Arm® Platform Security Architecture Firmware Framework 1.0

Architecture & Technology Group

Document number:	DEN 0063
Release Quality:	Release
Issue Number:	0
Confidentiality	Non-Confidential
Date of Issue:	19/06/2019

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

Release Information

The change history table lists the changes that have been made to this document. For a detailed list of changes, see *Change history* on page 97.

Date	Version	Confidentiality	Change
June 2017	0.5-Dev-0	Confidential	First development release.
June 2018	0.9-Alpha-0	Confidential	Aligned with PSA Security Model and updated terminology. Revised programming model and API. Scope now includes PSA Root of Trust interfaces.
August 2018	0.10-Alpha-1	Confidential	Changes and clarifications to the API.
October 2018	1.0-Beta-0	Non-confidential	Changes to concurrency and signal waiting model. Prepared for non-confidential release.
February 2019	1.0-Beta-1	Non-confidential	Expanded and clarified Isolation model. Expanded and clarified error handling and defined standard error conditions. Defined the C runtime for Secure Partitions. Incorporated feedback from implementations.
June 2019	1.0.0	Non-confidential	Final version of the 1.0 API. Added a type parameter to RoT Service requests. Changed API for halting a Secure Partition. Updated the Connection State model.

Arm® Platform Security Architecture Firmware Framework

Arm Non-Confidential Document Licence ("Licence")

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

The Document consists solely of commercial items. You 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.

If any of the provisions contained in this Licence conflict with any of the provisions of any click-through or signed written agreement with Arm relating to the Document, then the click-through or signed written agreement prevails over and supersedes the conflicting provisions of this Licence. This Licence may be translated into other languages for convenience, and you agree 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 validity, construction and performance of this Licence shall be governed by English Law.

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

Arm document reference: LES-PRE-21585

References

This document refers to the following documents:

Document Number	Title
ARM DDI 0553A.i	*Arm®v8-M Architecture Reference Manual*
ARM DEN 0079	*PSA Security Model*
ARM DEN 0083A	*Arm Trusted Base System Architecture for M*
ARM DEN 0072A	*PSA Trusted Boot and Firmware Update*
ARM IHI 0085	*PSA Attestation API*
ARM IHI 0086	*PSA Cryptography API*
ARM IHI 0087	*PSA Storage API*
GP_REQ_025	GlobalPlatform *Root of Trust Definitions and Requirements* v1.1, June 2018

Terms and abbreviations

This document uses the following 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 *PSA Security Model* for details.
Application RoT Service	This is an RoT Service within the Application Root of Trust domain.
Confused deputy problem	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.
IDAU	Implementation Defined Attribution Unit. Part of the Armv8-M architecture, see *Arm®v8-M Architecture Reference Manual*.
Implementation Defined	This is used to describe functional behavior that is not defined by this specification. Each specific implementation of the PSA Firmware Framework must define, document and consistently implement its chosen behavior. Firmware developers can choose to depend on IMPLEMENTATION DEFINED behavior, but must be aware that their code might not be portable to another implementation.
IPC	Interprocess communication. The PSA Firmware Framework specifies an IPC mechanism to provide a communication channel for requests between isolated firmware partitions.
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.

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.
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	An error that is caused by the misuse of a programming interface. A PROGRAMMER ERROR is in the caller of the interface, but it is detected by the implementor of the interface.
PSA	Platform Security Architecture
PSA Immutable Root of Trust	The hardware, code and data that cannot be modified following manufacturing. See *PSA Security Model* for details.
PSA Root of Trust	This defines the most trusted security domain within a PSA system. See *PSA Security Model* for details.
PSA RoT Service	This is an RoT Service within the PSA Root of Trust domain.
PSA Updateable Root of Trust	The Root of Trust firmware that can be updated following manufacturing. See *PSA Security Model* 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.
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 *Root of Trust Definitions and Requirements*.
RoT Service	A set of related security operations that are implemented in a Secure Partition. The server endpoint of a PSA IPC channel. Multiple RoT Services can coexist in a single Secure Partition.
SAU	Security Attribution Unit. See *Arm®v8-M Architecture Reference Manual*.
Secure Partition	A thread of execution with protected runtime state within the Secure Processing Environment. Container for the implementation of one or more RoT Services. Multiple Secure Partitions are allowed in a platform.
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 or an Application RoT Service.
SPM	Secure Partition Manager.

	Part of the PSA Firmware Framework that is responsible for isolating software in Partitions, managing the execution of software within Partitions, and providing IPC between Partitions.
Trusted Boot	Trusted Boot is technology to provide a chain of trust for all the components during boot.

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® Platform Security Architecture Firmware Framework).
- The number and release (DEN 0063 1.0 Release 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.

1 Introduction

Arm's *Platform Security Architecture* (PSA) is a holistic set of:

- Threat models.
- Security analyses.
- Hardware and firmware architecture specifications.
- Open source firmware reference implementations.
- Independent evaluation and certification scheme – PSA Certified™.

PSA provides a recipe, based on industry best practice, that allows security to be consistently designed in at both a hardware and firmware level.

The PSA Firmware Framework specification defines a standard programming environment and firmware interfaces for implementing and accessing security services within the Root of Trust for a device.

Standardization of the programming environment and *Application Programming Interface* (API) for secure services across a range of secure hardware implementations allows and encourages the reuse of firmware components. This reuse is essential for sustainably reducing the cost of developing and integrating secure devices.

The interfaces defined in this version of the specification are optimized for constrained, connected systems, for example IoT devices.

This manual includes:

- A rationale for the design.
- A high-level overview of the architecture.
- A detailed description of the components of the PSA Firmware Framework.
- A complete definition of the PSA Firmware Framework interfaces.

This manual includes guidance for both implementers and users of the interfaces defined in this specification.

1.1 Scope

This specification derives its requirements from the overall system security architecture that is defined in the *PSA Security Model*. *Figure 1* from the *PSA Security Model* shows the overall architecture and its component parts.

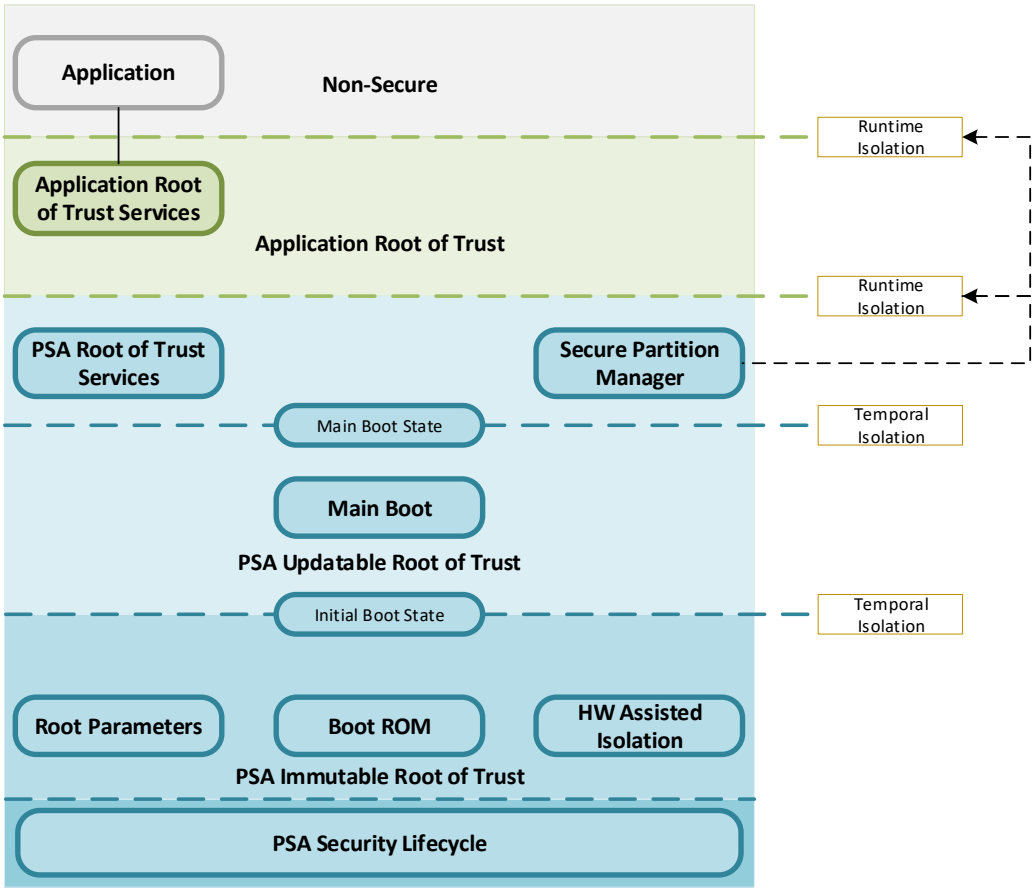


Figure 1 Overall PSA system architecture

The *PSA Security Model* divides execution within the system into two domains:

Domain	Contains
Non-secure Processing Environment (NSPE)	Application firmware OS kernel and libraries Most I/O peripherals
Secure Processing Environment (SPE)	Security firmware Security peripherals Root of Trust secrets

The *PSA Security Model* requires that no NSPE firmware or hardware can inspect or modify any SPE hardware, code or data.

Security functionality is exposed by PSA as a collection of *Root of Trust Services* (RoT Services). Each RoT Service is a set of related security functionality. For example, there might be an RoT Service for cryptography operations, and another for secure storage.

PSA further subdivides the SPE into two sub-domains: the PSA Root of Trust and the Application Root of Trust. The PSA Root of Trust provides the fundamental RoT Services to the system, as well as managing the isolated execution environment for the Application RoT Services.

The main components of the PSA Root of Trust are:

- The PSA Security Lifecycle, which identifies the production phase of the device and controls the availability of device secrets and sensitive capabilities, for example secure debug.
- The PSA Immutable Root of Trust, which is the hardware and non-modifiable firmware and data installed during manufacturing.
- The Trusted Boot and Firmware Update, which ensures the integrity and authenticity of the device firmware.
- The Secure Partition Manager, which manages isolation of the RoT Services, the IPC mechanism that allows software in one domain to make requests of another, and scheduling logic to ensure that requests are serviced.
- The PSA RoT Services, which provide essential cryptographic functionality and manage access to the immutable Roots of Trust for Application RoT Services.

This specification:

- Provides implementation requirements for the SPM.
- Defines a standard runtime environment for developing protected RoT Services, including the programming interfaces provided by the Secure Partition Manager for implementing and using RoT Services.
- Defines the standard interfaces for the PSA RoT Services.

A detailed specification of the other components of the PSA Root of Trust is in:

- The *PSA Security Model*.
- The *Arm Trusted Base System Architecture for M*.
- The *PSA Trusted Boot and Firmware Update*.

1.2 Design goals

1.2.1 Suitable for constrained devices

The interfaces defined in this document are designed to be suitable for a wide range of devices. The core of the PSA Firmware Framework provides the isolation and communication services. This foundation enables modular development and deployment of the protected security services that are required for different types of devices.

The communication mechanism enables multipart processing of service inputs, enabling implementation of the interface on devices with very limited memory. This allows RoT Services to present a natural API while reducing memory requirements for the RoT Service without compromising the isolation between SPE and NSPE.

1.2.2 Standard abstraction of hardware

Hardware flexibility

The PSA Firmware Framework is designed to be reasonably efficient to implement on different SoC architectures, while providing a consistent runtime environment for RoT Services. For example, the PSA Firmware Framework design is compatible with the following types of system:

- Armv8-M based SoCs, using TrustZone, or equivalent security IP, to protect the SPE.
- SoCs using multiple CPUs, providing an isolated CPU for the SPE and another for the NSPE.
- Armv7-M, Armv7-R and Armv8-R based SoCs, using an MPU to protect the SPE.
- Armv7-A or Armv8-A based SoCs, using TrustZone to protect the SPE.

Source-level portability of RoT Service code

A standard runtime environment for RoT Services across different hardware architectures and application operating systems allows the development and maintenance of trustworthy and reusable implementations of security functionality.

This also makes it easier to integrate the RoT Services from multiple vendors into a single device.

Implementation specific optimization

To enable compile-time and design-time optimization, the PSA Firmware Framework places no requirement on binary compatibility. Application and RoT Service developers will need to recompile their code against an appropriate implementation-defined version of the PSA Firmware Framework API headers to function correctly on that implementation.

1.2.3 Ease of use

The PSA Firmware Framework is designed to make it easy to develop RoT Services that support the management and applications of a device, when the RoT Services execute in an environment that is protected from the main application firmware.

Interfaces that are easy to program correctly reduce the risk of introducing logical programming errors and accelerate the development of software.

C Language

The PSA Firmware Framework API is defined in terms of the C99 specification, making it easy to adopt in systems that use the C and C++ programming languages.

Blocking functions

Most developers are familiar with synchronous functions which block waiting for the underlying task to complete before returning to the calling code. An asynchronous interface is difficult to design, challenging to port into different OS environments, and is generally difficult to use for developers familiar with synchronous APIs.

No shared state

Concurrent code is difficult to write correctly, and requires careful use of appropriate synchronization techniques when handling shared state. Individual RoT Services in the PSA Firmware Framework are designed to execute within a single thread with non-shared state. This eliminates the need to manage concurrent access, or review code for possible race conditions.

The lack of shared state results in services that are modular, enabling simpler integration of services from different vendors.

1.2.4 Mitigate coding error vulnerabilities

Programming errors are a very common cause of security vulnerabilities in systems: attackers will use them to cause behavior that is not intended by the firmware developer.

The PSA Firmware Framework is designed to prevent, detect and contain some common coding errors in RoT Services and in their clients. For example:

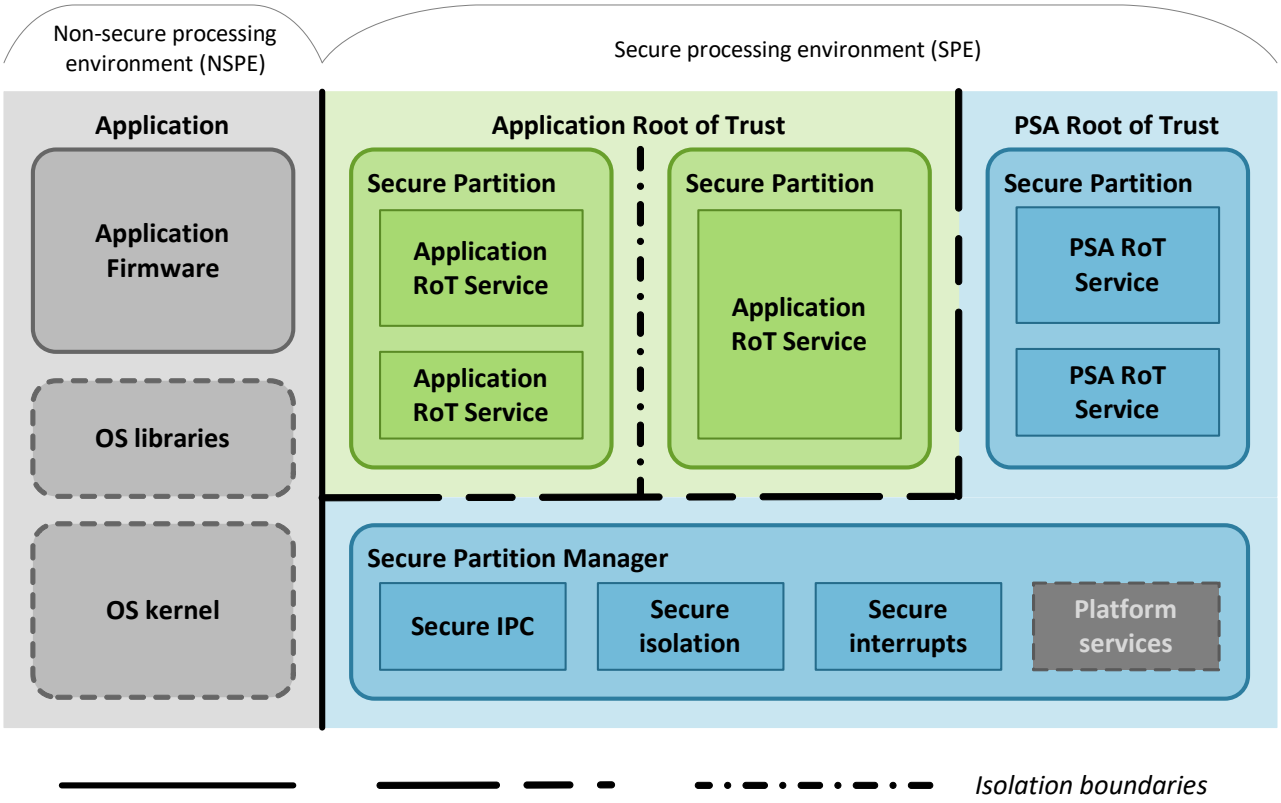
- The runtime isolation and execution model, and the IPC design eliminate some types of common error.
- The design of the PSA Firmware Framework API increases the ability for the SPM to detect programming errors in RoT Services and their clients.

- The SPM halts execution of code when a programmer error is detected.

The PSA Firmware Framework also provides APIs to enable a RoT Service to detect and contain programmer errors that are made by its clients.

2 Software architecture

This section provides an overview of the firmware architecture defined by the PSA Firmware Framework, and introduces the components and interfaces described by this specification. Section *Secure Processing Environment programming model* on page 20 explains the programmer's model for the Secure Processing Environment. *Programming API* on page 48 provides detailed API definitions for the interfaces defined by the PSA Firmware Framework.



Components of the system that are colored grey are outside of the scope of this specification.

Figure 2 Elements of the PSA Firmware Framework

2.1 Secure Partitions

The *PSA Security Model* describes various security services that run as part of the system's Root of Trust, divided into PSA RoT Services and Application RoT Services.

- The PSA RoT Services encapsulate and abstract the platform specific implementation of the security hardware and immutable secrets.
- The Application RoT Services provide the application and product specific security services.

Within the SPE, the RoT Services require an execution environment which provides access to resources, protection of its own code and data, and mechanisms to interact with other components in the system. In the PSA Firmware Framework, a Secure Partition is this execution environment.

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.

Each Secure Partition can host one or more RoT Services. Typically, related Services that share underlying functionality or data are implemented within the same Secure Partition.

A Secure Partition is either part of the PSA Root of Trust or part of the Application Root of Trust, depending on the services that it implements. This distinction is particularly important for isolation level 2, see *Isolation* on page 17.

Each Secure Partition has a persistent identifier, called a Partition ID, that can be used for access control within the system. These identifiers must be unique within the device, protected by the Secure Partition Manager at runtime and unchanged when firmware is rebooted or updated.

2.2 Secure Partition Manager

The *Secure Partition Manager* (SPM) is the most privileged firmware, which provides the fundamental security services to secure the PSA Root of Trust and enables isolated firmware components to communicate.

This specification defines the requirements for an SPM implementation, including the interfaces that it must provide to the Secure Partitions, and for the RoT Service clients in the NSPE which are depicted in Figure 3.

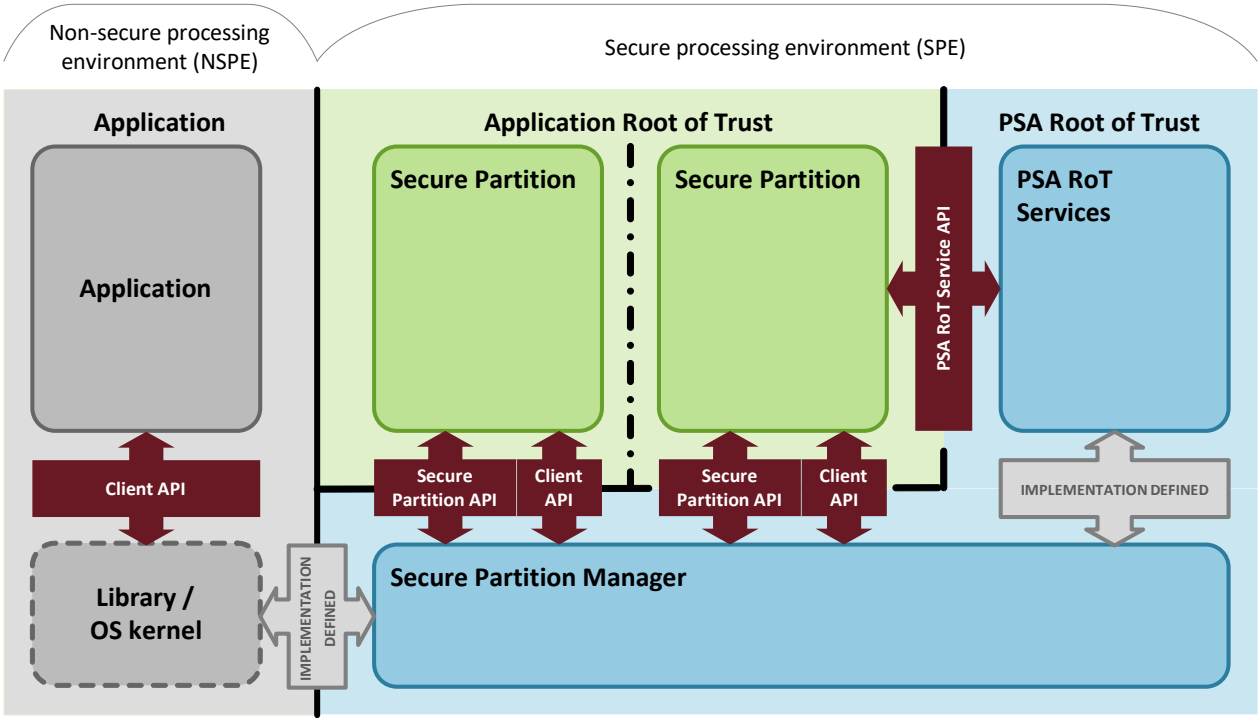


Figure 3 Scope of PSA Firmware Framework specification

The SPM implements:

- The isolation logic to separate the Secure Partitions.
- The IPC mechanism that allows software in one partition to make requests of another.
- Access to secure peripherals, if required, and interrupts.

- A mechanism to access the PSA RoT Services. This can be via IPC or by an IMPLEMENTATION DEFINED mechanism.
- Scheduling logic to ensure that requests are delivered to and processed by the Secure Partition.

The SPM provides these services via:

- The PSA Firmware Framework Client API.
- The PSA Firmware Framework Secure Partition API.
- The PSA RoT Services API.

These are defined in *Programming API* on page 48.

Some platforms include functionality that can only be accessed by firmware at the highest privilege level. For example, platform power control or control registers that are shared by secure and non-secure firmware. These *Platform services* must be implemented as part of the SPM, but the mechanism by which the NSPE firmware accesses these services is IMPLEMENTATION DEFINED.

2.3 Isolation

The *PSA Security Model* describes the use of isolation between software domains to protect the integrity and confidentiality of data assets. The SPM is responsible for isolating the different domains within the system. Depending on the level of isolation implemented, this can include:

- Isolation of the SPE from the NSPE.
- Isolation of the PSA Root of Trust from the Application Root of Trust.
- Isolation of Secure Partitions within the SPE.

The isolation must be enforced by platform hardware, which must be configured by the SPM. This ensures that the isolation applies to both software and hardware, and protects against both deliberate attack and defective software. See *Arm Trusted Base System Architecture for M* for PSA hardware isolation requirements and example implementations.

Increased isolation improves the security and robustness of the system by reducing its vulnerability to software defects, but this has a negative effect on additional hardware, memory, performance or energy. To support implementations that provide different security, performance and cost trade-offs, the *PSA Security Model* specifies multiple levels of isolation.

Table 1 provides a summary of the three supported isolation levels.

Table 1 PSA Isolation security levels

Isolation level	Purpose	Protection domains	Description
Level 1	SPE isolation.	Two	SPE is protected from access by Non-secure application firmware and hardware.
Level 2	PSA Root of Trust isolation.	Three	In addition to Level 1, the PSA Root of Trust is also protected from access by the Application Root of Trust.
Level 3	Maximum firmware isolation.	Three or more	In addition to Level 2, each Secure Partition is sandboxed and only permitted to access its own resources. This protects each Secure Partition from access by other Secure Partitions and protects the PSA Root of Trust from access by any Secure Partition.

The properties of each isolation boundary can also vary between implementations, and between different boundaries within a single implementation. Each implementation must document the isolation properties of the boundaries that it enforces. *Isolation architecture* on page 21 describes each level in more detail and specifies the requirements of a PSA Firmware Framework implementation.

Arm recommends that Secure Partition firmware is designed to run correctly with Level 3 isolation. For example, the Secure Partition firmware does not assume that data can be shared with another Secure Partition or the NSPE. This approach increases the portability of the firmware to run on multiple PSA implementations, and reduces the risk of introducing vulnerabilities related to data sharing.

Isolation within the NSPE

Sandboxing tasks within the application firmware is a well-known technique for providing improved robustness and security. This technique typically uses an MMU or MPU to only permit a task to access the code, data and devices that it needs to function correctly.

For example, if the hardware platform includes a Non-secure MPU, the Non-secure OS can use the MPU to implement isolation within the NSPE.

Isolating the application firmware in this way requires the use of an Non-secure OS that is designed for separating the OS kernel from the application tasks. The PSA Firmware Framework enables such a Non-secure OS to extend the isolation of application tasks into the SPE resources that each task uses, see *Client identification* on page 37.

Isolation of firmware within the NSPE is independent of the secure partitioning provided by the PSA Firmware Framework. System developers should use a product security analysis to determine the importance of this mitigation, which is beyond the scope of this specification.

2.4 RoT Services

Security functionality is exposed by PSA as a collection of RoT Services. Each RoT Service is a set of related security functionality, for example, there might be an RoT Service for symmetric cryptography operations, and another for random number generation.

Each RoT Service is either a PSA RoT Service provided by the PSA Root of Trust, or an Application RoT Service provided by the Application Root of Trust. See *Figure 2* on page 15.

Access to an RoT Service is provided by a programming API, which is typically defined in the C language. PSA defines standard programming APIs for accessing the PSA RoT Services. These are defined in *PSA RoT Services* on page 80.

An RoT Service can allow access only from the SPE, or from both the SPE and NSPE.

RoT Services are typically implemented within a Secure Partition. Each Secure Partition can implement one or more 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.

RoT Services can identify the Partition of the client, enabling the RoT Service to apply access control to resources and services that it provides to clients. For example, a secure storage service can use client-specific encryption keys to protect a client's persistent data from other clients.

2.5 Secure IPC

RoT Services are accessed from other Partitions via the PSA Secure IPC framework that is implemented in the SPM.

The IPC framework is a connection-based client and server model that provides remote procedure call behavior for calling clients. Clients can be in the NSPE or a Secure Partition. Servers implement an RoT Service within a Secure Partition.

An RoT Service will typically provide an API for its clients within a client library that encapsulates the use of the PSA IPC interface, as well as the server implementation within a Secure Partition.

An example RoT Service architecture is shown in *Figure 5* on page 30.

Secure IPC is session-based, and requests are always sent via a connection between the client and the RoT Service. IPC messaging is based on a request-and-response model. This is a model in which the client is blocked by the client request until the RoT Service has responded. This allows efficient implementation on smaller systems that use simple or no scheduling logic in the Non-secure application firmware.

The IPC design does not share memory between the client and the RoT Service. This improves security and flexibility for implementations:

- Shared memory across trust boundaries is a significant source of vulnerability in secure systems, because its contents must be treated as untrusted and volatile. Limiting the software that accesses shared memory reduces the risk of vulnerabilities in the system.
- Requiring the implementation to provide shared memory constrains the hardware system design, preventing implementations on hardware that provides physical separation between NSPE and SPE memory.

The IPC framework also provides no support for direct access to client memory from the RoT Service. If a client does not trust the server, providing the RoT Service with arbitrary access to client memory would be a breach of the security model.

Although direct access from the RoT Service to client memory is permitted on some systems, for example, when it is allowed by the isolation rules, Arm recommends that this direct access is not used by an RoT Service for the following reasons:

- The software will not be portable to all systems as the PSA isolation architecture does not guarantee access to client memory from the RoT Service.
- There is no mechanism for the server to check that the memory address from the client refers to memory that the client is permitted to access. A malicious client might try and use the RoT Service to disclose or tamper with memory belonging to another component in the system.

RoT Services in a Secure Partition can be clients of other RoT Services – as shown in *Figure 5* on page 30 – and can use the same RoT Service client library as non-secure clients. The blocking nature of the IPC design implies that a Secure Partition cannot depend on one of its own RoT Services as the chain of IPC requests will cause a deadlock in the firmware. The SPE must not include such dependency cycles between RoT Services.

2.6 Startup

Immediately following the initial stages of Trusted Boot, described by *PSA Trusted Boot and Firmware Update*, the components described in this specification must be securely initialized in sequence:

1. Initialize the SPM and secure IPC framework:
    a. Prepare the registry of Secure Partitions and RoT Services.
2. Configure all necessary hardware isolation components.

Unless the firmware is strongly coupled to the boot firmware, Arm recommends that the SPM fully configures the hardware isolation instead of relying on the configuration set up during boot.

3. Initialize all Secure Partitions:
    a. Initialize the private data, for example variables, stack and heap.
    b. Create the required execution context.
    c. Execute the entry point declared in the manifest file.
4. Process IPC connections and requests between Secure Partitions until they are all idle.
5. Load and verify the NSPE firmware, if not done by an earlier boot stage.
6. Start execution of the NSPE.

3 Secure Processing Environment programming model

This section provides detailed definitions, rules and guidance for the components in the PSA Firmware Framework. Specification of the manifest definition and programming interfaces is in *Programming API* on page 48.

The SPE hosts several Secure Partitions, managed by the *Secure Partition Manager* (SPM).

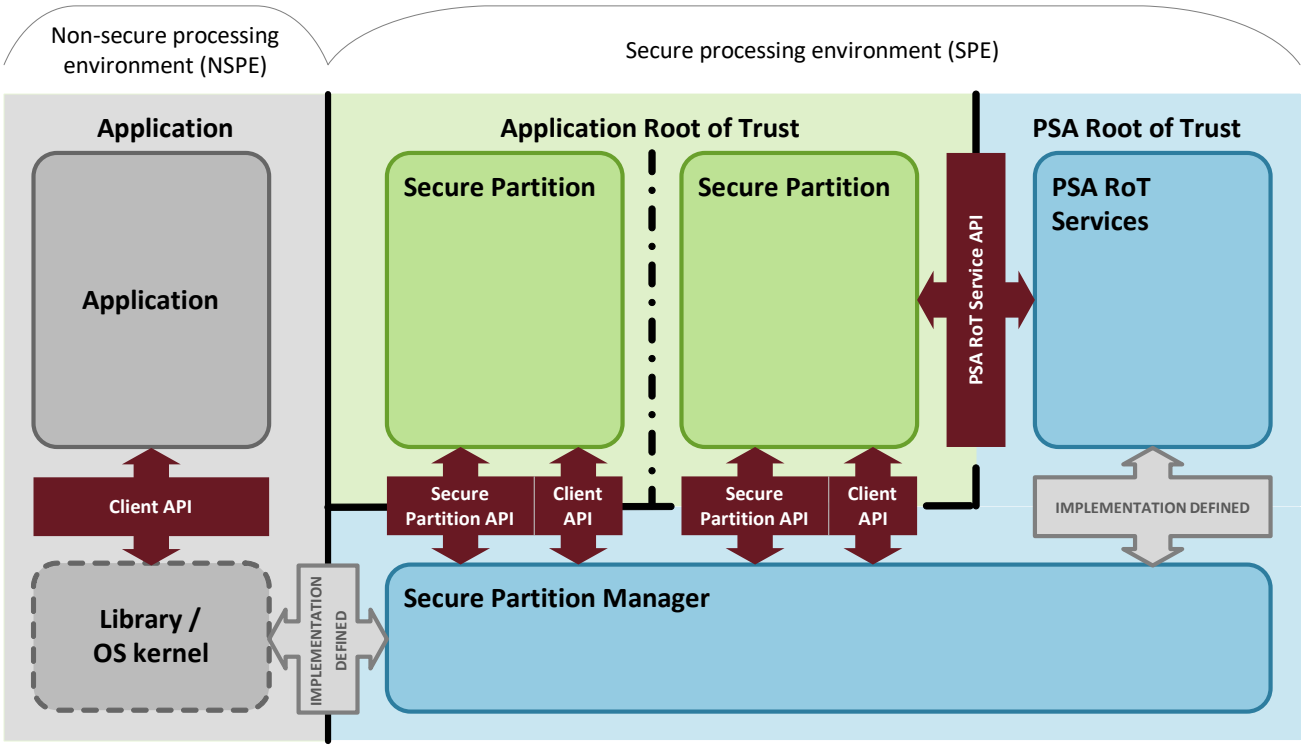


Figure 4 Interfaces in the PSA Firmware Framework

The SPM is permitted to fully isolate Secure Partitions from each other depending on the underlying hardware capabilities. This, and the lower levels of isolation, are described in *Isolation architecture* on page 21.

A Secure Partition provides a simple, protected, single-threaded runtime environment for development of secure services. *Secure Partitions* on page 25 describes the properties of Secure Partitions, how they are instantiated and their runtime behavior.

A Secure Partition contains one or more of the following elements:

- An RoT Service. *RoT Services* on page 30 describes the IPC model provided by the PSA Firmware Framework.
- A peripheral driver. *Secure peripheral drivers* on page 42 describes the Secure Partition support for managing secure peripherals.

PSA Firmware Framework defines standards for declaring Secure Partition properties and for providing the implemented APIs to Secure Partition source code. This allows source-level portability of Secure Partition firmware between different implementations of the PSA Firmware Framework.

PSA Firmware Framework does not define the *Application Binary Interface* (ABI) between Secure Partitions and the SPM, because this is specific to the platform and system architecture. Defining only a source code API allows design-time and compile-time optimizations to be provided by the implementation.

The C language is used to define the interfaces due to it being commonly found in OS libraries, application frameworks and ease of binding to other languages.

The implementations of the framework APIs and mechanisms by which they interact with the SPM are defined by the SPM provider.

PSA recommends that the SPM implementation also provides reference firmware source code that is used by the NSPE firmware to access the SPM IPC functionality.

3.1 Isolation architecture

This specification defines three primary levels of isolation for the firmware and hardware that provide different levels of security, performance and cost. Arm expects that each silicon partner offers products at each isolation level, so that Device Manufacturers can choose an implementation based on their specific use case.

The level of isolation is provided and enforced by the SPM, and applies to the memory addressable data and devices within the system.

Although secure applications require similar protection for persistent data stored on the device, this protection is not provided directly by the SPM or the isolation architecture. An implementation of the PSA Firmware Framework must include the Internal Trusted Storage RoT Service that provides the necessary confidentiality and integrity protection for persistent data as described in the *PSA Security Model*.

3.1.1 Memory Assets

The isolation architecture divides items in the system memory map into three classes of memory *asset* that require protection:

Table 2 Memory asset classes

Asset class	Memory items in this class
Code	Executable instructions. Constant data, in systems in which this is interleaved with executable instructions.
Constant data	Compile-time constants. Text strings or messages. Fixed tables and dictionaries. Configuration data.
Private data	Runtime program state, including: • Variables.

- Execution stacks.
- Allocation heap.
- Memory-mapped I/O regions.

3.1.2 Memory access rules

There are three *access methods* to memory assets:

- Read.
- Write.
- Execute.

The SPM must implement the following rules for protecting memory assets:

Table 3 Permitted access methods for memory assets

	Access rule	Rationale
I1	Only *Code* is executable.	Preventing execution of writable data mitigates the primary buffer-overrun attack vector. Preventing execution of read-only data reduces the attack surface available for *Return-Oriented Programming* (ROP) and *Jump-Oriented Programming* (JOP) attacks.
I2	Only *Private data* is writable.	Preventing modification of *Code* and *Constant data* assets mitigates accidental or malicious attempts to corrupt the execution control flow.

These rules are summarized in the following table:

	Asset class		
Access method	**Code**	**Constant data**	**Private data**
Read	Yes	Yes	Yes
Write			Yes
Execute	Yes		

NSPE access to its own assets is outside the scope of the PSA Firmware Framework specification. However, Arm recommends that the NSPE implements the same rules for memory accesses to its own assets.

3.1.3 Protection domains

Each isolation level divides the system into a set of non-overlapping *protection domains*.

At each isolation level, there is one protection domain that contains the SPM. Every other protection domain must trust the code in the SPM's protection domain for the following reasons:

- The SPM configures the protection mechanisms that enforce the isolation.
- The SPM requires access to the memory assets of any protection domain that uses the PSA Firmware Framework APIs to implement these APIs.

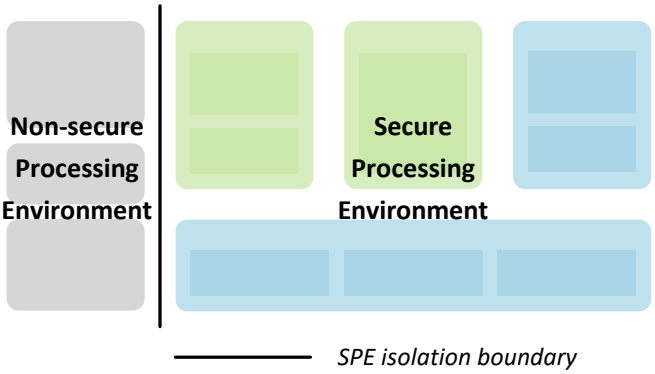
The SPM protects assets in a protection domain from access by other protection domains. The minimum set of protections that must always be implemented are defined by *Mandatory isolation rules* on page 23. The SPM can also implement the *Optional isolation rules* on page 24 to improve resistance to attack and robustness against implementation errors. These rules use the protection domain definitions and the *needs protection from* relationships between protection domains that are defined as follows:

Isolation level 1

Level 1 has an isolation boundary between the SPE and the NSPE.

The protection domains and the required protection for isolation level 1 are as follows:

Protection domain	Needs protection from
Non-secure Processing Environment (NSPE)	-
Secure Processing Environment	NSPE

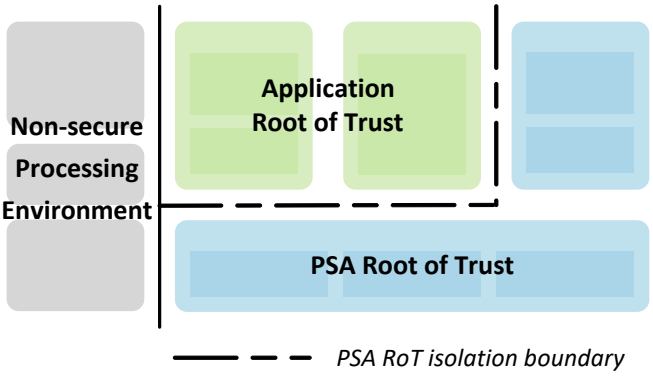


Isolation level 2

Level 2 introduces an isolation boundary between the PSA Root of Trust and the Application Root of Trust.

The protection domains and the required protection for isolation level 2 are as follows:

Protection domain	Needs protection from
NSPE	-
Application Root of Trust	NSPE
PSA Root of Trust	NSPE Application Root of Trust

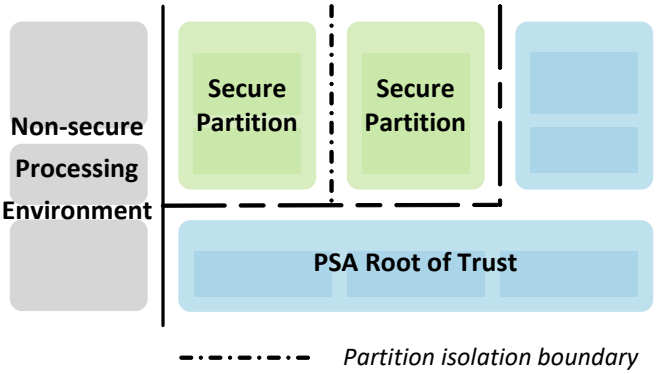


Isolation level 3

Level 3 introduces isolation boundaries between each of the Secure Partitions in the Application Root of Trust.

The protection domains and the required protection for isolation level 3 are as follows:

Protection domain	Needs protection from
NSPE	-
Secure Partition	NSPE Other Secure Partitions
PSA Root of Trust	NSPE Secure Partitions



3.1.4 Mandatory isolation rules

A protection domain is the owner of its memory assets. Access to these assets can originate from the same domain or from other domains.

The isolation rules cover both direct and indirect access:

- A direct access is one issued by the PE executing a load or store instruction in the domain firmware.

- An indirect access is one issued by a peripheral device that is managed by the domain.

The SPM must implement the following rules for protecting memory assets:

Table 4 Mandatory isolation rules

	Mandatory Isolation rule	Rationale
I3	If domain A needs protection from domain B, then *Private data* in domain A cannot be accessed by domain B.	Protecting the confidentiality and integrity of runtime state belonging to a domain.

3.1.5 Optional isolation rules

The following additional measures can be implemented by the SPM to improve isolation-based protection. These measures are not required to comply with the *PSA Security Model*, but these measures enhance the protection against common software attack techniques and implementation errors.

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.

Table 5 Optional isolation rules

	Optional isolation rule	Rationale
I4	If domain A needs protection from domain B, then *Code* and *Constant data* in domain A is not readable or executable by domain B.	Protecting the confidentiality of read-only assets. This strengthens rule **I3** to protect all assets from untrusted domains.
I5	*Code* in a domain is not executable by any other domain.	Eliminating shared *Code* reduces the attack surface available for ROP and JOP attacks.
I6	All assets in a domain are private to that domain and cannot be accessed by any other domain, with the following exception: The domain containing the SPM can only access *Private data* and *Constant data* assets of other domains when required to implement the PSA Firmware Framework API.	Only permitting access to assets that are within the domain mitigates a *Confused deputy attack*, and isolates an exploit to the domain in which it originates. If implemented, this rule upgrades and replaces rules **I3**, **I4** and **I5**.

SPM implementation note:

Implementation of recommendations **I4**, **I5** and **I6** is optional:

- The implementation can depend on system capability.
- The implementation can increase the code footprint and reduce the runtime performance of the system.
- The implementation can increase the complexity of the SPM functionality.

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.

3.1.6 Violating the isolation rules

If an attempt is made to violate an isolation rule which is implemented by the SPM, the SPM must implement one of the following behaviors:

- The access is detected by the SPM and is treated as an Internal fault in the execution context that made the access. This will terminate the execution context. See *Internal fault* on page 44.
- The access is ignored:
  - Reads will return an UNKNOWN or IMPLEMENTATION DEFINED value.
  - Writes will be ignored.

Arm recommends that isolation rule violations are treated as an Internal fault whenever it is possible for the SPM to detect the violation.

3.2 Secure Partitions

A Secure Partition is specified by source code that consists of:

- A Secure Partition manifest, with the following properties:
  - Written in JSON format.
  - Conforms to the schema defined in this specification. See *Manifest definition* on page 49.
  - Fully describes the Secure Partition identity and resources. See *Secure Partition manifest* on page 26.
- Secure Partition code, with the following properties:
  - Written in the C or C++ programming language.
  - Executes in a runtime environment based on the C99 freestanding requirements. See *Secure Partition C runtime* on page 55.
  - Single-threaded, executing an infinite loop that waits for inputs. See *Secure Partition execution* on page 27.
  - Runtime state which is private, that is not shared with other Secure Partitions.
  - Communicates with other Secure Partitions using the API defined in this specification.
  - Implements the RoT Services listed in the Secure Partition manifest.

The implementation of the PSA Firmware Framework must include tools that process all the Secure Partition manifests as part of building the SPE firmware. These tools:

- Ensure that manifests follow the rules in this specification.
- Provide information generated from the manifests to the Secure Partition code at build time.
- Provide information to the SPM for instantiating and managing the Secure Partitions and RoT Services.

3.2.1 Secure Partition identity

Every Secure Partition has an *Identity* (ID). The Secure Partition ID is a 32-bit signed integer that provides a local, unique, persistent and trusted identity for client endpoints in the IPC framework.

Every Secure Partition also has a symbolic name that is defined by the Secure Partition developer, for use by Secure Partition source code.

Secure Partition IDs are allocated by an IMPLEMENTATION DEFINED mechanism.

The SPM Implementation provides a binding of the allocated Secure Partition IDs to the Secure Partition names in a standard source header file. The SPM provides the client Secure Partition ID to RoT Services in every IPC message that is delivered.

Secure Partition IDs must be positive. Negative values are reserved for client endpoints that originate in the NSPE, so the NSPE cannot forge the client ID of a Secure Partition.

Secure Partition IDs must be unique within the system.

Secure Partition IDs must be fixed across updates. Having fixed Secure Partitions allows them to be used for authenticating client endpoint ownership of data on the device, and ensures that the data remains private to the originating Secure Partition.

SPM implementation note:

The SPE build tools can allocated Secure Partition IDs by either manual or automated mechanisms. For example, the PSE build tools might implement one of the following approaches:

- The SPE build tools can require that the system integrator provide a system manifest file that defines the Secure Partition ID for each Secure Partition that is included in the system. This makes it easier to maintain a persistent ID in subsequent versions of the firmware.
- The SPE build tools can allocate the Secure Partition IDs automatically, and generate a database of the mapping which is used on subsequent builds of the firmware to maintain persistent Secure Partition IDs.

3.2.2 Secure Partition manifest

Each Secure Partition must have resource requirements declared in a manifest file. The SPM uses the manifest file to assemble and allocate resources within the SPE. The manifest includes the following:

- A Secure Partition name.
- A list of implemented RoT Services.
- Access to other RoT Services.
- Memory requirements.
- Scheduling hints.
- Peripheral memory-mapped I/O regions and interrupts.

The manifest file is a contract between the Secure Partition developer and the SPM environment. A manifest with requirements that cannot be fulfilled will be rejected by the build system.

Manifest files are analyzed by the SPE build tools to validate dependencies, and produce a runtime binary that satisfies the required isolation level.

The manifest files define values and identifiers that are used by the Secure Partition source code to refer to the Secure Partitions and their resources. The SPE build tools generate these definitions when processing the manifest files for inclusion in the Secure Partition source code.

Access control between Secure Partitions must be specified in the manifest files. Access control is achieved by listing the dependencies to other RoT Services. Connection to an RoT Service is not permitted if the service is not declared in the client Secure Partition manifest.

Each RoT Service listed creates a dependency from the client Partition to the RoT Service Partition. Within the resulting network of dependencies, there must be no circular dependencies between Secure Partitions. If there were circular dependencies between Secure Partitions, these would result in deadlock because the IPC requests block the client. For the same reason, a Secure Partition must not make an IPC request to an RoT Service that is defined within itself.

PSA recommends that the build system removes Secure Partitions that cannot be called at runtime. A Secure Partition is called if one or more of the following are true:

- The manifest defines one or more RoT Services that can be called from the NSPE.
- An RoT Service in the Secure Partition is listed in another Secure Partition's dependencies.
- The manifest declares an interrupt source.

Additional memory regions must be explicitly declared in the manifest files.

For more information about the manifest attributes, see *Manifest definition* on page 49, and for the Secure Partition JSON schema see *Secure Partition manifest schema* on page 87.

3.2.3 Secure Partition execution

Each Secure Partition has a single thread of execution which is managed by the SPM.

A Secure Partition begins execution from the declared `entry_point` symbol in the manifest.

Following initialization, the Secure Partition thread must be structured as a loop that repeatedly waits for input before processing the input.

A Secure Partition must never exit its loop or return from the entry point. To do so would be a programming error. If interrupted, a Secure Partition will always resume execution from the location it was interrupted at.

Inputs to a Secure Partition are all in the form of signals. Each signal represents a distinct input source for the Secure Partition.

Signals

A signal is an asynchronous notification sent to a Secure Partition to notify it of an input that requires action. Each Secure Partition has up to 32 different signals. A signal is represented as a single-bit value within a 32-bit integer.

Each signal must be one of the following types:

Table 6 Secure Partition signal types

Signal type	Purpose	Asserted when
RoT Service	The primary IPC mechanism in the PSA Firmware Framework.	A message is queued for the RoT Service.
Interrupt	Managing hardware interrupts in a Secure Partition.	A bound interrupt line is asserted.
Doorbell	A notification mechanism between Secure Partitions.	A Secure Partition has notified this one.

Unassigned	This signal is not used in this Secure Partition.	Never.
Reserved	The signal is reserved for future use.	Never.

The following signals are reserved in all Secure Partitions:

Signal number	Description
`0x00000001U`	Reserved
`0x00000002U`	Reserved
`0x00000004U`	Reserved
`0x00000008U`	`PSA_DOORBELL`

The remaining 28 general signals can be associated with other inputs to the Secure Partition. The use of each signal value is specific to the Secure Partition and can be different or unassigned in another Secure Partition.

The Secure Partition manifest lists all interrupts, RoT Services and signal names for use by the Secure Partition source code. The SPM implementation allocates a signal value to each interrupt and each RoT Service, and provides source header files that bind those values to the symbolic names defined by the Secure Partition manifest files. The location and names of the pre-processor symbols are defined in *Manifest attributes* on page 49.

Waiting for signals

To wait for signals or test the current signal state, a Secure Partition calls `psa_wait()`.

This function takes a parameter indicating one of the following behaviors:

- The call blocks until at least one signal is asserted.
- The call polls the current signal status.

The call returns a signal mask that indicates the set of asserted signals, which may be any of the signal types.

The function also takes a signal mask which indicates the set of signals that the caller is waiting for. Signals that are not included in the mask are ignored when waiting, and are removed from the returned set of signals.

Filtering signals in this way when calling `psa_wait()` is useful in certain scenarios, for example:

- Waiting for a specific peripheral interrupt signal without polling.
- Ignoring a specific interrupt signal, if the hardware peripheral cannot disable the interrupt source.
- Ignoring an RoT Service signal while still processing a message for that service. This technique allows an RoT Service to enforce one-at-a-time processing of requests.

Handling signals

A signal remains asserted until it is processed by the Secure Partition.

A Secure Partition indicates it has completed its processing of a signal using a PSA function. The specific function depends on the type of signal:

- An interrupt signal requires a call to `psa_eoi()` to indicate that the interrupt has been handled at the source, and the SPM can reactivate the interrupt line. The call to `psa_eoi()` will clear the signal unless the source reasserts the interrupt. See *Secure peripheral drivers* on page 42.
- An RoT Service signal requires `psa_get()` to retrieve the message, and then appropriate processing to handle and complete the message. See *Processing RoT Service messages* on page 35.
- The doorbell signal requires `psa_clear()` to clear it.

The signal value is used when calling `psa_get()` and `psa_eoi()` to indicate which signal is being handled.

If multiple signals are returned by `psa_wait()`, the Secure Partition decides the order in which to process them. The Secure Partition is not required to process all asserted signals before calling `psa_wait()` again. Any signals that have not been handled will remain asserted, and will be returned by `psa_wait()`.

Signal delivery

The SPM delivers asserted signals to a Secure Partition immediately, causing a return from a blocking call to `psa_wait()`, except in the following situations:

- If the Secure Partition has supplied a signal mask when calling `psa_wait()`, only the requested subset of signals is considered for unblocking the Secure Partition.
- The SPM can postpone delivery of an RoT Service signal until the SPM has the necessary resources available, for example, enough memory to store an active message.

Even when there are asserted signals in a Secure Partition, the SPM can give control to the NSPE first, and then only schedule the Secure Partition when the NSPE is idle.

The SPM must eventually deliver all signals and RoT Service messages.

3.2.4 Scheduling Secure Partitions

The PSA Firmware Framework allows situations in which multiple clients have sent requests to one or more RoT Services, and in which one or more interrupt signals are asserted, all at the same time. In such situation the SPM decides how to execute all Secure Partitions that have signals to process.

The SPM must meet the following requirements:

- Deliver all RoT Service messages to the target Secure Partition.
- Execute a Secure Partition which has one or more signals to service.

The SPM has flexibility in how it meets these requirements. This flexibility allows a simple SPM implementation for more constrained devices, and a more complex, preemptively scheduled SPM on systems that support it.

An SPM implementation is not required to support fair scheduling of Secure Partitions.

Scheduling rules

The scheduling state of a Secure Partition is one of the following:

- A Secure Partition is *blocked* if it is waiting on an API call that requires an external event before completing. API calls that block a Secure Partition are `psa_connect()`, `psa_call()`, `psa_close()` and `psa_wait()`.
- A Secure Partition is *running* if it is currently executing code on the processor.
- A Secure Partition is *ready to run* if it is not *blocked* or *running*.

A *blocked* Secure Partition becomes *ready to run* in the following situations:

Blocking call	**Transitions to *ready to run* when**
`psa_wait()`	One of the filtered signals is asserted.
`psa_connect()`	The RoT Service completes the connection request.
`psa_call()`	The RoT Service completes the request.
`psa_close()`	The RoT Service completes the disconnection request.

A *running* Secure Partition becomes *ready to run* if it is preempted by an interrupt.

A *running* Secure Partition becomes *blocked* if it makes a blocking call.

The following scheduling requirements must be met by all SPM implementations:

- If the NSPE is idle and there are one or more Secure Partitions *ready to run*, the SPM must execute one of the Secure Partitions.
- When the execution of a Secure Partition is preempted then its execution context must be stored so that it can be resumed when the Secure Partition is next scheduled. The context can be saved and restored by software or hardware.

Scheduling recommendations

In more complex systems, Arm recommends that the following scheduling priority rules are followed by the SPM implementation when multiple Secure Partitions are *ready to run*. The rules are applied in sequence until one Secure Partition is selected for execution:

1. A Secure Partition with an asserted interrupt signal is given higher priority for execution than other Secure Partitions which are *ready to run*.
2. A Secure Partition with a high priority asserted interrupt signal is given higher priority for execution than a Secure Partition with a low priority asserted interrupt signal.
3. Priority for execution is based on the Secure Partition priority requested in the manifest.
4. The most recently preempted Secure Partition is given a higher priority than other Secure Partitions.

When a `psa_connect()`, `psa_call()` or `psa_close()` request is sent to an RoT Service, either:

- The SPM immediately executes the Secure Partition that implements the RoT Service.
- The SPM queues the request for the RoT Service, marks the RoT Service's Secure Partition as *ready to run*, and then reschedules the Secure Partition once the client task is *blocked*.

3.3 RoT Services

Root of Trust Services (RoT Services) that are implemented within a Secure Partition will typically follow the structure shown in Figure 5.

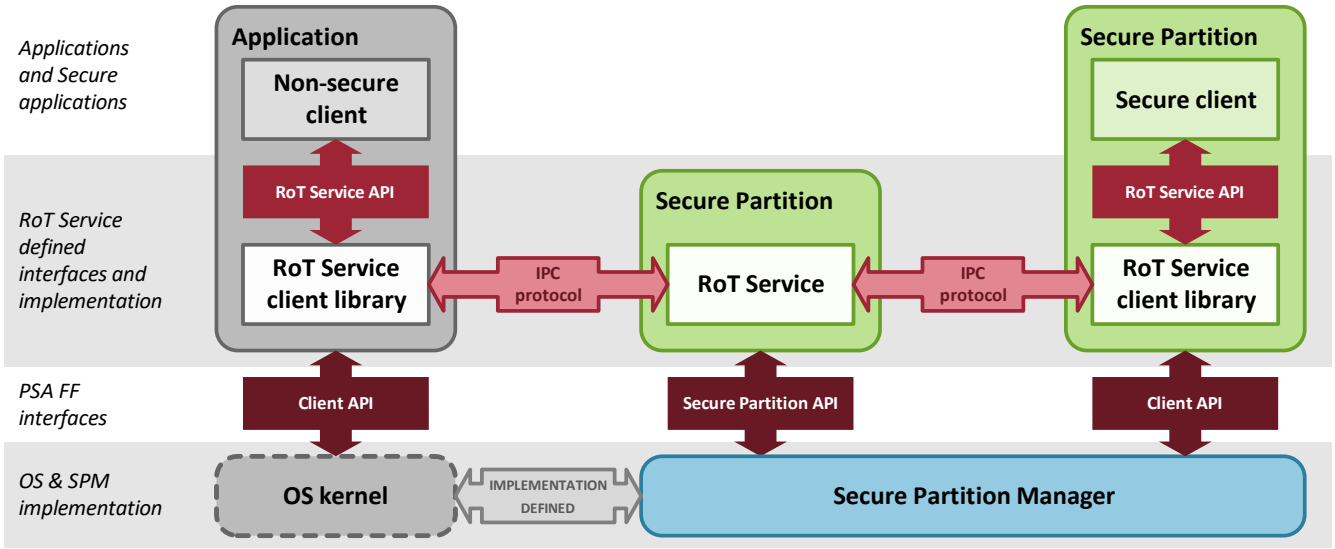


Figure 5 Example RoT Service architecture

Client firmware uses a secure IPC connection to access the RoT Service within the Secure Partition. Clients of the service are either in a Secure Partition or in the NSPE if permitted by the RoT Service declaration in the manifest files.

The RoT Service developer defines an IPC protocol, which is the message structure and sequence, used between the client and server to provide the service.

Arm recommends that the RoT Service developer also defines an RoT Service API and implementation to encapsulate the use of the IPC protocol, and improve the usability of the service for client firmware.

The secure IPC framework is provided by the SPM implementation, which must implement the Client API and Secure Partition API specified in this manual. See *Programming API* on page 48.

SPM implementation note:

The Client API can be used from tasks within the NSPE as well as from Secure Partitions.

The implementation of the Client API functions can be different in each case. For example:

- The implementation of the Client API used by Non-secure clients is provided as a library that works with the Non-secure RTOS to send requests via the SPM.
- The implementation of the Client API used by Secure Partition clients directly uses SPM services.

3.3.1 Defining RoT Services

Some of the properties of an RoT Service must be declared in the Secure Partition manifest file. The properties define the identity, version and authorized access for the RoT Service.

RoT Service identification

An RoT Service is identified by its *RoT Service ID* (SID). An SID is a 32-bit number which is associated with a symbolic name in the Secure Partition manifest. The SID is made available to the client source code via compilation pre-processor symbols. The location and name of the pre-processor symbols is defined in *services* on page 52..

It is recommended that source code and manifest references to SIDs use the symbolic name as defined via the SPM manifest files. Use of the symbolic name will ensure that the RoT Service is correctly identified at runtime.

The SID identifies the IPC protocol between the client and the RoT Service, which defines the valid message payload and response formats and the message sequence. It is recommended that RoT Service developers create a client-side library that encapsulates the IPC protocol to match the Secure Partition code, as shown in *Figure 5* on page 30. In such designs, the SID does not need to be visible to users of the RoT Service.

RoT Service developers must allocate a new SID if the IPC protocol used by the associated RoT Service is changed and becomes incompatible with the original protocol. SIDs must be unique in the SPE.

RoT Services can be sourced from multiple developers, so PSA defines an SID numbering convention to help prevent SID collisions between vendors. The SID numbering convention splits the SID into two sub-identifiers, this allows RoT Service developers to use a number range without risking a collision with a different RoT Service. Arm recommends that bits [31:12] uniquely identify the vendor of the RoT Service. The remaining bits [11:0] can be used at the discretion of the vendor.

RoT Services that do not have a vendor-allocated SID must be given a SID in the range for Vendor ID `0x00000`. Vendor IDs `0x00001-0x0000F` are reserved as described in Table 7.

Table 7 SID numbering convention

Vendor ID (20 bits)	Function ID (12 bits)	Description
0x00000	-	Local definitions in manifest
0x00001 - 0x0000E	-	Standard RoT services
0x0000F	-	Non-production services, for example, for testing and validation
0x00010 - 0xFFFFF	-	Globally unique definitions according to a public ledger with a defined process for acquiring a vendor ID

RoT Service versioning

It is expected that an implementation of an RoT Service will evolve over time, either to add or change functionality or to fix defects. Although the client and RoT Service firmware will be built concurrently in many systems, providing explicit version control allows flexibility in firmware design and allows detection of a mismatched client and RoT Service implementation.

When RoT Services are declared in the manifest, they must specify a version number and a version policy. During connection, the client provides a requested version number. The version policy is one of the following:

- **Strict** – this requires an exact match of the requested version and the RoT Service version. This is recommended if the RoT Service does not need to provide compatibility between different versions of client and service.
- **Relaxed** - this permits a requested version that is equal or lower than the RoT Service version. This policy is recommended to support compatible changes in the RoT Service, when the client and RoT Service might be independently updated.

If the version matching fails during connection, the SPM will treat this as a PROGRAMMER ERROR.

If a relaxed version policy is used for a RoT Service, then an incompatible change to the RoT Service can use a new SID to ensure that the client and RoT Service are using a matching IPC protocol.

It is possible to implement more complex client and RoT Service interactions relating to version or compatibility control as part of the specific RoT Service IPC protocol.

A client can use the `psa_version()` function to determine the availability and version of a specific RoT Service in the system.

RoT Service access control

The SID provided in a call to `psa_connect()` must refer to an RoT Service that the caller is authorized to access. If access is not authorized, the SPM will treat this as a PROGRAMMER ERROR.

Authorization of a connection depends on the type of client:

- For a connection from a Secure Partition, the RoT Service must be listed as an external SID in the Partition manifest.
- For a connection from the NSPE, the RoT Service must have permitted non-secure access in the manifest.

See *Manifest definition* on page 49 for details on how these controls can be specified in the manifest.

3.3.2 Using RoT Services

The Client API for the IPC framework is very simple, and comprises three main functions:

- `psa_connect()` establishes an IPC connection.
- `psa_call()` sends requests on an IPC connection.
- `psa_close()` terminates an IPC connection.

The Client API is specified in *Client API* on page 64 and there is an example of IPC client code in *Example of an RoT Service and client* on page 93.

Connections

Before using an RoT Service, the client must first connect to the RoT Service. Then, as soon as a connection is established, the client makes requests to the RoT Service. Finally, the client closes the connection to release any associated resources.

Depending on the RoT Service and the requirements of the client, an RoT Service connection is either used transiently for a short sequence of requests or it can be maintained for a program's lifetime for use whenever required.

Transient connections avoid the need for global data to retain the connection handle and provide better encapsulation. However, in the case of simple RoT Services which have no failure modes, using a long-lived connection ensures that the RoT Service is called with lower latency and without the risk of resource exhaustion, which can occur during connection.

A client connects to an RoT Service using `psa_connect()`. When it is connected, a handle is returned to the client. The client then uses the handle to refer to the connection in subsequent service requests or to close the connection.

Attempting to connect to missing or incompatible RoT Services, or without access permission is a PROGRAMMER ERROR. If runtime discovery of services is necessary, `psa_version()` will return the availability and version of an RoT Service.

If resources cannot be allocated for the connection or the RoT Service cannot otherwise complete the connection, an error is returned.

Once a client has finished using an RoT Service it must use the `psa_close()` function to release any Secure Partition and SPM resources for the connection.

Handles returned by `psa_connect()` are only valid in the same client until it is used in a call to `psa_close()`. See *Handles* on page 41 for further details about the use of handles.

SPM implementation note:

It is recommended that the SPM allocates all necessary resources for delivering messages during the creation of the connection. This ensures that `psa_call()` does not fail due to resource exhaustion in the SPM, and that any errors returned from `psa_call()` are from the RoT Service itself.

Requesting services

As soon as a connection is successfully established, a client sends a request to an RoT Service by using the connection handle with the `psa_call()` function.

The `psa_call()` function takes a parameter to indicate the type of request, and an array of both input memory regions and output memory regions. The input memory regions are used to provide any necessary parameters and data for the operation. The output memory regions provide space for the RoT Service to return data to the caller.

The use of each individual input or output parameter is governed by the IPC protocol that is defined by the RoT Service. The API allows up to four separate memory buffers to be used as parameters. An RoT Service client

library can pass memory buffers provided by its caller directly to the RoT Service, at the same time as adding any operation information in separate parameters.

`psa_call()` is a blocking operation and it will only return once the operation has completed and all outputs have been provided by the RoT Service. The array of output memory regions is updated by the call to report how much output data was returned in each parameter. In addition to the output parameters, the return value from `psa_call()` can be used to provide an error code or a simple status code from the RoT Service.

The SPM will terminate the connection if a PROGRAMMER ERROR occurs, which can also cause the client to be panicked. See *Abnormal connection termination* on page 39 and *Panics* on page 46.

A client can only make one single request at a time on a connection because the request blocks the client.

Service request parameters

The following rules apply to the input and output parameters provided to an RoT Service by `psa_call()`:

- Memory regions passed to `psa_call()` can be accessed by the SPM at any time until the call returns. After the call returns, the SPM must not access the memory regions and the client can safely reuse the memory. This makes it safe for a client library to use the execution stack for parameters to `psa_call()`.
- The memory regions, and the arrays describing them, must all lie in memory that is accessible to the caller. The SPM must check all memory references and treat any failure as a PROGRAMMER ERROR. See *Memory references* on page 41.

Overlapping of the request parameters, when a single memory location is referenced by more than one parameter, can lead to different results, depending on the system and SPM design. Table 8 describes the permitted behavior when a single memory location is read and/or written via more than one parameter in a single invocation of `psa_call()`.

Table 8 Behavior of overlapped parameters to `psa_call()`

1st access	2nd access	Result	Comments
Read	Read	Not secure	Reads are inconsistent if there is a concurrent update to the memory location by another agent.
Read	Write	OK	–
Write	Read	IMPLEMENTATION DEFINED	2nd read either returns the original or the modified value.
Write	Write	IMPLEMENTATION DEFINED	The final value is either the 1st or the 2nd value written.

These behaviors lead to some rules and recommendations regarding the use of overlapped parameters by RoT Services:

- Clients must never overlap input parameters because of the risk of a double-fetch inconsistency. Implementations are permitted to treat overlapping input parameters as a PROGRAMMER ERROR.
- If an RoT Service allows arbitrary overlap between an input and output parameter, the result will be IMPLEMENTATION DEFINED in most cases.

There are some specific situations in which an input and an output parameter overlap, which are safe. For example:

- When the service reads the input parameter, performs any processing and finally, writes the output parameter.

- When the input and output parameters refer to the same buffer, and the service never writes more data than it has read.

SPM implementation note:

The SPM has some flexibility in when the data is copied out of the client's input memory regions. This must occur after `psa_call()` is invoked and before the data is copied to the Secure Partition by `psa_read()`.

One possible approach is to do this on demand when `psa_read()` is called. However, in distributed systems it can be beneficial to marshal all of the client data across a physical IPC boundary at the point of `psa_call()` and then only transfer it into the Secure Partition when `psa_read()` is called.

Similarly, the SPM has flexibility in when data is copied back into the client's output memory regions. The data in the Secure Partition buffer must be copied out of the Secure Partition's memory during the call to `psa_write()`, but it does not need to be copied into the client memory until the call to `psa_call()` returns.

3.3.3 Processing RoT Service messages

The RoT Service side of the IPC framework is provided by the Secure Partition API. It comprises six functions in the following three groups:

1. `psa_get()` and `psa_reply()` retrieve and complete a single message for an RoT Service.
2. `psa_read()`, `psa_write()` and `psa_skip()` allow the RoT Service to obtain message parameters from the client and send response data back.
3. `psa_set_rhandle()` is a support function that allows the RoT Service to associate some RoT Service-side state with a connection.

The Secure Partition API is specified in *Secure Partition API* on page 70 and there is an example of IPC server code in *Example of an RoT Service and client* on page 93.

Message types

The SPM delivers IPC messages to a Secure Partition in order to manage the lifecycle of a connection from a client and to process individual requests. The three types of message are shown in Table 9:

Table 9 IPC message types

Message type	Message type value	Description
Connection	`PSA_IPC_CONNECT`	Sent by the `psa_connect()` function to establish a new connection.
Request	`>= 0`	Sent by the `psa_call()` function to make a request. The value is provided by the caller to `psa_call()`.
Disconnection	`PSA_IPC_DISCONNECT`	Sent by the `psa_close()` function to drop a connection.

An individual connection always starts with a *connection message*. If the connection is accepted, this is followed by zero or more *request messages* and finally, a *disconnection message* will end the sequence of messages on the connection.

Message delivery

The IPC messages are logically queued by the SPM for each RoT Service. When an RoT Service has an outstanding message, the SPM will assert the Secure Partition signal for the RoT Service that is defined in the manifest.

After the Secure Partition detects the RoT Service signal from a call to `psa_wait()`, it uses `psa_get()` to retrieve the message. This populates a `psa_msg_t` structure with the following information:

Table 10 Members of the psa_msg_t data structure

Member	Description
`type`	One of the message type values described in *Message types* on page 35.
`handle`	A handle to the message, required for further message processing. General rules about handles are provided in *Handles* on page 41.
`client_id`	The identity of the client that sent the message. See *Client identification* on page 38.
`rhandle`	A value that can be used by the RoT Service to associate resources with this connection. See *Managing connection resources* on page 38.
`in_size[]`	An array of values that report the sizes of the input parameters to a `psa_call()` request.
`out_size[]`	An array of values that report the sizes of the output parameters to a `psa_call()` request.

Sometimes the SPM is unable to deliver the message at the time that `psa_get()` is called. For example, this might be due to temporary resource exhaustion in the SPM, or because the SPM validates the message parameters when `psa_get()` is called. The return value from `psa_get()` indicates whether the message was delivered or not.

Each message can only be retrieved once: after a successful call to `psa_get()`, the message is delivered to the Secure Partition and is no longer in the message queue for the RoT Service.

If there was only one message queued for the RoT Service, then calling `psa_get()` will cause the RoT Service signal to be cleared. Otherwise, the RoT Service signal will remain asserted to indicate that there are more messages queued for the RoT Service.

The action required by the RoT Service for each type of message is described in the following sections:

- *Connection messages* below.
- *Request messages* below.
- *Disconnection messages* on page 37.

Connection messages

An RoT Service receives a *connection message* in response to a client's call to `psa_connect()`. A *connection message* is identified by a message type value of `PSA_IPC_CONNECT`. The SPM will have verified that the requested version matches the RoT Service version policy declared in the manifest.

The RoT Service optionally implements additional authorization checks based on the client ID provided in the message. This is described in *Client identification* on page 38.

The RoT Service optionally allocates or reserves Secure Partition resources for the connection. The SPM maintains a *reverse handle* for each connection which the RoT Service can use to remember the association between the connection and the allocated resources. This is described in *Managing connection resources* on page 38.

The RoT Service uses `psa_reply()` to either accept the connection, and expect further messages from the connection, or reject the connection and receive no new messages for the connection.

Request messages

Every time a client invokes `psa_call()` on a connection, a *request message* is sent to the RoT Service. A *request message* is identified by a message type value which is positive. For these messages, the `psa_msg_t` structure

provides the RoT Service with information about the client and connection, the client supplied message type, and the sizes of all of the client supplied input and output parameters.

The typical steps on receipt of a *request message* are:

1. Identify the operation being requested using the `psa_msg_t::type` value.
2. Based on the requested operation, determine what other parameters must be read from the client and read them using .
3. Carry out the requested operation.
4. Copy any response data back to the client output parameter buffers.
5. Complete the message using `psa_reply()`, and provide the client with a status code.

The RoT Service validates the client parameters and enforces correct usage of the IPC protocol. See *Errors within an RoT Service* on page 39 for recommendations on what to validate and how to report an error.

The client parameters are not directly accessible by the RoT Service. Instead, the input parameters must be copied into the Secure Partition's private memory using `psa_read()`, and the responses are copied back to the output parameters using `psa_write()`.

Parameters are either for input or for output, but never both.

`psa_read()` and `psa_write()` enforce a read/write-once policy on the client parameter data. There is no way to rewind the positions of the `psa_read()` and `psa_write()` operations.

Request parameters can safely overlap in some circumstances. See *Service request parameters* on page 34 for details.

For parameters of a known size, for example, small data structures, the Secure Partition can copy the parameter to memory allocated on its stack. For larger data structures, it might be more appropriate to allocate memory from a heap to copy the parameter data. For variable-length data buffers, the Secure Partition can use heap allocation as well, or use the streaming behavior of the `psa_read()` and `psa_write()` functions. When the client data can be processed by the Secure Partition sequentially, the Secure Partition can allocate a smaller, fixed-size buffer and read the input parameter block by block. This block by block also approach works for response data being copied to the client using `psa_write()`.

If the total bytes copied by calls to `psa_write()` are fewer than the parameter size reported in the corresponding `out_size` member of the `psa_msg_t` object, then the corresponding `psa_outvec` object in the client is updated by the SPM to report the actual number of bytes written by the RoT Service. This exact reporting simplifies returning variable-sized data to a client.

Disconnection messages

The *disconnection message* indicates to the RoT Service that the connection is being terminated and no further messages will be received for this connection. A *disconnection message* is identified by a message type value of `PSA_IPC_DISCONNECT`.

A *disconnection message* can be received for any of the following reasons:

- The client terminates the connection by calling `psa_close()`.
- The RoT Service terminates the connection by calling `psa_reply()` with a status code of `PSA_ERROR_PROGRAMMER_ERROR` on a previous *request message*.
- The SPM terminates the connection if the client provides invalid parameters to `psa_call()`.

The RoT Service must release resources associated with the client connection before calling `psa_reply()` to complete the disconnection process.

SPM implementation note:

The SPM must ignore the status code in the call to `psa_reply()` for a *disconnection message*.

The client call to `psa_close()` must not return until after the RoT Service has completed processing of the *disconnection message*. This guarantees that any RoT Service resources used by the connection are immediately available for use by other connections.

Client identification

The RoT Service identifies the client for a connection using the `client_id` member of the `psa_msg_t` object returned by the call to `psa_get()`. `client_id` is a 32-bit signed integer.

The binding of a client to a specific `client_id` is protected by the SPM. The binding is also persistent, that is, a specific client has the same `client_id` following a reboot or a firmware update. These properties allow an RoT Service to use `client_id` for client authentication and to authorize access to the transient or persistent resources that it manages.

A positive `client_id` indicates that the client is in the SPE and this is the Secure Partition ID of the client. See *Secure Partition identity* on page 26 for additional details.

A negative `client_id` indicates that the client is in the NSPE.

SPM implementation note:

An NSPE `client_id` is provided by the NSPE OS via the SPM or directly by the SPM.

In systems in which the NSPE OS enforces non-secure task isolation, it can be useful for the NSPE to provide distinct `client_id` values for each non-secure task. This allows the non-secure tasks to benefit from `client_id`-based access control provided by an RoT Service. This also ensures that the isolation of the non-secure tasks extends to any SPE resources used by those tasks.

In implementations in which an NSPE `client_id` is provided by the NSPE:

- The NSPE operating system must provide the `client_id` for each connection.
- The SPM must verify that the provided `client_id` is an NSPE `client_id`.

In implementations in which NSPE `client_id` values are provided by the SPM, the same negative `client_id` must be used for all connections.

Managing connection resources

Some RoT Services provide functionality which has no per-connection state, for example, a service to report a secure time value. Other RoT Services will need to maintain some internal protected state for a connection that is used by subsequent requests on the connection. For example, a secure signature function could maintain the hash state over several calls that provide additional data before creating and returning the signature.

When the RoT Service can only support a single connection at a time, it can statically allocate the private state and use a private variable to determine whether it is currently in use by a connection or available for a new connection. *Example of an RoT Service and client* on page 93 illustrates this.

More flexibly, the RoT Service can allocate the connection state and then associate the allocated state with the connection using the *reverse handle* provided by the IPC framework. The RoT Service can bind an arbitrary 32-bit value to the connection, for example an object pointer or array index, by using `psa_set_rhandle()` while processing a *connection* or *request message*. The reverse handle value is always provided in the `psa_msg_t` data for subsequent messages on the connection. The connection's reverse handle can be updated later, if required.

Once the connection state is no longer required, for example, when a *disconnection message* is received, the connection state can be deallocated.

Errors within an RoT Service

Some programming errors can be unambiguously detected by the SPM implementation. Other errors can only be detected while executing an RoT Service. For example, an invariant check within the Secure Partition might detect a programming error, a fault or the activity of an attacker. It is recommended that the Secure Partition reports such unrecoverable errors to the SPM by calling `psa_panic()`, which will terminate execution of that Secure Partition.

Each RoT Service defines an IPC protocol that specifies the structure, sequence and semantics of the messages and responses for that RoT Service. Arm recommends that the RoT Service enforce the IPC protocol and treat invalid messages as a PROGRAMMER ERROR. Some examples of message validity checks that can be performed include the following:

- The specific operation requested by the client is known to the RoT Service.
- The specific operation requested by the client is appropriate for the current state of the RoT Service and the connection.
- The input vectors have valid sizes for the specific operation.
- The additional message data has valid format and values for the specific operation requested.
- The additional message data will not overflow any internal buffers used to process the operation.
- The output vectors are correctly sized for the specific operation.

The RoT Service indicates such an error to the SPM by completing the message with a call to `psa_reply()`, using the status code `PSA_ERROR_PROGRAMMER_ERROR`. See *Abnormal connection termination* below.

Abnormal connection termination

A connection is terminated normally when the client calls `psa_close()`.

A connection can also be terminated abnormally by the SPM or the RoT Service. This happens if a PROGRAMMER ERROR in the client is detected by the SPM or the RoT Service, while processing a client call to `psa_call()`. See *Programmer error* on page 45 for a full definition of PROGRAMMER ERROR and how an RoT service can cause the termination of a connection.

In some cases, the abnormal termination of the connection will cause the client to be panicked, which may in turn cause the system to be restarted. See *Panics* on page 46.

If the SPM does not restart the system in response to the PROGRAMMER ERROR, then termination of the connection has the following effects:

- No further *request messages* will be received by the RoT Service for the connection.
- The RoT Service will receive a *disconnection message* for the connection to release resources and reset state associated with the connection.
- The failing call to `psa_call()` will return `PSA_ERROR_PROGRAMMER_ERROR`.
- Subsequent calls to `psa_call()` on the same connection will immediately return `PSA_ERROR_PROGRAMMER_ERROR`.
- The client must call `psa_close()` to close the connection.

SPM implementation note:
The client call to `psa_close()` must not return until after the RoT Service has completed processing of the *disconnection message*, as is required for normal connection termination.

Arm recommends that the SPM waits until the RoT Service has completed processing of the disconnection message before returning from the failing call to `psa_call()` in the client. This approach is shown in *Connection state model* on page 85.

Concurrent connections

A single RoT Service can manage multiple concurrent connections, if it has the necessary resource. Each connection is separately maintained by the SPM and it can have a distinct reverse handle to identify the RoT Service state belonging to that connection.

Each connection can be used to send independent messages to the RoT Service. For example, in a system that provides the preemptive scheduling of tasks, these messages can be queued while a Secure Partition processes another request.

Similarly, two RoT Services within the same Secure Partition might have messages that are concurrently queued.

Summary of message datatypes and functions

Table 11 provides a summary of the members of `psa_msg_t` depending on the message type and Table 12 provides a summary of the message processing functions in the Secure Partition API that can be used with each type of message.

Table 11 Summary of `psa_msg_t` fields by message type

Member	Connection message	Request message	Disconnection message
`type`	`PSA_IPC_CONNECT`	`>= 0`	`PSA_IPC_DISCONNECT`
`handle`	valid	valid	valid
`client_id`	valid	valid	valid
`rhandle`	`NULL`	as set by RoT Service (a)	as set by RoT Service (a)
`in_size[]`	- (b)	sizes of `in_vec[]` regions	- (b)
`out_size[]`	- (b)	sizes of `out_vec[]` regions	- (b)

(a) After `psa_set_rhandle()` is called on a message for a connection, all future messages on that connection will return the provided `rhandle` value.

(b) These values are undefined for *connection* and *disconnection messages*.

Table 12 Summary of Secure Partition API functions by message type

API	Connection message	Request message	Disconnection message
`psa_read()`	- (a)	read input parameter	- (a)
`psa_write()`	- (a)	write output parameter	- (a)
`psa_skip()`	- (a)	skip input parameter	- (a)
`psa_set_rhandle()`	set connection resource	set connection resource	- (b)
`psa_reply()`	accept/refuse connection	complete request	complete disconnection

(a) These operations are invalid on a *connection* or *disconnection message* and calling them is a PROGRAMMER ERROR.

(b) Setting the `rhandle` for a connection during disconnection has no observable effect.

3.3.4 Handles

The `psa_connect()` and `psa_get()` functions return handles to the caller. These handles are a local representation of an object that resides in the SPM. Handles are used:

- In the client to refer to a connection to an RoT Service.
- In the Secure Partition to refer to a message that has been received.

Handles are only valid for the caller and object type that they are created for. They must not be passed or shared with other components.

Handles are SPM-defined numerical values which the SPM uses to identify a specific entity. The combination of the Secure Partition ID, the object type and the handle uniquely identify a specific object in the SPM.

SPM implementation note:

The scheme used to allocate handle values is an implementation decision by the SPM.

For example, it is possible that distinct handles in different Secure Partitions have the same handle value and are resolved by the SPM in relation to the containing Partition. Alternatively, the handle values can all be globally distinct within the system and the SPM will verify that the calling Partition is the valid owner of the handle.

Handles are not required to include the type of the object within the handle value. For example, a message handle and a connection handle within the same Secure Partition can have the same numerical value.

A handle is valid only if all the following are true:

- The handle has been returned from `psa_connect()` or `psa_get()` to the same partition that uses it.
- The handle has not been subsequently used in a call to `psa_close()` or `psa_reply()`, respectively.
- The handle refers to the same type of object that is expected by the API.

The SPM must treat any use of an invalid handle as a PROGRAMMER ERROR. See *Error handling* on page 43.

The zero-value *null handle* is special for the following reasons:

- It is not a valid handle and must never be allocated or returned by the SPM.
- `psa_close()` must ignore calls that pass the *null handle*.
- Other PSA functions must treat *null handle* as an invalid handle.

These requirements allow the *null handle* to be assigned to variables used in clients and RoT Services, indicating that there is no current connection or message. The *null handle* is defined as `PSA_NULL_HANDLE`.

SPM implementation note:

Arm recommends that handle values are not reused quickly within a Partition. This improves robustness and resistance to attack. It also prevents exploitation of use-after-free type defects by turning these into detectable failures.

Arm also recommends that handles are not direct object references. This design is harder to protect against an attacker that is attempting to create a forged handle value.

3.3.5 Memory references

The SPM is the only component in the PSA Firmware Framework that is required to access memory belonging to another component as part of the implementation of the PSA Firmware Framework APIs. The SPM must verify that all memory accesses that it makes outside of the SPM maintain the confidentiality and integrity of the protection domains.

A *memory reference* is a start address and an associated size that refers to a region of the processor's memory address space. In this definition a memory reference is shown as `{start, size}`.

A *zero-length memory reference* is one which has the form `{?, 0}`. The size must be zero. The start address of a zero-length memory reference can safely take any value and must be ignored by the implementation. Providing a zero-length memory reference to an API indicates either:

- An absent optional memory parameter.
- A variable memory buffer of size zero.

A non-zero-length memory reference includes all memory addresses in the range `[start, start + (size - 1)]`.

A memory reference cannot cover a region that extends past the end of addressable memory.

A non-zero-length memory reference must refer to a memory region that the API caller can read.

Some API calls also require the region to be writeable to the caller.

See the individual API descriptions in *Programming API* on page 48 for any specific constraints on memory references.

SPM implementation note:

The SPM must verify that each memory reference is valid and must ensure that a verified memory reference cannot be modified before it is used to transfer data. Supplying an invalid memory reference to one of the APIs defined in this specification results in a PROGRAMMER ERROR, see *Programmer error* on page 45.

The SPM does not have to validate memory references immediately after they are provided to the SPM, but it must validate the references before they are used to transfer data. For example, the SPM is allowed to postpone the validation of memory references provided to `psa_call()` until the point at which the message is delivered in response to the RoT Service calling `psa_get()`. If validation fails at this point, the message cannot be delivered, and the SPM must terminate the connection. In this situation, the SPM can respond to `psa_get()` in one of the following ways:

- The SPM returns `PSA_ERROR_DOES_NOT_EXIST` to indicate that the message could not be delivered. The SPM then terminates the client connection as described in *Abnormal connection termination* on page 39.
- The SPM delivers a *disconnection message* instead of the client request and proceeds with connection termination.

3.3.6 RoT Service example

An example RoT Service and client is included in *Example of an RoT Service and client* on page 93. It demonstrates basic cryptographic hashing functionality as a service.

3.4 Secure peripheral drivers

An SPE peripheral, for example a cryptographic accelerator, can be made accessible to a Secure Partition, enabling a secure device driver to be implemented within an isolated Secure Partition. Implementing a device driver in a Secure Partition, instead of as part of the SPM, can result in:

- Improved system robustness and security against failure or exploitation.
- Portability across different SPM implementations.

To create a secure driver, the *Memory Mapped I/O* (MMIO) regions and interrupt bindings are declared in the Secure Partition manifest. See *Manifest definition* on page 49 for details.

Direct Memory Access (DMA) transactions within the SPE must be policed by the PSA Root of Trust.

A Secure Partition always has exclusive access to an MMIO region. Secure Partitions are not allowed to share MMIO regions with other Secure Partitions and MMIO regions are not allowed to overlap.

The Secure Partition must use `psa_wait()` to wait for an interrupt signal. When the Secure Partition detects the asserted signal, it must then manage the peripheral as appropriate for the interrupt before using `psa_eoi()` to tell the SPM that the interrupt has been serviced. The Secure Partition can now return to waiting for an interrupt.

SPM implementation note:

When a secure hardware interrupt is asserted, the SPM:

1. Acknowledges the interrupt and masks the hardware interrupt line.
2. Identifies the Secure Partition which has registered the interrupt in the manifest.
3. Asserts the IRQ signal for the Secure Partition, as defined in the manifest.
4. Schedules the Secure Partition, see *Scheduling Secure Partitions* on page 29.

When the Secure Partition calls `psa_eoi()`, the SPM:

1. Unmasks the hardware interrupt line.
2. Clears the IRQ signal for the Secure Partition.

To interact with other software components, the Secure Partition can also implement an RoT Service and use the Secure Partition doorbell mechanism. Long-running RoT Service requests in the same Secure Partition will adversely affect interrupt response times, so Arm recommends that the Secure Partition developer considers one or more of the following techniques to reduce the time before the interrupt signal status is queried:

- Only provide quick RoT Service operations within the same Secure Partition as a secure interrupt handler, which keeps the time to return to the call to `psa_wait()` bounded.
- Break long-running operations into steps, and check for interrupts between each step by calling `psa_wait()` with the `timeout` parameter set to `PSA_POLL`.
- Use a short RoT Service request to initiate a long-running operation. When the operation completes, the RoT Service uses the client's doorbell to signal the completion. On receipt of the doorbell signal, the client makes another short RoT Service request to retrieve the results of the operation.

Even when using these techniques, the latency involved in scheduling a Secure Partition to handle interrupts might not be acceptable for certain types of peripheral. The SPM can implement its own independent device driver model, which provides improved response latency, however, this comes at the cost of portability and security. The SPM's independent device driver model is not within the scope of this specification.

3.5 Error handling

There are different types of error that might be encountered during execution of the SPM implementation or an RoT Service. Each type of error has different underlying causes and requires a specific response when detected in the SPM or in an RoT Service. Table 13 describes the three types of error:

Table 13 Types of error

Error type	Occurrence	Typical cause
Internal fault	The internal state of the SPM or Secure Partition is invalid.	This can indicate one of the following issues: • A logic error in the implementation. • An attack on the system. • A hardware fault. This should not occur in a correctly functioning system.
Programmer error	The caller or client is using the API or IPC protocol incorrectly.	This can indicate one of the following issues: • A logic error in the calling firmware. • An attempted attack on the interface. This should not occur in a correctly functioning system.
Transient failure	A required logical or physical resource is not available or functioning.	This can indicate one of the following issues: • Resource exhaustion. • Concurrent activity on the device. • Other external factors. This category includes all errors which cannot be avoided by correct use of the API.

Errors that are not detected or are detected but not handled appropriately can result in a cascade of incorrect program execution. These errors are a reliability concern, but also present a security threat if an attacker can influence or exploit the resulting execution in the system. It is important to mitigate these risks using processes and techniques that improve prevention, detection and containment of errors.

The PSA Firmware Framework recommends a robust response to errors for two reasons:

- Implementation errors are a common source of security vulnerability. A robust response helps to force developers to identify and fix the underlying logic defects.
- Detected errors are a common symptom of an attempted attack. A robust response makes exploratory and brute-force attacks more difficult to carry out.

The following sections describe the design and support provided by the PSA Firmware Framework for error handling.

3.5.1 Internal fault

Management

Arm recommends that that the following approach is used for managing Internal faults:

- Detect and eliminate Internal faults during development.
- Detect and contain Internal faults safely and securely in production systems.

Cause

This occurs when the internal state of the component is invalid.

This can be the result of a logic error in the implementation, it might be an indication of an attack on the system, or it might be the result of a hardware fault.

Detection

Detection of Internal faults is a valuable tool during firmware development. Arm recommends that detection of Internal faults is also deployed in production systems, when it does not compromise required performance of the system.

Note that `assert()` is typically removed in production builds, and permanent fault detection must use a different code construction, for example calling `psa_panic()`.

Response

It is recommended that an Internal fault is handled as a critical error and result in restart of the component or the system.

If an RoT Service detects an Internal fault it can use `psa_panic()` to halt execution, and this will cause the SPM to panic the Secure Partition that contains the RoT Service. See *Panics* on page 46.

3.5.2 Programmer error

Programmer errors are particularly important in the PSA Firmware Framework because these are caused by the accidental or deliberate misuse of an API which crosses a protection boundary. Identifying and preventing invalid use of an interface reduces the attack surface of the implementation.

When this term is used elsewhere in this specification for this meaning, it is written as PROGRAMMER ERROR.

Management

Arm recommends that that the following approach is used for managing Programmer errors:

- Detect and eliminate Programmer errors from callers and clients during development.
- Detect and contain Programmer errors within the SPM and RoT Services during development and in production systems.

Cause

This occurs when the caller or client is using the API or IPC protocol incorrectly.

This error is predictable based on the call parameters or the sequence of calls and is the result of logic errors in the calling firmware or indication of an attempted attack on the interface.

Detection

The SPM must verify the parameters as described for each API, see the *Programmer error* sections for each API in *Programming API* on page 48.

Each RoT Service must verify that each request received conforms to the IPC protocol it has defined. Arm recommends that the RoT Service treats an invalid message as a PROGRAMMER ERROR. See *Errors within an RoT Service* on page 39 for examples of validity checks that the RoT Service can implement.

Response

If the SPM detects a PROGRAMMER ERROR in a caller, it must respond as required in the specific API documentation.

If an RoT Service detects a PROGRAMMER ERROR in a client message, it reports this to the SPM by replying to the message with the status code `PSA_ERROR_PROGRAMMER_ERROR`. This status code will cause the SPM to drop the connection, and no further requests will be received by the RoT Service for the connection. See *Abnormal connection termination* on page 39.

If the source of the programmer error is a Secure Partition, the SPM must panic the Secure Partition in response to a PROGRAMMER ERROR. See *Panics* on page 46.

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 an error code to the NSPE task that indicates a programmer error. See *Standard error codes* on page 47 and the *Programmer error* section in *Client API* on page 64 which defines the required error codes for each function.

SPM implementation note:

The SPM cannot directly terminate the execution of an individual NSPE task when a PROGRAMMER ERROR is detected because NSPE tasks are typically managed by NSPE firmware.

Arm recommends the specified errors codes are also sent to any NSPE management firmware. The NSPE management firmware can then decide to pass those error codes back to the calling task, or to use its own functionality for terminating an execution context.

For simple systems, it can be more appropriate for the SPM to restart the entire system when a programmer error is detected in either the SPE or NSPE.

3.5.3 Transient failure

This category covers all errors which cannot be avoided by correct use of the API.

Management

Arm recommends that that the following approach is used for managing Transient failures:

- Detect, report and handle Transient failures at the appropriate level in the system.

Cause

This occurs when a required logical or physical resource is not available or functioning.

This can be due to resource exhaustion, concurrent activity on the device, or external factors. For example, if a radio link could not be established.

Detection

It is important that firmware checks for Transient failure errors after operations that can fail. This provides robust and reliable response to unpredictable system or hardware problems.

Response

Transient failures are typically reported to the caller. This enables a higher-level component to decide about the appropriate way to deal with the error.

The PSA Firmware Framework provides some common status codes for reporting SPM and RoT Service errors. See *Standard error codes* on page 47 and the individual function specification in *Programming API* on page 48. RoT Services can also define their own error codes for responding to *request messages*.

3.5.4 Panics

A *panic* is the abnormal termination of an execution context. Following a panic, no further execution occurs in that context.

The SPM must panic the execution of a Secure Partition in the following situations:

- When the Secure Partition calls `psa_panic()`.
- When the SPM detects a PROGRAMMER ERROR in the Secure Partition.

- When a `psa_call()` made by the Secure Partition is invalid and the receiving RoT Service replies with the status code `PSA_ERROR_PROGRAMMER_ERROR`.

When a Secure Partition panics, the SPE cannot continue normal execution, as defined in this specification. The behavior of the SPM following a Secure Partition panic is IMPLEMENTATION DEFINED – Arm recommends that the SPM causes the system to restart in this situation. This behavior is recommended for several reasons:

- An individual Secure Partition cannot be reset and restarted in isolation.
- A Secure Partition may have state maintained on behalf of clients that will be destroyed when restarting the service. There is no mechanism to re-synchronize the clients.
- It is not possible to determine at the point of panic how much corruption has occurred within the Secure Partition and elsewhere in the SPE.

SPM implementation note:

The following approaches are examples of alternative responses to a Secure Partition panic:

- The SPM securely stores or reports diagnostics related to the panic.
- The SPM halts execution of the system and waits for a secure debug agent to be attached to the device, to allow examination of the faulting firmware.

The SPM can also implement a range of responses depending on the lifecycle state of the device.

3.5.5 Standard error codes

PSA Firmware Framework APIs use the convention that status codes that are negative indicate an error, and zero or positive values indicate success. These are identified in the API by the `psa_status_t` type or functions that return a `psa_handle_t` type.

Status codes `-129` to `-248` are for use by PSA specifications. These codes are defined in the current PSA specification, or are reserved for future PSA specifications. Status codes in this range are used in the following ways:

- Indicating specific operations that the SPM must implement for specific types of message.
- A set of standard error codes that cover failure conditions that are common to many RoT Services.
- RoT Service specific error codes for standard PSA APIs.

Status codes in this range must only be used as defined in a PSA specification.

A PSA Firmware Framework implementation can define error codes in the range -249 to -256 for IMPLEMENTATION DEFINED purposes. For example, these can be used by a PSA Firmware Framework implementation to communicate specific error conditions between the SPM and the implementations of the Client and Secure Partition APIs.

A RoT Service can define error codes in the ranges -1 to -128 and -257 to MIN_INT32 for RoT Service specific error conditions.

Table 14 defines the common error codes and reserved ranges for the PSA Firmware Framework. See the error code macros and function definitions in *Status codes* on page 58 for details on their usage.

Table 14 Standard error codes

Status code name	Value	Condition
Success	`>= 1`	RoT Service specific status code.
PSA_SUCCESS	`0`	General success status code.

RoT Service error	-1 to -128	RoT Service specific error code.
PSA_ERROR_PROGRAMMER_ERROR	-129	The connection was terminated due to PROGRAMMER ERROR.
PSA_ERROR_CONNECTION_REFUSED	-130	The caller is not permitted to connect to the RoT Service.
PSA_ERROR_CONNECTION_BUSY	-131	The caller is unable to connect to the RoT Service.
PSA_ERROR_GENERIC_ERROR	-132	A error that does not correspond to a specific failure cause.
PSA_ERROR_NOT_PERMITTED	-133	The requested action is denied by a policy.
PSA_ERROR_NOT_SUPPORTED	-134	The requested operation or a parameter is not supported.
PSA_ERROR_INVALID_ARGUMENT	-135	The parameters passed to the RoT Service are invalid.
PSA_ERROR_INVALID_HANDLE	-136	A handle parameter is not valid.
PSA_ERROR_BAD_STATE	-137	The requested action is not valid in the current state.
PSA_ERROR_BUFFER_TOO_SMALL	-138	An output buffer parameter is too small.
PSA_ERROR_ALREADY_EXISTS	-139	An identifier or index is already in use.
PSA_ERROR_DOES_NOT_EXIST	-140	An identified resource does not exist.
PSA_ERROR_INSUFFICIENT_MEMORY	-141	There is not enough runtime memory.
PSA_ERROR_INSUFFICIENT_STORAGE	-142	There is not enough persistent storage.
PSA_ERROR_INSUFFICIENT_DATA	-143	A data source has insufficient capacity left.
PSA_ERROR_SERVICE_FAILURE	-144	Failure within the RoT Service.
PSA_ERROR_COMMUNICATION_FAILURE	-145	Communication failure with another service or component.
PSA_ERROR_STORAGE_FAILURE	-146	Storage failure that may have led to data loss.
PSA_ERROR_HARDWARE_FAILURE	-147	General hardware failure.
Reserved	-148 to -248	Reserved for PSA RoT Services.
SPM Implementation error	-249 to -256	Reserved for the SPM implementation.
RoT Service error	<= -257	RoT Service specific error code.

4 Programming API

This section provides the full definition of the following PSA Firmware Framework elements:

- The *Manifest definition* on page 49.
- The *Secure Partition C runtime* on page 55.
- The *Status codes* on page 58.
- The *Client API* on page 64.
- The *Secure Partition API* on page 70.

The following section introduces the *PSA RoT Services* on page 80 and provides references to these API specifications.

Each API has a respective header file that must be provided by the implementation. Some of the APIs are available to both Secure Partitions and NSPE application firmware, and some are only available to Secure Partition firmware. This is summarized in the following table:

Table 15 Standard PSA source header files

Header file name	API	Availability
`<psa/error.h>`	Status and error codes	SPE and NSPE
`<psa/client.h>`	Client API	SPE and NSPE
`<psa/service.h>`	Secure Partition API	SPE
`<psa/crypto.h>`	Cryptography API	SPE and NSPE
`<psa/initial_attestation.h>`	Initial Attestation API	SPE and NSPE
`<psa/internal_trusted_storage.h>`	Internal Trusted Storage API	SPE and NSPE
`<psa/lifecycle.h>`	RoT Lifecycle API	SPE

The PSA Firmware Framework APIs are all defined in the C language. The APIs make use of standard C data types as defined in the ISO C99 specification.

Secure Partition code that is written in C++ must use `extern "C"` when including the API header files to avoid C++ name mangling.

4.1 Manifest definition

Each Secure Partition manifest must be in JSON. The manifest must conform to the JSON schema defined in this specification. See *Secure Partition manifest schema* on page 87.

See *Secure Partition manifest* on page 26 for information about the purpose of Secure Partition manifests.

The manifest information is used to generate the following set of source header files, that are required by the Secure Partition source code files.

Table 16 Manifest-derived PSA source header files

Header file name	API	Note
`<psa_manifest/pid.h>`	Secure Partition IDs	Macro definitions that map from Secure Partition names to Secure Partition IDs.
`<psa_manifest/sid.h>`	RoT Service IDs	Macro definitions derived from manifest files for RoT Service IDs and RoT Service versions.
`<psa_manifest/`*`manifestfilename`*`.h>`	Manifest definitions	A set of source header files, one for each Secure Partition, containing internal definitions for the Secure Partition implementation. Each file name is based on the name of the Secure Partition's manifest file. The name must not collide with other header files.

4.1.1 Manifest attributes

Each manifest attribute has one or more of the following properties:

Table 17 Properties of manifest attributes

Attribute property	Rule
Required	Attribute must always be present in each manifest file.
Optional	Attribute must be supported by the SPM implementation.
Unique	Attribute value is unique within the set of manifest files for the platform.

The manifest attributes in Table 18 are defined for Secure Partitions:

Table 18 Secure Partition manifest attributes

Attribute	Properties
`psa_framework_version`	Required
`name`	Required, Unique
`type`	Required
`description`	Optional
`priority`	Required
`entry_point`	Required, Unique
`stack_size`	Required
`heap_size`	Optional
`services`	Optional, Unique
`dependencies`	Optional
`mmio_regions`	Optional, Unique
`irqs`	Optional, Unique

Each of the attributes is described in detail below.

psa_framework_version

Properties: Required.

This attribute indicates the version of the PSA Firmware Framework specification this manifest conforms to.

This attribute must take the value `[1.0]` for a manifest file written against this version of the specification.

name

Properties: Required, Unique.

A Secure Partition must have a symbolic name for source code to refer to the Secure Partition ID. This preserves source portability if the Secure Partition ID is changed on a different platform. The format of the name must follow the rules of a C preprocessor macro.

The Secure Partition IDs header file, `<psa_manifest/pid.h>`, must include a definition of the Secure Partition ID using the `name` attribute from the manifest and the ID value allocated by the SPM:

```
#define name id-value
```

The name can also be used by build system tools to prevent naming conflicts with data and subroutines outside of the Secure Partition.

See *Secure Partition identity* on page 26 for more information about Secure Partition IDs.

type

Properties: Required.

This attribute indicates whether the Secure Partition is a part of the PSA Root of Trust or is part of the Application Root of Trust.

Type must be assigned one of the following values:

- `"APPLICATION-ROT"`
- `"PSA-ROT"`

description

Properties: Optional.

This attribute contains a human-readable description and comments for the Secure Partition.

priority

Properties: Required.

Each Secure Partition must be assigned one of the following priority groups:

- `"HIGH"`
- `"NORMAL"`
- `"LOW"`

The `priority` attribute is ignored by SPMs that do not implement any priority-based scheduling. Assigning more specific priorities is IMPLEMENTATION DEFINED.

entry_point

Properties: Required, Unique.

This attribute indicates the Secure Partition entry point in the form of a C function symbol. A single entry-point must be provided, and it must have the following signature:

```
void entry_point(void);
```

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

stack_size

Properties: Required.

This attribute indicates the Secure Partition's stack size in bytes.

The size value is represented either as a positive integer or as a hexadecimal string.

heap_size

Properties: Optional.

This attribute indicates the Secure Partition's heap size in bytes.

The size value is represented either as a positive integer or as a hexadecimal string.

If this attribute is specified in the manifest, then the value must be greater than 0.

If this attribute is not specified in the manifest, then the SPM can assume the size is 0.

If the framework does not implement the APIs defined in *Dynamic memory allocation* on page 56, then a manifest which specifies a `heap_size` must produce a build error.

services

Properties: Optional, Unique.

A Secure Partition must declare all the RoT Services that it implements. This includes at least three required attributes for each RoT Service:

- The `sid` attribute is the RoT Service ID value.
- The `name` attribute is a symbolic name for the RoT Service. This is used to generate symbols for the RoT Service ID, version and signal value, which are required in client and RoT Service implementation source code. The name is also used to identify the service in the `dependencies` attribute of other Secure Partition manifest files.
- The `non_secure_clients` attribute is a Boolean to indicate if the RoT Service is accessible to NSPE clients. RoT Services are always accessible to SPE clients.

The RoT Service IDs header file, `<psa_manifest/sid.h>`, must include the following definition of the RoT Service ID using the `name` and `sid` attributes from the manifest:

```
#define name_SID      sid
```

The Secure Partition header file, `<psa_manifest/`*`manifestfilename`*`.h>`, must include the following definition of the RoT Service signal number using the `name`attribute from the manifest:

```
#define name_SIGNAL   VALUE
```

In this definition, `VALUE` is a signal value that is assigned by the SPM build system. A Secure Partition has a maximum of 28 assigned signals. For more information about signals, see *Signals* on page 27.

Each RoT Service can also specify two optional attributes:

- The `version` attribute is a version number for the RoT Service, which is a non-zero positive integer.
- The `version_policy` attribute indicates which version numbers are allowed when connecting to an RoT Service.

A connection will be rejected if the requested version number does not match the policy defined for the RoT Service. The `version_policy` attribute can indicate one of the following policies:

- `"STRICT"`: the SPM only allows connections when `requested_version == version`.
- `"RELAXED"`**:** the SPM only allows connections when `requested_version <= version`.

The `version` and `version_policy` attributes do not need to be specified. If they are not specified in the manifest, the RoT Service will have default attributes of `version=1` and `version_policy="STRICT"`. More complex schemes for handling version compatibility can be directly implemented by the RoT Service after it receives connection requests from clients.

The RoT Service IDs header file, `<psa_manifest/sid.h>`, must include the following definition of the RoT Service version using the `name` and `version` attributes:

```
#define name_VERSION  version
```

The following excerpt shows the attributes for an example RoT Service within the manifest file `psa_example_service.json`:

```
"services": [
{"name": "PSA_EXAMPLE_SERVICE",
sid": "0x123",
 "non_secure_clients": true,
 "version": 1,
 "version_policy": "STRICT",
```

```
 "description": "PSA example service"
}]
```

If the SPM allocates signal value `0x00000010u` to this service, then this service definition will result in the following symbol definitions:

- In `<psa_manifest/sid.h>`:

```
#define PSA_EXAMPLE_SERVICE_SID      0x123u
#define PSA_EXAMPLE_SERVICE_VERSION  1
```

- In `<psa_manifest/psa_example_service.h>`:

```
#define PSA_EXAMPLE_SERVICE_SIGNAL   0x00000010u
```

dependencies

Properties: Optional.

This attribute lists the RoT Services which the Secure Partition code depends on and is authorized to access. The attribute is a list of the RoT Service names.

If access between a client Secure Partition and an RoT Service is not specified in the manifest, then the client is not allowed to connect to the RoT Service.

All referenced RoT Services must be implemented in some other Secure Partition in the SPE. For example, a Partition must have the following line in the manifest if it wants to access an RoT Service called "PSA_CRYPTO_AES":

```
"dependencies": [ "PSA_CRYPTO_AES" ]
```

mmio_regions

Properties: Optional, Unique.

This attribute is a list of MMIO region objects which the Secure Partition needs access to. For example, this is required by a Secure Partition to implement a device driver.

An MMIO region can only be assigned to a single Secure Partition. Secure Partitions are not allowed to share MMIO regions with other Secure Partitions. This exclusive access is also enforced at runtime by an SPM which implements isolation level 3.

An MMIO region is defined as either of the following:

- `numbered_region`
- `named_region`

A numbered region consists of a `base` address and a `size`. The size must be represented either as a positive integer or as a hexadecimal string. The base address must be represented as a hexadecimal string.

A named region consists of a string `name` identifier. The string identifier references an external definition, which is resolved in an IMPLEMENTATION DEFINED manner. This is helpful for implementations that do not duplicate static MMIO information in the manifest files themselves. The string identifiers are not defined in this specification and, as a result, are not portable.

An MMIO region must include a `permission` attribute. The following permissions are available:

- `READ-ONLY`
- `READ-WRITE`

MMIO regions must not overlap.

SPM implementation note:

The mechanism to compare the named regions with the numbered regions is IMPLEMENTATION DEFINED. For example, one of the following approaches can be used:

- An automated tool can parse the `named_region` element, consult a database for the base address and size, and then replace the `named_region` element in the manifest with a `numbered_region`. This allows a separate tool to compare the addresses and sizes directly without indirection.
- An automated tool can parse the `named_region` and `numbered_region` elements, then use this information to insert `assert()` statements into a source file. The source file is expected to have the symbols for the `named_regions` defined. In this case, the conflicts are reported as errors by the compiler.

If the hardware architecture contains different cacheable and shareable memory attributes, then the MMIO regions must be treated as non-cacheable.

irqs

Properties: Optional, Unique.

This attribute is a list of *Interrupt requests* (IRQs) which are assigned to the Secure Partition.

A Secure Partition always has exclusive access to an assigned IRQ. Secure Partitions are not allowed to share IRQs with other Secure Partitions.

Each IRQ specified must provide a `signal` attribute. Thisattribute contains a symbolic name for the signal, used by the SPM to indicate when the interrupt is asserted.

The Secure Partition header file, `<psa_manifest/manifestfilename.h>`, must include a definition of the IRQ signal number using the `signal` attribute from the manifest:

```
#define signal VALUE
```

In this definition, VALUE is a signal number that is assigned by the SPM build system. A Secure Partition has a maximum of 28 assigned signals. For information about signals see *Signals* on page 27.

Each IRQ source is declared using the `source` attribute. This is a string which identifies the interrupt source in an IMPLEMENTATION DEFINED manner. For example, the PSA Firmware Framework implementation can interpret `source` as one of the following:

- A valid IRQ or exception number for the platform.
- An name that identifies an interrupt source defined in an existing platform abstraction layer.

The following excerpt shows two types of declaration with two IRQ sources:

```
"irqs": [
  {
    "source": "17",
    "signal": "RTC"
  },
  {
    "source": "CRYPTOCELL_IRQN",
    "signal": "CRYPTO_INTR"
  }
]
```

If no RoT Service is defined in the `services` attribute, then at least one IRQ must be declared.

4.2 Secure Partition C runtime

The firmware in a Secure Partition does not have access to normal OS services because the isolation architecture limits the access to other components. The Secure Partition can only use services provided directly by the SPM implementation and RoT Services provided via the IPC mechanism.

In addition to the Secure Partition API, the implementation must provide a basic C language runtime and library for Secure Partition firmware. This runtime is a subset of standard C99, subject to the following constraints:

- A full freestanding implementation of the C99 language, excluding:
  - Floating point types.
  - Wide character types.
  - Variable-length arrays.
- Independence of Secure Partition global and static data.
  - Each Secure Partition has its own instance of the C runtime.
  - Writable global and static data in the Secure Partition is not shared with other Secure Partitions.
- The Secure Partition execution entry point does not return, and is defined in the manifest file.
- Implementation of the following standard headers:
  - `<limits.h>`
  - `<stdarg.h>`
  - `<stdbool.h>`
  - `<stddef.h>`
  - `<stdint.h>`
- Implementation of selected functions:
  - `memcmp()`
  - `memcpy()`
  - `memmove()`
  - `memset()`
- The following functions are optional, but if present, they must conform to additional requirements:
  - `assert()` – on failure, execution is halted by calling `psa_panic()`.
  - `malloc()` – allocated memory is set to zero.
  - `free()` – memory is scrubbed before being released.
  - `realloc()` – released memory is scrubbed and allocated memory is set to zero.
  - `printf()` – a limited set of specifiers and output via a secure channel.

The following sections provide details on these requirements for the Secure Partition C runtime.

4.2.1 Global and static variables

A Secure Partition's runtime state is private to the Secure Partition, see *Memory Assets* on page 21. This includes local, static and global variables, and memory allocated from the dynamic heap. An SPM that implements isolation level 3 must also protect the Secure Partition runtime state from direct access by other Secure Partitions, see *Protection domains* and *Mandatory isolation rules* on page 22.

To provide this separation, each Secure Partition must behave as if it is a separate instance of the C runtime. This is the same behavior that is provided for executable applications in a general purpose OS.

The following are some of the implications of these rules:

- In general, two symbols with the same name in different Secure Partitions refer to separate objects.

- If two Secure Partitions define global variables with the same name, the variables will have separate instances allocated in the runtime memory.
- If two Secure Partitions both use a common library of code that defines a global or static variable, each Secure Partition will access a separate instance of that variable.
- If two Secure Partitions both use a common library of code, the SPE is permitted, but is not required, to share a single copy of the *Code* and *Constant data* with both Secure Partitions.

> **SPM implementation note:**
>
> These rules and their implications need to be considered in the approach taken by the SPE build tools to compiling, linking and loading the SPE firmware.

4.2.2 Runtime management

After initialization, Secure Partition code begins execution from the `entry_point` symbol that is defined in the Secure Partition manifest file. This must have the following signature:

```
void entry_point(void);
```

The entry point function must not return.

4.2.3 Dynamic memory allocation

If the SPM implementation supports dynamic memory allocation, then it must implement the following features:

- The `heap_size` manifest attribute, which specifies the size of the dynamic memory allocation region for the Secure Partition.
- The `malloc()`, `free()` and `realloc()` functions from `<stdlib.h>`, which must implement the additional requirements specified below.
- The allocator heap is private to the Secure Partition. Regions allocated by calling `malloc()` in a Secure Partition, must be released by calling `free()` in the same Partition.
- If a heap has been specified in the Secure Partition Manifest, the allocator must be initialized prior to the execution of the Secure Partition entry point.
- It is a PROGRAMMER ERROR to call the allocation functions if `heap_size` has not been specified in the Secure Partition manifest file. Arm recommends that the SPM detects this error at build time.

These additional requirements on the memory allocation functions reduce the risk of defects leading to vulnerabilities in the Secure Partition. Providing this behavior in the implementation provided by the SPM results in more secure and more efficient Secure Partition firmware.

malloc()

The allocated memory is initialized to zero before being returned to the caller.

Initializing the memory reduces the risk in case of accidental use of uninitialized heap memory.

free()

The region of memory being freed must be scrubbed before being returned to the allocation pool. For example, the memory can be overwritten with a constant value, or with data that is required for managing the dynamic allocator.

Scrubbing the memory ensures that confidential data in the memory cannot be disclosed later through a defect: for example, a use after free or a heap buffer over-read. This also meets a requirement in some security standards regarding the removal of secrets from memory.

realloc()

If `realloc()` is also provided, it must have the combined properties of `malloc()` and `free()`:

- Freed memory is scrubbed.
- Newly allocated memory is filled with zeros.

4.2.4 Buffer and string manipulation

The following memory manipulation functions from `<string.h>` must be implemented with standard C99 definitions:

- `memcmp()`
- `memcpy()`
- `memmove()`
- `memset()`

Additional memory and string functions are optional.

Zero-terminated string functions are a risk in security services. If the Secure Partition does need to process text data, consider using length-bounded versions of the string functions or using static analysis tools to identify unsafe code.

4.2.5 Diagnostics and trace

assert()

Arm recommends that the SPM implements `assert()` from `<assert.h>`, which is useful during firmware development.

This macro tests a condition, which on failure issues diagnostics to a trace or error output and then calls `psa_panic()` to terminate the execution of the calling Secure Partition.

printf()

The ability to report firmware status and diagnostics via a trace port or output log is valuable during firmware development. A commonly used tool for this is a form of the `printf()` function that automatically outputs to a channel that the developer can access.

In its standard form this function is complex, requires a large code and data footprint, and carries high risk of implementation or programmer errors that can lead to vulnerabilities.

If an SPM implementation provides the `printf()` function, Arm recommends:

- The function provides a limited set of format specifiers. This includes the most valuable formats, and eliminates the most expensive and risky formats.
- The output channel is secure. This prevents disclosure of confidential information or information that would help an attacker exploit the system.

The following format specifiers must be supported:

Table 19 Supported format specifiers for printf()

Specifier	Type	Output
`%d` or `%i`	Integral, see length modifiers below.	Signed decimal integer.
`%u`	Integral, see length modifiers below.	Unsigned decimal integer.
`%x`	Integral, see length modifiers below.	Hexadecimal integer.

%p	Pointer: `void*`	Hexadecimal pointer address.
%s	Pointer to zero-terminated string: `char*`	String of characters, without the zero terminator.
%%	None.	A single % character.

The integer specifiers expect an argument of type `int`. The following length modifiers must be used before the format specifier character to correctly process arguments of other integral types:

Length modifier	Argument type
none	`int` `int32_t`
`l`	`long int`
`ll`	`long long int` `int64_t`
`z`	`size_t`

The integral formats can optionally be padded to a fixed width by including a padding modifier before the format specifier character:

Padding modifier	Output format
`0`*«n»*	Pad the number with leading zeros, outputting a minimum of *«n»* characters.

On success, `printf()` will return the number of characters output. If any unsupported specifiers or modifiers are encountered, `printf()` will return a negative error code.

SPM implementation note:

The PSA Firmware Framework implementation is responsible for providing an output channel for `printf()`. The implementation must also implement a security policy for the output from `printf()`, to prevent disclosure of confidential data or leakage of information that an attacker can use against the system.

The output channel can be implemented in different ways, depending on what is available in the system. For example, it could be one of the following:

- A UART acting as a console output.
- A trace or debugging hardware channel.
- A RAM buffer than can be inspected by a debugger or extracted by an API.
- Stored in NVM for later extraction.

The security policy can take different forms, for example:

- `printf()` is implemented to have no output in production builds, or if the device is in the *provisioned* lifecycle state.
- The output channel is secure, and requires authorization to access the content.

4.3 Status codes

These are common status and error codes for all SPM and RoT Service APIs. See *Standard error codes* on page 47 for a summary of the SPM and common RoT Service error codes.

The common codes are provided for use by RoT Services to reduce the need for the RoT Service to define a service-specific error code, and to make it simpler to propagate these error conditions through a chain of RoT

Service calls. The PSA Firmware Framework does not require a RoT Service to use any of the common error codes defined here.

The following client API elements must be defined in a header file <psa/error.h>:

```
#define PSA_SUCCESS

#define PSA_ERROR_PROGRAMMER_ERROR
#define PSA_ERROR_CONNECTION_REFUSED
#define PSA_ERROR_CONNECTION_BUSY
#define PSA_ERROR_GENERIC_ERROR
#define PSA_ERROR_NOT_PERMITTED
#define PSA_ERROR_NOT_SUPPORTED
#define PSA_ERROR_INVALID_ARGUMENT
#define PSA_ERROR_INVALID_HANDLE
#define PSA_ERROR_BAD_STATE
#define PSA_ERROR_BUFFER_TOO_SMALL
#define PSA_ERROR_ALREADY_EXISTS
#define PSA_ERROR_DOES_NOT_EXIST
#define PSA_ERROR_INSUFFICIENT_MEMORY
#define PSA_ERROR_INSUFFICIENT_STORAGE
#define PSA_ERROR_INSUFFICIENT_DATA
#define PSA_ERROR_SERVICE_FAILURE
#define PSA_ERROR_COMMUNICATION_FAILURE
#define PSA_ERROR_STORAGE_FAILURE
#define PSA_ERROR_HARDWARE_FAILURE
#define PSA_ERROR_INVALID_SIGNATURE

typedef int32_t psa_status_t;
```

The details of these definitions are provided in sections 4.3.1 to 4.3.2 below. See *Reference header files* on page 90 for a reference version of this header file.

4.3.1 Macros

PSA_SUCCESS

Status code to indicate general success.

```
#define PSA_SUCCESS ((psa_status_t)0)
```

This is a generic return value from calls to `psa_call()` to indicate success of the operation.

A Secure Partition must use this value in a call to `psa_reply()` for a *connection message* when it accepts the connection. This will result in the client receiving a valid connection handle from `psa_connect()`.

A Secure Partition can use this value in a call to `psa_reply()` for a *request message* to indicate success of the operation. Using this macro is optional as the RoT Service can use the status code to report request-specific additional information.

PSA_ERROR_PROGRAMMER_ERROR

Status code that indicates a PROGRAMMER ERROR in the client.

```
#define PSA_ERROR_PROGRAMMER_ERROR ((psa_status_t)-129)
```

When a Secure Partition uses this status code in a call to `psa_reply()`, the SPM will terminate the connection. Replying with this status code is recommended practice if the client has sent an unrecognizable, incorrectly formatted or invalid message to the RoT Service.

If the RoT Service needs to reply with a non-terminating error when it detects an invalid request, the RoT Service can use one of PSA_ERROR_INVALID_ARGUMENT, PSA_ERROR_INVALID_HANDLE, PSA_ERROR_BAD_STATE, PSA_ERROR_BUFFER_TOO_SMALL or PSA_ERROR_NOT_SUPPORTED as appropriate for the error condition.

If PSA_ERROR_PROGRAMMER_ERROR is returned by psa_call():

- All further calls to psa_call() on the same connection will immediately return PSA_ERROR_PROGRAMMER_ERROR.
- The client must close the connection with psa_close().

See *Abnormal connection termination* on page 39.

PSA_ERROR_CONNECTION_REFUSED

Status code that indicates that the caller is not permitted to connect to an RoT Service.

```
#define PSA_ERROR_CONNECTION_REFUSED ((psa_status_t)-130)
```

This status is the return value from psa_connect() if the RoT Service or SPM does not permit a connection from the client.

A Secure Partition can uses this status code in a call to psa_reply() for a *connection message* when it rejects the connection for a non-transient reason, for example, if this call will never succeed for this client. This rejection will result in the client receiving PSA_ERROR_CONNECTION_REFUSED from the call to psa_connect().

If the SPM or RoT Service cannot accept the connection because of a transient error, for example resource exhaustion, it uses PSA_ERROR_CONNECTION_BUSY instead.

PSA_ERROR_CONNECTION_BUSY

Status code that indicates that the caller cannot connect to an RoT Service.

```
#define PSA_ERROR_CONNECTION_BUSY ((psa_status_t)-131)
```

This status is the return value from psa_connect() if the RoT Service or SPM is currently unable to establish a connection.

A Secure Partition uses this status code in a call to psa_reply() for a *connection message* when it rejects the connection for a transient reason. This will result in the client receiving PSA_ERROR_CONNECTION_BUSY from the call to psa_connect().

If the RoT Service will never accept the connection from this client, it uses PSA_ERROR_CONNECTION_REFUSED instead.

PSA_ERROR_GENERIC_ERROR

A status code that indicates an error that does not correspond to any defined failure cause.

```
#define PSA_ERROR_GENERIC_ERROR ((psa_status_t)-132)
```

RoT Services can use this error code if none of the other standard error codes are applicable.

PSA_ERROR_NOT_PERMITTED

A status code that indicates that the requested action is denied by a policy.

```
#define PSA_ERROR_NOT_PERMITTED ((psa_status_t)-133)
```

RoT Services can reply with this error code when the request parameters are recognized as valid and supported, and a policy explicitly denies the requested operation.

If a subset of the parameters of a function call identify a forbidden operation, and another subset of the parameters are not valid or not supported, the RoT Service can reply with PSA_ERROR_NOT_PERMITTED, PSA_ERROR_NOT_SUPPORTED, or PSA_ERROR_INVALID_ARGUMENT.

PSA_ERROR_NOT_SUPPORTED

A status code that indicates that the requested operation or a parameter is not supported.

```
#define PSA_ERROR_NOT_SUPPORTED ((psa_status_t)-134)
```

RoT Services can reply with this error code when a specific operation request is not implemented by the specific instance of the RoT Service.

This error code is recommended for indicating that optional functionality in a standard specification is not implemented by the RoT Service.

If a combination of parameters is recognized and identified as not valid, prefer to return PSA_ERROR_INVALID_ARGUMENT instead.

PSA_ERROR_INVALID_ARGUMENT

A status code that indicates that the parameters passed to the RoT Service are invalid.

```
#define PSA_ERROR_INVALID_ARGUMENT ((psa_status_t)-135)
```

RoT Services can reply with this error any time a request parameter or combination of parameters are recognized as invalid.

The RoT Service can alternatively treat this situation as a PROGRAMMER ERROR and reply with PSA_ERROR_PROGRAMMER_ERROR, which will terminate the connection.

RoT Services should use a more specific error code for certain situations, for example PSA_ERROR_INVALID_HANDLE, PSA_ERROR_DOES_NOT_EXIST, or PSA_ERROR_ALREADY_EXISTS.

PSA_ERROR_INVALID_HANDLE

A status code that indicates that a handle parameter is not valid.

```
#define PSA_ERROR_INVALID_HANDLE ((psa_status_t)-136)
```

RoT Services can reply with this error any time a handle parameter in a request is invalid.

This error code is useful for RoT Services that use service-defined handles to identify resources that it manages on behalf of clients. The RoT Service can alternatively treat this situation as a PROGRAMMER ERROR and reply with PSA_ERROR_PROGRAMMER_ERROR, which will terminate the connection.

An invalid connection handle in the Client API or message handle in the Secure Partition API is always treated as a PROGRAMMER ERROR, and will not return PSA_ERROR_INVALID_HANDLE.

PSA_ERROR_BAD_STATE

A status code that indicates that the requested action cannot be performed in the current state.

```
#define PSA_ERROR_BAD_STATE ((psa_status_t)-137)
```

RoT Services can reply with this error when one an operation is requested out of sequence.

The RoT Service can alternatively treat this situation as a PROGRAMMER ERROR and reply with PSA_ERROR_PROGRAMMER_ERROR, which will terminate the connection.

PSA_ERROR_BUFFER_TOO_SMALL

A status code that indicates that an output buffer parameter is too small.

```
#define PSA_ERROR_BUFFER_TOO_SMALL ((psa_status_t)-138)
```

RoT Services can reply with this error if an output parameter is too small for the requested output data.

If the client can be expected to know the correct size of output parameter before making the request, the RoT Service can alternatively treat this situation as a PROGRAMMER ERROR and reply with `PSA_ERROR_PROGRAMMER_ERROR`, which will terminate the connection.

RoT Services should preferably return this error code only in cases when performing the operation with a larger output buffer would succeed. However, implementations can return `PSA_ERROR_BUFFER_TOO_SMALL` if a request has other invalid or unsupported parameters.

PSA_ERROR_ALREADY_EXISTS

A status code that indicates that an identifier or index is already in use.

```
#define PSA_ERROR_ALREADY_EXISTS ((psa_status_t)-139)
```

RoT Services can reply with this error if a request is attempting to reuse an identifier or a resource index that is already in use or allocated.

A handle or index that is invalid is not treated as in use. For invalid parameters the RoT Service should reply with `PSA_ERROR_PROGRAMMER_ERROR`, `PSA_ERROR_INVALID_HANDLE`, or `PSA_ERROR_INVALID_ARGUMENT`.

PSA_ERROR_DOES_NOT_EXIST

A status code that indicates that an identified resource does not exist.

```
#define PSA_ERROR_DOES_NOT_EXIST ((psa_status_t)-140)
```

RoT Services can reply with this error if a request identifies a resource that has not been created or is not present.

A handle or index that is invalid is not treated as missing. For invalid parameters the RoT Service should reply with `PSA_ERROR_PROGRAMMER_ERROR`, `PSA_ERROR_INVALID_HANDLE`, or `PSA_ERROR_INVALID_ARGUMENT`.

`psa_get()` returns this value if the SPM cannot deliver a message to the Secure Partition following the assertion of the RoT Service signal. See *Processing RoT Service messages* on page 35.

PSA_ERROR_INSUFFICIENT_MEMORY

A status code that indicates that there is not enough runtime memory.

```
#define PSA_ERROR_INSUFFICIENT_MEMORY ((psa_status_t)-141)
```

RoT Services can reply with this error if runtime memory required for the requested operation cannot be allocated.

If the operation involves multiple components, this error can refer to available memory in any of the components.

PSA_ERROR_INSUFFICIENT_STORAGE

A status code that indicates that there is not enough persistent storage.

```
#define PSA_ERROR_INSUFFICIENT_STORAGE ((psa_status_t)-142)
```

RoT Services can reply with this error if the operation involves storing data in non-volatile memory, and when there is insufficient space on the host media.

Operations that do not directly store persistent data can also return this error code if the implementation requires a mandatory log entry for the requested action and the log storage space is full.

PSA_ERROR_INSUFFICIENT_DATA

A status code that indicates that a data source has insufficient capacity left.

```
#define PSA_ERROR_INSUFFICIENT_DATA ((psa_status_t)-143)
```

RoT Services can reply with this error if the operation attempts to extract data from a source which has been exhausted. For example, the source might be a generator for a pseudorandom bit stream.

PSA_ERROR_SERVICE_FAILURE

A status code that indicates an error within the RoT Service.

```
#define PSA_ERROR_SERVICE_FAILURE ((psa_status_t)-144)
```

RoT Services can reply with this error if it unable to operate correctly. For example, if an essential initialization operation failed.

For failures that are related to hardware peripheral errors, the RoT Service can reply with `PSA_ERROR_COMMUNICATION_FAILURE` or `PSA_ERROR_HARDWARE_FAILURE`.

PSA_ERROR_COMMUNICATION_FAILURE

A status code that indicates a communication failure between the RoT Service and another service or component.

```
#define PSA_ERROR_COMMUNICATION_FAILURE ((psa_status_t)-145)
```

RoT Services can reply with this error if there is a fault in the communication between the RoT Service and another service or peripheral used to provide the requested service. A communication failure may be transient or permanent depending on the cause.

Warning: If an RoT Service returns this error, it is undetermined whether the requested action has completed or not.

An RoT Service should return `PSA_SUCCESS`, or other success code, on successful completion whenever possible. However, an RoT Service can return `PSA_ERROR_COMMUNICATION_FAILURE` if the request was completed successfully in an external component but there was a breakdown of communication before this was reported to the RoT Service.

PSA_ERROR_STORAGE_FAILURE

A status code that indicates a storage failure that may have led to data loss.

```
#define PSA_ERROR_STORAGE_FAILURE ((psa_status_t)-146)
```

RoT Services can reply with this error to indicate that some persistent storage is corrupted. A storage failure does not indicate that any data that was previously read is invalid. However, this previously read data may no longer be readable from storage.

It should not be used for:

- Corruption of volatile runtime memory — this is an internal fault.

- A communication error between the RoT Service and the storage hardware — use PSA_ERROR_COMMUNICATION_FAILURE.
- When the storage is in a valid state but is full — use PSA_ERROR_INSUFFICIENT_STORAGE.

RoT Services should only use this error code to report a permanent storage corruption. However, application writers should keep in mind that transient errors while reading the storage may be reported using this error code.

PSA_ERROR_HARDWARE_FAILURE

A status code that indicates that a hardware failure was detected.

```
#define PSA_ERROR_HARDWARE_FAILURE ((psa_status_t)-147)
```

RoT Services can reply with this error to report a general hardware fault. A hardware failure may be transient or permanent depending on the cause.

PSA_ERROR_INVALID_SIGNATURE

A status code that indicates that a signature, MAC or hash is incorrect.

```
#define PSA_ERROR_INVALID_SIGNATURE ((psa_status_t)-149)
```

RoT Services can reply with this error to report when a verification calculation completes successfully, and the value to be verified is incorrect.

4.3.2 Types

psa_status_t

This is a status code and is used as the return type of an RoT Service call.

```
typedef int32_t psa_status_t;
```

A zero or positive value indicates success, the interpretation of the value depends on the specific operation.

A negative integer value indicates an error.

Some of the status code values have special meanings when used to reply to a RoT Service message, see the status code descriptions in the documentation for psa_reply() on page 77.

4.4 Client API

These are the common interface elements that provide the client endpoint of an IPC connection. This API is used for both RoT Service clients operating in the NSPE and when Secure Partition firmware needs to invoke another RoT Service.

The following client API elements must be defined in a header file <psa/client.h>:

```
#define PSA_FRAMEWORK_VERSION
#define PSA_VERSION_NONE
#define PSA_NULL_HANDLE
#define PSA_HANDLE_IS_VALID(handle)
#define PSA_HANDLE_TO_ERROR(handle)
#define PSA_MAX_IOVEC
#define PSA_IPC_CALL

typedef int32_t psa_handle_t;
typedef struct psa_invec psa_invec;
```

```
typedef struct psa_outvec psa_outvec;

uint32_t psa_framework_version(void);
uint32_t psa_version(uint32_t sid);
psa_handle_t psa_connect(uint32_t sid, uint32_t version);
psa_status_t psa_call(psa_handle_t handle, int32_t type,
                      const psa_invec *in_vec, size_t in_len,
                      psa_outvec *out_vec, size_t out_len);
void psa_close(psa_handle_t handle);
```

In addition, `<psa/client.h>` must provide all the API elements from `<psa/error.h>` as defined in *Status codes* on page 58.

The details of these definitions are provided in sections 4.4.1 to 4.4.3 below. See *Reference header files* on page 90 for a reference version of this header file.

Using RoT Services on page 32 provides further information on how these APIs work together.

Some APIs will panic in the case of a PROGRAMMER ERROR. See *Programmer error* on page 45.

4.4.1 Macros

PSA_FRAMEWORK_VERSION

This value reports the version of the PSA Framework API that is being used to build the calling firmware.

```
#define PSA_FRAMEWORK_VERSION (0x0100u)
```

The version value is a 16-bit integer with the major version in bits[15:8] and the minor version in bits[7:0].

An implementation of this version of the specification, v1.0, must report the value `0x0100`.

PSA_VERSION_NONE

This is the return value from `psa_version()` if the requested RoT Service is not present in the system. See `psa_version()` on page 67 for more details.

```
#define PSA_VERSION_NONE (0u)
```

PSA_NULL_HANDLE

This is the zero-value *null handle*.

```
#define PSA_NULL_HANDLE ((psa_handle_t)0)
```

This is an invalid handle and must never be allocated or returned by the SPM.

`psa_close()` must ignore calls that pass the *null handle*, at the same time, other PSA functions must treat it as an invalid handle.

This value can be assigned to variables used in clients and RoT Services, indicating that there is no current connection or message.

PSA_HANDLE_IS_VALID()

This macro tests whether a handle value returned by `psa_connect()` is valid.

```
#define PSA_HANDLE_IS_VALID(handle) ((psa_handle_t)(handle) > 0)
```

This macro can be used to check whether a call to `psa_connect()` was successful.

This test does not detect all possible invalid handles, and is only appropriate for validating the psa_handle_t value returned by psa_connect(). See *Handles* on page 41 for the full definition of a valid handle.

PSA_HANDLE_TO_ERROR()

This macro converts the handle value returned from a failed call psa_connect() into an error code.

```
#define PSA_HANDLE_TO_ERROR(handle) ((psa_status_t)(handle))
```

When psa_connect() fails, the psa_handle_t value that is returned can be interpreted as a status code. This macro can be used to convert the invalid handle value into an error code.

PSA_MAX_IOVEC

This value is the maximum number of input and output vectors for a request to psa_call().

```
#define PSA_MAX_IOVEC (4u)
```

The combined number of input and output vectors to psa_call() cannot exceed four. So, this also individually limits the number of input vectors and the number of output vectors that are reported in psa_msg_t.

PSA_IPC_CALL

This value is an IPC message type that indicates a generic client request.

```
#define PSA_IPC_CALL (0)
```

The type member of the psa_msg_t object returned by psa_get() takes this value for a client request following a call to psa_call() when PSA_IPC_CALL is provided as the type parameter. See *Request messages* on page 36.

4.4.2 Types

psa_handle_t

This type is used for handles, which are indirect references to elements of the PSA Firmware Framework that are within the SPM. This type is used for active connections and for active messages.

```
typedef int32_t psa_handle_t;
```

A zero value indicates the *null handle*, which is always an invalid handle.

A negative value, for example, returned from psa_connect(), indicates an error.

psa_invec

This structure identifies a read-only input memory region provided to an RoT Service. psa_call() is invoked with an array of between zero and four psa_invec objects.

```
typedef struct psa_invec {
    const void *base;
    size_t len;
} psa_invec;
```

The start address of the memory buffer is base and len is its size in bytes. If len is zero, this indicates an empty buffer and base is ignored.

psa_outvec

This structure identifies a writable output memory region provided to an RoT Service. psa_call() is invoked with an array of between zero and four psa_outvec objects.

```
typedef struct psa_outvec {
    void *base;
    size_t len;
} psa_outvec;
```

The start address of the memory buffer is `base` and `len` is its size in bytes. If `len` is zero, this indicates a non-existent buffer and `base` is ignored.

On return from `psa_call()`, the `len` value will update to indicate the number of bytes of data written to the buffer by the RoT Service.

4.4.3 Functions

psa_framework_version()

This function retrieves the version of the PSA Framework API that is implemented.

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

Parameters

`void`	None

Return

`uint32_t version`	The version of the PSA Framework implementation that is providing the runtime services to the caller. The major and minor version are encoded as follows: `version[15:8]` – major version number `version[7:0]` – minor version number

Usage

An implementation of this version of the specification, v1.0, must return the value `0x0100`.

> **SPM implementation note:**
>
> For systems in which all SPE and NSPE firmware is linked together this function may be implemented as an inline constant expression that returns `PSA_FRAMEWORK_VERSION`.

psa_version()

This function retrieves the version of an RoT Service or indicates that an RoT Service is not present on the system.

```
uint32_t psa_version(uint32_t sid);
```

Parameters

`uint32_t sid`	ID of the RoT Service to query.

Return

`PSA_VERSION_NONE`	The RoT Service is not implemented, or the caller is not allowed to access the service.
`> 0`	The version of the implemented RoT Service.

Usage

If the RoT Service is not implemented or if the caller is not authorized to access it, this call will return `PSA_VERSION_NONE`.

psa_connect()

This function connects to an RoT Service by its SID.

```
psa_handle_t psa_connect(uint32_t sid, uint32_t version);
```

Parameters

`uint32_t sid`	ID of the RoT Service to connect to.
`uint32_t version`	Requested version of the RoT Service.

Return

`> 0`	A handle for the connection.
`PSA_ERROR_CONNECTION_REFUSED`	The SPM or RoT Service has refused the connection.
`PSA_ERROR_CONNECTION_BUSY`	The SPM or RoT Service cannot make the connection now.

Programmer error

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

- The RoT Service ID is not present.
- The RoT Service version is not supported, see *RoT Service versioning* on page 32.
- The caller is not allowed to access the RoT Service, see *RoT Service access control* on page 32.

A PROGRAMMER ERROR will result in one of the following behaviors:

- Panic the caller.
- Fail the call with error code `PSA_ERROR_CONNECTION_REFUSED`.

If the caller is a Secure Partition, a PROGRAMMER ERROR must result in a panic. If the caller is in the NSPE, it is IMPLEMENTATION DEFINED whether a PROGRAMMER ERROR will panic or return `PSA_ERROR_CONNECTION_REFUSED`.

Usage

A client connects to an RoT Service using `psa_connect()`. When it is connected, a handle is returned to the client, the client must use the handle to refer to the connection in subsequent calls to `psa_call()` and `psa_close()`.

The connection can be refused by the SPM or RoT Service and will return `PSA_ERROR_CONNECTION_REFUSED` or `PSA_ERROR_CONNECTION_BUSY` errors in some situations. For example:

- The SPM has reached the limit of concurrent connections.
- The RoT Service has reached the limit of concurrent client connections.
- The RoT Service rejected the client because of a service-specific condition.

An error response of `PSA_ERROR_CONNECTION_BUSY` indicates that retrying the connection later might be successful.

The `PSA_HANDLE_IS_VALID()` and `PSA_HANDLE_TO_ERROR()` macros can be used to process the value returned by `psa_connect()`.

When `psa_connect()` is called, the SPM delivers a *connection message* from the client to the Secure Partition that implements the RoT Service. For more information about handling connections and messages see *Processing RoT Service messages* on page 35.

psa_call()

This function calls an RoT Service on an established connection.

```
psa_status_t psa_call(psa_handle_t handle, int32_t type,
                      const psa_invec *in_vec, size_t in_len,
                      psa_outvec *out_vec, size_t out_len);
```

Parameters

`psa_handle_t handle`	A handle to an established connection.
`int32_t type`	The request type. Must be zero or positive.
`const psa_invec *in_vec`	Array of input `psa_invec` structures.
`size_t in_len`	Number of input `psa_invec` structures.
`psa_outvec *out_vec`	Array of output `psa_outvec` structures.
`size_t out_len`	Number of output `psa_outvec` structures.

Return

`>= 0`	RoT Service-specific status code.
`< 0`	RoT Service-specific error code.
`PSA_ERROR_PROGRAMMER_ERROR`	The connection has been terminated by the RoT Service. See the description of PROGRAMMER ERROR that follows.

Programmer error

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

- An invalid `handle` was passed.
- The connection is already handling a request.
- `type < 0`
- An invalid memory reference was provided.
- `in_len + out_len > PSA_MAX_IOVEC`
- The message is unrecognized by the RoT Service or incorrectly formatted.

A PROGRAMMER ERROR will result in one of the following behaviors:

- Panic the caller.
- Fail the call with error code `PSA_ERROR_PROGRAMMER_ERROR`.

If the caller is a Secure Partition, a PROGRAMMER ERROR must result in a panic. If the caller is in the NSPE, it is IMPLEMENTATION DEFINED whether a PROGRAMMER ERROR will panic or return `PSA_ERROR_PROGRAMMER_ERROR`.

If this call returns `PSA_ERROR_PROGRAMMER_ERROR`, when a valid connection handle was provided, then all subsequent calls to `psa_call()` with the same connection handle will immediately return `PSA_ERROR_PROGRAMMER_ERROR`. The client must close the connection using `psa_close()`. See *Abnormal connection termination* on page 39.

Usage

The caller indicates a specific operation using the `type` parameter. This can be any positive value. If the RoT Service only has a single operation, `PSA_IPC_CALL` can be used as the `type`, which has the value zero.

The caller can optionally send additional parameters to the RoT Service in the form of a payload using `in_vec`. The caller can optionally provide one or more buffers to receive a response using `out_vec`.

The input and output buffers are in the form of arrays of I/O vectors. The number of vectors is specified by `in_len` and `out_len`, and there must be four or fewer vectors in total, such that `in_len + out_len <= 4`.

Any I/O vector of length zero is allowed and will be treated as an empty or non-existent vector by the framework.

If `in_len` is zero, then `in_vec` is ignored.

If `out_len` is zero, then `out_vec` is ignored.

As soon as the message completes, the `len` member of each element in the `out_vec` array is updated by the framework to reflect the actual data written by the RoT Service.

Service request parameters on page 34 provides detailed rules about the parameters, for example, what can happen if parameters overlap.

psa_close()

This function closes a connection to an RoT Service. Calling this function sends a *disconnection message* to the RoT Service so it can clean up any resources associated with the connection.

```
void psa_close(psa_handle_t handle);
```

Parameters

`psa_handle_t handle`	A handle to an established connection, or the *null handle*.

Return

void	Success.

Programmer error

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

- An invalid `handle` was provided that is not the *null handle*.
- The connection is currently handling a request. For example, this can happen if an NSPE client is multi-threaded.

A PROGRAMMER ERROR will result in one of the following behaviors:

- Panic the caller.
- Return with no effect.

If the caller is a Secure Partition, a PROGRAMMER ERROR must result in a panic. If the caller is in the NSPE, it is IMPLEMENTATION DEFINED whether a PROGRAMMER ERROR will panic or return.

Usage

This function will only return after the RoT Service has completed processing the disconnection.

This function will have no effect if called with the *null handle*.

4.5 Secure Partition API

The following elements of the framework provide the Secure Partition interface to the SPM for handling inputs and responding to IPC messages.

The following partition API elements must be defined in a header file `<psa/service.h>`:

```
#define PSA_POLL
#define PSA_BLOCK
#define PSA_WAIT_ANY
#define PSA_DOORBELL
```

```
#define PSA_IPC_CONNECT
#define PSA_IPC_DISCONNECT

typedef uint32_t psa_signal_t;
typedef struct psa_msg_t psa_msg_t;

psa_signal_t psa_wait(psa_signal_t signal_mask, uint32_t timeout);
void psa_set_rhandle(psa_handle_t msg_handle, void *rhandle);
psa_status_t psa_get(psa_signal_t signal, psa_msg_t *msg);
size_t psa_read(psa_handle_t msg_handle, uint32_t invec_idx, void *buffer, size_t num_bytes);
size_t psa_skip(psa_handle_t msg_handle, uint32_t invec_idx, size_t num_bytes);
void psa_write(psa_handle_t msg_handle, uint32_t outvec_idx,
               const void *buffer, size_t num_bytes);
void psa_reply(psa_handle_t msg_handle, psa_status_t status);
void psa_notify(int32_t partition_id);
void psa_clear(void);
void psa_eoi(psa_signal_t irq_signal);
void psa_panic(void);
```

In addition, `<psa/service.h>` must provide all the API elements from `<psa/error.h>` and `<psa/client.h>` as defined in *Status codes* on page 58 and *Client API* on page 64, respectively.

The details of these definitions are provided in sections 4.5.1 to 4.5.3 below. See *Reference header files* on page 90 for a reference version of this header file.

Processing RoT Service messages on page 35 provides further information on how these APIs work together.

Some APIs will panic in the case of a PROGRAMMER ERROR. See *Programmer error* on page 45.

4.5.1 Macros

PSA_POLL

A timeout value that requests a polling wait operation.

```
#define PSA_POLL (0x00000000u)
```

`PSA_POLL` is used as the timeout for `psa_wait()` to query the current signal status for the Secure Partition.

PSA_BLOCK

A timeout value that requests a blocking wait operation.

```
#define PSA_BLOCK (0x80000000u)
```

`PSA_BLOCK` is used as the timeout for `psa_wait()` to wait until a Secure Partition signal is asserted before returning.

PSA_WAIT_ANY

A mask value that includes all Secure Partition signals.

```
#define PSA_WAIT_ANY (0xFFFFFFFFu)
```

`PSA_WAIT_ANY` is used as the signal mask for `psa_wait()` when waiting for any signal to be asserted.

PSA_DOORBELL

This value is the signal number for the Secure Partition doorbell.

```
#define PSA_DOORBELL (0x00000008u)
```

PSA_DOORBELL can be used:

- To select the doorbell as a signal to wait for when calling psa_wait().
- To detect a doorbell notification in the signals returned by psa_wait().

PSA_IPC_CONNECT

This value is the IPC message type that indicates a new connection.

```
#define PSA_IPC_CONNECT (-1)
```

The type member of the psa_msg_t object returned by psa_get() takes this value for a new connection request following a call to psa_connect(). See *Connection messages* on page 36.

PSA_IPC_DISCONNECT

This value is the IPC message type that indicates the end of a connection.

```
#define PSA_IPC_DISCONNECT (-2)
```

The type member of the psa_msg_t object returned by psa_get() takes this value when the connection is terminated. See *Disconnection messages* on page 37.

4.5.2 Types

psa_signal_t

This type stores a set of one or more Secure Partition signals.

```
typedef uint32_t psa_signal_t;
```

A set of signals is represented as a bitmask. Each signal is a value with a single one-bit, that is, (1U << n) for 0 <= n <= 31.

psa_msg_t

This structure describes a message received by an RoT Service after calling psa_get().

```
typedef struct psa_msg_t {
    int32_t type;
    psa_handle_t handle;
    int32_t client_id;
    void *rhandle;
    size_t in_size[PSA_MAX_IOVEC];
    size_t out_size[PSA_MAX_IOVEC];
} psa_msg_t;
```

The message type will be one of the following values:

Table 20 Types of message received by an RoT Service

psa_msg_t::type	Message type	Description
PSA_IPC_CONNECT	*Connection*	New connection request sent by the psa_connect() function.
>= 0	*Request*	Client request sent by the psa_call() function.
PSA_IPC_DISCONNECT	*Disconnection*	Notification of disconnection, sent by the psa_close() function, or after a connection is terminated by the RoT Service.

The message `handle` is a reference generated by the SPM to the message returned by `psa_get()`. The handle is used to identify the message in the PSA functions that are used to process RoT Service messages.

The message `client_id` is the Partition ID of the sender of the message. For details of its definition and how it is used, see *Client identification* on page 38.

The message *reverse handle*, `rhandle`, is useful for binding a connection to some application-specific data or function pointer within the RoT Service implementation. The `rhandle` for a connection is `NULL` until `psa_set_rhandle()` is used.

The array `in_size` provides the size of each client input vector in bytes. The array `out_size` provides the size of each client output vector in bytes. It is recommended that an RoT Service checks that `in_size` and `out_size` are valid for this function.

Any input or output vector of length zero is permitted and will be treated as an empty or non-existent vector by the framework. When less than four vectors are provided to `psa_call()` for either input or output, then the remaining vectors have zero length and the `in_size` and `out_size` elements for these vectors will be zero.

4.5.3 Functions

psa_wait()

This function returns the Secure Partition interrupt signals that have been asserted from a subset of signals provided by the caller. Optionally, this call will block until at least one of those signals is asserted.

```
psa_signal_t psa_wait(psa_signal_t signal_mask, uint32_t timeout);
```

Parameters

`psa_signal_t signal_mask`	A set of signals to query. Signals that are not in this set will be ignored.
`uint32_t timeout`	A timeout bitfield value, see below.

Return

`> 0`	At least one signal is asserted.
`0`	No signals are asserted. This case is only seen when a polling timeout is used.

Programmer error

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

- `signal_mask` does not include any assigned signals.

A PROGRAMMER ERROR will panic the caller.

Usage

This call will block until at least one of the requested signals is asserted if the timeout mode is blocking, otherwise it will return the current signal state immediately.

The `timeout` parameter has two bitfields:

Field	Bits	Description
`MODE`	`timeout[31]`	Specifies either blocking (1) or polling (0) operation.
`RES`	`timeout[30:0]`	Reserved for future use.

When `MODE` is one, `psa_wait()` blocks the caller until one of the requested signals is asserted.

When `MODE` is zero, `psa_wait()` returns immediately with the current signal state masked by the requested signal set. The return value will be zero if no signals are asserted.

Callers must set RES to zero. SPM implementations must ignore the value of RES. These requirements ensure compatibility for the intended future use of this field.

The values PSA_BLOCK and PSA_POLL can be used by callers to specify the timeout parameter with the appropriate MODE value. The value PSA_WAIT_ANY can be used by callers to wait for any signal. For example, the following code will block until one or more signals are asserted:

```
psa_signal_t signals;
Signals = psa_wait(PSA_WAIT_ANY, PSA_BLOCK);
```

psa_get()

This function retrieves the message which corresponds to a given RoT Service signal and removes the message from the RoT Service queue.

```
psa_status_t psa_get(psa_signal_t signal, psa_msg_t *msg);
```

Parameters

psa_signal_t signal	The signal value for an asserted RoT Service.
psa_msg_t *msg	Pointer to psa_msg_t object to receive the message.

Return

PSA_SUCCESS	Success, *msg will contain the delivered message.
PSA_ERROR_DOES_NOT_EXIST	The message could not be delivered.

Programmer error

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

- signal has more than a single bit set.
- signal does not correspond to an RoT Service.
- The RoT Service signal is not currently asserted.
- The msg pointer provided is not a valid memory reference.

A PROGRAMMER ERROR will panic the caller.

Usage

After an RoT Service message is signaled, this function is used to retrieve the message details. Each message can only be retrieved once.

If the SPM is unable to deliver a message at the time that psa_get() is called, for example, due to resource availability within the SPM or because of failed validation of the client's parameters, this function must return PSA_ERROR_DOES_NOT_EXIST.

Calling psa_get() when no message has been sent is a PROGRAMMER ERROR. The caller must only call psa_get() after a RoT Service signal is returned by psa_wait(). See *Signals* on page 27.

The state of the signal on return from psa_get() indicates if additional messages have been sent to the RoT Service by other clients. See *Signal delivery* on page 29.

SPM implementation note:

If this function returns PSA_ERROR_DOES_NOT_EXIST when retrieving a valid message, the SPM must reassert the signal later when resource is available to indicate to the Secure Partition that there is a message waiting for the RoT Service.

psa_set_rhandle()

This function associates some RoT Service private data with a client connection.

```
void psa_set_rhandle(psa_handle_t msg_handle, void *rhandle);
```

Parameters

psa_handle_t `msg_handle`	Handle for the client's message.
`void *rhandle`	Reverse handle value to be associated with the connection.

Return

void	Success, `rhandle` will be provided with all subsequent messages delivered on this connection.

Programmer error

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

- `msg_handle` is invalid.

A PROGRAMMER ERROR will panic the caller.

Usage

On success the `rhandle` is retained by the implementation and provided in all future messages on that connection as part of the `psa_msg_t` structure. `rhandle` is private to the calling Secure Partition and the SPM – it cannot be read or modified by any other Secure Partition.

If a Secure Partition calls `psa_set_rhandle()` multiple times on the same connection, `psa_get()` will return the `rhandle` value from the last call.

psa_read()

This function reads a message parameter or part of a message parameter from a client input vector.

```
size_t psa_read(psa_handle_t msg_handle, uint32_t invec_idx, void *buffer, size_t num_bytes);
```

Parameters

psa_handle_t `msg_handle`	Handle for the client's message.
`uint32_t invec_idx`	Index of the input vector to read from. Must be less than `PSA_MAX_IOVEC`.
`void *buffer`	Buffer in the Secure Partition to copy the requested data to.
`size_t num_bytes`	Maximum number of bytes to read from the client input vector.

Return

`> 0`	Number of bytes copied.
`0`	There was no remaining data in this input vector.

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` is equal to, or greater than, `PSA_MAX_IOVEC`.
- The memory reference for `buffer` is invalid or not writable.

A PROGRAMMER ERROR will panic the caller.

Usage

Streams up to the next `num_bytes` bytes of client input vector `invec_idx` in the message identified by `msg_handle` to the Secure Partition `buffer`. Returns the number of bytes copied.

If `num_bytes` is less than, or equal to, the available data in the input vector then `num_bytes` are copied to `buffer`, and the remaining data in the input vector can be read by subsequent calls to `psa_read()` with the same `msg_handle` and `invec_idx`.

If `num_bytes` is greater than the remaining data in the input vector, then the remaining input bytes are copied to `buffer` and the call returns the number of bytes copied. Any space after this in `buffer` is not modified. Subsequent calls of `psa_read()` or `psa_skip()` with the same message input vector will report that there is no more data in the vector.

RoT Services determine how much data is available to read from the message based on the `in_size[]` attribute of the `psa_msg_t` message returned from `psa_get()`. If an input vector has not been provided by the client, then the corresponding `in_size[]` value for that vector is zero.

psa_skip()

This function skips over part of a client input vector.

```
size_t psa_skip(psa_handle_t msg_handle, uint32_t invec_idx, size_t num_bytes);
```

Parameters

psa_handle_t `msg_handle`	Handle for the client's message.
`uint32_t invec_idx`	Index of the input vector to skip from. Must be less than `PSA_MAX_IOVEC`.
`size_t num_bytes`	Maximum number of bytes to skip in the client input vector.

Return

`> 0`	Number of bytes skipped.
`0`	There was no remaining data in this input vector.

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` is equal to or greater than `PSA_MAX_IOVEC`.

A PROGRAMMER ERROR will panic the caller.

Usage

Advances the current read offset by skipping up to `num_bytes` bytes for input vector `invec_idx` in the message identified by `msg_handle`. The function returns the number of bytes skipped.

If `num_bytes` is greater than the remaining size of the input vector then the remaining size of the input vector is returned. Subsequent calls of `psa_read()` or `psa_skip()` with the same message input vector will report that there is no more data in the vector.

Calling `psa_skip()` is equivalent to calling `psa_read()` and then discarding the data that was read.

psa_write()

This function writes a message response to a client output vector.

```
void psa_write(psa_handle_t msg_handle, uint32_t outvec_idx,
               const void *buffer, size_t num_bytes);
```

Parameters

psa_handle_t `msg_handle`	Handle for the client's message.
`uint32_t outvec_idx`	Index of the output vector to write to. Must be less than `PSA_MAX_IOVEC`.
`const void *buffer`	Buffer with the data to write.
`size_t num_bytes`	Number of bytes to write to the client output vector.

Return

void	Success.

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` is equal to, or greater than, `PSA_MAX_IOVEC`.
- The memory reference for `buffer` is invalid.
- The call attempts to write data past the end of the client output vector.

A PROGRAMMER ERROR will panic the caller.

Usage

Appends `num_bytes` of data from `buffer` to the client output vector `outvec_idx` in the message identified by `msg_handle`. Sequential calls using the same `msg_handle` and `outvec_idx` will be concatenated in the output vector.

Attempting to write data beyond the end of the client output vector is a PROGRAMMER ERROR. The initial size of each output vector is provided by the `out_size[]` attribute of the `psa_msg_t` object for the message. If no output vector has been passed by the client, then the corresponding `out_size[]` for that vector is zero.

SPM implementation note:

The total number of bytes written to a single parameter must be reported to the client by updating the `len` member of the `psa_outvec` structure for the parameter before returning from `psa_call()`.

psa_reply()

Completes handling of a specific message and unblocks the client.

```
void psa_reply(psa_handle_t msg_handle, psa_status_t status);
```

Parameters

`psa_handle_t msg_handle`	Handle for the client's message.
`psa_status_t status`	Message result value to be reported to the client.

Return

void	Success.

Programmer error

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

- `msg_handle` is invalid.
- An invalid status code is specified for the type of message, see usage below.

A PROGRAMMER ERROR will panic the caller.

Usage

A status code must be specified, which will normally be sent to the client.

For a *connection message*, the status codes must conform to those in Table 21. Using any other status codes is a PROGRAMMER ERROR.

Table 21 Status codes for a *connection message*

Status code name	Value	Condition
PSA_SUCCESS	0	Accept the client connection.
PSA_ERROR_CONNECTION_REFUSED	-130	Refuse the client connection, indicating a permanent error.
PSA_ERROR_CONNECTION_BUSY	-131	Fail the client connection, indicating a transient error.

For a *request message*, when the client has made an invalid request the RoT Service can request that the connection is terminated by calling `psa_reply()` with status code `PSA_ERROR_PROGRAMMER_ERROR`. See *Abnormal connection termination* on page 39.

All other status codes are reported to the client. The status codes for a *request message* are summarized in Table 22:

Table 22 Status codes for a *request message*

Status code name	Value	Condition
Success	>= 1	RoT Service specific status code.
PSA_SUCCESS	0	Generic status code to indicate success.
Error	-1 to -128	RoT Service specific error code.
PSA_ERROR_PROGRAMMER_ERROR	-129	Terminate the connection.
PSA_ERROR_CONNECTION_REFUSED	-130	Reserved for *connection message*.
PSA_ERROR_CONNECTION_BUSY	-131	Reserved for *connection message*.
Error	-132 to -256	PSA common error code.
Error	<= -257	RoT Service specific error code.

For a *disconnection* message, the status code is ignored.

psa_notify()

This function sends a `PSA_DOORBELL` signal to a specific Secure Partition.

```
void psa_notify(int32_t partition_id);
```

Parameters

`int32_t partition_id`	Secure Partition ID of the target partition.

Return

void	Success.

Programmer error

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

- `partition_id` does not correspond to a Secure Partition.

A PROGRAMMER ERROR will panic the caller.

Usage

`psa_notify()` is used to asynchronously wake up another Secure Partition. The receiving partition uses `psa_wait()` to detect, or wait for, assertion of its `PSA_DOORBELL` signal.

The value of `partition_id` must be greater than zero as the target of notification must be a Secure Partition, providing a Non-secure Partition ID is a PROGRAMMER ERROR.

The target Secure Partition doorbell will remain asserted until the Secure Partition calls `psa_clear()`.

psa_clear()

This function clears the `PSA_DOORBELL` signal.

```
void psa_clear(void);
```

Return

void	Success.

Programmer error

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

- The Secure Partition's doorbell signal is not currently asserted.

A PROGRAMMER ERROR will panic the caller.

Usage

After calling `psa_clear()`, the doorbell signal will remain clear until `psa_notify()` is called with the Secure Partition's ID.

psa_eoi()

This function informs the SPM that an interrupt has been handled.

```
void psa_eoi(psa_signal_t irq_signal);
```

Parameters

`psa_signal_t irq_signal`	The interrupt signal that has been processed.

Return

void	Success.

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.
- `irq_signal` is not currently asserted.

A PROGRAMMER ERROR will panic the caller.

Usage

This is used after processing an interrupt signal. Calling `psa_eoi()` will re-enable the interrupt line, and so this action needs be taken after managing the source of the interrupt, otherwise the signal is likely to be reasserted immediately.

psa_panic()

This function will terminate execution within the calling Secure Partition and will not return.

```
void psa_panic(void);
```

Return

Does not return

Usage

This function can be used by a Secure Partition when it detects an Internal fault to halt execution. `psa_panic()` is equivalent to the C standard library `abort()` function for a Secure Partition. It will not return.

In many cases, this will result in the SPM implementation restarting the system, see *Panics* on page 46.

SPM implementation note:
The SPM Implementation can provide this interface as a macro instead of a C function in order to provide additional diagnostic information about the origin of the panic.

5 PSA RoT Services

Along with the SPM, the PSA Root of Trust also includes some PSA RoT Services. These services provide access to the fundamental Root of Trust secrets, enforcing the policies for their use, and as defined in the *PSA Security Model*.

The PSA RoT Service APIs abstract platform implementation details, enabling a common interface for Application RoT Service software.

These services can be implemented directly within the SPM, by a secure hardware element accessed via the SPM, or as RoT Services within one or more PSA RoT Secure Partitions. The details of the implementation will depend on the platform hardware architecture.

The PSA RoT Services include:

- Cryptographic operations, including:
  - Symmetric ciphers.
  - Asymmetric algorithms.
  - Hash algorithms.
  - Key derivation.
  - Random number generation.
- Initial Attestation. This service reports the immutable device identity and boot state for use by an Application Attestation Service.

- Internal Trusted Storage. This service provides secure storage of small data items, for example, cryptographic keys.
- PSA RoT Lifecycle. This service reports the lifecycle state of the PSA RoT.

5.1 PSA Cryptography API

This PSA RoT Service is an implementation of the *PSA Cryptography API*. The specification for this API is in a separate manual available from Arm.

This API is designed to support isolation of the application from the key material, allowing the implementation of the API to enforce policies relating to key usage and ownership.

Implementations of the PSA Firmware Framework that provide isolation levels 2 or 3 must implement the PSA Cryptography Service within the PSA RoT and isolated from Application Root of Trust clients.

5.2 PSA Initial Attestation API

This PSA RoT Service is an implementation of the Initial Attestation interface in the *PSA Attestation API*. The specification for this API is in a separate manual available from Arm.

This API is designed either to directly sign data or to bootstrap trust in other attestation schemes that are layered on top of the PSA RoT.

Implementations of the PSA Firmware Framework that provide isolation levels 2 or 3 must implement the PSA Initial Attestation Service within the PSA RoT and isolated from Application Root of Trust clients.

5.3 PSA Internal Trusted Storage API

This PSA RoT Service is an implementation of the Internal Trusted Storage module in the *PSA Storage API*. The specification for this API is in a separate manual available from Arm.

This API is designed to provide confidentiality and integrity protection of limited storage of persistent data against physical and logical attacks.

Implementations of the PSA Firmware Framework that provide isolation levels 2 or 3 must implement the PSA Internal Trusted Storage Service within the PSA RoT and isolated from Application Root of Trust clients.

5.4 PSA RoT Lifecycle API

5.4.1 Overview

The security and behavior of the PSA RoT is dependent on the lifecycle state of the PSA RoT. The *PSA Security Model* identifies a primary set of lifecycle states for the PSA RoT which indicate the availability of the PSA RoT secrets and the overall security state of the PSA RoT.

This state is an essential property in the Initial Attestation, to report on the security state of the device to external services, but it can also influence the operation of the PSA RoT and Application RoT Services in order to protect both device and user assets on the device when outside of the *Secured* state. The PSA RoT Lifecycle API defined here allows these services to determine the security state of the PSA RoT for these purposes.

Implementation lifecycle

Instantiations of the PSA RoT do not need to implement the *PSA Security Model* lifecycle precisely:

- The implementation can ignore states defined in the *PSA Security Model* that are not used. For example, the implementation may not be able to implement *Recoverable PSA RoT Debug*.
- The implementation can have multiple distinct states that correspond to a single *PSA Security Model* state. For example, *NSPE Debug* and *Application RoT Debug* are both types of *Non-PSA RoT Debug*.

However, every implementation lifecycle state must correspond to exactly one of the *PSA Security Model* states.

The PSA RoT Lifecycle API is designed to report the implementation lifecycle state in a format that allows:

- Implementation-independent RoT Services to extract the *PSA Security Model* lifecycle state.
- Implementation-specific RoT Services to extract the full lifecycle state.

This is achieved by forming the lifecycle state value as a combination of the *PSA Security Model* lifecycle state and the Implementation state.

The following elements provide the interface to the PSA RoT Service for querying the PSA RoT lifecycle state.

The following partition API elements must be defined in a header file `<psa/lifecycle.h>`:

```
#define PSA_LIFECYCLE_PSA_STATE_MASK
#define PSA_LIFECYCLE_IMP_STATE_MASK

#define PSA_LIFECYCLE_UNKNOWN
#define PSA_LIFECYCLE_ASSEMBLY_AND_TEST
#define PSA_LIFECYCLE_PSA_ROT_PROVISIONING
#define PSA_LIFECYCLE_SECURED
#define PSA_LIFECYCLE_NON_PSA_ROT_DEBUG
#define PSA_LIFECYCLE_RECOVERABLE_PSA_ROT_DEBUG
#define PSA_LIFECYCLE_DECOMMISSIONED

uint32_t psa_rot_lifecycle_state(void);
```

The details of these definitions are provided in sections 5.4.2 to 5.4.3 below.

5.4.2 Macros

PSA_LIFECYCLE_PSA_STATE_MASK

A mask value that extracts the main lifecycle state from a lifecycle state returned by `psa_rot_lifecycle_state()`.

```
#define PSA_LIFECYCLE_PSA_STATE_MASK (0xff00u)
```

Implementation independent firmware must always mask the value returned by `psa_rot_lifecycle_state()` with `PSA_LIFECYCLE_PSA_STATE_MASK` to identify the current lifecycle state of the PSA RoT.

PSA_LIFECYCLE_IMP_STATE_MASK

A mask value that extracts the IMPLEMENTATION DEFINED lifecycle state from a lifecycle state returned by `psa_rot_lifecycle_state()`.

```
#define PSA_LIFECYCLE_IMP_STATE_MASK (0x00ffu)
```

PSA_LIFECYCLE_UNKNOWN

A lifecycle state value returned by `psa_rot_lifecycle_state()` when the device state is not known.

```
#define PSA_LIFECYCLE_UNKNOWN (0x0000u)
```

PSA_LIFECYCLE_ASSEMBLY_AND_TEST

A lifecycle state value returned by `psa_rot_lifecycle_state()` when the device is in the *Device assembly and Test* state defined by the *PSA Security Model*.

```
#define PSA_LIFECYCLE_ASSEMBLY_AND_TEST (0x1000u)
```

PSA_LIFECYCLE_PSA_ROT_PROVISIONING

A lifecycle state value returned by `psa_rot_lifecycle_state()` when the device is in the *PSA RoT Provisioning* state defined by the *PSA Security Model*.

```
#define PSA_LIFECYCLE_PSA_ROT_PROVISIONING (0x2000u)
```

PSA_LIFECYCLE_SECURED

A lifecycle state value returned by `psa_rot_lifecycle_state()` when the device is in the *Secured* state defined by the *PSA Security Model*.

```
#define PSA_LIFECYCLE_SECURED (0x3000u)
```

PSA_LIFECYCLE_NON_PSA_ROT_DEBUG

A lifecycle state value returned by `psa_rot_lifecycle_state()` when the device is in the *Non PSA RoT Debug* state defined by the *PSA Security Model*.

```
#define PSA_LIFECYCLE_NON_PSA_ROT_DEBUG (0x4000u)
```

PSA_LIFECYCLE_RECOVERABLE_PSA_ROT_DEBUG

A lifecycle state value returned by `psa_rot_lifecycle_state()` when the device is in the *Recoverable PSA RoT Debug* state defined by the *PSA Security Model*.

```
#define PSA_LIFECYCLE_RECOVERABLE_PSA_ROT_DEBUG (0x5000u)
```

PSA_LIFECYCLE_DECOMMISSIONED

A lifecycle state value returned by `psa_rot_lifecycle_state()` when the device is in the *Decommissioned* state defined by the *PSA Security Model*.

```
#define PSA_LIFECYCLE_DECOMMISSIONED (0x6000u)
```

5.4.3 Functions

psa_rot_lifecycle_state()

This function retrieves the current PSA RoT lifecycle state.

```
uint32_t psa_rot_lifecycle_state(void);
```

Parameters

`void`	None.

Return

`uint32_t state`	The current security lifecycle state of the PSA RoT. The PSA state and implementation state are encoded as follows:

	`version[15:8]` – PSA lifecycle state
	`version[7:0]` – IMPLEMENTATION DEFINED state

Usage

This API can be used to determine the current lifecycle state of the PSA RoT, as defined in the *PSA Security Model*.

The PSA Security Model defined state is reported in bits[15:8], and any IMPLEMENTATION DEFINED state in bits[7:0]. The `PSA_LIFECYCLE_PSA_STATE_MASK` and `PSA_LIFECYCLE_IMP_STATE_MASK` constants are provided to extract these parts of the return value.

Appendix A Connection state model

Figure 6 shows the state machine of permitted behavior for a connection between a single client and an RoT Service. This is the state machine as observed by the SPM, which manages the connection. There are no connection states that are observed by both the client and the RoT Service because of the design of the API.

The connection states:

The **PENDING/*** states all appear identical to the RoT Service. The RoT Service signal is asserted, but the message has not been delivered to the RoT Service. The RoT Service will call `psa_get()` in these states.

The **DISCONNECTING/*** states all appear identical to the RoT Service. A *disconnection message* has been delivered to the RoT Service but has not yet been completed by the RoT Service.

PENDING/connect – a connection has been requested by a client.

CONNECTING –the RoT Service is processing a *connection message*.

IDLE – a valid connection that can be used for sending messages.

PENDING/request – a request has been sent to an RoT Service.

ACTIVE – the RoT Service is processing a *request message*.

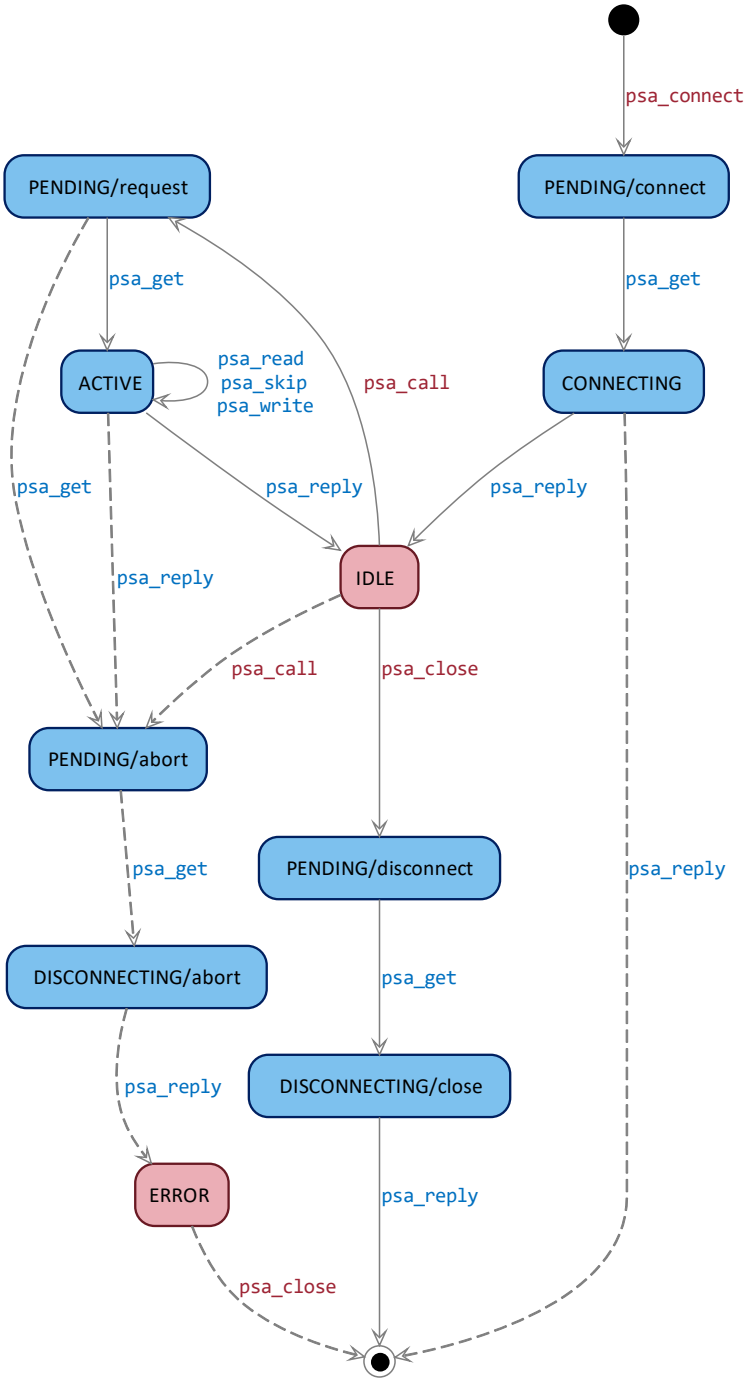
PENDING/disconnect – a disconnection has been requested by the client.

DISCONNECTING/close – the RoT Service is processing a *disconnection message* from a client `psa_close()` request.

PENDING/abort – a disconnection has been triggered due to a PROGRAMMER ERROR.

DISCONNECTING/abort – the RoT Service is processing a *disconnection message* due to abnormal connection termination.

ERROR – a connection that has been terminated by the RoT Service following a PROGRAMMER ERROR detected by the SPM or by the RoT Service.



Red states are observed by the client, and red operations represent API calls initiated by the client.
Blue states are observed by the RoT Service, and blue operations represent API calls initiated by the RoT Service.
Solid lines show the normal operation flow; dashed lines show alternate flows resulting from error conditions.

Figure 6 IPC connection states and operations

A connection handle is used by the client to refer to a connection. The RoT Service is not provided with an explicit identifier for the connection by the SPM, but the RoT Service can identify messages from a connection by associating an `rhandle` value while processing the *connection message*. The state of a connection, as described here, is observed by the RoT Service as a result of the *connection*, *request* and *disconnection messages* received from the SPM.

Multiple clients can concurrently send message to the same or other RoT Service within a single Secure Partition, and the Secure Partition can retrieve more than one of these messages concurrently. The state machine described here applies individually to each connection and message managed by the Secure Partition.

A connection does not exist until a client initiates a connection with an RoT Service, using the `psa_connect()` function in the Client API. The connection exists until the client calls the `psa_close()` function on the handle returned by `psa_connect()`. When a connection is entering an ERROR state, the SPM can release all resources associated with the connection.

Table 23 Permitted state transitions for connections, from the client perspective

	`psa_connect()`	**`psa_call()`**	**`psa_close()`**
Start	IDLE Stop	-	-
IDLE	-	IDLE ERROR	Stop
ERROR	-	ERROR	Stop

Table 24 Observable state transitions for connections, from the RoT Service perspective

	`psa_get()`	**`psa_read()`** **`psa_skip()`** **`psa_write()`**	**`psa_reply()`**	*Client calls* **`psa_call()`** *or* **`psa_close()`**
PENDING/connect	CONNECTING	-	-	-
CONNECTING	-	PROGRAMMER ERROR	IDLE Stop	-
IDLE *(invisible)*	-	-	-	PENDING/request PENDING/disconnect PENDING/abort
PENDING/request	ACTIVE PENDING/abort	-	-	-
ACTIVE	-	ACTIVE	IDLE PENDING/abort	-
PENDING/disconnect	DISCONNECTING/close	-	-	-
PENDING/abort	DISCONNECTING/abort	-	-	-
DISCONNECTING/*	-	PROGRAMMER ERROR	Stop	-

A connection that reaches ERROR represents a PROGRAMMER ERROR. See *Error handling* on page 43.

Appendix B Secure Partition manifest schema

Each Secure Partition manifest must conform to the following JSON schema:

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "description": "schema for a partition description.",
  "type": "object",
  "required": ["psa_framework_version", "name", "type", "priority", "entry_point",
"stack_size"],
  "anyOf": [
    {"required" : ["services"]},
    {"required" : ["irqs"]}
  ],
  "properties": {
    "psa_framework_version" : {
      "description": "Version of the PSA Firmware Framework specification this manifest
conforms to.",
      "enum": [1.0]
    },
    "name": {
      "description": "Alphanumeric C macro for referring to a partition. (all capital)",
      "$ref": "#/definitions/c_macro"
    },
    "type" : {
      "description": "Whether the partition is unprivileged or part of the trusted computing
base.",
    "enum": ["APPLICATION-ROT", "PSA-ROT"]
    },
    "description": {
      "description": "Human readable description",
      "type": "string"
    },
    "priority": {
      "description": "Partition task priority.",
      "enum": ["LOW", "NORMAL", "HIGH"]
    },
    "entry_point": {
      "description": "C symbol name of the partition's entry point. (unmangled, use extern C
if needed)",
      "$ref": "#/definitions/c_symbol"
    },
    "stack_size": {
      "description": "Partition's task stack size in bytes.",
      "$ref": "#/definitions/positive_integer_or_hex_string"
    },
    "heap_size": {
      "description": "Partition's task heap size in bytes.",
      "$ref": "#/definitions/positive_integer_or_hex_string"
    },
    "mmio_regions": {
      "description": "List of Memory-Mapped IO region objects which the partition has access
to.",
      "type": "array",
      "items": {
        "anyOf": [
          { "$ref": "#/definitions/named_region" },
          { "$ref": "#/definitions/numbered_region" }
        ]
      },
      "uniqueItems": true
```

```
    },
    "services ": {
      "description": "List of RoT Service objects which the partition implements.",
      "type": "array",
      "items": { "$ref": "#/definitions/service" },
      "uniqueItems": true
    },
    "dependencies": {
      "description": "List of SID which the partition code depends on and allowed to
access.",
      "type": "array",
      "items": { "$ref": "#/definitions/c_macro" },
      "uniqueItems": true
    },
    "irqs": {
      "description": "List of IRQ objects which the partition implements.",
      "type": "array",
      "items": { "$ref": "#/definitions/irq" },
      "uniqueItems": true
    }
  },
  "definitions": {
    "c_macro": { "type": "string", "pattern": "^[A-Z_][A-Z0-9_]*$" },
    "c_symbol": { "type": "string", "pattern": "^[a-zA-Z_][a-zA-Z0-9_]*$" },
    "hex_string": { "type": "string", "pattern": "^0x(0*[1-9a-fA-F][0-9a-fA-F]*)$",
"minLength": 3, "maxLength": 10 },
    "positive_integer": { "type": "integer", "exclusiveMinimum": true, "minimum": 0 },
    "positive_integer_or_hex_string": {
      "anyOf": [
        { "$ref": "#/definitions/positive_integer" },
        { "$ref": "#/definitions/hex_string" }
      ]
    },
    "named_region": {
      "description": "MMIO region which is described by its C macro name and access
permissions.",
      "required": ["name", "permission"],
      "properties": {
        "name": {
          "description": "Alphanumeric C macro for referring to the region.",
          "$ref": "#/definitions/c_macro"
        },
        "permission": {
          "description": "Access permissions for the region.",
          "enum": [ "READ-ONLY", "READ-WRITE"]
        }
      }
    },
    "numbered_region": {
      "description": "MMIO region which is described by its base address, size and access
permissions.",
      "required": ["base", "size", "permission"],
      "properties": {
        "base": {
          "description": "The base address of the region.",
          "$ref": "#/definitions/hex_string"
        },
        "size": {
          "description": "Size in bytes of the region.",
          "$ref": "#/definitions/positive_integer_or_hex_string"
        },
        "permission": {
```

```
          "description": "Access permissions for the region.",
          "enum": [ "READ-ONLY", "READ-WRITE"]
        }
      }
    },
    "service": {
      "required": ["name", "sid", "non_secure_clients],
      "properties": {
        "name": {
          "description": "Service name prefix used to generate the RoT Service name, version
and signal C macros for use from source code (all capital)",
          "$ref": "#/definitions/c_macro"
        },
"sid": {
          "description": "The integer value of the RoT Service ID",
          "$ref": "#/definitions/positive_integer_or_hex_string"
        },
        "non_secure_clients":{
          "description": "Denote whether the RoT Service is exposed to non-secure clients.",
          "type": "boolean"
        },
        "version": {
          "description": "Optional: Version number of the RoT Service's interface.",
          "$ref": "#/definitions/positive_integer",
 "default": "1"
        },
        "version_policy": {
          "description": "Optional: Version policy to apply on connections to the RoT
Service.",
          "enum": ["STRICT", "RELAXED"],
 "default": "STRICT"
        },
        "description": {
          "description": "Human readable description",
          "type": "string"
        },
      }
    },
    "irq": {
      "required": ["signal", "source"],
      "properties": {
        "source": {
          "description": "Interrupt line number or name for registering to ISR table entry
and enable/disable the specific IRQ once received.",
          "type": "string"
        },
        "signal": {
          "description": "Alphanumeric C macro for referring to the IRQ's signal value. (all
capital)",
          "$ref": "#/definitions/c_macro"
        },
        "description": {
          "description": "Human readable description",
          "type": "string"
        },
      }
    }
  }
}
```

Appendix C Reference header files

The following are example or template versions of the standard header files for the PSA Firmware Framework.

The implementation of the PSA Firmware Framework must provide variations of these files for building both NSPE and Secure Partition firmware.

psa/error.h

```
/* psa/error.h

Standard error codes for the SPM and RoT Services

As defined in PSA Firmware Framework v1.0
*/

#ifndef PSA_ERROR_H
#define PSA_ERROR_H

typedef int32_t psa_status_t;

#define PSA_SUCCESS                     ((psa_status_t)0)

#define PSA_ERROR_PROGRAMMER_ERROR      ((psa_status_t)-129)
#define PSA_ERROR_CONNECTION_REFUSED    ((psa_status_t)-130)
#define PSA_ERROR_CONNECTION_BUSY       ((psa_status_t)-131)
#define PSA_ERROR_GENERIC_ERROR         ((psa_status_t)-132)
#define PSA_ERROR_NOT_PERMITTED         ((psa_status_t)-133)
#define PSA_ERROR_NOT_SUPPORTED         ((psa_status_t)-134)
#define PSA_ERROR_INVALID_ARGUMENT      ((psa_status_t)-135)
#define PSA_ERROR_INVALID_HANDLE        ((psa_status_t)-136)
#define PSA_ERROR_BAD_STATE             ((psa_status_t)-137)
#define PSA_ERROR_BUFFER_TOO_SMALL      ((psa_status_t)-138)
#define PSA_ERROR_ALREADY_EXISTS        ((psa_status_t)-139)
#define PSA_ERROR_DOES_NOT_EXIST        ((psa_status_t)-140)
#define PSA_ERROR_INSUFFICIENT_MEMORY   ((psa_status_t)-141)
#define PSA_ERROR_INSUFFICIENT_STORAGE  ((psa_status_t)-142)
#define PSA_ERROR_INSUFFICIENT_DATA     ((psa_status_t)-143)
#define PSA_ERROR_SERVICE_FAILURE       ((psa_status_t)-144)
#define PSA_ERROR_COMMUNICATION_FAILURE ((psa_status_t)-145)
#define PSA_ERROR_STORAGE_FAILURE       ((psa_status_t)-146)
#define PSA_ERROR_HARDWARE_FAILURE      ((psa_status_t)-147)
#define PSA_ERROR_INVALID_SIGNATURE     ((psa_status_t)-149)

#endif
```

psa/client.h

```
/* psa/client.h

Client API for accessing RoT Services

As defined in PSA Firmware Framework v1.0
*/

#ifndef PSA_CLIENT_H
#define PSA_CLIENT_H

#include <psa/error.h>
```

```
#define PSA_FRAMEWORK_VERSION (0x0100u)
uint32_t psa_framework_version(void);

uint32_t psa_version(uint32_t sid);
#define PSA_VERSION_NONE (0u)

typedef int32_t psa_handle_t;
#define PSA_NULL_HANDLE ((psa_handle_t)0)

#define PSA_MAX_IOVEC (4u)

typedef struct psa_invec {
    const void *base;
    size_t len;
} psa_invec;

typedef struct psa_outvec {
    void *base;
    size_t len;
} psa_outvec;

#define PSA_IPC_CALL (0)

psa_handle_t psa_connect(uint32_t sid, uint32_t version);
psa_status_t psa_call(psa_handle_t handle, int32_t type,
                      const psa_invec *in_vec, size_t in_len,
                      psa_outvec *out_vec, size_t out_len);
void psa_close(psa_handle_t handle);

#endif
```

psa/service.h

```
/* psa/service.h

Secure Partition API for implementing Secure Partitions

As defined in PSA Firmware Framework v1.0
*/

#ifndef PSA_SERVICE_H
#define PSA_SERVICE_H

#include <psa/error.h>
#include <psa/client.h>

typedef uint32_t psa_signal_t;

#define PSA_WAIT_ANY  (0xFFFFFFFFu)
#define PSA_DOORBELL  (0x00000008u)

#define PSA_POLL      (0x00000000u)
#define PSA_BLOCK     (0x80000000u)

psa_signal_t psa_wait(psa_signal_t signal_mask, uint32_t timeout);

typedef struct psa_msg_t {
    int32_t type;
    psa_handle_t handle;
    int32_t client_id;
    void *rhandle;
```

```
    size_t in_size[PSA_MAX_IOVEC];
    size_t out_size[PSA_MAX_IOVEC];
} psa_msg_t;

#define PSA_IPC_CONNECT    (-1)
#define PSA_IPC_DISCONNECT (-2)

psa_status_t psa_get(psa_signal_t signal, psa_msg_t *msg);
void psa_set_rhandle(psa_handle_t msg_handle, void *rhandle);
size_t psa_read(psa_handle_t msg_handle, uint32_t invec_idx, void *buffer, size_t num_bytes);
size_t psa_skip(psa_handle_t msg_handle, uint32_t invec_idx, size_t num_bytes);
void psa_write(psa_handle_t msg_handle, uint32_t outvec_idx,
               const void *buffer, size_t num_bytes);
void psa_reply(psa_handle_t msg_handle, psa_status_t status);

void psa_notify(int32_t partition_id);
void psa_clear(void);

void psa_eoi(psa_signal_t irq_signal);

void psa_panic(void);

#endif
```

psa/lifecycle.h

```
/* psa/lifecycle.h

PSA RoT API for querying the PSA RoT Lifecycle state

As defined in PSA Firmware Framework v1.0
*/

#ifndef PSA_LIFECYCLE_H
#define PSA_LIFECYCLE_H

#define PSA_LIFECYCLE_PSA_STATE_MASK            (0xff00u)
#define PSA_LIFECYCLE_IMP_STATE_MASK            (0x00ffu)

#define PSA_LIFECYCLE_UNKNOWN                   (0x0000u)
#define PSA_LIFECYCLE_ASSEMBLY_AND_TEST         (0x1000u)
#define PSA_LIFECYCLE_PSA_ROT_PROVISIONING      (0x2000u)
#define PSA_LIFECYCLE_SECURED                   (0x3000u)
#define PSA_LIFECYCLE_NON_PSA_ROT_DEBUG         (0x4000u)
#define PSA_LIFECYCLE_RECOVERABLE_PSA_ROT_DEBUG (0x5000u)
#define PSA_LIFECYCLE_DECOMMISSIONED            (0x6000u)

uint32_t psa_rot_lifecycle_state(void);

#endif
```

Appendix D Example of an RoT Service and client

This section is for information only and provides an example pattern of:

- How a client can communicate with an RoT Service using the PSA Client API.
- How an RoT Service handles incoming signals and messages.
- How to create a manifest for a Secure Partition.

Note:

- This example is for illustrative purposes only and is not guaranteed to compile nor conform to any cryptography interface that is specified in PSA documents.
- Arm recommends that Firmware use the *PSA Cryptography API* for hash operations.

psa_sha256_partition.json

A manifest file for the RoT Service implementation.

```
{
  "psa_framework_version": 1.0,
  "name": "CRYPTO_PARTITION",
  "type": "PSA-ROT",
  "priority": "LOW",
  "entry_point": "psa_sha256_main",
  "stack_size": "0x400",
  "heap_size": "0x100",
  "services": [{
      "name": "PSA_SHA256",
      "sid": "0x0000F000",
      "non_secure_clients": true,
      "version": 1,
      "version_policy": "STRICT"
    }
  ]
}
```

psa_sha256.h

The public header file providing the client API.

```
#include <psa/client.h>

// Initialise a SHA256 hash operation
psa_status_t psa_sha256_init(psa_handle_t* phandle);

// Update a SHA256 hash operation with additional data
psa_status_t psa_sha256_update(psa_handle_t handle, const void* buf, size_t len);

// Return the SHA256 hash result and release the operation resources
psa_status_t psa_sha256_final(psa_handle_t handle, void* phash);
```

psa_sha256_priv.h

The private header file defining the RoT Service IPC protocol.

```
enum {
       PSA_SHA256_UPDATE,
       PSA_SHA256_FINAL
};
```

psa_sha256.c

Implementation of the client API.

```
#include <psa/client.h>
#include "psa_manifest/sid.h"
#include "psa_sha256.h"
#include "psa_sha256_priv.h"

psa_status_t psa_sha256_init(psa_handle_t* phandle) {
	psa_handle_t h;
	psa_status_t r;

	h = psa_connect(PSA_SHA256_SID, PSA_SHA256_VERSION);
	if (PSA_HANDLE_IS_VALID(h)) {
		*phandle = h;
		r = PSA_SUCCESS;
	} else {
		*phandle = PSA_NULL_HANDLE;
		r = PSA_HANDLE_TO_ERROR(h);
	}
	return r;
}

psa_status_t psa_sha256_update(psa_handle_t handle, const void* buf, size_t len) {
	psa_invec invec;

	invec.base = buf;
	invec.len = len;

	return psa_call(handle, PSA_SHA256_UPDATE, &invec, 1, NULL, 0);
}

psa_status_t psa_sha256_final(psa_handle_t handle, void* phash) {
	psa_outvec outvec;
	psa_status_t r;

	outvec.base = phash;
	outvec.len = PSA_SHA256_HASH_SIZE;

	r = psa_call(handle, PSA_SHA256_FINAL, NULL, 0, &outvec, 1);
	psa_close(handle);
	return r;
}
```

This is not intended to be an optimized example:

- A better implementation would allow a data buffer to be supplied in `psa_sha256_final()`, so hashes on a fully in-memory buffer can be computed in a single call to `psa_sha256_final()`.
- Instead of being fixed as SHA256, this API might be better designed to take the hash type as an initialization parameter, so it can work for different hash algorithms.

It shows how using `iovecs` allows the client library to pass the client buffer directly to the RoT Service while the message header is sent as a separate parameter. The memory usage of the client library is independent of the data being hashed.

psa_sha256_service.c

RoT Service implementation.

```
#include <psa/service.h>
```

```
#include "psa_sha256_priv.h"
#include "psa_manifest/psa_sh256_partition.h"

static hash_sha256_state_t hash_state;
static int inuse = 0;

static int psa_sha256_process(const psa_msg_t* msg);

void psa_sha256_main(void) {
       uint32_t signals;
       psa_msg_t msg;
       int r;

       for (;;) {
              signals = psa_wait(PSA_WAIT_ANY, PSA_BLOCK);

              if (signals & PSA_SHA256_SIGNAL) {
                     if (psa_get(PSA_SHA256_SIGNAL, &msg) != PSA_SUCCESS)
                            continue;
                     if (msg.type >= 0) {
                            assert (inuse == 1);
                            psa_reply(msg.handle, psa_sha256_process(&msg));
                     } else if (msg.type == PSA_IPC_CONNECT) {
                            /* only allow one connection at a time as
                               Hash_state is a static resource
                            */
                            if (inuse) {
                                   r = PSA_ERROR_CONNECTION_BUSY;
                            } else {
                                   r = PSA_SUCCESS;
                                   inuse = 1;
                                   hash_sha256_init(&hash_state);
                            }
                            psa_reply(msg.handle, r);
                     } else if (msg.type == PSA_IPC_DISCONNECT) {
                            assert (inuse == 1);
                            inuse = 0;
                            psa_reply(msg.handle, PSA_SUCCESS);
                     } else {
                            // cannot get here? [broken SPM]
                            psa_panic();
                     }
              }
       }
}

// limit the memory used by the RoT Service, independently of the size of
// the data sent by the client

#define MAX_BUF_SIZE = 512

// Process a single call for the SHA256 SF
int psa_sha256_process(const psa_msg_t* msg) {
       uint8_t bytes[MAX_BUF_SIZE];
       size_t len;

       // terminate connection if message protocol is invalid
       int r = PSA_ERROR_PROGRAMMER_ERROR;

       if (msg->in_size[1] == 0 && msg->in_size[2] == 0 && msg->in_size[3] == 0 &&
           msg->out_size[1] == 0 && msg->out_size[2] == 0 && msg_out_size[3] == 0) {
              switch (msg->type) {
```

```
                default:
                        // invalid operation
                        break;
                case PSA_SHA256_UPDATE:
                        if (msg->out_size[0] == 0) {
                                // read the data and process it through the hash
                                // read it in blocks of up to MAX_BUF_SIZE bytes
                                for (;;) {
                                        len = psa_read(msg->handle, 0, &bytes, MAX_BUF_SIZE);
                                        if (len == 0)
                                                break;
                                        hash_sha256_update(&hash_state, &bytes, len);
                                }
                                r = PSA_SUCCESS;
                        }
                        break;
                case PSA_SHA256_FINAL:
                        if (msg->in_size[0] == 0 && msg->out_size[0] == PSA_SHA256_HASH_SIZE) {
                                hash_sha256_final(&hash_state, &bytes);
                                psa_write(msg->handle, 0, &bytes, PSA_SHA256_HASH_SIZE);
                                r = PSA_SUCCESS;
                        }
                        break;
                }
        }
        return r;
}
```

This is also not intended to be a perfect example:

- The RoT Service could allow multiple concurrent clients by allocating the hash state for each connection and storing the address in the connection `rhandle` using `psa_set_rhandle()`. The `rhandle` is then returned to the RoT Service in subsequent calls and can be freed during `psa_sha256_final()` or when processing the *disconnection message*.
- The transfer buffer could be static instead of on the stack, or dynamically allocated.

Appendix E Change history

Changes between version 1.0 beta 1 and version 1.0.0

Manifest changes

- Breaking: Replaced `line_num` and `line_name` attributes with `source` attribute in the `irqs` manifest attribute.
- Breaking: Generate the pre-processor symbol names for the RoT Service ID, version and signal from the service's name attribute.
    - The SID symbol is changed from `name` to `name_SID`.
    - The signal symbol is changed from `signal` to `name_SIGNAL`.
    - The version symbol is now provided as `name_VERSION`.

API changes

- Breaking: Added `psa_panic()` to indicate an Internal fault in a Secure Partition. This replaces `abort()`, which is no longer required to be provided by the runtime support.
- Breaking: Introduced a message type parameter to the `psa_call()` function which is delivered as part of the `psa_msg_t` data to the RoT Service.
- Clarification: Refer to message types as *connection*, *request* and *disconnection messages*, rather than by the `PSA_IPC_*` macros.
- Breaking: Changed the values of the defined message types `PSA_IPC_CONNECT`, `PSA_IPC_CALL` and `PSA_IPC_DISCONNECT`. Moved `PSA_IPC_CALL` to <psa/client.h>.
- Clarification: Change references to 'minor version' for RoT Services to 'version' and updated the description of RoT Service versioning to not require a reference to semantic versioning.
- Enhancement: Added `PSA_HANDLE_IS_VALID()` and `PSA_HANDLE_TO_ERROR()` macros.

Other changes

- Defect fix: added `psa_close()` to the list of blocking functions in *Scheduling rules* on page 29.
- Clarification: tighten the wording of the rules relating to signal delivery in *Signal delivery* on page 29.
- Clarification: Provided a reference version of the <psa/lifecycle.h> header file.
- Clarification: Updated *Connection state model* on page 85, to fully replicate the abnormal connection termination and synchronous disconnection behaviors of the IPC mechanism.
- Clarification: Included *Platform services* in Figure 2, to illustrate how non-PSA services can be included in a system.
- Clarification: Changed the way isolation boundaries are drawn to show that they are not all identical, and updated the text to describe this and the requirement for each implementation to document its isolation properties.
- Clarification: Added additional text about the use of isolation within the NSPE.
- Relaxation: Relaxed the rule about freeing dynamic memory: this memory must be scrubbed, but not necessarily set to zero.
- Clarification: Updated the summary of requirements for the Secure Partition C Runtime.
- Clarification: Updated the table showing availability of the APIs for NSPE and Secure Partition firmware.

Changes between version 1.0 beta 0 and version 1.0 beta 1

Manifest changes

- Documented `psa_framework_version` in the manifest attributes.
- Made `heap_size` optional and require that the SPM provides memory allocation functions if the SPM accepts a manifest that specifies a heap size.
- Clarified the rules for generating symbols for signals and IDs.
- Added the `<psa/pid.h>` header file that collates all the Secure Partition IDs for the system.
- Removed the Secure Partition `id` attribute from the manifest. Partition IDs are now allocated by an IMPLEMENTATION DEFINED mechanism.
- Removed the `linker_pattern` attribute from the manifest. Identifying the content source code and objects for a Secure Partition is done via an IMPLEMENTATION DEFINED mechanism

API changes

- Renamed `PSA_DROP_CONNECTION` to `PSA_ERROR_PROGRAMMER_ERROR`.
- Moved the description of the error codes from the `psa_reply()` description to the Error Handling section.
- Defined the C runtime requirements for Secure Partitions, including requirements for handling of global symbols and a subset of the C standard library. See *Secure Partition C runtime* on page 55.
- Added the *PSA RoT Lifecycle API* on page 81.
- Adopted many of the error codes from the PSA RoT APIs as common error codes, defined in a new `<psa/error.h>` header file, for use by any RoT Service. See *Status codes* on page 58.
- Standardized the names of error codes to be prefixed with `PSA_ERROR_`. This changes the name of `PSA_PROGRAMMER_ERROR`, `PSA_CONNECTION_BUSY` and `PSA_CONNECTION_REFUSED`.
- Changed the error code returned by `psa_get()` when the message could not be delivered. It now returns `PSA_ERROR_DOES_NOT_EXIST`.
- Renumbered all the PSA error codes, which are now allocated from the reserved range –129 to –248.
- Reserved error codes from –249 to –256 for SPM implementation defined error conditions.
- Moved the definition of `PSA_MAX_IOVEC` from `psa/service.h` to `psa/client.h`.

Other changes

- Rewrote *Isolation architecture* on page 21 to improve the clarity and precision of the protection rules.
- Defined IMPLEMENTATION DEFINED in the *Terms and Abbreviations*.
- Reworked Error Handling section, introducing the PROGRAMMER ERROR and Panic terms to replace fatal/unrecoverable error.
- Clarified the required responses to PROGRAMMER ERROR in SPE and NSPE clients.
- Added Design goal section to the introduction providing more context for the design decisions in this specification.
- Clarified the required response to an attempted violation of the isolation rules.
- Provided external URL links to the other PSA architectural specifications.
- Included example/reference versions of the Firmware Framework API header files.

Changes between version 0.10 and version 1.0 beta 0

Manifest changes

- `source_files` list attribute has been replaced by `linker_pattern`.
- `extern_sids` is renamed to `dependencies`.
- Schema and example updated.

API changes

- `psa_wait_any()` and `psa_wait_interrupt()` have been combined into `psa_wait()`.

Other changes

- Required SPMs to provide full concurrency for RoT Services and signals, no flexibility is permitted. This allows a single wait function and simplifies the description, Signal delivery and Scheduling and behavior of the signals following message delivery. This also enable SPE->NSPE services to be implemented effectively using the existing IPC mechanism.
- Replaced state diagram and rewrote the text in Appendix A.
- Standardized the location of defined and generated C source header files. Header files are now scoped by being in a sub-directory rather than by using a name prefix.
- Many formatting updates.

Changes between version 0.9 and version 0.10

Manifest changes

- Manifest is now versioned against the `PSA_FRAMEWORK_VERSION`.
- Manifest `identifier` field is renamed to `sid` to be consistent with the Service ID concept in the text.
- Partition ID in the Secure Partition manifest can now be represented in decimal as well as hex.
- Secure Partition manifest schema no longer permits undefined attributes.
- Globbing is now permitted when specifying sources in a Secure Partition manifest.
- Named IRQ support added to schema.
- Support for optional human readable description added to Partition declaration.
- Support for optional human readable description added to service declaration.
- Support for optional human readable description added to IRQ declaration.

API changes

- `psa_error_t` renamed to `psa_status_t`.
- All functions that accept or return signals now use the `psa_signal_t` type.
- `PSA_CONNECTION_ACCEPTED` replaced by `PSA_SUCCESS` in connection messages.
- `PSA_CONNECTION_BUSY` added to indicate transient error conditions during calls to `psa_connect()`, and the PSA error codes renumbered.
- A return value added to `psa_get()` to indicate if delivery of the message is successful. This allows late validation of message parameters by the SPM or deferral of message delivery if SPM resources are not available.
- `psa_identity()` removed and replaced with `client_id` in `psa_msg_t`.
- `psa_end()` renamed to `psa_reply()`.
- The *null handle* made invalid in calls to `psa_reply()`.

- Disconnection must be acknowledged – RoT Services must call `psa_reply()` for `PSA_IPC_DISCONNECT` messages, and the client call to `psa_close()` can only return after this has completed.
- Report actual bytes output from `psa_call()` for each output parameter by requiring the SPM to update the `psa_outvec` structures.
- Reserved vendor IDs reassigned.

Other changes

- "Building Secure Partitions" section removed.
- APIs separated, and labels updated in Figure 3.
- Manifest file example added to Appendix C.
- The behavior when overlapping parameters are sent to an RoT Service has been clarified.
- References to the PSA Cryptography API draft specifications added.
- Minor fixes, clarifications and additions in response to feedback.

Changes between version 0.5 and version 0.9

Threat model and Security analysis

Version 0.5 contained a preliminary threat model and security analysis for IoT devices, including the need for a hardware-protected firmware Root of Trust on the device. The *PSA Security Model* now provides a more complete and structured analysis of the security requirements for the PSA Root of Trust, so this is no longer included in this manual.

Terminology

During the development of the *PSA Security Model*, some of the terminology used in the previous version of the PSA Firmware Framework has been revised to align better with related security terminology.

Table 25 summarizes the main changes in the terminology since version 0.5 of the PSA Firmware Framework specification.

Table 25 Changes in PSA terminology between v0.5 and 1.0

New term	Version 0.5 term	Comment
Secure Partition	Secure/Trusted Partition	Trusted Partitions in v0.5 are now Secure Partitions that are within the PSA Root of Trust domain.
PSA Immutable Root of Trust	Secure Hardware	The hardware, code and data that cannot be modified following manufacturing. See the *PSA Security Model* for details.
PSA Updateable Root of Trust	SPM + Trusted Partitions	The Root of Trust firmware that can be updated following manufacturing. See *PSA Security Model* for details.
PSA Root of Trust	Secure Hardware + SPM + Trusted Partitions	This defines the most trusted security domain within a PSA system. See *PSA Security Model* for details.
Application Root of Trust	Secure Partitions	This is the security domain in which additional security services are implemented. See *PSA Security Model* for details.
Root of Trust Service	Secure/Trusted Function	Abbreviated as RoT Service.

PSA RoT Service	Trusted Function	This is an RoT Service within the PSA Root of Trust.
Application RoT Service	RoT Service	This is an RoT Service within the Application Root of Trust.
Secure Processing Environment (SPE)	SPE (no change)	This is the security domain that includes the PSA Root of Trust and Application Root of Trust.
Non-secure Processing Environment (NSPE)	NSPE (no change)	This is the security domain outside of the SPE. The Application domain, typically containing the application firmware and hardware.
Service ID (SID)	Secure Function ID (SFID)	The identifier used for a PSA RoT Service or an Application RoT Service.

Deprecated API calls

Some of the API features and functions described in version 0.5 have been removed from the specification:

- Asynchronous connection. This feature is not needed as all RoT Services are now declared in manifests. As a result, the SPM does not need a Partition to call `port_create()` to know about the existence of an RoT Service.
- There is no longer a client-side `wait()` or `wait_any()`.
- There is no longer a `port_create()` or `port_close()` function for Secure Partitions because the port numbers are now signals declared in the manifest.
- `accept()` and `reject()` have been incorporated into `psa_reply()` by providing specific return codes.
- Most IPC error codes have been removed and replaced with stricter function requirements relating to non-recoverable errors, memory reference validation and support for client termination by the RoT Service.