the partition and disable the interrupt
    return PSA_FLIH_SIGNAL|PSA_FLIH_DISABLE;
}
```

The Secure Partition thread context responds to the signal and re-enables the interrupt if required:

```
...

psa_signal_t s = psa_wait(PSA_WAIT_ANY, PSA_BLOCK);
if (s & IRQ1_SIGNAL)
{
    // Read the FLIH result data.
    // There is no race condition as the interrupt is disabled
    handle_status_change(irq1_status);
    // Reset signal before enabling the interrupt
    psa_reset_signal(IRQ1_SIGNAL);
    psa_irq_enable(IRQ1_SIGNAL);
}

...
```

In this pattern, the signal must be reset before the interrupt is re-enabled, but there is no race condition on access to the shared data because the FLIH function cannot run again while the interrupt is disabled.

7 Deprecated features

From version 1.1, the Secure Partition doorbell feature is deprecated.

- Implementations of the framework must still support this functionality as described in the version 1.0 and version 1.1 Extension specifications.
- Secure Partition developers are recommended to use other mechanisms to replace the use of this feature.

This feature might be removed in a future version of this specification.

7.1 Background and rationale

The doorbell feature was provided in version 1.0 to help in the development of complex peripheral drivers. An example scenario is described in *Arm® Platform Security Architecture Firmware Framework* [FF-M] §3.4.

The main use cases that inspired this feature are better supported with the use of *FLIH functions* in an implementation of version 1.1. As a result the need for the doorbell feature is significantly reduced.

Also, the doorbell feature is hard to secure, and provides no additional data from the signalling Secure Partition.

7.2 Changes to the Programming API

From version 1.1, the following APIs are deprecated:

- `PSA_DOORBELL`
- `psa_notify()`
- `psa_clear()`

An implementation is permitted to report a diagnostic if it detects the use of one of the APIs in Secure Partition code. It is recommended that any such diagnostic can be enabled or disabled by a developer using the framework implementation.

8 Miscellaneous changes

The following additional relaxations and clarifications are made for version 1.1:

- [RoT Service terminology and requirements](#)
- [Use of the manifest type attribute in isolation level 1 on page 73](#)
- [Availability of the PSA Lifecycle API in NSPE on page 74](#)
- [Relaxation of memory access rules for Constant data on page 74](#)
- [Reworking the optional isolation rules on page 75](#)
- [Replace the term 'reverse handle' with 'rhandle' on page 79](#)
- [Symbolic definition of Secure Partition resources on page 79](#)
- [Implementation defined maximum call type value on page 80](#)
- [Secure Partition RoT Service status codes on page 81](#)

8.1 RoT Service terminology and requirements

In Arm® Platform Security Architecture Firmware Framework [FF-M] version 1.0, the term [Root of Trust Service](#) (RoT Service) is not consistently used to mean precisely the same thing. This leads to some confusion within the specification, in particular, there are conflicting statements about the requirements for the deployment and isolation of [PSA RoT Services](#).

The following changes are made for version 1.1:

- Distinct terminology introduced and used for the different meanings of 'RoT Service'.
- Resolve the inconsistent description of PSA RoT Services.
- Provide clear rules for the deployment and isolation of PSA RoT Services and the use of Secure Partitions within the PSA Root of Trust.

8.1.1 The meaning of 'Root of Trust Service'

When 'Root of Trust Service' is used in version 1.0 text it could mean one of the following:

1. The general concept of a security service within a Root of Trust. This is the concept described in [Root of Trust Definitions and Requirements \[GP-ROT\]](#) as a *Root of Trust Security Service*.
2. A security service within the [PSA Root of Trust](#) or [Application Root of Trust](#), as defined in FF-M.
3. A [PSA RoT Service](#) or [Application RoT Service](#), which is implemented within a Secure Partition using the APIs defined in FF-M.

Changes to the specification

- Definition 1 is infrequently used. In version 1.1, any use of 'RoT Service' with this intended meaning will be referenced to its definition in [\[GP-ROT\]](#).
- Definition 3 is a subset of definition 2. For example, PSA RoT Services that are implemented directly within the [Secure Partition Manager](#) (SPM) would match definition 2, but not definition 3.

In version 1.1, the term 'RoT Service' is used for definition 2, and the term [Secure Partition RoT Service](#) is introduced for definition 3. The term 'RoT Service' can be used for definition 3 if the context is clearly related to a Secure Partition RoT Service.

8.1.2 PSA RoT Services and Secure Partitions

The purpose of FF-M is to provide a common programming and runtime model for writing isolated security services, which can be deployed into different implementations of the framework, on different types of underlying system architecture. This programming model is provided in FF-M in the definition of Secure Partitions and the APIs for communication between a client and a Secure Partition RoT Service.

The Secure Partition programming model is essential for RoT Services that are not provided by the implementation of the framework. When an RoT Service is provided by the framework implementation, there is more flexibility in how it is deployed. For example:

- A PSA RoT Service, such as *PSA Cryptography API* [\[PSA-CRYPT\]](#), is defined as a C programming API. Although the service implementation must be protected within the PSA Root of Trust, the service is not required to be implemented within a Secure Partition using the FF-M framework APIs. This is illustrated in Figure 4 of [\[FF-M\]](#) §3, where the PSA RoT Services are shown within the PSA Root of Trust, with an [IMPLEMENTATION DEFINED](#) interface to the SPM.
- A PSA RoT Service does not need to be isolated from the SPM, which is required for Application RoT Services at isolation level 2 or 3. Application RoT Services must be deployed in a Secure Partition. Isolation level 3, defined in [\[FF-M\]](#) §3.1.3, places the PSA RoT Services within the same PSA Root of Trust protection domain as the SPM. Secure Partitions within the Application Root of Trust must be protected from each other and isolated from the SPM.

A framework is still permitted to use the Secure Partition model and interfaces for PSA RoT Services, where this is advantageous for the implementation. In this case, the implementation retains the following flexibility for these PSA RoT Secure Partitions:

- The framework is permitted to restrict the communication models to a subset of those available for Application RoT Secure Partitions.
- The framework is permitted to restrict the RoT Service types to a subset of those available for Application RoT Secure Partitions.
- The framework is not required to isolate a PSA RoT Secure Partition from the SPM by using a Secure Partition protection domain.

Changes to the specification

Clarifying the scope of the isolation architecture

In §2.1, the following paragraph:

Each Secure Partition is a single thread of execution and is the smallest unit of isolation. If the strongest isolation level is implemented, every Secure Partition is isolated from every other Secure Partition.

This is modified to the following:

Each Secure Partition is a single thread of execution and is the smallest unit of isolation defined by this specification. If the strongest isolation level is implemented, every Secure Partition is isolated from every other Secure Partition.

Additional text will be added to §2.3 to describe the flexibility for frameworks to implement isolation within the protection domains defined by FF-M. This matches with the recommendation to use isolation within the NSPE to provide increased robustness and resilience.

Clarifying the RoT Service definition

In §2.4, the following text describes the deployment of RoT Services:

PSA RoT Services that permit access from the NSPE, and all Application RoT Services, must be implemented in a Secure Partition. These services must be accessed using the PSA Secure IPC framework that is defined in this specification. This provides a consistent and portable mechanism for implementing and accessing the service from both Secure Partitions and from the NSPE.

PSA RoT Services that are only available to the SPE can either be implemented using the IPC framework as already described, or in an implementation defined manner within the SPM and PSA Root of Trust.

This is replaced with the following text:

Application RoT Services must be implemented in a Secure Partition within the Application Root of Trust.

The framework is permitted to implement PSA RoT Services either within a Secure Partition, or in an **IMPLEMENTATION DEFINED** way. At Isolation level 2 or 3, the PSA RoT Services must be implemented within the PSA Root of Trust protection domain.

An RoT Service implemented within a Secure Partition must be accessed using the FF-M communication API that is defined in this specification. This provides a consistent and portable mechanism for implementing and accessing the service from both Secure Partitions and from the NSPE.

PSA RoT Services that are only available to the SPE can either be implemented using the IPC framework as already described, or in an **IMPLEMENTATION DEFINED** manner within the SPM and PSA Root of Trust.

Clarifying the protection domains definition

In §3.1.3, in the definition of *Isolation level 3*, references to 'Secure Partition' are replaced by 'Application RoT Secure Partition'.

This clarifies that the Secure Partition protection domains at isolation level 3 do not include Secure Partitions that the framework can optionally use within the PSA Root of Trust.

Relaxation of the Secure Partition type attribute

The type attribute in the Secure Partition manifest is relaxed. Support for the "PSA-ROT" value is optional in version 1.1.

If the implementation does not support Secure Partitions within the PSA Root of Trust, then the following rules apply:

- PSA RoT Services are integrated into the PSA Root of Trust in an IMPLEMENTATION DEFINED manner.
- The implementation reports an error if a Secure Partition manifest specifies type: "PSA-ROT". The type attribute is still required in manifests for a Secure Partitions in the Application Root of Trust.

If the implementation permits Secure Partitions within the PSA Root of Trust, then the following rules apply:

- PSA RoT Services can be integrated into the PSA Root of Trust using a combination of Secure Partitions and IMPLEMENTATION DEFINED mechanisms.
- PSA RoT Services that are deployed in a Secure Partition must use the FF-M communication framework.
- For Secure Partitions in the PSA Root of Trust, the implementation is permitted to support a different set of communication models than for Secure Partitions in the Application Root of Trust. For example, the framework might require the use of the SFN model for a PSA RoT Secure Partition, but support both models for an Application RoT Secure Partition.
- For Secure Partitions in the PSA Root of Trust, the implementation is permitted to support a different set of RoT Service types than for Secure Partitions in the Application Root of Trust. For example, the framework might require all PSA RoT Services to be stateless, but support stateless and connection-based Application RoT Services.

8.2 Use of the manifest type attribute in isolation level 1

A Secure Partition manifest file must declare the type of the Secure Partition, which identifies whether the Secure Partition is part of the [PSA Root of Trust](#) or the [Application Root of Trust](#).

When isolation level 1 is implemented, there are only two protection domains: the NSPE and the SPE. There is no practical distinction between Secure Partitions in the PSA Root of Trust and those in the Application Root of Trust. The version 1.0 specification did not describe the effect of the type attribute for such implementations.

8.2.1 Changes to the specification

Clarification of the Secure Partition type attribute

In §4.1.1, the definition of the type attribute will be extended to describe the behavior for implementations that provide level 1 isolation. In these systems, the type attribute must have a supported value, but does not affect the placement of the Secure Partition within the architecture. In other words, all Secure Partitions are located within the SPE protection domain.

See also [PSA RoT Services and Secure Partitions on page 71](#) for related changes to this attribute.

8.3 Availability of the PSA Lifecycle API in NSPE

In [FF-M] version 1.0, Table 15 in §4 *Programming API* specifies the availability of the APIs defined in that specification and by the PSA RoT Service API specifications.

In version 1.0, the RoT Lifecycle API is only available to callers in the SPE.

For version 1.1, this API is also available to the NSPE.

This information provided by this API is not difficult to provide to the NSPE, and disclosing the information does not pose a security risk.

8.4 Relaxation of memory access rules for Constant data

The memory access rules in [FF-M] version 1.0, §3.1.2 prohibit an implementation from making *Constant data* executable. From rule I1 in Table 3:

Table 8 Excerpt from *Permitted access methods for memory assets* in version 1.0

ID Access rule	Rationale
I1 Only Code is executable	Preventing execution of writable data mitigates the primary buffer-overflow attack vector. Preventing execution of read-only data reduces the attack surface available for Return-Oriented Programming (ROP) and Jump-Oriented Programming (JOP) attacks.

This is challenging for systems that implement higher levels of isolation, and increased isolation between protection domains, such as the optional rule I4 or I6 from §3.1.5. These implementations require significantly more resources in the memory protection hardware for an increased number of different asset memory regions that require different access permissions.

However, isolation rules like I4 and I6 significantly reduce or eliminate the visibility of *Constant data* assets across protection domains. This provides a significant mitigation against using the data for ROP or JOP attacks, by reducing the availability of suitable executable gadgets for the attack. As a consequence, the prohibition on allowing execute access to *Constant data* adds only a small additional mitigation for systems that implement stronger isolation rules.

For version 1.1:

- Isolation rule **I1** is relaxed to only cover execution of *Private Data*.
- A new, optional isolation rule **I7** is introduced to cover execution of *Constant data*.

8.4.1 Changes to the specification

Rule **I1** in Table 3 is changed to the following:

Table 9 Update for *Permitted access methods for memory assets* in version 1.1

ID Access rule	Rationale
I1 <i>Private data</i> is not executable	Preventing execution of writable data mitigates the primary buffer-overflow attack vector.

The summary table that follows Table 3 is also updated:

Table 10 Updated *Summary of asset access rules* in version 1.1

Access method	Asset class		
	Code	Constant data	Private data
Read	Yes	Yes	Yes
Write	No	No	Yes
Execute	Yes	IMPLEMENTATION DEFINED ^a	No

^a We recommend that *Constant data* is not executable. See example rule **I7** in *Additional isolation policies*.

Policy **I7**, *Constant data* is not executable, is added to the optional isolation policies **I4**, **I5**, and **I6**, that are described in the *Additional isolation policies* appendix. See [Reworking the optional isolation rules](#) for details.

8.5 Reworking the optional isolation rules

Section §3.1.5 *Optional isolation rules* does not meet the original intentions for this section.

- These are meant to be **examples** of additional policies that an implementation **might** enforce.
- It is not clear that implementations can implement other isolation policies that achieve equivalent security benefits.
- Rule **I4** and **I5**, as currently written, imply that the implementation cannot use shared code libraries across multiple protection domains. This constraint is undesirable for small systems.

None of these rules are mandatory, under any specified isolation level. This reflects the diversity of hardware systems which can support a compliant implementation, and the variation in required security level for end products.

The 'rules' defined in this section are better presented as examples of additional isolation policies that can be enforced by an implementation. This is better suited to being an appendix, instead of a section within the normative text of the specification. In that form, it is easier to present the optional policies as responses to various attack techniques, which encourages each implementation to identify appropriate policies.

8.5.1 Changes to the specification

Section §3.1.5 is changed to the following:

3.1.5 Additional isolation policies

Additional measures can be implemented by the SPM to improve isolation-based protection. These measures are not required to comply with [\[PSA-SM\]](#), but these measures enhance the protection against common software attack techniques and implementation errors. The decision to implement additional policies is a trade-off that considers the isolation capability of the device, the security requirements for the product, and the footprint and runtime costs of the chosen policies.

Each implementation must document the properties of the isolation boundaries that it enforces. This allows developers to determine whether the implementation meets the security requirements of their product.

[Example additional isolation policies](#) provides some examples of additional isolation policies that can be implemented, and the attacks which the policies mitigate.

A new appendix is added to the specification:

Appendix: Example additional isolation policies

§3.1 *Isolation architecture* describes the mandatory isolation rules that must be enforced by the SPM, in order to provide a secure execution environment for the SPM and RoT Services. An implementation is permitted to implement additional isolation policies that provide some protection against common software attack techniques and software implementation errors.

The additional policies that are implemented will depend on a number of factors:

- The capability of the isolation hardware in the system.
- The sensitivity of the end product to runtime and footprint overheads resulting from the isolation policy.
- The additional complexity in the SPM and runtime support required to implement the policy.
- The required security for the product in relation to the attacks that are mitigated by the policy.

The following examples of attacks and mitigating isolation policies are provided:

- Preventing disclosure of secrets and sensitive code across a protection boundary. See [policy I4](#) and [I6](#).
- Reducing available [Return-Oriented Programming](#) (ROP) and [Jump-Oriented Programming](#) (JOP) gadgets by restricting execution to Code assets, and by limiting code accessibility. See [policy I4](#), [I5](#), [I6](#), and [I7](#).
- Reduce exploitability of implementation errors by preventing access to any other domain's code or data. See [policy I6](#).

This list is not exhaustive: there are other attacks that might require mitigation, and other policies that can provide broadly similar protection against the listed attacks.

Policy I4

Policy	If domain A needs protection from domain B, then <i>Code</i> and <i>Constant data</i> in domain A, which is not part of a shared library, is not executable or readable by domain B.
Threats	Disclosure of secrets and sensitive code. Execution of sensitive code sequences outside of the intended domain.
Attacks	Direct read of the <i>Code</i> and <i>Constant data</i> assets of a protected domain. Execution of the <i>Code</i> assets of a protected domain, either directly, or via a ROP or JOP technique.
Mitigations	Strengthens isolation rule I3 to protect all assets from untrusted domains. Reduces available ROP and JOP gadgets in untrusted domains.
Comments	Shared code libraries are described explicitly in this policy, because this code is accessible to multiple protection domains.

Policy I5

Policy	<i>Code</i> in a domain, which is not part of a shared library, is not executable by any other domain.
Threats	Execution of sensitive code sequences outside of the intended domain. Execution of code in sequences not intended by the designer,
Attacks	Execution of the <i>Code</i> assets of a protected domain in an untrusted domain. Using ROP and JOP techniques to implement attacker-controlled behavior within a protection domain.
Mitigations	Eliminating accessible <i>Code</i> reduces the gadgets available for ROP and JOP attacks.
Comments	This reduces the code accessible to a domain to exactly the code that is required for the domain to execute. This is the 'principle of least privilege' for what the domain can execute. Large shared libraries reduce the effectiveness of this isolation policy, as they can provide a good source of gadgets for exploitation. Typically, each domain only uses a small portion of the functionality in a large shared library.

Policy I6

Policy	<p>All assets in a domain are private to that domain and cannot be accessed by any other domain, with the following exception:</p> <p>The domain containing the SPM can only access <i>Private data</i> and <i>Constant data</i> assets of other domains when required to implement the PSA Firmware Framework API.</p>
Threats	<p>Disclosure of secrets and sensitive code.</p> <p>Tampering with protected domain data.</p> <p>Execution of sensitive code sequences outside of the intended domain.</p> <p>Execution of code in sequences not intended by the designer,</p>
Attacks	<p>Direct access to memory assets of a protected domain.</p> <p>Direct modification of data belonging to another domain.</p> <p>Indirect access to protected assets via a vulnerability in another domain: a Confused deputy attack.</p> <p>Execution of the <i>Code</i> assets of a protected domain in an untrusted domain.</p> <p>Using ROP and JOP techniques to implement attacker-controlled behavior within a protection domain.</p>
Mitigations	<p>Strengthens isolation rule I3 to protect all assets from all other domains.</p> <p>Eliminating access to other domain data reduces the assets that can be disclosed or tampered with via a Confused deputy attack.</p> <p>Eliminating accessible <i>Code</i> reduces the gadgets available for ROP and JOP attacks.</p>
Comments	<p>This policy effectively upgrades mandatory rule I3, and additional policies I4 and I5.</p>

SPM implementation note:

In implementations that enforce rule **I6**, the technique used to provide special access by the SPM depends on the platform capabilities. For example, this can be achieved by one of the following techniques:

- Allowing direct access by the SPM to all other domains' memory assets.
- Denying direct access by the SPM to all other domains' memory assets by default, and only permitting access when copying data to or from API parameters that are in the caller's domain.
- Denying direct access by the SPM to all other domains' memory assets, and marshalling all API parameters to and from the SPM through a dedicated region of memory shared between the calling domain and the SPM.

Policy I7

Policy	<i>Constant data</i> is not executable
Threats	Execution of code in sequences not intended by the designer,
Attacks	Execution of <i>Constant data</i> assets that correspond to legitimate code sequences. Using <i>ROP</i> and <i>JOP</i> techniques to implement attacker-controlled behavior within a protection domain.
Mitigations	Preventing execution of read-only data reduces the gadgets available for <i>ROP</i> and <i>JOP</i> attacks.
Comments	<p>This policy is recommended for an implementation that has a lower level of isolation. In particular, for systems where the <i>Code</i> and <i>Constant data</i> are visible to all protection domains.</p> <p>This policy provides less benefit in an implementation that already restricts the visibility of <i>Constant data</i> between protection domains. For example, policies I4 or I6 already provide most of the mitigation that is provided by I7.</p>

8.6 Replace the term ‘reverse handle’ with ‘rhandle’

In version 1.1, the new terms *stateless handle* and *connection handle* specify handles for different types of RoT Service. The existing term *null handle* also defines a specific value of the same `psa_handle_t` type.

However, the version 1.0 term ‘reverse handle’ is not an RoT Service handle or a message handle. This similarity in naming is misleading, and the *rhandle* feature does not require the value to be treated like a handle.

In this document, and in future FF-M specifications, the ‘reverse handle’ functionality will be referred to as the *rhandle* feature.

8.7 Symbolic definition of Secure Partition resources

8.7.1 `stack_size` (attribute)

This existing attribute has an extended definition for version 1.1.

The value of `stack_size` must indicate the stack memory usage of the Secure Partition.

If the value of the `stack_size` attribute is a decimal or hexadecimal value, this is used as the stack requirement in bytes.

In version 1.0, no other type of value is permitted.

In version 1.1, a non-numerical value is resolved in in *IMPLEMENTATION DEFINED* manner. This permits an implementation to support the use of symbolic constants that reference an external definition.

Implementation note:

The implementation of the framework is not required to provide dedicated stack in situations where the isolation rules do not require this.

For example, in a system providing isolation level 1, the framework can use a single execution stack for all Secure Partitions that are using the [SFN model](#). This stack would have to be large enough for the deepest chain of calls between these Secure Partitions.

8.7.2 heap_size (attribute)

This existing attribute has an extended definition for version 1.1.

The value of heap_size must indicate the heap memory usage of the Secure Partition.

If the value of the heap_size attribute is a decimal or hexadecimal value, this is used as the heap requirement in bytes.

In version 1.0, no other type of value is permitted.

In version 1.1, a non-numerical value is resolved in [IMPLEMENTATION DEFINED](#) manner. This permits an implementation to support the use of symbolic constants that reference an external definition.

Implementation note:

The implementation of the framework is not required to provide dedicated heap resources in situations where the isolation rules do not require this.

8.8 Implementation defined maximum call type value

In version 1.0, the message type provide by the client to `psa_call()` and received by the service in `psa_msg_t::type` is constrained to be a zero-or-positive `int32_t` value.

Implementations would like to be able to limit the message type to a smaller range, in order to pack this value with other parameter data during call processing. For example, limiting the value to a 16-bit value.

The assumption is that Secure Partition RoT Service developers will not have made use of the large range of message type values available in version 1.0, and so this change will not disrupt existing Secure Partition code.

In version 1.1, new constant values [PSA_CALL_TYPE_MIN](#) and [PSA_CALL_TYPE_MAX](#) are introduced.

[PSA_CALL_TYPE_MIN](#) is defined to be zero, and [PSA_CALL_TYPE_MAX](#) is an [IMPLEMENTATION DEFINED](#) value between $2^{15} - 1$ and $2^{31} - 1$ inclusive (`0x7FFF - 0x7FFFFFFF`).

Rationale

The IPC part of the identifier is dropped as these values are valid for IPC and SFN model Secure Partitions. We expect this is less confusing for maintenance of new code, than to retain alignment with the existing `PSA_IPC_XXX` constant values.

The existing value, `PSA_IPC_CALL`, is at the lower end of the valid range of call type values.

8.8.1 Changes to the specification

In the *Client API*, §4.4.1, the following definitions are added:

PSA_CALL_TYPE_MIN (macro)

This is the minimum value that can be passed as the type parameter in a call to `psa_call()`.

```
#define PSA_CALL_TYPE_MIN (0)
```

A type value less than `PSA_CALL_TYPE_MIN` is a **PROGRAMMER ERROR**.

PSA_CALL_TYPE_MAX (macro)

This is the maximum value that can be passed as the type parameter in a call to `psa_call()`.

```
#define PSA_CALL_TYPE_MAX /* implementation-defined value */
```

A type value greater than `PSA_CALL_TYPE_MAX` is a **PROGRAMMER ERROR**.

Implementation note:

The value must satisfy the constraint: $0x7FFF \leq \text{PSA_CALL_TYPE_MAX} \leq 0xFFFFFFFF$.

In the definition of `psa_call()` in §4.4.3:

- The type parameter description is changed to state:
The request type. Must be in the range `PSA_CALL_TYPE_MIN` to `PSA_CALL_TYPE_MAX`.
- The condition in the **Programmer error** section is changed to:
`type < PSA_CALL_TYPE_MIN || type > PSA_CALL_TYPE_MAX`

Elsewhere in the document, descriptions of the range of message type values are changed to refer to `PSA_CALL_TYPE_MIN` and `PSA_CALL_TYPE_MAX`.

8.9 Secure Partition RoT Service status codes

§3.5.5 *Standard error codes* describes the value ranges for status codes, as well as describing a set of standard error codes for use by the standard PSA API specifications. Details of the common error codes are provided in the API definition in §4.3 *Status codes*.

The latter section does not repeat the information about ranges that are reserved for SPM implementation use, reserved for future PSA API specifications, and available for RoT Service developers to use. This has resulted in developers not seeing the reserved range requirements, and then allocating RoT Service error codes that collide with those allocated by PSA RoT Service APIs, such as the PSA Cryptography API.

8.9.1 Changes to the specification

In §4.3 *Status codes*, describe the reservation of value ranges for status codes and provide the following summary table:

Table 15 Status code ranges

Status code range	Values	Description
<i>Success</i>	≥ 1	RoT Service specific status codes.
<i>PSA_SUCCESS</i>	0	General success status code.
<i>RoT Service error</i>	-1 to -128	RoT Service specific error codes.
<i>Standard error codes</i>	-129 to -248	Reserved for PSA RoT Services.
<i>SPM Implementation error</i>	-249 to -256	Reserved for the SPM implementation.
<i>RoT Service error</i>	≤ -257	RoT Service specific error codes.

The following API definitions are added to §4.3.1, to help RoT Service developers allocate error codes for their RoT Service:

PSA_ERROR_ROT_SERVICE_BASE (macro)

Starting value for the first range of RoT Service error codes.

```
#define PSA_ERROR_ROT_SERVICE_BASE ((psa_status_t) -1)
```

This is the first value available for a RoT Service developer to use as an error code. The end of this range is [PSA_ERROR_ROT_SERVICE_LIMIT](#).

An error code, *error*, allocated by a RoT Service in the first range, must satisfy the condition:

```
error <= PSA_ERROR_ROT_SERVICE_BASE && error >= PSA_ERROR_ROT_SERVICE_LIMIT
```

PSA_ERROR_ROT_SERVICE_LIMIT (macro)

End value for the first range of RoT Service error codes.

```
#define PSA_ERROR_ROT_SERVICE_LIMIT ((psa_status_t) -128)
```

This is the last value available for a RoT Service developer to use as an error code. The start of this range is [PSA_ERROR_ROT_SERVICE_BASE](#).

An error code, *error*, allocated by a RoT Service in the first range, must satisfy the condition:

```
error <= PSA_ERROR_ROT_SERVICE_BASE && error >= PSA_ERROR_ROT_SERVICE_LIMIT
```

PSA_ERROR_ROT_SERVICE_BASE_2 (macro)

Starting value for the second range of RoT Service error codes.

```
#define PSA_ERROR_ROT_SERVICE_BASE_2 ((psa_status_t) -257)
```

This is the first value from the second block of error codes that are available for a RoT Service developer to use. The second range has no limit.

An error code, `error`, allocated by a RoT Service in the second range, must satisfy the condition:

`error <= PSA_ERROR_ROT_SERVICE_BASE_2`

Appendix A: Reference header files

The API elements in this specification, once finalized, will be defined the `psa/error.h`, `psa/client.h`, `psa/service.h`, and `psa/framework_feature.h` header files.

The following listings are examples of the header file definitions for the API elements described in this document. This can be used as a starting point or reference for an implementation.

Note:

Not all of the API elements are fully defined. An implementation must provide the full definition.

The header will not compile without these missing definitions, and might require reordering to satisfy C compilation rules.

`psa/framework_feature.h`

```
/* psa/framework_feature.h
Firmware framework API for discovering feature implementation
As defined in PSA Firmware Framework v1.1
*/

#ifndef PSA_FRAMEWORK_FEATURE_H
#define PSA_FRAMEWORK_FEATURE_H

#define PSA_FRAMEWORK_ISOLATION_LEVEL /* implementation-defined value */
#define PSA_FRAMEWORK_HAS_MM_IOVEC /* implementation-defined status */

#endif // PSA_FRAMEWORK_FEATURE_H
```

`psa/error.h`

```
/* Updates to the psa/error.h header file
*/

#define PSA_ERROR_ROT_SERVICE_BASE ((psa_status_t) -1)
#define PSA_ERROR_ROT_SERVICE_LIMIT ((psa_status_t) -128)
#define PSA_ERROR_ROT_SERVICE_BASE_2 ((psa_status_t) -257)
```

psa/client.h

```
/* Updates to the psa/client.h header file
 */

#define PSA_CALL_TYPE_MIN (0)
#define PSA_CALL_TYPE_MAX /* implementation-defined value */
#define PSA_FRAMEWORK_VERSION (0x0101u)
uint32_t psa_framework_version(void);
```

psa/service.h

```
/* Updates to the psa/service.h header file
 */

typedef uint32_t psa_flih_result_t;
#define PSA_FLIH_NO_SIGNAL ((psa_flih_result_t) 0U)
#define PSA_FLIH_SIGNAL ((psa_flih_result_t) 1U)
#define PSA_FLIH_DISABLE ((psa_flih_result_t) 2U)
#define PSA_FLIH_PANIC ((psa_flih_result_t) ~0U)
void psa_reset_signal(psa_signal_t irq_signal);
void psa_irq_enable(psa_signal_t irq_signal);
void psa_irq_disable(psa_signal_t irq_signal);
uint8_t psa_mmio_read8(uintptr_t addr);
uint16_t psa_mmio_read16(uintptr_t addr);
uint32_t psa_mmio_read32(uintptr_t addr);
void psa_mmio_write8(uintptr_t addr,
                     uint8_t value);
void psa_mmio_write16(uintptr_t addr,
                      uint16_t value);
void psa_mmio_write32(uintptr_t addr,
                      uint32_t value);
const void * psa_map_invec(psa_handle_t msg_handle,
                           uint32_t invec_idx);
void psa_unmap_invec(psa_handle_t msg_handle,
                     uint32_t invec_idx);
void * psa_map_outvec(psa_handle_t msg_handle,
                      uint32_t outvec_idx);
void psa_unmap_outvec(psa_handle_t msg_handle,
                      uint32_t outvec_idx,
                      size_t len);
```

Appendix B: Summary of manifest attributes

This appendix is a summary of objects and attributes used in the Secure Partition manifest file.

The Secure Partition manifest file is a JSON file consisting of a single [Secure Partition](#) object.

B.1 Secure Partition object

B.1.1 Required attributes

Table 16 Required Secure Partition attributes

Name	Type	Description
psa_framework_version	enum: 1.0 or 1.1	Version of the Firmware Framework for M specification this manifest conforms to.
name	string: c_macro	Alphanumeric C macro for referring to a partition.
type	enum: "APPLICATION-ROT" or "PSA-ROT"	Whether the partition is unprivileged or part of the trusted computing base. v1.1: support for "PSA-ROT" is optional.
priority	enum: "LOW", "NORMAL", or "HIGH"	Partition task priority.
model	enum: "IPC" or "SFN"	The communication model that this Secure Partition uses. <i>New in v1.1.</i>
entry_point	string: c_symbol	C symbol name of an IPC model partition's entry point. v1.0: required v1.1: required for IPC model.
stack_size	integer or string	Secure Partition thread context's stack size. v1.0: integer or hex_string v1.1: integer, hex_string , or symbolic string

B.1.2 Optional attributes

Table 17 Optional Secure Partition attributes

Name	Type	Description	Default
description	string	Human readable description.	<i>null</i>
entry_init	string: c_symbol	C symbol name of an SFN model partition's optional initialization function. <i>New in v1.1:</i> optional for SFN model	<i>null</i>
heap_size	integer or string	Secure Partition's heap size. v1.0: integer or hex_string v1.1: integer, hex_string , or symbolic string	0
mmio_regions	array of Named Region and Numbered Region objects	List of Memory-Mapped IO region objects which the partition has access to.	<i>null</i>
services	array of Service objects	List of RoT Service objects which the partition implements.	<i>null</i>
irqs	array of IRQ objects	List of IRQ objects which the partition implements.	<i>null</i>
dependencies	array of string	List of RoT Service names which the partition code depends on and is permitted to access.	<i>null</i>

B.1.3 Example

This is an example Secure Partition object for a SFN model Secure Partition:

```
{
  "description": "My Secure Partition",
  "psa_framework_version": 1.1,
  "name": "MY_SP",
  "type": "APPLICATION-ROT",
  "priority": "NORMAL",
  "model": "SFN",
  "entry_init": "my_sp_init",
  "stack_size": "0x800",
  "services": [
    {
      "description": "My example stateless RoT Service",
      "name": "MY_ROT_SERVICE",
      "sid": "0x00022001",
```

(continues on next page)

(continued from previous page)

```
        "version": 2,
        "version_policy": "RELAXED",
        "non_secure_clients": true,
        "connection_based": false,
        "stateless_handle": "auto",
        "mm_iovec": "disable"
    },
    {
        "description": "My second RoT Service",
        "name": "MY_OTHER_ROT_SERVICE",
        "sid": "0x00022002",
        "version": 1,
        "version_policy": "RELAXED",
        "non_secure_clients": true,
        "connection_based": true,
        "mm_iovec": "enable"
    }
],
"dependencies": [
    "PSA_CRYPTO",
    "PSA_TRUSTED_STORAGE"
]
}
```

B.2 Service object

B.2.1 Required attributes

Table 18 Required Service attributes

Name	Type	Description
name	string: c_macro	RoT Service name. This is used as a prefix for the RoT Service SID, signal, version and SFN symbols.
sid	integer or hex_string	The integer value of the RoT Service ID
non_secure_clients	boolean	Indicate whether the RoT Service is exposed to non-secure clients.
connection_based	boolean	Specify the type of service, use true to indicate a connection-based RoT Service , or false to indicate a stateless RoT Service . New in v1.1

B.2.2 Optional attributes

Table 19 Optional Service attributes

Name	Type	Description	Default
description	string	Human readable description.	<i>null</i>
version	integer	Version number of the RoT Service interface.	1
version_policy	enum: "STRICT" or "RELAXED"	Version policy to apply on connections to the RoT Service.	"STRICT"
stateless_handle	integer, hex_string or "auto"	The index for the stateless handle in a stateless RoT Service . The framework will allocate the index if "auto" is specified. <i>New in v1.1: optional for a stateless RoT Service</i>	"auto"
mm_iovec	enum: "enable" or "disable"	Enable the MM-IOVEC functionality for this RoT Service. <i>New in v1.1</i>	"disable"

B.2.3 Example

```
{  
  "description": "My example stateless RoT Service",  
  "name": "MY_ROT_SERVICE",  
  "sid": "0x00022001",  
  "version": 2,  
  "version_policy": "RELAXED",  
  "non_secure_clients": true,  
  "connection_based": false,  
  "stateless_handle": "auto",  
  "mm_iovec": "disable"  
}
```

B.3 Named Region object

B.3.1 Required attributes

Table 20 Required Named Region attributes

Name	Type	Description
name	string: c_macro	Alphanumeric C macro for referring to the region.
permission	enum: "READ-ONLY" or "READ-WRITE"	Access permissions for the region.

B.3.2 Example

```
{  
  "name": "CRYPTOCELL_312",  
  "permission": "READ-WRITE"  
}
```

B.4 Numbered Region object

B.4.1 Required attributes

Table 21 Required Numbered Region attributes

Name	Type	Description
base	string: hex_string	The base address of the region.
size	string: hex_string	Size in bytes of the region.
permission	enum: "READ-ONLY" or "READ-WRITE"	Access permissions for the region.

B.4.2 Example

```
{  
  "base": "0x20004000",  
  "size": "0x1000",  
  "permission": "READ-WRITE"  
}
```

B.5 IRQ object

B.5.1 Required attributes

Table 22 Required IRQ attributes

Name	Type	Description
source	string	Interrupt line number or name for registering to ISR table entry and enable/disable the specific IRQ once received.
signal	string: c_macro	Alphanumeric C macro for referring to the IRQ's signal value.

Not valid in v1.1: replaced by irq.name attribute.

Table 22 (continued)

Name	Type	Description
name	string: <code>c_macro</code>	Interrupt name. This is used as a prefix for the interrupt signal and FLIH function symbols. <i>New in v1.1:</i> this replaces the use of the <code>irq.signal</code> attribute.
handling	enum: "FLIH" or "SLIH"	The handling pattern for the interrupt. <i>New in v1.1</i>

B.5.2 Optional attributes

Table 23 Optional IRQ attributes

Name	Type	Description	Default
description	string	Human readable description.	<i>null</i>

B.5.3 Example

```
{
  "description": "The secure timer interrupt",
  "name": "S_TIMER",
  "source": "SECURE_TIMER",
  "handling": "FLIH"
}
```

B.6 Typed string attributes

B.6.1 `c_macro`

An alphanumeric string that is used to construct C pre-processor symbols.

The string is all uppercase, must start with a letter, and can contain underscore characters.

B.6.2 `c_symbol`

An alphanumeric string that is used to construct C identifiers.

The string must start with a letter, and can contain underscore characters.

B.6.3 hex_string

The hexadecimal representation of a 32-bit, non-zero, unsigned integer.

The string starts with "0x", followed by between one and eight hexadecimal digits.

Appendix C: Migrating Secure Partitions to version 1.1

The version 1.1 framework APIs are compatible with the version 1.0 API. However, when a Secure Partition manifest specifies that it is using framework version 1.1, there are some necessary, incompatible changes that must be made to the manifest.

This section provides a guide to the changes that you need to make when migrating a Secure Partition developed for version 1.0 onto a framework that implements version 1.1.

C.1 Using an unmodified version 1.0 Secure Partition

If the framework supports the IPC model, and you do not need to take advantage of any of the new features in version 1.1, then an existing version 1.0 Secure Partition will work without modification in the new framework.

Otherwise, you will need to make some changes to your code.

Note:

It is possible to use the same source code (but not manifest files) for both version 1.0 and version 1.1 Secure Partitions. The `PSA_FRAMEWORK_VERSION` pre-processor macro enables appropriate code to be selected to match the framework.

C.2 Update the manifest to version 1.1

The first step when you want to use one or more of the version 1.1 features is to update the Secure Partition to version 1.1.

C.2.1 Manifest changes

1. Update the `psa_framework_version` attribute to version 1.1:

```
- "psa_framework_version": 1.0,  
+ "psa_framework_version": 1.1,
```

2. Add the `model` attribute at the top level of the manifest, specifying the IPC model:

```
+ "model": "IPC",
```

3. For each RoT Service, add the connection-based attribute within the service specification, with the value `true`:

```
+ "connection-based": true,
```

4. For each interrupt, within the `irq` specification:

- Replace the `signal` attribute within the `irq` specification, with a `name` attribute.
- Add the `handling` attribute, with the value `"SLIH"`.

```
- "signal": "MY_IRQ_SIG",
+ "name": "MY_IRQ",
+ "handling": "SLIH",
```

C.2.2 Source code changes

In a Secure Partition that uses version 1.1, some changes are needed to code dealing with interrupts:

1. The symbolic name of the interrupt is defined differently, and uses of these symbols will need to be updated in the source code.
 - In version 1.0, the name is `«signal»`, where `«signal»` is the value of the interrupt's `signal` attribute in the manifest.
 - In version 1.1, the name is `«name»_SIGNAL`. where `«name»` is the value of the interrupt's `name` attribute in the manifest.
2. Interrupts are initially disabled in a version 1.1 Secure Partition. Explicit calls to `psa_irq_enable()` need to be made for each interrupt during Secure Partition initialization, or when programming the peripheral interrupt.

C.3 Using version 1.1 features

After migrating the Secure Partition to version 1.1, the following sections summarize the changes required to use the new features in existing Secure Partition and RoT Service code.

C.3.1 Using the SFN model

To use the SFN model in a Secure Partition, the following changes need to be made to a Secure Partition using the IPC model. See also [Secure Functions on page 24](#).

1. Change the `model` attribute at the top level of the Secure Partition manifest, specifying the SFN model:

```
- "model": "IPC",
+ "model": "SFN",
```

2. Remove the `entry_point` attribute from the manifest, optionally replacing it with an `entry_init` attribute if your Secure Partition requires initialization before any of the RoT Service SFNs are called. Either:

```
- "entry_point": "my_sp_main",
```

or:

```
- "entry_point": "my_sp_main",
+ "entry_init": "my_sp_init",
```

3. If initialization is required, refactor the initialization code from the version 1.0 entry point function into the version 1.1 entry initialization function.

After initialization, the `entry_init` function returns the following values:

- Return `PSA_SUCCESS` if initialization succeeds.
 - Return `PSA_SUCCESS` if initialization is partially successful, and you want some SFNs to receive messages. RoT Services that are non-operational must respond to connection requests with `PSA_ERROR_CONNECTION_REFUSED`.
 - Return an error status if the initialization failed, and no SFNs within the Secure Partition must be called.
4. Add a Secure Function (SFN) to process messages for each RoT Service specified in the manifest. Each SFN will have the following prototype:

```
psa_status_t «name»_sfn(const psa_msg_t* msg);
```

where «name» is the service's name attribute from the manifest in lowercase.

Refactor the message handling code for each RoT Service into the SFNs:

- Each SFN will receive *connection*, *request* and *disconnection messages* for that RoT Service.
- The reply to the message occurs when the SFN returns, using the return value as the response status to the client.

C.3.2 Using a stateless RoT Service

To change a connection-based RoT Service into a stateless RoT Service, the following changes need to be made. See also [Stateless Root of Trust services on page 33](#).

1. Change the `connection-based` attribute in the service specification in the manifest, to `false`:

```
- "connection-based": true,  
+ "connection-based": false,
```

2. Optionally, specify the stateless handle index used to construct the [stateless handle](#) for the RoT Service. For example, to allocate the index 1:

```
+ "stateless_handle": 1,
```

By default, the implementation will allocate this value for the RoT Service.

3. Rework the Secure Partition code that handles messages for the RoT Service:
 - Remove code that handles *connection* and *disconnection messages*. You can assume (or assert) that every message received for that RoT Service is a *request message*.
 - Ensure that the Secure Partition does not call `psa_set_rhandle()` on a message for that RoT Service.
4. Rework all of the client calls to the RoT Service:
 - Remove calls to `psa_connect()` and `psa_close()`.
 - Replace the [connection handle](#) used in calls to `psa_call()` with the [stateless handle](#). The stateless handle has the name «name»_HANDLE where «name» is the service's name attribute from the manifest.

C.3.3 Using MM-IOVEC

To use memory mapped iovecs in an RoT Service, the following changes need to be made. See also [Memory-mapped IOVECs on page 41](#).

1. Check that the framework you are using supports MM-IOVEC, or consider implementing the changes conditionally in your code using the `PSA_FRAMEWORK_HAS_MM_IOVEC` pre-processor macro.
2. Enable MM-IOVEC for the RoT Service, by adding the `mm_iovec` attribute to the service specification in the manifest file, and giving it the value "enable":

```
+  "mm_iovec": "enable",
```

3. Rework the RoT Service code which reads input vectors and writes output vectors. Only do this for vectors where this provides a significant reduction in memory usage or improvement in performance, without introducing memory-safety vulnerabilities.
 - Replace the use of `psa_read()` to copy data from a client input vector, with calls to `psa_map_invec()`, and optionally `psa_unmap_invec()`. For example:

```
size_t len;
- uint8_t buffer[BUF_SIZE];
-
- n = psa_read(msg.handle, INBUF_IDX, &buffer, sizeof(buffer));
- // use the data in buffer[0..len]
+ const uint8_t *vec;
+
+ len = msg.in_size[INBUF_IDX];
+ if (len > 0) {
+     vec = psa_map_invec(msg.handle, INBUF_IDX);
+     // use the data in vec[0..len]
+ } // leave the framework to unmap the vector
```

To make the code portable to implementations that do not support MM-IOVEC, use `PSA_FRAMEWORK_HAS_MM_IOVEC` to conditionally include the appropriate code. For example:

```
size_t len;
+#if PSA_FRAMEWORK_HAS_MM_IOVEC
+ const uint8_t *vec;
+
+ len = msg.in_size[INBUF_IDX];
+ if (len > 0) {
+     vec = psa_map_invec(msg.handle, INBUF_IDX);
+     // use the data in vec[0..len]
+ }
+ #else
    uint8_t buffer[BUF_SIZE];

    len = psa_read(msg.handle, INBUF_IDX, &buffer, sizeof(buffer));
    // use the data in buffer[0..n]
+ #endif
```

- Replace the use of `psa_write()` to copy data into a client output vector, with calls to `psa_map_outvec()` and `psa_unmap_outvec()`. For example:

```

    size_t len;
-   uint8_t buffer[BUF_SIZE];
-
-   // construct the output data in buffer[0..BUF_SIZE],
-   // and set len to the output size
-   psa_write(msg.handle, OUTBUF_IDX, &buffer, len);
+   uint8_t *vec;
+   size_t sz;
+
+   sz = msg.out_size[OUTBUF_IDX];
+   if (sz > 0) {
+       vec = psa_map_outvec(msg.handle, OUTBUF_IDX);
+       // construct the output data in vec[0..sz],
+       // and set len to the output size
+       psa_unmap_outvec(msg.handle, OUTBUF_IDX, len);
+   }

```

To make the code portable to implementations that do not support MM-IOVEC, use `PSA_FRAMEWORK_HAS_MM_IOVEC` to conditionally include the appropriate code.

C.3.4 Using FLIH

To use *First-level interrupt handling* instead of SLIH for a Secure Partition interrupt, the following changes need to be made. See also *Enhancements for Secure Partition peripheral drivers* on page 51.

1. Change the handling attribute in the irq specification in the manifest, specifying "FLIH" handling:

```

-   "handling": "SLIH",
+   "handling": "FLIH",

```

2. Add an *FLIH function* to the Secure Partition with the following signature:

```
psa_flih_result_t «name»_flih(void);
```

where «name» is the interrupt's name attribute from the manifest in lowercase.

3. Rework the interrupt handling code:

- Any interaction with the peripheral that has low-latency requirements must be moved into the FLIH function.
- Any interaction with other Secure Partitions, or with the message processing APIs must remain in the *Secure Partition thread context*.
- If all of the handling code has been moved into the FLIH function, then:
 - The FLIH function returns `PSA_FLIH_NO_SIGNAL`.
 - The call to `psa_eoi()` for this interrupt is removed.
- If the FLIH function requires that further processing is done in the Secure Partition thread context, then:
 - The FLIH function returns `PSA_FLIH_SIGNAL`.
 - The call to `psa_eoi()` is replaced with a call to `psa_reset_signal()` from the Secure Partition thread context.
- If the interrupt is still enabled when the FLIH function returns the value `PSA_FLIH_SIGNAL`, then

the Secure Partition thread context must handle potential race conditions when it access any data shared with the FLIH function.

- We recommend that the Secure Partition thread context calls `psa_reset_signal()` first, before processing the data.
 - The Secure Partition thread context can use `psa_irq_disable()` and `psa_irq_enable()` to create a critical section, if this is necessary to safely update data shared with the FLIH function.
4. Review read and write access to peripheral MMIO registers, and consider using the [Register access functions for MMIO on page 63](#) instead.

Appendix D: Comparison between FF-M and TF-M frameworks

This appendix provides a more detailed analysis of the existing frameworks, comparing their strengths and weaknesses in different system use cases. This analysis provides the rationale for the addition described in [Secure Functions on page 24](#).

D.1 Background

D.1.1 The IPC model

Version 1.0 of *Arm® Platform Security Architecture Firmware Framework* [FF-M] describes a programming model, communication framework and API that is based around one or more secure execution contexts called *Secure Partitions*.

Each Secure Partition is programmed like an individual C program or task — polling for messages and other events, and responding to them. The communication API presents session-based connections to secure services, on which structured, synchronous requests are made by clients.

The communication broker (the *SPM*) in this framework also acts as a secure client identity provider, enabling more complex resource ownership and access control designs to be implemented with secure services.

Secure services are able to connect and makes requests as the client of other secure services, utilizing the same communication framework.

This overall design is typically referred to as the *IPC model* within the *Trusted Firmware-M* [TF-M] documentation.

This architecture diagram provides an overview of the components within the design:

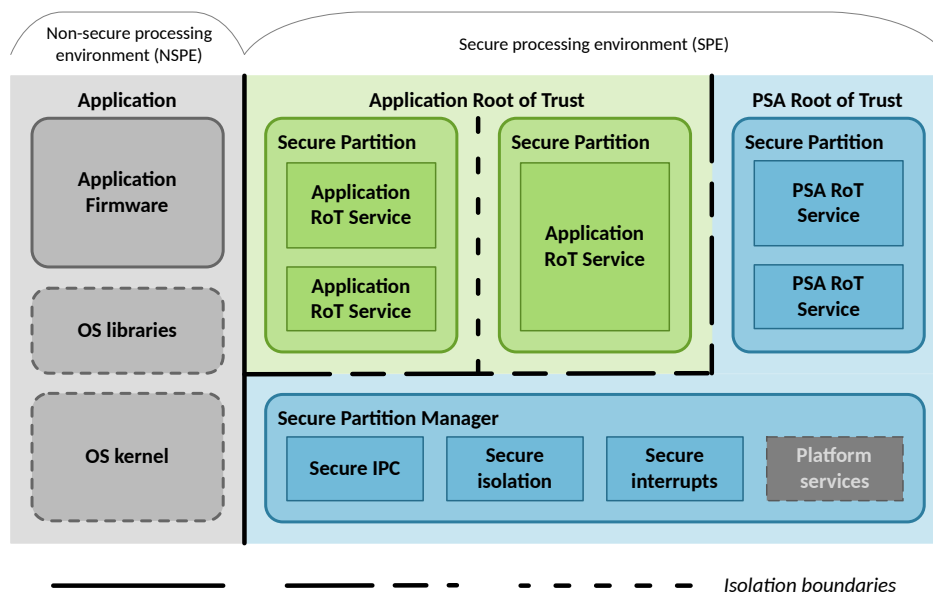


Figure 1 Elements of the FF-M v1.0 Architecture

D.1.2 The Library model

Version 1.0 of the *Trusted Firmware-M* [TF-M] project introduced a much simpler abstraction for secure services. The programming model is based around a set of secure service functions, each of which handles requests from a corresponding client-side function. The secure service functions are run as callbacks from the framework, and the framework is in control of the execution context and sequence in which secure service handlers run.

There is no concept of a connection in this framework. Each request to a secure service runs as an individual function call.

The constrained systems that this framework is designed for typically have a single non-secure client and single protected domain containing secure services. There is no real requirement for a client identity.

Secure services that require the use of other secure services can typically just call the underlying functionality directly, as there is no need to cross a protection boundary.

This communication framework is referred to as the *Library model* within the TF-M documentation.

D.2 Analysis

D.2.1 One or two architectures?

The IPC model provides significant control for a service developer to manage execution within each Secure Partition. The API permits the deferred completion of requests, and the ability to process multiple messages from different clients simultaneously within a single Secure Partition execution thread. The overall programming model is easier to reason about when integrating multiple Secure Partitions, which together enable concurrent handling of different secure services, in an implementation that provides a high isolation level. The API design requires that request data has to be copied between the client and service, mitigating many memory-safety vulnerabilities.

The Library model is simple to describe, and leads to a simple implementation in a system with two protection domains, where each secure service function must complete execution before another can start. Direct access to client memory is assumed by the API (parameter buffers are passed by address), which prevents the API being used in systems demanding higher isolation.

The IPC model is well suited for more complex systems and product designs, and the Library model is effective for simpler systems. However, there are two main challenges with the current situation:

- System and product requirements are not binary: there is a spectrum of system complexity and product security needs. For systems that fall in between the two points addressed by the FF-M IPC model and the TF-M Library model, there is no framework that is a good fit for the system.
- It is difficult to describe the two models and APIs in a single architecture, due to their current differences. Providing a single architecture which could span these use cases would enable framework implementations to be a better fit for more systems.

D.2.2 Scaling and Flexibility

The challenges with constructing a unified framework architecture, that includes both the IPC model and the Library model stem from the current frameworks' inability to scale effectively.

Scaling the IPC model

The IPC model does not scale down efficiently:

- Simple stateless or one-shot secure operations require a connection. Creating a temporary connection significantly increases the runtime cost; there is not always a good place to stash a connection handle.
- Simple secure services still require boilerplate code in the Secure Partition to handle signals and dispatch requests to their respective service handlers.
- The thread-based programming model requires that the framework manages extra execution contexts, and switching between them to process requests, even for an implementation that only provides isolation level 1.

Scaling the Library model

The Library model does not scale up efficiently or safely:

- Adding more protection domains breaks the assumptions that enabled simplicity with just two domains:
 - Secure service functions must be run in a separate isolation domain and execution context to the SPM or framework.
 - Client identity is required as there can be more than one client for a service, and the service must respect the isolation of the client domains.
 - Calls between secure services cannot be direct function calls, and might need to be managed by the SPM to cross an isolation boundary.
 - Secure services cannot be run on a single SPE execution stack.

Although an isolation level 2 design could be constructed by implementing the two-domain architecture between each pair of protection domains that need to communicate, this is not a simple design. The emergent, complex behavior of this system is derived from the implementation decisions instead of being defined by the framework architecture.

- Concurrent secure service execution either requires:
 - Grouping secure service functions, where secure service functions within the same group are run sequentially (that is, without concurrency), but secure service functions in different groups can interrupt or interleave with each other.
 - Allowing any secure service function to interrupt or interleave with execution of others.

Both approaches require additional execution contexts (in particular, they prevent the re-use of the message dispatcher's stack), even with isolation level 1.

Both also present the main risk of concurrent execution: shared data. Mitigating the risks of shared data, without isolating the concurrent contexts, requires synchronization primitives and increases the difficulty of ensuring that secure services are error-free.

- Mitigation of common memory-safety errors in secure services requires changing the API. Even when the request broker validates the client memory parameters, providing direct access to client memory leaves the secure service vulnerable to double-fetch bugs, data alignment errors, buffer overruns, and permits pointer-chasing using unvalidated memory addresses passed as data. The current API requires that **every** secure service implementation must be code reviewed carefully to mitigate against these vulnerabilities. In contrast, the API for the IPC model places mitigation for these risks within the framework implementation, and not in every single secure service.

Appendix E: Implementing session-less RoT Services

This appendix examines the options for optimizing [Secure Partition RoT Service](#) requests, when the RoT Service operations do not make any use of the session-based features of the version 1.0 API.

This analysis provides the rationale for the [Stateless Root of Trust services on page 33](#) extension.

E.1 Background

A client of a [Root of Trust Service](#) accesses the service with RoT Service API. A simple approach to implementing the RoT Service API is to wrap this around calls to the FF-M Client API.

In general, there are two typical usages for an RoT Service API:

- Session-based API usage.
- Session-less API usage.

Implementing a session-based API

Session-based API usage needs an initial process to establish the session, and subsequent operations are based on this session.

An FF-M connection handle can be embedded in the session instance object, and `psa_connect()` can be called during the session setup operation. The performance overhead of `psa_connect()` is diluted in this case, since there are multiple subsequent session operations for each call to `psa_connect()`.

Here is a simplified example of an RoT Service API implementation that takes this approach:

```
int32_t RoTServiceA_Open(uint32_t *p_hsession)
{
    psa_handle_t handle;

    handle = psa_connect(ROT_SERVICE_A_SID, ROT_SERVICE_A_VERSION);
    if (!PSA_HANDLE_IS_VALID(handle) {
        return PSA_HANDLE_TO_ERROR(handle);
    }

    /*
     * Here is the simplest scenario.
     * In practice, the session object has other members as well as the connection handle.
     */
    *p_hsession = (uint32_t)handle;

    return (int32_t)PSA_SUCCESS;
}

int32_t RoTServiceA_Control1(uint32_t hsession)
{
```

(continues on next page)


```

    return (int32_t)psa_call((psa_handle_t)hsession,
                            PSA_IPC_CALL,
                            NULL, 0, NULL, 0);
}

int32_t RoTServiceA_Control2(uint32_t hsession)
{
    return (int32_t)psa_call((psa_handle_t)hsession,
                            PSA_IPC_CALL + 1,
                            NULL, 0, NULL, 0);
}

void RoTServiceA_Close(uint32_t hsession)
{
    psa_close((psa_handle_t)hsession);
}

```

Implementing a session-less API

Implementing a session-less API efficiently using the FF-M API is more challenging. This type of API does not have an explicit session, and has no session setup operation.

In the FF-M version 1.0 framework, a connection is mandatory for accessing services. This requires the client to maintain a handle while accessing an RoT Service.

Here is a simplified example of a session-less RoT Service API implementation using the FF-M Client API:

```

psa_status_t RoTServiceB(void)
{
    psa_handle_t handle;
    psa_status_t status;

    handle = psa_connect(ROT_SERVICE_B_SID, ROT_SERVICE_B_VERSION);
    if (!PSA_HANDLE_IS_VALID(handle) {
        return PSA_HANDLE_TO_ERROR(handle);
    }

    status = psa_call(handle, PSA_IPC_CALL, NULL, 0, NULL, 0);

    psa_close(handle);

    return status;
}

```

There are two aspects of this approach that affect the runtime performance of the RoTServiceB() API:

- If RoTServiceB() is a frequently called API, the accumulated duration for calling psa_connect() and psa_close() is significant.
- There is a SID-lookup process within psa_connect() which is difficult to optimize, because each RoT Service ID is allocated by the RoT Service developer.

To improve the performance for calling session-less secure services, the following approaches can be considered:

- Increase the SID lookup performance.
- Increase the performance of the individual framework calls.
- Reduce the number of calls to `psa_connect()` and `psa_close()`.

Increasing the performance of individual framework functions is a framework implementation issue. In the following analysis, we consider how to eliminate calls to these framework functions for session-less RoT Service APIs.

E.2 Analysis

Before analyzing options, we consider the mechanisms for integrating an RoT Service API. Typically, the client needs to integrate the object files which contain the RoT Service API implementation. Here are the common scenarios:

1. There is no isolation between clients, the same object is shared by all clients.
2. There is isolation between clients, and the object is shared as read-only between clients. For example, this is what [\[TF-M\]](#) does with its Secure Partition Runtime Library (SPRTL) within the SPE.
3. There is isolation between clients, and the object is duplicated and linked with each client that uses the API.

For the example implementations of `RoTServiceA` and `RoTServiceB` above, the source compiles to object code with no writable data. This allows the framework to use any of the three mechanisms for integrating the RoT Service API with Secure Partition client code.

Optimizing `RoTServiceB()`

If the developer of `RoTServiceB()` wants to avoid always calling `psa_connect()` and `psa_close()`, one obvious approach is to save the connection handle somewhere, and reuse this for subsequent requests.

For example, `RoTServiceB()` could be rewritten as follows:

```
static psa_handle_t saved_handle_b;

psa_status_t RoTServiceB(void)
{
    psa_handle_t handle;

    handle = saved_handle_b;
    if (handle == PSA_NULL_HANDLE) {
        handle = psa_connect(ROT_SERVICE_B_SID, ROT_SERVICE_B_VERSION);
        if (!PSA_HANDLE_IS_VALID(handle)) {
            return PSA_HANDLE_TO_ERROR(handle);
        }
        saved_handle_b = handle;
    }

    return psa_call(handle, PSA_IPC_CALL, NULL, 0, NULL, 0);
}
```

If this handle is saved in a global area, it has the following effects:

- A non-const variable is needed to save the handle, and the object will contain a read-write data section.
- There are multiple clients that could call the API, but the framework assumes that each connection has at most one outstanding request. There is also no specified behavior if one task calls `psa_close()` on a handle that is currently used by another task in a call to `psa_call()`.
So each client should have its own connection handle, and multiple handles need to be saved. When used from a Secure Partition, the RoT Service API can allocate memory in the Secure Partition or rely on separate handle variables being allocated by the framework (if the framework use mechanism 3 above). For NPSE clients, this approach requires knowledge of the NSPE runtime environment, resulting in a non-portable RoT Service API.
- Even in a system where the same connection handle can be used by all clients, it is not possible to store this handle in a location that all clients can read in systems which implement higher levels of isolation. It is also a security risk for a trusted service to use a shared connection handle that could be tampered with by malicious code.

These disadvantages make this approach problematic. An alternative is to delegate the handle storage to the system using an abstracted API which avoids explicit use of shared global variables (especially NSPE clients):

```
psa_status_t RoTServiceB(void)
{
    psa_handle_t handle;

    handle = GET_ROT_HANDLE(CLIENT_ID);
    if (!PSA_HANDLE_IS_VALID(handle)) {
        handle = psa_connect(ROT_SERVICE_B_SID, ROT_SERVICE_B_VERSION);
        if (!PSA_HANDLE_IS_VALID(handle)) {
            return PSA_HANDLE_TO_ERROR(handle);
        }
        SET_ROT_HANDLE(CLIENT_ID, handle);
    }

    return psa_call(handle, PSA_IPC_CALL, NULL, 0, NULL, 0);
}
```

The disadvantages with this approach include:

- This involves an abstracted API within the RoT Service API library implementation. Ideally, the RoT Service API library should not have system dependencies, but now the SPE and NSPE need to implement the `GET_ROT_HANDLE()` and `SET_ROT_HANDLE()` functionality.
- In a simple system without memory management API, a custom allocating implementation is required.
- Need mechanisms to retrieve the caller's client ID to let abstract API find corresponding handle for this client.

Conclusion

None of these approaches work in a portable way, or they introduce new APIs that have to be made to work on each implementation into which the RoT Service is integrated.

To make a useful improvement for session-less APIs, support is required from the Firmware Framework itself.

E.3 Framework options

Idea 1: Fixed handle values

Idea: What will happen if the handle value is decided and known by clients at build time?

Then the situation would change as we do not need to store the handle after connection, since the handle value is already known.

The notes in [FF-M] §3.3.4 describe a connection handle allocation strategy in which different Secure Partitions can use the same handle value for different connections. This suggests that it is possible to make each client have their own connection to an RoT Service, using the same, fixed handle value.

An RoT Service API that follows the session-less pattern can work using this type of connection. `psa_call()` can be invoked directly with this handle value, if the handle is connected implicitly to the RoT Service by the framework. The resulting implementation of `RoTServiceB()` would look like this:

```
psa_status_t RoTServiceB(void)
{
    return psa_call(ROT_SERVICE_B_FIXED_HANDLE, PSA_IPC_CALL, NULL, 0, NULL, 0);
}
```

Ideally, the framework automatically makes the fixed handle ready for use before the client runs, or when the client first calls `psa_call()` with this handle. If the client has to explicitly ensure this handle is connected, this would reintroduce many of the client logic challenges that we want to solve.

Idea 2: No connection at all

Idea: Are there other opportunities for optimizing the framework for session-less RoT Service APIs?

The typical code for a session-less RoT Service API (see [original RoT ServiceB](#)) creates transient connections for each and every request. As a result, the RoT Service implementation cannot make use of the *handle* feature of the Secure Partition API for these calls, and does not do anything during the *connection* and *disconnection* messages that it receives.

If all of the requests for the RoT Service use the same pattern, then the framework can eliminate all of the following activities without affecting the functionality of the RoT Service:

- The client explicitly connecting to the RoT service using `psa_connect()`.
- The framework delivering *connection* or *disconnection* messages to the RoT Service.
- The RoT Service using the `rhandle` feature.

If all of this functionality is removed for a session-less RoT Service, this enables a simpler framework design for this type of service.

Conclusion

Although there are scenarios in which only one of these ideas is needed for an RoT Service, providing this flexibility adds complexity to the understanding of the features and the framework design and implementation. Combining these two ideas results in a simpler feature that provides substantial benefit for a large number of RoT Services and enables efficient framework implementation. This is what has been done in [Stateless Root of Trust services on page 33](#).

An existing RoT Service which includes a mixture of session-less type APIs and session-based APIs can still take advantage of stateless RoT Services when migrating to version 1.1. This is achieved by defining two RoT Services in the Secure Partition manifest, where one is connection-based and one is stateless, and using the appropriate handle in different parts of the RoT Service API.

Appendix F: Change history

F.1 Changes between *Alpha (Issue 0)* and *Beta (Issue 0)*

Manifest changes

- Clarified the behavior of the `type` attribute for implementations that provide isolation level 1. See [Use of the manifest type attribute in isolation level 1 on page 73](#).

API changes

- Simplified the interfaces for managing interrupt state. See [Secure Partition API changes for interrupt control on page 61](#).
- Introduced an `IMPLEMENTATION DEFINED` limit for message type values. See [Implementation defined maximum call type value on page 80](#).
- Changed the way that an FLIH function interacts with the SPM.
 - New return codes `PSA_FLIH_DISABLE` and `PSA_FLIH_PANIC` have been added to replace calls to `psa_irq_disable()` and `psa_panic()` for the common use cases in FLIH functions.
 - The availability of the Secure Partition API in FLIH functions is now `IMPLEMENTATION DEFINED`, providing flexibility for framework implementations. FLIH functions are already peripheral-specific, so it is acceptable to introduce a framework-specific dependency on FLIH functions that require additional interfaces to the SPM. See [FLIH Execution model on page 56](#).
- Added constants that define the error code ranges that are available for RoT Service developers to use. See [Secure Partition RoT Service status codes on page 81](#).

Other changes

- Made support for [Connection-based RoT Services](#) optional for an implementation of version 1.1.
- Clarified the allocation of reserved ranges of status codes in the API definition as well as the Programming Model. See [Secure Partition RoT Service status codes on page 81](#).
- Deprecated the Secure Partition doorbell feature. This includes the following APIs:
 - `psa_notify()`
 - `psa_clear()`
 - `PSA_DOORBELL`

See [Deprecated features on page 69](#).

- Revised the optional isolation rules in §3.1.5: this section is now split between a summary of the benefit of providing additional isolation policies, and an appendix that provides an expanded explanation of the example rules and their benefits. See [Reworking the optional isolation rules on page 75](#).