# PSA Storage API 1.0

## Architecture & Technology Group

| | |
|---|---|
| Document number: | ARM IHI 0087 |
| Release Quality: | Release |
| Issue Number: | 0 |
| Confidentiality: | Non-Confidential |
| Date of Issue: | 17/06/2019 |

## Contents

# About this document

## Release Information

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

| Date | Changes |
|---|---|
| 2019-02-25 | *Release 1.0 Beta 2* |
| 2019-06-19 | *Release 1.0 Rel*<br>• The protected storage API now supports flags `PSA_STORAGE_FLAG_NO_CONFIDENTIALITY` and `PSA_STORAGE_FLAG_NO_REPLAY_PROTECTION`<br>• Error values now use standard PSA error codes, which are now defined in <psa/error.h>.<br>• Input parameters are now separate from output parameters. There are no longer any in/out parameters.<br>• Size types have been replaced with `size_t` instead of `uint32_t`. |

## PSA Storage API

## Proprietary Notice

**Arm Non-Confidential Document Licence ("Licence")**

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

Arm document reference: LES-PRE-21585

## Potential for change

The contents of this specification are subject to change.

In particular, the following may change:

- Feature addition, modification, or removal
- Parameter addition, modification, or removal
- Numerical values, encodings, bit maps

## Conventions

### Typographical conventions

The typographical conventions are:

*italic*

Introduces special terminology, and denotes citations.

**bold**

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

`monospace`

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

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

SMALL CAPITALS

Used for some common terms such as IMPLEMENTATION DEFINED.

Used for a few terms that have specific technical meanings, and are included in the Glossary.

Red text

Indicates an open issue.

Blue text

Indicates a link. This can be

- A cross-reference to another location within the document
- A URL, for example http://infocenter.arm.com

### Numbers

Numbers are normally written in decimal. Binary numbers are preceded by `0b`, and hexadecimal numbers by `0x`.

In both cases, the prefix and the associated value are written in a monospace font, for example `0xFFFF0000`. To improve readability, long numbers can be written with an underscore separator between every four characters, for example `0xFFFF_0000_0000_0000`. Ignore any underscores when interpreting the value of a number.

## Pseudocode descriptions

This book uses a form of pseudocode to provide precise descriptions of the specified functionality. This pseudocode is written in a monospace font. The pseudocode language is described in the Arm Architecture Reference Manual.

## Assembler syntax descriptions

This book is not expected to contain assembler code or pseudo code examples.

Any code examples are shown in a `monospace` font.

## Current status and anticipated changes

First draft, major changes and re-writes to be expected.

## 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 (PSA Storage API).
- The number and release (ARM IHI 0087 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, an open source firmware reference implementation, and an independent evaluation and certification scheme. 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 Storage APIs described in this document are important components of the PSA that provide key/value storage interfaces for use with device-protected storage. They describe the interface for the storage provided by the PSA Root of Trust (the PSA Internal Trusted Storage API), as well as an interface for external protected storage (the PSA Protected Storage API).

The PSA Internal Trusted Storage API (PSA ITS) must be implemented in the PSA Root of Trust as described in the PSA Security Model specification.

If there are no Application Root of Trust (ARoT) services that rely on it, the PSA Protected Storage API (PS API) may be implemented in the NSPE. Otherwise, the PS API must be implemented in the ARoT.

## 1.1 Use Cases and Rationale

There are two use cases covered here: secure storage for device intimate data (PSA Internal Trusted Storage), and protection for data-at-rest (PSA Protected Storage).

PSA Internal Trust Storage aims at providing a place for devices to store their most intimate secrets, either to ensure data privacy or data integrity. For example, a device identity key requires confidentiality, whereas an authority public key is public data but requires integrity. Other critical values that are part of PSA services, such as secure time values, monotonic counter values or firmware image hashes, will need trusted storage as well.

PSA Protected Storage is meant to protect larger data sets against physical attacks. It aims to provide the ability for a firmware developer to store data onto external flash, with a promise of data-at-rest protection, including device-bound encryption, integrity, and replay protection. It should be possible to select the appropriate protection level, e.g. encryption only, or integrity only, or all three, depending on the threat model of the device and the nature of its deployment.

## 1.2 Technical Background

Modern embedded platforms have multiple types of storage, each with different security properties.

Most embedded MCUs have on-chip flash storage that can be made inaccessible except to software running on the MCU. If the storage is made inaccessible to software other than that of the PSA Root of Trust (PRoT), then it can be used to store key material, replay protection values, or other data critical to the secure operation of the device.

In addition, many platforms also have external storage that requires confidentiality, integrity, and replay protection from attackers with physical access to the device.

By providing consistent APIs for accessing storage, NSPE and SPE software can be written in a platform-independent manner, improving portability between PSA-supported platforms.

## 1.3 The PSA Protected Storage API

The PSA Protected Storage API (PS API) is the general-purpose API that most developers should use. It is intended to be used to protect storage media that are external to the MCU package.

On platforms using external storage for this API that do not provide hardware protection (such as remote locations or tamper proof enclosures), implementations of this API should provide authenticated encryption of the data, as well as replay protection values stored using the PSA Internal Trusted Storage API.

The API provides flags, `PSA_STORAGE_FLAG_NO_CONFIDENTIALITY` and `PSA_STORAGE_FLAG_NO_REPLAY_PROTECTION`, enabling the caller to request a lower level of protection.

`PSA_STORAGE_FLAG_NO_CONFIDENTIALITY` requests integrity but not confidentiality. For example, this might be selected when storing other party's public keys. This flag does not affect replay protection.

`PSA_STORAGE_FLAG_NO_REPLAY_PROTECTION` requests confidentiality and integrity protection of the data as controlled by `PSA_STORAGE_FLAG_NO_CONFIDENTIALITY`, but does not require the implementation to store data that would detect replacement with a previously valid value. For all other data objects, the implementation must ensure that the version returned is the most recently stored version.

> **Note** This is usually achieved by creating a hash table or tree of all the file tags and storing the root in ITS. Some implementations may only store the root and recreate the tree at boot - in which case when it detects and error it cannot tell which file has been tampered with and must reject all attempts to read replay protected files.

The implementation is permitted to treat these flags as indicative and to apply a higher level of protection if it does not implement every protection class. It MUST NOT apply a lower level of protection than that selected.

However, it MUST treat the `PSA_STORAGE_FLAG_WRITE_ONCE` flag as definitive if it is supported.

When reporting meta data, `psa_ps_get_info()` should report the actual protection level applied and not the requested level.

## 1.4 The PSA Internal Trusted Storage API

The PSA Internal Trusted Storage API (PITS API) is a more specialized API. Uses of this API will be less common. It is intended to be used for assets that must be placed inside internal flash. Some examples of assets that require this are replay protection values for external storage and keys for use by components of the PSA Root of Trust.

Storing assets that don't fit this requirement is permissible. In fact, it is expected that many platforms will have the PSA Protected Storage API call directly into the PSA internal Trusted Storage API. This can be done on platforms that do not have external flash, for example.

While this document makes no requirements about the size of the storage available by this API, it is expected to be limited, and therefore should be used for small, security critical, values.

As the Internal Storage is implicitly confidential and protected from replay, the implementation can ignore the flags requesting lower levels of protection. However, it must honor the `PSA_STORAGE_FLAG_WRITE_ONCE` flag.

## 1.5 UIDs

`uids` in this API are defined as `uint64_t`. This is obviously much larger than would be used on any system. This large namespace was chosen to allow Trusted Services to easily manage assets on behalf of other services.

For example, consider a PSA Crypto service running as a Trusted Service. When a service running in another Trusted Partition requests key storage from the PSA Crypto service, the PSA Crypto service can concatenate the partition ID of the requesting service (which is `int32_t`) with the key identifier (which is `uint32_t`) to generate the `uid` of the PSA ITS entry for the key. This allows the PSA Crypto service to easily manage isolation between the key namespaces of its various clients.

Requirements for `uid`:

- The value zero (0) is reserved for `uids` and will result in an error if passed to any of the storage API functions.
- Each partition can use any of the non-zero `uid` in the full 64-bit range.
- `uid` namespaces are independent. Using a `uid` in partition A has no impact on the UIDs or data assets in partition B.
- Data assets are always private. There is no mechanism that enables partition A to access a data asset owned by partition B.

The implication is that the implementation cannot divide the `uid` range between partitions, but it must use the partition ID in addition to the `uid` to identify a specific data asset.

## 1.6 Atomicity of Operations

In the event of power failure or other interruption of operations that modify storage, implementations of these APIs must maintain the following properties:

Atomicity:

After the operation, the data assets of the storage service either contain the new data or are unchanged. Atomicity should be guaranteed in all situations - such as invalid request, software crash or power cycle - and must not result in corruption of the data assets. The only exceptions to this are situations involving storage failures or corruption.

Consistency:

In this API, each operation is individually atomic. A multi-threaded application using this API must not be able to observe any intermediate state in the data assets. If thread B calls the API while thread A is in the middle of an operation that modifies a data asset, thread B must either see the state of the asset before or the state of the asset after the operation requested by thread A.

Isolation:

A partition using the storage service cannot cause a change in the data assets belonging to a different partition.

Durability:

When an operation that modifies storage returns to the caller, the data is persisted. System reset or power fail at this point will not revert the data assets to the previous state.

## 1.7 Components

| **Component** | **Description** |
|---|---|
| PSA Internal Trusted Storage API | The storage API described in this document intended for access to internal flash memory |
| PSA Internal Trusted Storage Service | A PSA Root of Trust (PRoT) Service that implements the PSA Internal Trusted Storage API |
| PSA Protected Storage API | The general-purpose storage API described in this document. |
| PSA Protected Storage Service | A service, implemented either in the Application Root of Trust (ARoT) or the NSPE, that implements the PSA Protected Storage API |

| | |
|---|---|
| Secure Partition Manager | The entity in the PSA Secure Processing Environment responsible for marshalling between the various secure services |

# 2 Requirements

## 2.1 PSA Protected Storage Requirements

1. The technology and techniques used by the PSA Protected Storage Service MUST allow for frequent writes and data updates
2. If writing to external storage, the PSA Protected Storage Service MUST provide confidentiality - unless the caller specifically requests integrity only.
3. Confidentiality for a PSA Protected Storage service may be provided by cryptographic ciphers using device-bound keys, a tamper resistant enclosure, or an inaccessible deployment location, depending on the threat model of the deployed system. If using counter-based encryption, the service must ensure a fresh key and nonce pair is used for each object instance encrypted.
4. If writing to external storage, the PSA Protected Storage Service MUST provide integrity protection
5. Integrity protection for a PSA Protected Storage service may be provided by cryptographic Message Authentication Codes (MAC) or signatures generated using device-bound keys, a tamper resistant enclosure, or an inaccessible deployment location, depending on the threat model of the deployed system.
6. If writing to external storage, the PSA Protected Storage Service MUST provide replay protection by writing replay protection values through the PSA Internal Trusted Storage API, unless the caller specifically requests no replay protection.
7. If providing services to secure partitions, and the PSA Isolation level is 3, the PSA Protected Storage Service MUST provide protection from one PSA partition accessing the storage assets of a different partition
8. The PSA Protected Storage Service MUST use the partition ID information associated with each request for its access control mechanism
9. If the PSA Protected Storage Service is providing services to other ARoT services, it MUST be implemented inside the ARoT itself
10. If implemented inside the ARoT, the PSA Protected Storage Service MAY use helper services outside of the ARoT to perform actual read and write operations through the external interface or file system
11. In the event of power failures or unexpected flash write failures, the implementation must attempt to fallback to allow retention of old content.
12. The creation of a `uid` with value 0 (zero) must be treated as an error.

## 2.2 PSA Internal Trusted Storage Requirements

1. The storage underlying the PSA Internal Trusted Storage Service MUST be protected from read and modification by attackers with physical access to the device.
2. The storage underlying the PSA Internal Trusted Storage Service MUST be protected from direct read or write access from software partitions outside of the PSA Root of Trust (PRoT).
3. The technology and techniques used by the PSA Internal Trusted Storage Service MUST allow for frequent writes and data updates.
4. The PSA Internal Trusted Storage Service MAY provide confidentiality using cryptographic ciphers.
5. The PSA Internal Trusted Storage Service MAY provide integrity protection using cryptographic Message Authentication Codes (MAC) or signatures.

6. The PSA Internal Trusted Storage Service MUST provide protection from one PSA partition accessing the storage assets of a different partition.
7. The PSA Internal Trusted Storage Service MUST use the partition ID information associated with each request for its access control mechanism.
8. The medium and methods utilized by a PSA Internal Trusted Storage Service must provide confidentiality within the threat model of the system.
9. The medium and methods utilized by a PSA Internal Trusted Service must provide integrity within the threat model of the system.
10. If the device supports the RECOVERABLE_PSA_ROT_DEBUG Lifecycle State, then the PSA Internal Trusted Storage Service MUST provide confidentiality and integrity using cryptographic primitives with keys unavailable in the RECOVERABLE_PSA_ROT_DEBUG state.
11. In the event of power failures or unexpected flash write failures, the implementation must attempt to fallback to allow retention of old content.
12. In the extreme case of storage medium being completely non-accessible, no assurances can be made about the availability of the old content.
13. The `PSA_STORAGE_FLAG_WRITE_ONCE` must be enforced when the PSA RoT Lifecycle of the device is SECURED or NON_PSA_ROT_DEBUG. It must not be enforced when the device is in the PSA_ROT_PROVISIONING state.
14. The creation of a `uid` with value 0 (zero) must be treated as an error.

# 3 Theory of Operation

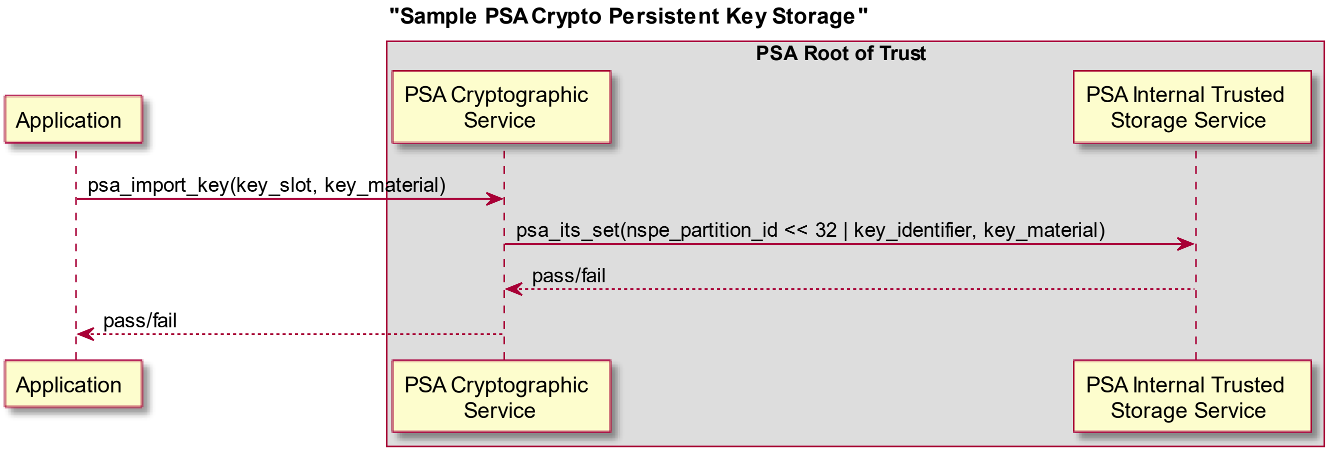
## 3.1 PSA Internal Trusted Storage API

The PSA Trusted Storage Service that implements this API is not expected to replace the need for a filesystem that resides on external storage. Instead, it's intended to be used to interface to a small piece of storage that is only accessible to software that is part of the PSA Root of Trust. The API can be made accessible to the Non-Secure Processing Environment as well as the Secure Processing Environment.

Internally the PSA Trusted Storage Service should be designed such that one partition cannot access the data owned by another partition. The method of doing this is not specified here, but one method would be to store metadata with the data indicating the partition that owns it.

A simple use-case is described below. This is illustrative and not prescriptive.

## 3.2 PSA Cryptographic API Implementation


"Sample PSA Crypto Persistent Key Storage"
PSA Root of Trust
Application
PSA Cryptographic Service
PSA Internal Trusted Storage Service
psa_import_key(key_slot, key_material)
psa_its_set(nspe_partition_id << 32 | key_identifier, key_material)
pass/fail
pass/fail
Application
PSA Cryptographic Service
PSA Internal Trusted Storage Service

Sample Storage

# 4 Reference Implementation

A reference implementation of PSA Internal Trusted Storage is being developed by the Trusted Firmware-M project. The integration guide is available at https://git.trustedfirmware.org/trusted-firmware-m.git/tree/docs/user_guides/services/tfm_sst_integration_guide.md. The reference code is available at https://git.trustedfirmware.org/trusted-firmware-m.git/tree/secure_fw/services/secure_storage/.

# 5 API Reference

## 5.1 General Definitions

These definitions MUST be defined in the header file `psa/storage_common.h`.

### 5.1.1 `psa_storage_info_t` (struct)

A container for metadata associated with a specific `uid`.

```
struct psa_storage_info_t {
    size_t capacity;
    size_t size;
    psa_storage_create_flags_t flags;
};
```

**Fields:**

| | |
|---|---|
| `capacity` | The allocated capacity of the storage associated with a `uid`. |
| `size` | The size of the data associated with a `uid`. |
| `flags` | The flags set when the `uid` was created. |

### 5.1.2 `psa_storage_create_flags_t` (type)

Flags used when creating a data entry.

```
typedef uint32_t psa_storage_create_flags_t;
```

### 5.1.3 `psa_storage_uid_t` (type)

A type for `uid` used for identifying data.

```
typedef uint64_t psa_storage_uid_t;
```

### 5.1.4 `PSA_STORAGE_FLAG_NONE` (macro)

```
#define PSA_STORAGE_FLAG_NONE  0u
```

No flags to pass.

### 5.1.5 `PSA_STORAGE_FLAG_WRITE_ONCE` (macro)

```
#define PSA_STORAGE_FLAG_WRITE_ONCE  (1u << 0)
```

The data associated with the `uid` will not be able to be modified or deleted. Intended to be used to set bits in `psa_storage_create_flags_t`.

### 5.1.6 `PSA_STORAGE_FLAG_NO_CONFIDENTIALITY` (macro)

```
#define PSA_STORAGE_FLAG_NO_CONFIDENTIALITY  (1u << 1)
```

The data associated with the `uid` is public and therefore does not require confidentiality. It therefore only needs to be integrity protected.

### 5.1.7 `PSA_STORAGE_FLAG_NO_REPLAY_PROTECTION` (macro)

```
#define PSA_STORAGE_FLAG_NO_REPLAY_PROTECTION  (1u << 2)
```

The data associated with the `uid` does not require replay protection. This may permit faster storage - but it permits an attacker with physical access to revert to an earlier version of the data.

### 5.1.8 `PSA_STORAGE_SUPPORT_SET_EXTENDED` (macro)

```
#define PSA_STORAGE_SUPPORT_SET_EXTENDED  (1u << 0)
```

Flag indicating that `psa_ps_create` and `psa_ps_set_extended` are supported.

PSA storage specific error codes.

### 5.1.9 `PSA_ERROR_INVALID_SIGNATURE` (macro)

A PSA storage specific error code.

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

The signature on the data is invalid.

### 5.1.10 `PSA_ERROR_DATA_CORRUPT` (macro)

A PSA storage specific error code.

```
#define PSA_ERROR_DATA_CORRUPT  ((psa_status_t)-152)
```

The data on the underlying storage is corrupt.

## 5.2 Internal Trusted Storage API

These definitions MUST be defined in the header file psa/internal_trusted_storage.h.

### 5.2.1 PSA_ITS_API_VERSION_MAJOR (macro)

```
#define PSA_ITS_API_VERSION_MAJOR  1
```

The major version number of the PSA ITS API. It will be incremented on significant updates that may include breaking changes.

### 5.2.2 PSA_ITS_API_VERSION_MINOR (macro)

```
#define PSA_ITS_API_VERSION_MINOR  0
```

The minor version number of the PSA ITS API. It will be incremented in small updates that are unlikely to include breaking changes.

### 5.2.3 psa_its_set (function)

Create a new, or modify an existing, uid/value pair.

```
psa_status_t psa_its_set(psa_storage_uid_t uid,
                         size_t data_length,
                         const void * p_data,
                         psa_storage_create_flags_t create_flags);
```

**Parameters:**

| | |
|---|---|
| uid | The identifier for the data. |
| data_length | The size in bytes of the data in p_data. |
| p_data | A buffer containing the data. |
| create_flags | The flags that the data will be stored with. |

**Returns:** psa_status_t

A status indicating the success/failure of the operation.

| | |
|---|---|
| PSA_SUCCESS | The operation completed successfully. |
| PSA_ERROR_NOT_PERMITTED | The operation failed because the provided uid value was already created with PSA_STORAGE_FLAG_WRITE_ONCE. |
| PSA_ERROR_NOT_SUPPORTED | The operation failed because one or more of the flags provided in create_flags is not supported or is not valid. |
| PSA_ERROR_INSUFFICIENT_STORAGE | The operation failed because there was insufficient space on the storage medium. |
| PSA_ERROR_STORAGE_FAILURE | The operation failed because the physical storage has failed (Fatal error). |
| PSA_ERROR_INVALID_ARGUMENT | The operation failed because one of the provided pointers (p_data) is invalid, for example is NULL or references memory the caller cannot access. |

**Description:**

Stores data in the internal storage.

### 5.2.4 psa_its_get (function)

Retrieve data associated with a provided UID.

```
psa_status_t psa_its_get(psa_storage_uid_t uid,
                         size_t data_offset,
                         size_t data_size,
                         void * p_data,
                         size_t * p_data_length);
```

**Parameters:**

| | |
|---|---|
| uid | The uid value. |
| data_offset | The starting offset of the data requested. |
| data_size | The amount of data requested. |
| p_data | On success, the buffer where the data will be placed. |
| p_data_length | On success, this will contain size of the data placed in p_data. |

**Returns:** psa_status_t

A status indicating the success/failure of the operation.

| | |
|---|---|
| PSA_SUCCESS | The operation completed successfully. |
| PSA_ERROR_DOES_NOT_EXIST | The operation failed because the provided uid value was not found in the storage. |
| PSA_ERROR_STORAGE_FAILURE | The operation failed because the physical storage has failed (Fatal error). |
| PSA_ERROR_INVALID_ARGUMENT | The operation failed because one of the provided arguments( p_data, p_data_length) is invalid, for example is NULL or references memory the caller cannot access. In addition, this can also happen if data_offset is larger than the size of the data associated with uid. |

**Description:**

Retrieves up to data_size bytes of the data associated with uid, starting at data_offset bytes from the beginning of the data. Upon successful completion, the data will be placed in the p_data buffer, which must be at least data_size bytes in size. The length of the data returned will be in p_data_length. If data_size is 0, the contents of p_data_length will be set to zero.

### 5.2.5 psa_its_get_info (function)

Retrieve the metadata about the provided uid.

```
psa_status_t psa_its_get_info(psa_storage_uid_t uid,
                              struct psa_storage_info_t * p_info);
```

**Parameters:**

| | |
|---|---|
| uid | The uid value. |
| p_info | A pointer to the psa_storage_info_t struct that will be populated with the metadata. |

**Returns:** psa_status_t

A status indicating the success/failure of the operation.

| | |
|---|---|
| PSA_SUCCESS | The operation completed successfully. |

| | |
|---|---|
| `PSA_ERROR_DOES_NOT_EXIST` | The operation failed because the provided `uid` value was not found in the storage. |
| `PSA_ERROR_STORAGE_FAILURE` | The operation failed because the physical storage has failed (Fatal error). |
| `PSA_ERROR_INVALID_ARGUMENT` | The operation failed because one of the provided pointers (`p_info`) is invalid, for example is `NULL` or references memory the caller cannot access. |

**Description:**

Retrieves the metadata stored for a given `uid` as a `psa_storage_info_t` structure.

### 5.2.6 `psa_its_remove` (function)

Remove the provided key and its associated data from the storage.

```
psa_status_t psa_its_remove(psa_storage_uid_t uid);
```

**Parameters:**

| | |
|---|---|
| `uid` | The `uid` value. |

**Returns:** `psa_status_t`

A status indicating the success/failure of the operation.

| | |
|---|---|
| `PSA_SUCCESS` | The operation completed successfully. |
| `PSA_ERROR_INVALID_ARGUMENT` | The operation failed because one or more of the given arguments were invalid (null pointer, wrong flags and so on). |
| `PSA_ERROR_DOES_NOT_EXIST` | The operation failed because the provided key value was not found in the storage. |
| `PSA_ERROR_NOT_PERMITTED` | The operation failed because the provided key value was created with `PSA_STORAGE_FLAG_WRITE_ONCE`. |
| `PSA_ERROR_STORAGE_FAILURE` | The operation failed because the physical storage has failed (Fatal error). |

**Description:**

Deletes the data from internal storage.

## 5.3 Protected Storage API

These definitions MUST be defined in the header file `psa/protected_storage.h`.

### 5.3.1 `PSA_PS_API_VERSION_MAJOR` (macro)

```
#define PSA_PS_API_VERSION_MAJOR  1
```

The major version number of the PSA PS API. It will be incremented on significant updates that may include breaking changes.

### 5.3.2 `PSA_PS_API_VERSION_MINOR` (macro)

```
#define PSA_PS_API_VERSION_MINOR  0
```

The minor version number of the PSA PS API. It will be incremented in small updates that are unlikely to include breaking changes.

### 5.3.3 psa_ps_set (function)

Create a new or modify an existing key/value pair.

```
psa_status_t psa_ps_set(psa_storage_uid_t uid,
                        size_t data_length,
                        const void * p_data,
                        psa_storage_create_flags_t create_flags);
```

**Parameters:**

| | |
|---|---|
| `uid` | The identifier for the data. |
| `data_length` | The size in bytes of the data in `p_data`. |
| `p_data` | A buffer containing the data. |
| `create_flags` | The flags indicating the properties of the data. |

**Returns:** `psa_status_t`

A status indicating the success/failure of the operation.

| | |
|---|---|
| `PSA_SUCCESS` | The operation completed successfully. |
| `PSA_ERROR_NOT_PERMITTED` | The operation failed because the provided `uid` value was already created with `PSA_STORAGE_FLAG_WRITE_ONCE`. |
| `PSA_ERROR_INVALID_ARGUMENT` | The operation failed because one or more of the given arguments were invalid. |
| `PSA_ERROR_NOT_SUPPORTED` | The operation failed because one or more of the flags provided in `create_flags` is not supported or is not valid. |
| `PSA_ERROR_INSUFFICIENT_STORAGE` | The operation failed because there was insufficient space on the storage medium. |
| `PSA_ERROR_STORAGE_FAILURE` | The operation failed because the physical storage has failed (Fatal error). |
| `PSA_ERROR_GENERIC_ERROR` | The operation failed because of an unspecified internal failure. |

**Description:**

The newly created asset has a capacity and size that are equal to `data_length`.

### 5.3.4 psa_ps_get (function)

Retrieve data associated with a provided `uid`.

```
psa_status_t psa_ps_get(psa_storage_uid_t uid,
                        size_t data_offset,
                        size_t data_size,
                        void * p_data,
                        size_t * p_data_length);
```

**Parameters:**

| | |
|---|---|
| `uid` | The `uid` value. |
| `data_offset` | The starting offset of the data requested. |
| `data_size` | The amount of data requested. |
| `p_data` | On success, the buffer where the data will be placed. |
| `p_data_length` | On success, will contain size of the data placed in `p_data`. |

**Returns:** psa_status_t

A status indicating the success/failure of the operation.

| | |
|---|---|
| PSA_SUCCESS | The operation completed successfully. |
| PSA_ERROR_INVALID_ARGUMENT | The operation failed because one of the provided arguments( p_data, p_data_length) is invalid, for example is NULL or references memory the caller cannot access. In addition, this can also happen if data_offset is larger than the size of the data associated with uid. |
| PSA_ERROR_DOES_NOT_EXIST | The operation failed because the provided uid value was not found in the storage. |
| PSA_ERROR_STORAGE_FAILURE | The operation failed because the physical storage has failed (Fatal error). |
| PSA_ERROR_GENERIC_ERROR | The operation failed because of an unspecified internal failure. |
| PSA_ERROR_DATA_CORRUPT | The operation failed because of an authentication failure when attempting to get the key. |
| PSA_ERROR_INVALID_SIGNATURE | The operation failed because the data associated with the uid failed authentication. |

**Description:**

Retrieves up to data_size bytes of the data associated with uid, starting at data_offset bytes from the beginning of the data. Upon successful completion, the data will be placed in the p_data buffer, which must be at least data_size bytes in size. The length of the data returned will be in p_data_length. If data_size is 0, the contents of p_data_length will be set to zero.

### 5.3.5 psa_ps_get_info (function)

Retrieve the metadata about the provided uid.

```
psa_status_t psa_ps_get_info(psa_storage_uid_t uid,
                             struct psa_storage_info_t * p_info);
```

**Parameters:**

| | |
|---|---|
| uid | The identifier for the data. |
| p_info | A pointer to the psa_storage_info_t struct that will be populated with the metadata. |

**Returns:** psa_status_t

A status indicating the success/failure of the operation.

| | |
|---|---|
| PSA_SUCCESS | The operation completed successfully. |
| PSA_ERROR_INVALID_ARGUMENT | The operation failed because one or more of the given arguments were invalid (null pointer, wrong flags and so on). |
| PSA_ERROR_DOES_NOT_EXIST | The operation failed because the provided uid value was not found in the storage. |
| PSA_ERROR_STORAGE_FAILURE | The operation failed because the physical storage has failed (Fatal error). |
| PSA_ERROR_GENERIC_ERROR | The operation failed because of an unspecified internal failure. |
| PSA_ERROR_DATA_CORRUPT | The operation failed because of an authentication failure when attempting to get the key. |

| | |
|---|---|
| PSA_ERROR_INVALID_SIGNATURE | The operation failed because the data associated with the uid failed authentication. |

**Description:**

Retrieves the metadata stored for a given uid as a psa_storage_info_t structure.

### 5.3.6 psa_ps_remove (function)

Remove the provided uid and its associated data from the storage.

```
psa_status_t psa_ps_remove(psa_storage_uid_t uid);
```

**Parameters:**

| | |
|---|---|
| uid | The identifier for the data to be removed. |

**Returns:** psa_status_t

A status indicating the success/failure of the operation.

| | |
|---|---|
| PSA_SUCCESS | The operation completed successfully. |
| PSA_ERROR_INVALID_ARGUMENT | The operation failed because one or more of the given arguments were invalid (null pointer, wrong flags and so on). |
| PSA_ERROR_DOES_NOT_EXIST | The operation failed because the provided uid value was not found in the storage. |
| PSA_ERROR_NOT_PERMITTED | The operation failed because the provided uid value was created with PSA_STORAGE_FLAG_WRITE_ONCE. |
| PSA_ERROR_STORAGE_FAILURE | The operation failed because the physical storage has failed (Fatal error). |
| PSA_ERROR_GENERIC_ERROR | The operation failed because of an unspecified internal failure. |

**Description:**

Removes previously stored data and any associated metadata, including rollback protection data.

### 5.3.7 psa_ps_create (function)

```
psa_status_t psa_ps_create(psa_storage_uid_t uid,
                           size_t capacity,
                           psa_storage_create_flags_t create_flags);
```

**Parameters:**

| | |
|---|---|
| uid | A unique identifier for the asset. |
| capacity | The allocated capacity, in bytes, of the uid. |
| create_flags | Flags indicating properties of the storage. |

**Returns:** psa_status_t

| | |
|---|---|
| PSA_SUCCESS | The storage was successfully reserved. |
| PSA_ERROR_STORAGE_FAILURE | The operation failed because the physical storage has failed (Fatal error). |
| PSA_ERROR_INSUFFICIENT_STORAGE | capacity is bigger than the current available space. |
| PSA_ERROR_NOT_SUPPORTED | The function is not implemented or one or more create_flags are not supported. |
| PSA_ERROR_INVALID_ARGUMENT | uid was 0 or create_flags specified flags that are not defined in the API. |

| | |
|---|---|
| PSA_ERROR_GENERIC_ERROR | The operation has failed due to an unspecified error. |
| PSA_ERROR_ALREADY_EXISTS | Storage for the specified uid already exists. |

**Description:**

Reserves storage for the specified uid. Upon success, the capacity of the storage is capacity, and the size is 0.

It is only necessary to call this function for assets that will be written with the psa_ps_set_extended function. If only the psa_ps_set function is needed, calls to this function are redundant.

This function cannot be used to replace an existing asset, and attempting to do so will return PSA_ERROR_ALREADY_EXISTS.

If the PSA_STORAGE_FLAG_WRITE_ONCE flag is passed, psa_ps_create will return PSA_ERROR_NOT_SUPPORTED.

This function is optional. Consult the documentation of your chosen platform to determine if it is implemented, or perform a call to psa_ps_get_support. This function must be implemented if psa_ps_get_support returns PSA_STORAGE_SUPPORT_SET_EXTENDED.

### 5.3.8 psa_ps_set_extended (function)

```
psa_status_t psa_ps_set_extended(psa_storage_uid_t uid,
                                 size_t data_offset,
                                 size_t data_length,
                                 const void * p_data);
```

**Parameters:**

| | |
|---|---|
| uid | The unique identifier for the asset. |
| data_offset | Offset within the asset to start the write. |
| data_length | The size in bytes of the data in p_data to write. |
| p_data | Pointer to a buffer which contains the data to write. |

**Returns:** psa_status_t

| | |
|---|---|
| PSA_SUCCESS | The asset exists, the input parameters are correct and the data is correctly written in the physical storage. |
| PSA_ERROR_STORAGE_FAILURE | The data was not written correctly in the physical storage. |
| PSA_ERROR_INVALID_ARGUMENT | The operation failed because one or more of the preconditions listed above regarding data_offset, size, or data_length was violated. |
| PSA_ERROR_DOES_NOT_EXIST | The specified uid was not found. |
| PSA_ERROR_NOT_SUPPORTED | The implementation of the API does not support this function. |
| PSA_ERROR_GENERIC_ERROR | The operation failed due to an unspecified error. |
| PSA_ERROR_DATA_CORRUPT | The operation failed because the existing data has been corrupted. |
| PSA_ERROR_INVALID_SIGNATURE | The operation failed because the existing data failed authentication (MAC check failed). |
| PSA_ERROR_NOT_PERMITTED | The operation failed because it was attempted on an asset which was written with the flag PSA_STORAGE_FLAG_WRITE_ONCE. |

**Description:**

Sets partial data into an asset based on the given identifier, data_offset, data length and p_data.

Before calling this function, the storage must have been reserved with a call to `psa_ps_create`. It can also be used to overwrite data in an asset that was created with a call to `psa_ps_set`.

Calling this function with `data_length` = 0 is permitted. This makes no change to the stored data.

This function can overwrite existing data and/or extend it up to the capacity for the `uid` specified in `psa_ps_create`, but cannot create gaps. That is, it has preconditions:

- `data_offset <= size`
- `data_offset + data_length <= capacity`

and postconditions:

- `size = max(size, data_offset + data_length)`
- `capacity` unchanged.

This function is optional. Consult the documentation of your chosen platform to determine if it is implemented, or perform a call to `psa_ps_get_support`. This function must be implemented if `psa_ps_get_support` returns `PSA_STORAGE_SUPPORT_SET_EXTENDED`.

### 5.3.9 `psa_ps_get_support` (function)

```
uint32_t psa_ps_get_support(void);
```

**Returns:** `uint32_t`

**Description:**

Returns a bitmask with flags set for all of the optional features supported by the implementation.

Currently defined flags are limited to:

- `PSA_STORAGE_SUPPORT_SET_EXTENDED`