PSA Cryptography API Specification

Release 1.0 beta2

Arm

2019-02-22

CHAPTER

ONE

INTRODUCTION

Arm's Platform Security Architecture (PSA) is a holistic set of threat models, security analyses, hardware and firmware architecture specifications, and an open source firmware reference implementation. 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 Cryptographic API (Crypto API) described in this document is an important component of the PSA that provides an interface to modern cryptographic primitives on resource-constrained devices. It constitutes an interface that is easy to comprehend while still providing access to the primitives used in modern cryptography. The interface does not require the user to have access to the key material, instead using opaque key handles.

This document is part of the Platform Security Architecture (PSA) family of specifications. It defines an interface for cryptographic services, including cryptography primitives and a key storage functionality.

This document includes:

- A *rationale* for the design.
- A *description of typical architectures* of implementations of this specification.
- A *high-level overview of the functionality* provided by the interface.
- General considerations *for implementers* of this specification and *for applications* that use the interface defined in this specification.

Refer to the companion document "Platform Security Architecture — cryptography and keystore interface" for a detailed definition of the API.

Companion documents will define *profiles* for this specification. A profile is a minimum mandatory subset of the interface that a compliant implementation must provide.

CHAPTER

TWO

DESIGN GOALS

Suitable for constrained devices

The interface defined in this document was designed to be suitable for a vast range of devices, from special-purpose cryptographic processors specialized to process data with a built-in key, through constrained devices running custom application code such as microcontrollers, to multi-application devices such as servers. As a consequence, the interface is modular and scalable.

- *Scalable*: you shouldn't pay for functionality that you don't need.
- *Modular*: larger devices implement larger subsets of the same interface, not different interfaces.

Because this specification is designed to be suitable for very constrained devices, including devices where memory is very limited, all operations on unbounded amounts of data allow *multipart* processing if the calculations on the data are performed in a streaming manner. This means that the application does not need to store the whole message in memory at a time.

Memory outside the keystore boundary is meant to be managed by the application. The interface is intended to allow implementations not to retain any data between function calls apart from the content of the keystore and other data that needs to be stored inside the keystore security boundary.

The interface does not expose the representation of keys and intermediate data except when required for interchange. This allows each implementation to choose optimal data representations. Implementations with multiple components are also free to choose which memory area to use for internal data.

A keystore interface

This specification is designed to allow cryptographic operations performed on a key to which the application does not have direct access. Except where required for interchange, applications access all keys indirectly, via a handle. The key material corresponding to that handle may reside inside a security boundary that prevents it from being extracted (except as permitted by a policy defined when the key is created).

Optional isolation

Implementations may optionally isolate the cryptoprocessor from the calling application, and may optionally further isolate multiple calling applications. The interface is designed to allow the implementation to be separated between a frontend and a backend. In an implementation with isolation, the frontend is the part of the implementation that is located in the same isolation boundary as the application, which the application accesses via function calls, and the backend is the part of the implementation that is located in a different environment which is protected from the frontend. The protection may be provided by a technology such as process isolation in an operating system, partition isolation with a virtual machine or partition manager, physical separation between devices, or any suitable technology. How the frontend and the backend communicate is out of scope of this specification.

In an implementation with isolation, the backend may serve more than one implementation instance. In this case, a single backend communicates with multiple instances of the frontend. The backend must enforce *caller isolation*: it must ensure that assets of one frontend are not visible to any other frontend. How callers are identified is out of scope of this specification. Implementations that provide caller isolation SHALL document how callers are identified. Implementations that provide isolation SHALL document any implementaion-specific extension of the API that may allow frontend instances to share data in any form.

In summary, there are three types of implementations:

- No isolation: there is no security boundary between the application and the cryptoprocessor. An example type of implementation with no isolation is a statically or dynamically linked library.
- Cryptoprocessor isolation: there is a security boundary beween the application and the cryptoprocessor, but the cryptoprocessor does not communicate with other applications. An example type of implementation with cryptoprocessor isolation is a cryptoprocessor chip that is a companion to an application processor.
- Caller isolation: there are multiple application instances, with a security boundary between the application instances among themselves as well as between the cryptoprocessor and the application instances. An example type of implementation with cryptoprocessor isolation is a cryptography service in a multiprocess environment.

Choice of algorithms

This specification defines a low-level cryptographic interface, where the caller explicitly chooses which algorithm and which security parameters to use. This is necessary to implement protocols that are inescapable in various use cases. The interface is designed to support widespread protocols and data exchange formats, as well as custom ones that applications may need to implement.

As a consequence, all cryptographic functionality operates according to the precise algorithm specified by the caller. (This does not apply to device-internal functionality which does not involve any form of interoperability, such as random number generation.) This specification does not include generic higher-level interfaces where the implementation chooses the best algorithm for a purpose, but higher-level libraries can be built on top of it.

Another consequence is that this specification permits the use of algorithms, key sizes and other parameters that are known to be insecure, but may be necessary to support legacy protocols or legacy data. Where major weaknesses are known, the algorithm description includes applicable warnings, but the lack of a warning does not and cannot indicate that an algorithm is secure in all circumstances. Application developers should research the security of the algorithms that they plan to use and decide according to their needs.

The interface is designed to facilitate algorithm agility. As a consequence, cryptographic primitives are presented through generic functions, with a parameter indicating the specific choice of algorithm. For example, there is a single function to calculate a message digest, taking a parameter which identifies the specific hash algorithm.

Ease of use

The interface is designed to be as easy to use as possible, given the aforementioned constraints on suitability for varied types of devices and on the freedom to choose algorithms.

In particular, the code flows are designed to reduce the chance of dangerous misuse. The interface is intended to make misuse harder than correct use, and for likely mistakes to result in test failures rather than subtle security issues. Implementations are encouraged to avoid leaking data when a function is called with invalid parameters, to the extent allowed by the C language and by implementation size constraints.

Example use cases

This section lists some of the use cases that were considered when designing this API. This list is not limitative, nor are all implementations required to support all use cases.

Network Security (TLS)

This API should provide everything needed to establish TLS connections on the device side: asymmetric key management inside a key store, symmetric ciphers, MAC, HMAC, message digests, and AEAD.

Secure Storage

This API should provide all primitives related to storage encryption, block- or file-based, with master encryption keys stored inside a key store.

Network Credentials

This API should provide network credential management inside a key store, e.g. for X.509-based authentication or pre-shared keys on enterprise networks.

Device Pairing

This API should provide support for key agreement protocols that are often used for secure pairing of devices over wireless channels, for example pairing an NFC token or a bluetooth device could make use of key agreement protocols upon first use.

Secure Boot

This API should provide primitives for use during firmware integrity and authenticity validation during a secure or trusted boot process.

Attestation

This API should provide primitives used in attestation activities. Attestation is the ability for a device to sign an arbitrary bag of bytes with a device private key and return the result to the caller. Several use cases are attached to this, from attestation of the device state to the ability to generate a key pair and prove that it has been generated inside a secure key store. The API provides access to the algorithms commonly used for attestation.

Factory Provisioning

It is expected that most IoT devices will receive a unique identity during a factory provisioning process or once deployed to the field. This API should provide the APIs necessary for populating a device with keys that represent that identity.

CHAPTER

THREE

FUNCTIONALITY OVERVIEW

This section provides a high-level overview of the functionality provided by the interface defined in this specification. Refer to the API definition for a detailed description.

Due to the modularity of the interface, almost every part of the library is optional. The only mandatory function is `psa_crypto_init`.

Library management

Before any use, applications must call `psa_crypto_init` to initialize the library.

Key management

Applications always access keys via a handle. This allows keys to be non-extractable, i.e. an application can perform operations using a key without having access to the key material. Non-extractable keys are bound to the device, can be rate-limited, and can have their usage restricted by policies.

Each key has a *lifetime* that determines when the key material is destroyed. There are two types of lifetimes: *volatile* and *persistent*.

Volatile keys

A *volatile* key is destroyed as soon as the application closes the handle to the key. When the application terminates, it conceptually closes all of its key handles. Conceptually, a volatile key is stored in RAM. Volatile keys have the lifetime `PSA_KEY_LIFETIME_VOLATILE`.

To create a volatile key:

1. Call `psa_allocate_key`.
2. Set the key's policy.
3. Provision the key material with `psa_import_key`, `psa_generate_key`, `psa_generator_import_key` or `psa_clone_key`.

To destroy a volatile key, call `psa_close_key` or `psa_destroy_key` (these functions are equivalent when called on a volatile key).

Persistent keys

A *persistent* key exists until it explicitly destroyed with psa_destroy_key or until it is wiped by the reset or destruction of the device. Persistent keys may be stored in different storage areas; this is indicated through different lifetime values. This specification defines a lifetime value PSA_KEY_LIFETIME_PERSISTENT which corresponds to a default storage area. Implementations may define alternative lifetime values corresponding to different storage areas with different retention policies, or to secure elements with different security characteristics.

To create a persistent key:

1. Call psa_create_key, specifying the desired lifetime for the key and the desired persistent identifier. The lifetime value specifies the storage area for the key data and metadata, and the identifier serves as a name. Lifetimes are namespaces for persistent keys: the same key identifier value with distinct lifetime values designates unrelated keys.
2. Set the key's policy.
3. Provision the key material with psa_import_key, psa_generate_key, psa_generator_import_key or psa_clone_key.

To release memory resources associated with a key but keep the key in storage, call psa_close_key. To access an existing persistent key, call psa_open_key with the same lifetime value and the same key identifier as the original call to psa_create_key.

To destroy a persistent key, open it (if it isn't already open) and call psa_destroy_key.

Recommendations of minimum standards for key management

Most implementations should provide the functions psa_import_key. The only exceptions are implementations that only give access to a key or keys that are provisioned by proprietary means and do not allow the main application to use its own cryptographic material.

Most implementations should provide psa_get_key_information, psa_get_key_lifetime and psa_get_key_policy since they are easy to implement and it is difficult to write applications and especially to diagnose issues without being able to check the metadata.

Most implementations should also provide psa_export_public_key if they support any asymmetric algorithm, since public-key cryptography often requires delivery of a public key that is associated with a protected private key.

Most implementations should provide psa_export_key. However, highly constrained implementations that are designed to work either only with short-term keys (no non-volatile storage) or only with long-term non-extractable keys may omit this function.

Usage policies

All keys have an associated policy that regulates what operations are permitted on the key. This specification defines policies that encode three kinds of attributes:

- The extractable flag PSA_KEY_USAGE_EXPORT determines whether the key material can be extracted. The extractable flag is encoded in the usage bitmask which has the type psa_key_usage_t.
- The usage flags PSA_KEY_USAGE_ENCRYPT, PSA_KEY_USAGE_SIGN, etc. determine whether the corresponding operation is permitted on the key. These flags are encoded in the usage bitmask as well.
- In addition to the usage bitmask, a policy specifies which algorithm is permitted with the key. This specification only defines policies that restrict keys to a single algorithm, which is in keeping with common practice and with security good practice.

Most implementations should provide the function `psa_set_key_policy`. Highly constrained implementations that only support slots with preset policies may omit this function.

Symmetric cryptography

This specification defines interfaces for message digests (hash functions), MAC (message authentication codes), symmetric ciphers and authenticated encryption with associated data (AEAD). For each type of primitive, the API includes two standalone functions (compute and verify, or encrypt and decrypt) as well as a series of functions that permit *multipart operations*.

The standalone functions are:

- `psa_hash_compute` and `psa_hash_compare` to calculate the hash of a message or compare the hash of a message with a reference value.
- `psa_mac_compute` and `psa_mac_verify` to calculate the MAC of a message of compare the MAC with a reference value.
- `psa_cipher_encrypt` and `psa_cipher_decrypt` to encrypt or decrypt a message using an unauthenticated symmetric cipher. The encryption function generates a random IV; to use a deterministic IV (which is not secure in general, but can be secure in some conditions that depend on the algorithm), use the multipart API.
- `psa_aead_encrypt` and `psa_aead_decrypt` to encrypt/decrypt and authenticate a message using an AEAD algorithm. These functions follow the interface recommended by RFC 5116.

Multipart operations

The API provides a multipart interface to hash, MAC, symmetric cipher and AEAD primitives. These interfaces process messages one chunk at a time, with the size of chunks determined by the caller. This allows processing messages that cannot be assembled in memory. The steps to perform a multipart operation are as follows:

1. Allocate an operation object. It is free to use any allocation strategy: stack, heap, static, etc.
2. Initialize the operation object by setting it to zero (either logical zero or all-bits-zero) or by calling one of the applicable macro `PSA_xxx_INIT` or function `psa_xxx_init`.
3. Associate a key with the operation using the applicable function: `psa_hash_setup`, `psa_mac_sign_setup`, `psa_mac_verify_setup`, `psa_cipher_encrypt_setup`, `psa_cipher_decrypt_setup`, `psa_aead_encrypt_setup`, `psa_aead_decrypt_setup`.
4. When encrypting data, generate or set an initialization vector (IV) or nonce or similar initial value such as an initial counter value. When decrypting, set the IV or nonce. For a symmetric cipher, to generate a random IV, which is recommended in most protocols, call `psa_cipher_generate_iv`. To set the IV, call `psa_cipher_set_iv`. For AEAD, call `psa_aead_generate_nonce` or `psa_aead_set_nonce`.
5. Call the applicable update function on successive chunks of the message: `psa_hash_update`, `psa_mac_update` or `psa_cipher_update`.
6. At the end of the message, call the applicable finishing function. There are three kinds of finishing function, depending on what to do with the verification tag.
   - Unauthenticated encryption and decryption does not involve a verification tag. Call `psa_cipher_finish`.
   - To calculate the digest or MAC or authentication tag of a message, call the applicable function to calculate and output the verification tag: `psa_hash_finish`, `psa_mac_sign_finish` or `psa_aead_finish`.

- To verify the digest or MAC of a message against a reference value or to verify the authentication tag at the end of AEAD decryption, call the applicable function to compare the verification tag with the reference value: `psa_hash_verify`, `psa_mac_verify_finish` or `psa_aead_verify`.

Calling the start/setup function may allocate resources inside the implementation. These resources are freed when calling the associated finishing function. In addition, each family of functions defines a function `psa_xxx_abort` which can be called at any time to free the resources associated with an operation.

Authenticated encryption

Having a multipart interface to authenticated encryption raises specific issues.

Multipart authenticated decryption produces partial results that are not authenticated. Applications must not use or expose partial results of authenticated decryption until `psa_aead_verify` has returned a success status, and must destroy all partial results without revealing them if `psa_aead_verify` returns a failure status. Revealing partial results (directly, or indirectly through the application's behavior) can compromise the confidentiality of all inputs that are encrypted with the same key.

For encryption, some common algorithms cannot be processed in a streaming fashion. For SIV mode, the whole plaintext must be known before the encryption can start; the multipart AEAD API is not meant to be usable with SIV mode. For CCM mode, the length of the plaintext must be known before the encryption can start; the application can call the function `psa_aead_set_lengths` to provide these lengths before providing input.

Key derivation and generators

This specification defines a mechanism for key derivation that allows splitting the output of the derivation into multiple keys as well as non-key outputs.

In an implementation with *isolation*, the intermediate state of the key derivation is not visible to the caller, and if an output of derivation is a non-exportable key, then this output cannot be recovered outside the isolation boundary.

Generators

A *generator* is an object that encodes a method to generate a finite stream of bytes. This data stream is computed by the cryptoprocessor and extracted in chunks. The intent of generators is that if two generators are constructed with the same parameters then they will produce the same outputs.

Some examples of generators are:

- A pseudorandom generator, initialized with a seed and other parameters.
- A key derivation function, initialized with a secret, a salt and other parameters.
- A key agreement function, initialized with a public key (peer key), a key pair (own key) and other parameters.

The lifecycle of a generator is as follows:

1. Setup: construct the object and set its parameters. The setup phase determines the generator's capacity, which is the length of the generated stream, i.e. the maximum number of bytes that can be generated with this generator.
2. Generate: read bytes from the stream defined by the generator. This can be done any number of times until the stream is exhausted because its capacity has been reached. Each generation step can either be used to populate a key object (`psa_generator_import_key`), or to read some bytes and extract them as cleartext (`psa_generator_read`).
3. Terminate: clear the generator object and release associated resources (`psa_generator_abort`).

A generator cannot be rewinded. Once a part of the stream has been read, it cannot be read again. This ensures that the same part of the generator output will not be used from different purposes.

Key derivation function

This specification defines functions to set up a key derivation. A key derivation consists of two parts:

1. Input collection. This is sometimes known as *extraction*: the operation "extracts" information from the inputs to generate a pseudorandom intermediate secret value.
2. Output generation. This is sometimes known as *expansion*: the operation "expands" the intermediate secret value to the desired output length.

The API uses a *generator object* to store the state of a key derivation operation. To perform a key derivation:

1. Initialize a generator object to zero or to `PSA_CRYPTO_GENERATOR_INIT`.
2. Call `psa_key_derivation_setup` to select a key derivation algorithm.
3. Call the functions `psa_key_derivation_input_bytes`, `psa_key_derivation_input_key` and `psa_key_agreement` to provide the inputs to the key derivation algorithm. Many key derivation algorithm take multiple inputs; the "step" parameter to these functions indicates which input is being passed.
4. Call `psa_generator_import_key` to create a derived key, or `psa_generator_read` to export the derived data. These functions may be called multiple times to read successive output from the key derivation.
5. Call `psa_generator_abort` to release the generator memory.

Here is an example of a use case where a master key is used to generate both a message encryption key and an IV for the encryption, and the derived key and IV are then used to encrypt a message.

1. Allocate a key slot for the derived message encryption key and set its policy.
2. Derive the message encryption material from the master key.
   (a) Initialize a generator object to zero or to `PSA_CRYPTO_GENERATOR_INIT`.
   (b) Call `psa_key_derivation_setup` with `PSA_ALG_HKDF` as the algorithm.
   (c) Call `psa_key_derivation_input_key` with the step `PSA_KDF_STEP_SECRET` and the master key.
   (d) Call `psa_key_derivation_input_bytes` with the step `PSA_KDF_STEP_INFO` and a public value that uniquely identifies the message.
   (e) Call `psa_generator_import_key` to create the derived message key.
   (f) Call `psa_generator_read` to generate the derived IV.
   (g) Call `psa_generator_abort` to release the generator memory.
3. Encrypt the message with the derived material.
   (a) Call `psa_cipher_encrypt_setup` with the derived encryption key.
   (b) Call `psa_cipher_set_iv` using the derived IV retrieved above.
   (c) Call `psa_cipher_update` one or more times to encrypt the message.
   (d) Call `psa_cipher_finish` at the end of the message.
4. Call `psa_destroy_key` to clear the generated key.

Asymmetric cryptography

The asymmetric cryptography part of this interface defines functions for asymmetric encryption, asymmetric signature and two-way key agreement.

Asymmetric encryption

Asymmetric encryption is provided through the functions `psa_asymmetric_encrypt` and `psa_asymmetric_decrypt`.

Hash-and-sign

The signature and verification functions `psa_asymmetric_sign` and `psa_asymmetric_verify` take a hash as one of their inputs. This hash should be calculated with `psa_hash_setup`, `psa_hash_update` and `psa_hash_finish` before calling `psa_asymmetric_sign` or `psa_asymmetric_verify`. To determine which hash algorithm to use, call the macro `PSA_ALG_SIGN_GET_HASH` on the corresponding signature algorithm.

Key agreement

This specification defines two functions for a Diffie-Hellman-style key agreement where each party combines its own private key with the peer's public key. The recommended function is `psa_key_agreement`, which calculates a shared secret and passes it as one of the inputs to a *key derivation function*. In case an application needs direct access to the shared secret, it can call `psa_key_agreement_raw_shared_secret`; note that in general the shared secret is not directly suitable for use as a key because it is biased.

Randomness and key generation

It is strongly recommended that implementations include a random generator consisting of a cryptographically secure pseudo-random generator (CSPRNG) which is adequately seeded with a cryptographic-quality hardware entropy source, commonly refered to as a true random number generator (TRNG). Constrained implementations may omit the random generation functionality if they do not implement any algorithm that requires randomness internally and they do not provide a key generation functionality — for example a special-purpose component for signature verification.

Applications should use `psa_generate_key`, `psa_encrypt_generate_iv` or `psa_aead_generate_iv` to generate suitably-formatted random data as applicable. In addition, the API includes a function `psa_generate_random` to generate and extract arbitrary random data.

Future additions

We plan to cover the following features in future drafts or future editions of this specification:

- Single-shot functions for symmetric operations.
- Multi-part operations for hybrid cryptography: hash-and-sign (e.g. for EdDSA), hybrid encryption (e.g. for ECIES).
- Key exchange and a more general interface to key derivation. This would enable deriving a non-extractable session key from non-extractable secrets without leaking the intermediate material.
- Key wrapping mechanisms, to extract and import keys in a protected form (encrypted and authenticated).

- Key discovery and slot discovery mechanisms. This would enable locating a key through its name or attributes rather than having to hard-code slot numbers, and finding a slot to contain a key prior to creating the key.
- An ownership and access control mechanism allowing a multi-client implementation to have privileged clients that are able to manage keys of other clients.

CHAPTER

FOUR

SAMPLE ARCHITECTURES

This section describes some possible architectures of implementations of the interface described in this specification. This list of architectures is not limitative and this section is entirely non-normative.

Single-partition architecture

In this architecture, there is no security boundary inside the system. The application code may access all the system memory, including the memory used by the cryptographic services described by this specification. Thus this architecture provides *no isolation*.

This architecture does not conform to the Arm Platform Security Architecture specification. However, it may be useful to provide cryptographic services using the same interface even on devices that cannot support any security boundary. Therefore, while this architecture is not the primary design goal of the API defined in the present specification, it is supported.

In this case, the functions in this specification simply execute the underlying algorithmic code. Security checks can be kept to a minimum since the cryptoprocessor cannot defend against a malicious application. Key import and export copy data inside the same memory space.

This architecture also describes a subset of some larger systems where the cryptographic services are implemented inside a high-security partition, separate from the code of the main application, but it shares this high-security partition with other platform security services.

Cryptographic token and single-application processor

This example system is composed of two partitions: one partition is a cryptoprocessor, and the other partition runs an application. There is a security boundary between the two partitions, so that the application cannot access the cryptoprocessor except through its public interface. Thus this architecture provides *cryptoprocessor isolation*. The cryptoprocessor includes some nonvolatile storage, a TRNG, and possibly some cryptographic accelerators.

There are multiple possible physical realizations: the cryptoprocessor may be a separate chip, a separate processor on the same chip, or a logical partition using a combination of hardware and software to provide the isolation. These realizations are functionally equivalent in terms of the offered software interface, but they would typically offer different levels of security guarantees.

The PSA crypto API in the application processor consists of a thin layer of code that translates function calls to remote procedure calls in the cryptoprocessor. All cryptographic computations are therefore performed inside the cryptoprocessor. Non-volatile keys are stored inside the cryptoprocessor.

Cryptoprocessor with no key storage

Like the *previous one*, this example system is composed of two partitions separated by a security boundary. Thus this architecture also provides *cryptoprocessor isolation*. Unlike the previous architecture, in this case, the cryptoprocessor does not have any secure persistent storage that could be used to store application keys.

If the cryptoprocessor is not capable of storing any cryptographic material, then there is little use for a separate cryptoprocessor, since all data would have to be imported by the application.

The cryptoprocessor can provide useful services if it is able to store at least one key. This may be a hardware unique key that is burnt to one-time programmable memory during the manufacturing of the device. This key may be used for one or more purposes including:

- Encrypt and authenticate data whose storage is delegated to the application processor.
- Communicate with a paired device.
- Allow the application to perform operations with keys that are derived from the hardware unique key.

Multi-client cryptoprocessor

This is an expanded variant of the *cryptographic token plus application architecture*. In this variant, the cryptoprocessor serves multiple applications that are mutually untrustworthy. This architecture provides *caller isolation*.

In this architecture, API calls are translated to remote procedure calls which encode the identity of the client application. The cryptoprocessor carefully segments its internal storage to ensure that a client's data is never leaked to another client.

Multi-cryptoprocessor architecture

In this example, the system includes multiple cryptoprocessors. Some reasons to have multiple cryptoprocessors include:

- Different compromises between security and performance for different keys. Typically this means a cryptoprocessor running on the same hardware as the main application and processing short-term secrets, and a secure element or similar separate chip that retains long-term secrets.
- Independent provisioning of certain secrets.
- A combination of a non-removable cryptoprocessor and removable ones (e.g. a smartcard or HSM).

The keystore implementation needs to dispatch each request to the correct processor. All requests involving a non-extractable key must be processed in the cryptoprocessor that holds that key. Other requests may target a cryptoprocessor based on parameters supplied by the application or based on considerations such as performance inside the implementation. A typical choice for dispatch is for the implementation to define ranges of key slot numbers, such that each range corresponds to one of the cryptoprocessors.

CHAPTER

FIVE

LIBRARY CONVENTIONS

Error handling

Almost all functions return a status indication of type `psa_status_t`. This is an enumeration of integer values, with 0 (`PSA_SUCCESS`) conveying successful operation and other values indicating errors. The exception is data structure accessor functions that cannot fail: such functions may return `void` or a data value.

All function calls must be implemented atomically:

- When a function returns a type other than `psa_status_t`, the requested action has been carried out.
- When a function returns the status `PSA_SUCCESS`, the requested action has been carried out.
- When a function returns another status of type `psa_status_t`, no action has been carried out. The content of output parameters is undefined, but otherwise the state of the system has not changed except has described below.

Generally speaking, functions that modify the system state (modifying the content of a key slot or its metadata) must leave the system state unchanged if they return an error code. However, there are a few exceptions to this general principle in exceptional conditions:

- The status `PSA_ERROR_BAD_STATE` indicates that a supplied parameter was not in a valid state for the requested action. The corresponding object may have been modified by the call and must not be used for any further action except to abort the corresponding object.
- The status `PSA_ERROR_INSUFFICIENT_CAPACITY` indicates that a generator has reached its maximum capacity. The generator object may have been modified by the call and any further attempt to read from the generator will return `PSA_ERROR_INSUFFICIENT_CAPACITY`.
- The status `PSA_ERROR_COMMUNICATION_FAILURE` indicates that the communication between the application and the cryptoprocessor has broken down. In this case, it may be impossible to know whether the action has been carried out. Upon detection of a communication failure, the cryptoprocessor must either finish carrying out the request or roll back to the original state, but the application may not be able to find out which of these two possibilities happened.
- The statuses `PSA_ERROR_STORAGE_FAILURE`, `PSA_ERROR_HARDWARE_FAILURE` and `PSA_ERROR_TAMPERING_DETECTED` may indicate data corruption in the system state. Thus, when a function returns one of these statuses, the system state may have changed compared to before the function call, even though the function call failed.
- Some system state cannot be rolled back, for example the internal state of the random number generator, or the content of logs if the implementation keeps access logs.

Unless otherwise documented, the content of output parameters is not defined when a function returns a status other than `PSA_SUCCESS`. Implementations should set output parameters to safe defaults to avoid leaking confidential data and to limit the risks in case an application does not properly handle all errors.

Parameter conventions

Pointer conventions

Unless explicitly stated in the documentation of a function, all pointers must be valid pointers to an object of the specified type.

A parameter is considered a *buffer* if it points to an array of bytes. A buffer parameter always has the type `uint8_t *` or `const uint8_t *`, and always has an associated parameter indicating the size of the array. Note that a parameter of type `void *` is never considered a buffer.

All parameters of pointer type must be valid, non-null pointers unless the pointer is to a buffer of length 0 or the function's documentation explicitly describes the behavior when the pointer is null. Implementations where a null pointer dereference usually aborts the application, passing `NULL` as a function parameter where a null pointer is not allowed should abort the caller in the habitual manner.

Pointers to input parameters may be in read-only memory. Output parameters must be in writable memory. Output parameters that are not buffers must also be readable, and the implementation must be able to write to a non-buffer output parameter and read back the same value, as explained in the section *"Stability of parameters"*.

Input buffer sizes

For input buffers, the parameter convention is:

- `const uint8_t *foo`: pointer to the first byte of the data. The pointer may be invalid if the buffer size is 0.
- `size_t foo_length`: size of the buffer in bytes.

The interface never uses input-output buffers.

Output buffer sizes

For output buffers, the parameter convention is:

- `uint8_t *foo`: pointer to the first byte of the data. The pointer may be invalid if the buffer size is 0.
- `size_t foo_size`: the size of the buffer in bytes.
- `size_t *foo_length`: on successful return, contains the length of the output in bytes.

The content of the data buffer and of `*foo_length` on error is unspecified unless explicitly mentioned in the function description. They may be unmodified or set to a safe default. On successful completion, the content of the buffer between the offsets `*foo_length` and `foo_size` is also unspecified.

Functions return `PSA_ERROR_BUFFER_TOO_SMALL` if the buffer size is insufficient to carry out the requested operation. The interface defines macros to calculate a sufficient buffer size for each operation that has an output buffer. These macros return compile-time constants if their arguments are compile-time constants, so they are suitable for static or stack allocation. Refer to individual functions' documentation for the associated output size macro.

Some functions always return exactly as much data as the size of the output buffer. In this case, the parameter convention changes to:

- `uint8_t *foo`: pointer to the first byte of the output. The pointer may be invalid if the buffer size is 0.
- `size_t foo_length`: the number of bytes to return in `foo` if successful.

Overlap between parameters

Output parameters that are not buffers may not overlap with any input buffer or with any other output parameter. Otherwise the behavior is undefined.

Output buffers may overlap with input buffers. If this happens, the implementation must return the same result as if the buffers did not overlap. In other words, the implementation must behave as if it had copied all the inputs into temporary memory, as far as the result is concerned. However application developers should note that overlap between parameters may affect the performance of a function call. Overlap may also affect the security of how memory is managed if the buffer is located in memory that the caller shares with another security context, as described in the section *"Stability of parameters"*.

Stability of parameters

In some environments, it is possible for the content of a parameter to change while a function is executing. It may also be possible for the content of an output parameter to be read before the function terminates. This can happen if the application is multithreaded. In some implementations, memory can be shared between security contexts, for example, between tasks in a multitasking operating system, or between a user land task and the kernel, or between the non-secure world and the secure world of a trusted execution environment. This section describes what implementations need or need not guarantee in such cases.

Parameters that are not buffers are assumed to be under the caller's full control. In a shared memory environment, this means that the parameter must be in memory that is exclusively accessible by the application. In a multithreaded environment, this means that the the parameter may not be modified during the execution and the value of an output parameter is undetermined until the function returns. The implementation may read an input parameter that is not a buffer multiple times and expect to read the same data. The implementation may write to an output parameter that is not a buffer and expect to read back the value that it last wrote. The implementation has the same permissions on buffers that overlap with a buffer in the opposite direction.

In an environment with multiple threads or with shared memory, the implementation shall access non-overlapping buffer parameters carefully in order to prevent any unsafety if the content of the buffer is modified or observed during the execution of the function. In an input buffer that does not overlap with an output buffer, the implementation shall read each byte of the input at most once. The implementation shall not read from an output buffer that does not overlap with an input buffer. Additionally, the implementation shall not write data to a non-overlapping output buffer if this data is potentially confidential and the implementation has not yet verified that outputting this data is authorized.

Key types and algorithms

Types and cryptographic keys and cryptographic algorithms are encoded separately. Each is encoded using an integral type, respectively `psa_key_type_t` and `psa_algorithm_t`.

There is some overlap in the information conveyed through keys and algorithms. Both types include enough information so that the meaning of an algorithm type value does not depend on what type of key it is used with and vice versa. However, the particular instance of an algorithm may depend on the key type. For example, the algorithm `PSA_ALG_GCM` can be instantiated as any AEAD algorithm using the GCM mode over a block cipher; the underlying block cipher is determined by the key type.

Key types do not encode the key size. For example AES-128, AES-192 and AES-256 share a key type `PSA_KEY_TYPE_AES`.

Structure of key and algorithm types

Both types use a partial bitmask structure which allows analyzing and building values from parts. However the interface defines constants so that applications do not need to depend on the encoding and an implementation may care about the encoding only for code size optimization.

The encodings follows a few conventions:

- The highest bit is a vendor flag. Values with this bit clear are reserved for values defined by this specification, and values with this bit set will not be defined by this specification.
- The next few highest bits indicate the corresponding algorithm category: hash, MAC, symmetric cipher, asymmetric encryption, etc.
- The following bits identify a family of algorithms in a category-dependent manner.
- In some categories and algorithm families, the lowest-order bits indicate a variant in a systematic way. For example, algorithm families that are parametrized around a hash function encode the hash in the 8 lowest bits.

Concurrent calls

In some environments, it is possible for an application to make calls to the PSA crypto API in separate threads. In such an environment, concurrent calls SHALL be performed correctly, as if the calls had been executed in sequence, provided that they obey the following constraints:

- There must not be any overlap between an output parameter of one call and an input or output parameter of another call. (Overlap between input parameters is permitted.)
- If a call modifies a key slot, then no other call must modify or use that key slot. *Using*, in this context, includes all functions of multipart operations using the key. (Concurrent calls that merely use the same key are permitted.)
- Concurrent calls may not use the same operation or generator object.

If any of these constraints is violated, the behavior is undefined.

Individual implementations may provide additional guarantees.

CHAPTER

SIX

IMPLEMENTATION CONSIDERATIONS

Implementation-specific aspects of the interface

Implementation profile

Implementations may implement a subset of the API and a subset of the available algorithms. The implemented subset is known as the implementation's profile. The documentation of each implementation SHALL document what profile it implements. Companion documents to this specification will define standard profiles.

Implementation-specific types

This specification defines some platform-specific types to represent data structures whose content depends on the implementation. These types are C `struct` types. In the associated header files, `crypto.h` declares the `struct` tags and `crypto_struct.h` provides a definition for the structures.

Implementation-specific macros

Some macros compute a result based on an algorithm or a key type. This specification provides a sample implementation of these macros which works for all standard types. If an implementation defines vendor-specific algorithms or key types, it must provide an implementation of such macros that takes all relevant algorithms and types into account. Conversely, an implementation that does not support a certain algorithm or key type may define such macros in a simpler way that does not take unsupported argument values into account.

Some macros define the minimum sufficient output buffer size for certain functions. In some cases, implementations are allowed to require a buffer size that is larger than the theoretical minimum. Implementations SHALL define minimum-size macros in such a way as to guarantee that a buffer of the resulting size is sufficient for the output of the corresponding function. Refer to each macro's documentation for the applicable requirements.

Porting to a platform

Platform assumptions

This specification is designed for a C89 platform. The interface is defined in terms of C macros, functions and objects.

This specification assumes 8-bit bytes. In this specification, "byte" and "octet" are used synonymously.

Platform-specific types

The specification makes use of some platform-specific types which should be defined in `crypto_platform.h` (possibly via a header included by this file). `crypto_platform.h` must define the following types:

- `uint8_t`, `uint16_t`, `uint32_t`: unsigned integer types with 8, 16 and 32 value bits respectively. These may be the types defined by the C99 header `stdint.h`.
- `psa_key_handle_t`: an unsigned integer type of the implementation's choice.

Cryptographic hardware support

Implementations are encouraged to make use of hardware accelerators where available. A future version of this specification will define a function interface to call drivers for hardware accelerators and external cryptographic hardware.

Security requirements and recommendations

Error detection

Implementations that provide isolation between the caller and the cryptography processing environment SHALL validate parameters to ensure that the cryptography processing environment is protected from attacks caused by passing invalid parameters.

Even implementations that do not provide isolation should strive to detect bad parameters and fail safe as much as possible.

Memory cleanup

Implementations SHALL wipe all sensitive data from memory when it is no longer used. Implementations should wipe sensitive data as soon as possible. In any case, all temporary data (such as stack buffers) used during the execution of a function shall be wiped before the function returns, and all data associated with an object (such as a multipart operation) shall be wiped at the latest when the object becomes inactive (for example, when a multipart operation is aborted).

The rationale for this non-functional requirement is to minimize the impact if the system is compromised. If sensitive data is wiped immediately after use, a data leak only leaks data that is currently in use, but does not compromise past data.

Safe outputs on error

Implementations SHALL ensure that no confidential data is written to output parameters before validating that the disclosure of this confidential data is authorized. This requirement is especially important on implementations where the caller may share memory with another security context as described in the section *"Stability of parameters"*.

In most cases, this specification does not define the content of output parameters when an error occurs. Implementations should ensure that the content of output parameters is as safe as possible in case it ends up being used due to an application flaw or a data leak. In particular, implementations should avoid placing partial output in output buffers if an action is interrupted. The definition of "safe" is left up to each implementation as different environments may require different compromises between implementation complexity, overall robustness and performance. Some common strategies include leaving output parameters unchanged in case of errors, or zeroing them out.

Attack resistance

Cryptographic code tends to manipulate high-value secrets from which other secrets can be unlocked. As such it is a high-value target for attacks. A vast body of literature exists on attack types such as side channel attacks and glitch attacks. Typical side channels include timing, cache access patterns, branch prediction access patterns, power consumption, radio emissions and more.

This specification does not place any particular requirement regarding attack resistance. Implementers should consider the attack resistance that is expected in each use case and design their implementation accordingly. Security standards that define targets for attack resistance may be applicable in certain use cases.

Other implementation considerations

Philosophy of resource management

This specification allows most functions to return `PSA_ERROR_INSUFFICIENT_MEMORY`. This gives implementations the freedom to manage memory as they please.

Nonetheless the interface is designed to allow conservative strategies for memory management. In particular, an implementation may avoid dynamic memory allocation altogether by obeying certain restrictions:

- Pre-allocate memory for a predefined number of key slots, each with sufficient memory for all key types that can be stored in that slot.
- For multipart operations, in an implementation without isolation, place all the data that needs to be carried over from one step to the next in the operation object. The application is then fully in control of how memory is allocated for the operation.
- In an implementation with isolation, pre-allocate memory for a predefined number of operations inside the cryptoprocessor.

CHAPTER

SEVEN

USAGE CONSIDERATIONS

Security recommendations

Always check for errors

Most functions in this API can return errors. All functions that can fail have the return type `psa_status_t`. A few functions cannot fail, and thus return `void` or some other type.

If an error occurs, unless otherwise specified, the content of output parameters is undefined and must not be used.

Some common causes of errors include:

- In implementations where the keys are stored and processed in a separate environment from the application, all functions that need to access the cryptography processing environment may fail due to an error in the communication between the two environments.
- If an algorithm is implemented with a hardware accelerator which is logically separate from the application processor, this accelerator may fail even when the application processor keeps running normally.
- All functions may fail due to a lack of resources, although some implementations may guarantee that certain functions always have sufficient memory.
- All functions that access persistent keys may fail due to a storage failure.
- All functions that require randomness may fail due to a lack of entropy. Implementations are encouraged to seed the random generator with sufficient entropy during the execution of `psa_crypto_init`; however some security standards require periodic reseeding from a hardware random generator which can fail.

Shared memory and concurrency

Some environment allow applications to be multithreaded. In some environments, applications may share memory with a different security context. In such environments, applications must be written carefully to avoid data corruption or leakage. This specification requires the application to obey certain constraints.

In general, this API allows either one writer or any number of simultaneous readers on any given object. In other words, if two or more calls access the same object concurrently, the behavior is well-defined only if all the calls are only reading from the object and do not modify it. Read accesses include reading memory via input parameters and reading key store content by using a key. For more details, refer to the section *"Concurrent calls"*.

If an application shared memory with another security contexts, it may pass shared memory blocks as input buffers or output buffers, but not as non-buffer parameters. For more details from the implementation's perspective, refer to the section *"Stability of parameters"*.

Cleaning up after use

In order to minimize the impact if the system is compromised, applications should wipe all sensitive data from memory when it is no longer used. This way, a data leak only leaks data that is currently in use, but does not compromise past data.

Wiping sensitive data includes:

- Clearing temporary buffers in the stack or on the heap.
- Aborting operations if they will not be finished.
- Destroying key slots that are no longer used.

CHAPTER

EIGHT

IMPLEMENTATION-SPECIFIC DEFINITIONS

psa_key_handle_t (type)

```
typedef _unsigned_integral_type_ psa_key_handle_t;
```

Key handle.

This type represents open handles to keys. It must be an unsigned integral type. The choice of type is implementation-dependent.

0 is not a valid key handle. How other handle values are assigned is implementation-dependent.

CHAPTER

NINE

LIBRARY INITIALIZATION

psa_crypto_init (function)

```
psa_status_t psa_crypto_init(void);
```

Returns: psa_status_t

PSA_SUCCESS

PSA_ERROR_INSUFFICIENT_MEMORY

PSA_ERROR_COMMUNICATION_FAILURE

PSA_ERROR_HARDWARE_FAILURE

PSA_ERROR_TAMPERING_DETECTED

PSA_ERROR_INSUFFICIENT_ENTROPY

Description: Library initialization.

Applications must call this function before calling any other function in this module.

Applications may call this function more than once. Once a call succeeds, subsequent calls are guaranteed to succeed.

If the application calls other functions before calling *psa_crypto_init()*, the behavior is undefined. Implementations are encouraged to either perform the operation as if the library had been initialized or to return *PSA_ERROR_BAD_STATE* or some other applicable error. In particular, implementations should not return a success status if the lack of initialization may have security implications, for example due to improper seeding of the random number generator.

CHAPTER

TEN

KEY POLICIES

psa_key_policy_t (type)

```
typedef struct psa_key_policy_s psa_key_policy_t;
```

The type of the key policy data structure.

Before calling any function on a key policy, the application must initialize it by any of the following means:

- Set the structure to all-bits-zero, for example:

```
psa_key_policy_t policy;
memset(&policy, 0, sizeof(policy));
```

- Initialize the structure to logical zero values, for example:

```
psa_key_policy_t policy = {0};
```

- Initialize the structure to the initializer `PSA_KEY_POLICY_INIT`, for example:

```
psa_key_policy_t policy = PSA_KEY_POLICY_INIT;
```

- Assign the result of the function psa_key_policy_init() to the structure, for example:

```
psa_key_policy_t policy;
policy = psa_key_policy_init();
```

This is an implementation-defined `struct`. Applications should not make any assumptions about the content of this structure except as directed by the documentation of a specific implementation.

psa_key_usage_t (type)

```
typedef uint32_t psa_key_usage_t;
```

Encoding of permitted usage on a key.

PSA_KEY_POLICY_INIT (macro)

```
#define PSA_KEY_POLICY_INIT  {0}
```

This macro returns a suitable initializer for a key policy object of type `psa_key_policy_t`.

PSA_KEY_USAGE_EXPORT (macro)

```
#define PSA_KEY_USAGE_EXPORT  ((psa_key_usage_t)0x00000001)
```

Whether the key may be exported.

A public key or the public part of a key pair may always be exported regardless of the value of this permission flag.

If a key does not have export permission, implementations shall not allow the key to be exported in plain form from the cryptoprocessor, whether through `psa_export_key()` or through a proprietary interface. The key may however be exportable in a wrapped form, i.e. in a form where it is encrypted by another key.

PSA_KEY_USAGE_ENCRYPT (macro)

```
#define PSA_KEY_USAGE_ENCRYPT  ((psa_key_usage_t)0x00000100)
```

Whether the key may be used to encrypt a message.

This flag allows the key to be used for a symmetric encryption operation, for an AEAD encryption-and-authentication operation, or for an asymmetric encryption operation, if otherwise permitted by the key's type and policy.

For a key pair, this concerns the public key.

PSA_KEY_USAGE_DECRYPT (macro)

```
#define PSA_KEY_USAGE_DECRYPT  ((psa_key_usage_t)0x00000200)
```

Whether the key may be used to decrypt a message.

This flag allows the key to be used for a symmetric decryption operation, for an AEAD decryption-and-verification operation, or for an asymmetric decryption operation, if otherwise permitted by the key's type and policy.

For a key pair, this concerns the private key.

PSA_KEY_USAGE_SIGN (macro)

```
#define PSA_KEY_USAGE_SIGN  ((psa_key_usage_t)0x00000400)
```

Whether the key may be used to sign a message.

This flag allows the key to be used for a MAC calculation operation or for an asymmetric signature operation, if otherwise permitted by the key's type and policy.

For a key pair, this concerns the private key.

PSA_KEY_USAGE_VERIFY (macro)

```
#define PSA_KEY_USAGE_VERIFY  ((psa_key_usage_t)0x00000800)
```

Whether the key may be used to verify a message signature.

This flag allows the key to be used for a MAC verification operation or for an asymmetric signature verification operation, if otherwise permitted by by the key's type and policy.

For a key pair, this concerns the public key.

PSA_KEY_USAGE_DERIVE (macro)

```
#define PSA_KEY_USAGE_DERIVE  ((psa_key_usage_t)0x00001000)
```

Whether the key may be used to derive other keys.

psa_key_policy_init (function)

```
psa_key_policy_t psa_key_policy_init(void);
```

Returns: psa_key_policy_t **Description:** Return an initial value for a key policy that forbids all usage of the key.

psa_key_policy_set_usage (function)

```
void psa_key_policy_set_usage(psa_key_policy_t *policy,
                              psa_key_usage_t usage,
                              psa_algorithm_t alg);
```

Parameters:

policy The key policy to modify. It must have been initialized as per the documentation for *psa_key_policy_t*.

usage The permitted uses for the key.

alg The algorithm that the key may be used for.

Returns: void **Description:** Set the standard fields of a policy structure.

Note that this function does not make any consistency check of the parameters. The values are only checked when applying the policy to a key slot with *psa_set_key_policy()*.

psa_key_policy_get_usage (function)

```
psa_key_usage_t psa_key_policy_get_usage(const psa_key_policy_t *policy);
```

Parameters:

policy The policy object to query.

Returns: psa_key_usage_t

The permitted uses for a key with this policy. **Description:** Retrieve the usage field of a policy structure.

psa_key_policy_get_algorithm (function)

```
psa_algorithm_t psa_key_policy_get_algorithm(const psa_key_policy_t *policy);
```

Parameters:

policy The policy object to query.

Returns: psa_algorithm_t

The permitted algorithm for a key with this policy. **Description:** Retrieve the algorithm field of a policy structure.

psa_set_key_policy (function)

```
psa_status_t psa_set_key_policy(psa_key_handle_t handle,
                                const psa_key_policy_t *policy);
```

Parameters:

handle Handle to the key whose policy is to be changed.

policy The policy object to query.

Returns: psa_status_t

PSA_SUCCESS Success. If the key is persistent, it is implementation-defined whether the policy has been saved to persistent storage. Implementations may defer saving the policy until the key material is created.

PSA_ERROR_INVALID_HANDLE

PSA_ERROR_OCCUPIED_SLOT

PSA_ERROR_NOT_SUPPORTED

PSA_ERROR_INVALID_ARGUMENT

PSA_ERROR_COMMUNICATION_FAILURE

PSA_ERROR_HARDWARE_FAILURE

PSA_ERROR_TAMPERING_DETECTED

PSA_ERROR_BAD_STATE The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

Description: Set the usage policy on a key slot.

This function must be called on an empty key slot, before importing, generating or creating a key in the slot. Changing the policy of an existing key is not permitted.

Implementations may set restrictions on supported key policies depending on the key type and the key slot.

psa_get_key_policy (function)

```
psa_status_t psa_get_key_policy(psa_key_handle_t handle,
                                psa_key_policy_t *policy);
```

Parameters:

handle Handle to the key slot whose policy is being queried.

policy On success, the key's policy.

Returns: psa_status_t

PSA_SUCCESS

PSA_ERROR_INVALID_HANDLE

PSA_ERROR_COMMUNICATION_FAILURE

PSA_ERROR_HARDWARE_FAILURE

PSA_ERROR_TAMPERING_DETECTED

PSA_ERROR_BAD_STATE The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

Description: Get the usage policy for a key slot.

CHAPTER

ELEVEN

KEY MANAGEMENT

psa_get_key_lifetime (function)

```
psa_status_t psa_get_key_lifetime(psa_key_handle_t handle,
                                  psa_key_lifetime_t *lifetime);
```

Parameters:

handle Handle to query.

lifetime On success, the lifetime value.

Returns: psa_status_t

PSA_SUCCESS Success.

PSA_ERROR_INVALID_HANDLE

PSA_ERROR_COMMUNICATION_FAILURE

PSA_ERROR_HARDWARE_FAILURE

PSA_ERROR_TAMPERING_DETECTED

PSA_ERROR_BAD_STATE The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

Description: Retrieve the lifetime of an open key.

psa_allocate_key (function)

```
psa_status_t psa_allocate_key(psa_key_handle_t *handle);
```

Parameters:

handle On success, a handle to a volatile key slot.

Returns: psa_status_t

PSA_SUCCESS Success. The application can now use the value of *handle to access the newly allocated key slot.

PSA_ERROR_INSUFFICIENT_MEMORY There was not enough memory, or the maximum number of key slots has been reached.

Description: Allocate a key slot for a transient key, i.e. a key which is only stored in volatile memory.

The allocated key slot and its handle remain valid until the application calls *psa_close_key()* or *psa_destroy_key()* or until the application terminates.

`psa_open_key` (function)

```
psa_status_t psa_open_key(psa_key_lifetime_t lifetime,
                          psa_key_id_t id,
                          psa_key_handle_t *handle);
```

Parameters:

`lifetime` The lifetime of the key. This designates a storage area where the key material is stored. This must not be *`PSA_KEY_LIFETIME_VOLATILE`*.

`id` The persistent identifier of the key.

`handle` On success, a handle to a key slot which contains the data and metadata loaded from the specified persistent location.

Returns: `psa_status_t`

`PSA_SUCCESS` Success. The application can now use the value of `*handle` to access the newly allocated key slot.

`PSA_ERROR_INSUFFICIENT_MEMORY`

`PSA_ERROR_EMPTY_SLOT`

`PSA_ERROR_INVALID_ARGUMENT` `lifetime` is invalid, for example *`PSA_KEY_LIFETIME_VOLATILE`*.

`PSA_ERROR_INVALID_ARGUMENT` `id` is invalid for the specified lifetime.

`PSA_ERROR_NOT_SUPPORTED` `lifetime` is not supported.

`PSA_ERROR_NOT_PERMITTED` The specified key exists, but the application does not have the permission to access it. Note that this specification does not define any way to create such a key, but it may be possible through implementation-specific means.

Description: Open a handle to an existing persistent key.

Open a handle to a key which was previously created with *`psa_create_key()`*.

`psa_create_key` (function)

```
psa_status_t psa_create_key(psa_key_lifetime_t lifetime,
                            psa_key_id_t id,
                            psa_key_handle_t *handle);
```

Parameters:

`lifetime` The lifetime of the key. This designates a storage area where the key material is stored. This must not be *`PSA_KEY_LIFETIME_VOLATILE`*.

`id` The persistent identifier of the key.

`handle` On success, a handle to the newly created key slot. When key material is later created in this key slot, it will be saved to the specified persistent location.

Returns: `psa_status_t`

`PSA_SUCCESS` Success. The application can now use the value of `*handle` to access the newly allocated key slot.

`PSA_ERROR_INSUFFICIENT_MEMORY`

`PSA_ERROR_INSUFFICIENT_STORAGE`

PSA_ERROR_OCCUPIED_SLOT There is already a key with the identifier `id` in the storage area designated by `lifetime`.

PSA_ERROR_INVALID_ARGUMENT `lifetime` is invalid, for example *PSA_KEY_LIFETIME_VOLATILE*.

PSA_ERROR_INVALID_ARGUMENT `id` is invalid for the specified lifetime.

PSA_ERROR_NOT_SUPPORTED `lifetime` is not supported.

PSA_ERROR_NOT_PERMITTED `lifetime` is valid, but the application does not have the permission to create a key there.

Description: Create a new persistent key slot.

Create a new persistent key slot and return a handle to it. The handle remains valid until the application calls *psa_close_key()* or terminates. The application can open the key again with *psa_open_key()* until it removes the key by calling *psa_destroy_key()*.

psa_close_key (function)

```
psa_status_t psa_close_key(psa_key_handle_t handle);
```

Parameters:

handle The key handle to close.

Returns: `psa_status_t`

PSA_SUCCESS

PSA_ERROR_INVALID_HANDLE

PSA_ERROR_COMMUNICATION_FAILURE

Description: Close a key handle.

If the handle designates a volatile key, destroy the key material and free all associated resources, just like *psa_destroy_key()*.

If the handle designates a persistent key, free all resources associated with the key in volatile memory. The key slot in persistent storage is not affected and can be opened again later with *psa_open_key()*.

If the key is currently in use in a multipart operation, the multipart operation is aborted.

CHAPTER

TWELVE

KEY IMPORT AND EXPORT

psa_import_key (function)

```
psa_status_t psa_import_key(psa_key_handle_t handle,
                            psa_key_type_t type,
                            const uint8_t *data,
                            size_t data_length);
```

Parameters:

handle Handle to the slot where the key will be stored. It must have been obtained by calling *psa_allocate_key()* or *psa_create_key()* and must not contain key material yet.

type Key type (a PSA_KEY_TYPE_XXX value). On a successful import, the key slot will contain a key of this type.

data Buffer containing the key data. The content of this buffer is interpreted according to type. It must contain the format described in the documentation of *psa_export_key()* or *psa_export_public_key()* for the chosen type.

data_length Size of the data buffer in bytes.

Returns: psa_status_t

PSA_SUCCESS Success. If the key is persistent, the key material and the key's metadata have been saved to persistent storage.

PSA_ERROR_INVALID_HANDLE

PSA_ERROR_NOT_SUPPORTED The key type or key size is not supported, either by the implementation in general or in this particular slot.

PSA_ERROR_INVALID_ARGUMENT The key slot is invalid, or the key data is not correctly formatted.

PSA_ERROR_OCCUPIED_SLOT There is already a key in the specified slot.

PSA_ERROR_INSUFFICIENT_MEMORY

PSA_ERROR_INSUFFICIENT_STORAGE

PSA_ERROR_COMMUNICATION_FAILURE

PSA_ERROR_STORAGE_FAILURE

PSA_ERROR_HARDWARE_FAILURE

PSA_ERROR_TAMPERING_DETECTED

PSA_ERROR_BAD_STATE The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

Description: Import a key in binary format.

This function supports any output from `psa_export_key()`. Refer to the documentation of `psa_export_public_key()` for the format of public keys and to the documentation of `psa_export_key()` for the format for other key types.

This specification supports a single format for each key type. Implementations may support other formats as long as the standard format is supported. Implementations that support other formats should ensure that the formats are clearly unambiguous so as to minimize the risk that an invalid input is accidentally interpreted according to a different format.

`psa_destroy_key` (function)

```
psa_status_t psa_destroy_key(psa_key_handle_t handle);
```

Parameters:

`handle` Handle to the key slot to erase.

Returns: `psa_status_t`

`PSA_SUCCESS` The slot's content, if any, has been erased.

`PSA_ERROR_NOT_PERMITTED` The slot holds content and cannot be erased because it is read-only, either due to a policy or due to physical restrictions.

`PSA_ERROR_INVALID_HANDLE`

`PSA_ERROR_COMMUNICATION_FAILURE` There was an failure in communication with the cryptoprocessor. The key material may still be present in the cryptoprocessor.

`PSA_ERROR_STORAGE_FAILURE` The storage is corrupted. Implementations shall make a best effort to erase key material even in this stage, however applications should be aware that it may be impossible to guarantee that the key material is not recoverable in such cases.

`PSA_ERROR_TAMPERING_DETECTED` An unexpected condition which is not a storage corruption or a communication failure occurred. The cryptoprocessor may have been compromised.

`PSA_ERROR_BAD_STATE` The library has not been previously initialized by `psa_crypto_init()`. It is implementation-dependent whether a failure to initialize results in this error code.

Description: Destroy a key.

This function destroys the content of the key slot from both volatile memory and, if applicable, non-volatile storage. Implementations shall make a best effort to ensure that any previous content of the slot is unrecoverable.

This function also erases any metadata such as policies and frees all resources associated with the key.

If the key is currently in use in a multipart operation, the multipart operation is aborted.

`psa_get_key_information` (function)

```
psa_status_t psa_get_key_information(psa_key_handle_t handle,
                                     psa_key_type_t *type,
                                     size_t *bits);
```

Parameters:

`handle` Handle to the key slot to query.

type On success, the key type (a PSA_KEY_TYPE_XXX value). This may be a null pointer, in which case the key type is not written.

bits On success, the key size in bits. This may be a null pointer, in which case the key size is not written.

Returns: psa_status_t

PSA_SUCCESS

PSA_ERROR_INVALID_HANDLE

PSA_ERROR_EMPTY_SLOT The handle is to a key slot which does not contain key material yet.

PSA_ERROR_COMMUNICATION_FAILURE

PSA_ERROR_HARDWARE_FAILURE

PSA_ERROR_TAMPERING_DETECTED

PSA_ERROR_BAD_STATE The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

Description: Get basic metadata about a key.

psa_set_key_domain_parameters (function)

```
psa_status_t psa_set_key_domain_parameters(psa_key_handle_t handle,
                                           psa_key_type_t type,
                                           const uint8_t *data,
                                           size_t data_length);
```

Parameters:

handle Handle to the slot where the key will be stored. This must be a valid slot for a key of the chosen type: it must have been obtained by calling *psa_allocate_key()* or *psa_create_key()* with the correct type and with a maximum size that is compatible with data. It must not contain key material yet.

type Key type (a PSA_KEY_TYPE_XXX value). When subsequently creating key material into handle, the type must be compatible.

data Buffer containing the key domain parameters. The content of this buffer is interpreted according to type. of *psa_export_key()* or *psa_export_public_key()* for the chosen type.

data_length Size of the data buffer in bytes.

Returns: psa_status_t

PSA_SUCCESS

PSA_ERROR_INVALID_HANDLE

PSA_ERROR_OCCUPIED_SLOT There is already a key in the specified slot.

PSA_ERROR_INVALID_ARGUMENT

PSA_ERROR_COMMUNICATION_FAILURE

PSA_ERROR_HARDWARE_FAILURE

PSA_ERROR_TAMPERING_DETECTED

PSA_ERROR_BAD_STATE The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

Description: Set domain parameters for a key.

Some key types require additional domain parameters to be set before import or generation of the key. The domain parameters can be set with this function or, for key generation, through the `extra` parameter of *psa_generate_key()*.

The format for the required domain parameters varies by the key type.

- For DSA public keys (*PSA_KEY_TYPE_DSA_PUBLIC_KEY*), the `Dss-Parms` format as defined by RFC 3279 §2.3.2.

```
Dss-Parms ::= SEQUENCE  {
   p       INTEGER,
   q       INTEGER,
   g       INTEGER
}
```

- For Diffie-Hellman key exchange keys (*PSA_KEY_TYPE_DH_PUBLIC_KEY*), the `DomainParameters` format as defined by RFC 3279 §2.3.3.

```
DomainParameters ::= SEQUENCE {
   p               INTEGER,                    -- odd prime, p=jq +1
   g               INTEGER,                    -- generator, g
   q               INTEGER,                    -- factor of p-1
   j               INTEGER OPTIONAL,           -- subgroup factor
   validationParms ValidationParms OPTIONAL
}
ValidationParms ::= SEQUENCE {
   seed            BIT STRING,
   pgenCounter     INTEGER
}
```

psa_get_key_domain_parameters (function)

```
psa_status_t psa_get_key_domain_parameters(psa_key_handle_t handle,
                                           uint8_t *data,
                                           size_t data_size,
                                           size_t *data_length);
```

Parameters:

handle Handle to the key to get domain parameters from.

data On success, the key domain parameters.

data_size Size of the `data` buffer in bytes.

data_length On success, the number of bytes that make up the key domain parameters data.

Returns: `psa_status_t`

PSA_SUCCESS

PSA_ERROR_INVALID_HANDLE

PSA_ERROR_EMPTY_SLOT There is no key in the specified slot.

PSA_ERROR_INVALID_ARGUMENT

PSA_ERROR_NOT_SUPPORTED

PSA_ERROR_COMMUNICATION_FAILURE

PSA_ERROR_HARDWARE_FAILURE

PSA_ERROR_TAMPERING_DETECTED

PSA_ERROR_BAD_STATE The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

Description: Get domain parameters for a key.

Get the domain parameters for a key with this function, if any. The format of the domain parameters written to `data` is specified in the documentation for *psa_set_key_domain_parameters()*.

psa_export_key (function)

```
psa_status_t psa_export_key(psa_key_handle_t handle,
                            uint8_t *data,
                            size_t data_size,
                            size_t *data_length);
```

Parameters:

handle Handle to the key to export.

data Buffer where the key data is to be written.

data_size Size of the `data` buffer in bytes.

data_length On success, the number of bytes that make up the key data.

Returns: `psa_status_t`

PSA_SUCCESS

PSA_ERROR_INVALID_HANDLE

PSA_ERROR_EMPTY_SLOT

PSA_ERROR_NOT_PERMITTED

PSA_ERROR_NOT_SUPPORTED

PSA_ERROR_BUFFER_TOO_SMALL The size of the `data` buffer is too small. You can determine a sufficient buffer size by calling *PSA_KEY_EXPORT_MAX_SIZE*(`type`, `bits`) where `type` is the key type and `bits` is the key size in bits.

PSA_ERROR_COMMUNICATION_FAILURE

PSA_ERROR_HARDWARE_FAILURE

PSA_ERROR_TAMPERING_DETECTED

PSA_ERROR_BAD_STATE The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

Description: Export a key in binary format.

The output of this function can be passed to *psa_import_key()* to create an equivalent object.

If the implementation of *psa_import_key()* supports other formats beyond the format specified here, the output from *psa_export_key()* must use the representation specified here, not the original representation.

For standard key types, the output format is as follows:

- For symmetric keys (including MAC keys), the format is the raw bytes of the key.

- For DES, the key data consists of 8 bytes. The parity bits must be correct.
- For Triple-DES, the format is the concatenation of the two or three DES keys.
- For RSA key pairs (*PSA_KEY_TYPE_RSA_KEYPAIR*), the format is the non-encrypted DER encoding of the representation defined by PKCS#1 (RFC 8017) as `RSAPrivateKey`, version 0.

```
RSAPrivateKey ::= SEQUENCE {
    version             INTEGER,  -- must be 0
    modulus             INTEGER,  -- n
    publicExponent      INTEGER,  -- e
    privateExponent     INTEGER,  -- d
    prime1              INTEGER,  -- p
    prime2              INTEGER,  -- q
    exponent1           INTEGER,  -- d mod (p-1)
    exponent2           INTEGER,  -- d mod (q-1)
    coefficient         INTEGER,  -- (inverse of q) mod p
}
```

- For DSA private keys (*PSA_KEY_TYPE_DSA_KEYPAIR*), the format is the representation of the private key `x` as a big-endian byte string. The length of the byte string is the private key size in bytes (leading zeroes are not stripped).
- For elliptic curve key pairs (key types for which *PSA_KEY_TYPE_IS_ECC_KEYPAIR* is true), the format is a representation of the private value as a `ceiling(m/8)`-byte string where `m` is the bit size associated with the curve, i.e. the bit size of the order of the curve's coordinate field. This byte string is in little-endian order for Montgomery curves (curve types `PSA_ECC_CURVE_CURVEXXX`), and in big-endian order for Weierstrass curves (curve types `PSA_ECC_CURVE_SECTXXX`, `PSA_ECC_CURVE_SECPXXX` and `PSA_ECC_CURVE_BRAINPOOL_PXXX`). This is the content of the `privateKey` field of the `ECPrivateKey` format defined by RFC 5915.
- For Diffie-Hellman key exchange key pairs (*PSA_KEY_TYPE_DH_KEYPAIR*), the format is the representation of the private key `x` as a big-endian byte string. The length of the byte string is the private key size in bytes (leading zeroes are not stripped).
- For public keys (key types for which *PSA_KEY_TYPE_IS_PUBLIC_KEY* is true), the format is the same as for *psa_export_public_key()*.

psa_export_public_key (function)

```
psa_status_t psa_export_public_key(psa_key_handle_t handle,
                                   uint8_t *data,
                                   size_t data_size,
                                   size_t *data_length);
```

Parameters:

handle Handle to the key to export.

data Buffer where the key data is to be written.

data_size Size of the `data` buffer in bytes.

data_length On success, the number of bytes that make up the key data.

Returns: `psa_status_t`

PSA_SUCCESS

PSA_ERROR_INVALID_HANDLE

PSA_ERROR_EMPTY_SLOT

PSA_ERROR_INVALID_ARGUMENT The key is neither a public key nor a key pair.

PSA_ERROR_NOT_SUPPORTED

PSA_ERROR_BUFFER_TOO_SMALL The size of the `data` buffer is too small. You can determine a sufficient buffer size by calling *PSA_KEY_EXPORT_MAX_SIZE*(*PSA_KEY_TYPE_PUBLIC_KEY_OF_KEYPAIR*(`type`), `bits`) where `type` is the key type and `bits` is the key size in bits.

PSA_ERROR_COMMUNICATION_FAILURE

PSA_ERROR_HARDWARE_FAILURE

PSA_ERROR_TAMPERING_DETECTED

PSA_ERROR_BAD_STATE The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

Description: Export a public key or the public part of a key pair in binary format.

The output of this function can be passed to *psa_import_key()* to create an object that is equivalent to the public key.

This specification supports a single format for each key type. Implementations may support other formats as long as the standard format is supported. Implementations that support other formats should ensure that the formats are clearly unambiguous so as to minimize the risk that an invalid input is accidentally interpreted according to a different format.

For standard key types, the output format is as follows:

- For RSA public keys (*PSA_KEY_TYPE_RSA_PUBLIC_KEY*), the DER encoding of the representation defined by RFC 3279 §2.3.1 as `RSAPublicKey`.

```
RSAPublicKey ::= SEQUENCE {
   modulus            INTEGER,    -- n
   publicExponent     INTEGER  }  -- e
```

- For elliptic curve public keys (key types for which *PSA_KEY_TYPE_IS_ECC_PUBLIC_KEY* is true), the format is the uncompressed representation defined by SEC1 §2.3.3 as the content of an ECPoint. Let `m` be the bit size associated with the curve, i.e. the bit size of `q` for a curve over `F_q`. The representation consists of:
  - The byte 0x04;
  - `x_P` as a `ceiling(m/8)`-byte string, big-endian;
  - `y_P` as a `ceiling(m/8)`-byte string, big-endian.
- For DSA public keys (*PSA_KEY_TYPE_DSA_PUBLIC_KEY*), the format is the representation of the public key `y = g^x mod p` as a big-endian byte string. The length of the byte string is the length of the base prime `p` in bytes.
- For Diffie-Hellman key exchange public keys (*PSA_KEY_TYPE_DH_PUBLIC_KEY*), the format is the representation of the public key `y = g^x mod p` as a big-endian byte string. The length of the byte string is the length of the base prime `p` in bytes.

psa_copy_key (function)

```
psa_status_t psa_copy_key(psa_key_handle_t source_handle,
                          psa_key_handle_t target_handle,
                          const psa_key_policy_t *constraint);
```

Parameters:

source_handle The key to copy. It must be a handle to an occupied slot.

target_handle A handle to the target slot. It must not contain key material yet.

constraint An optional policy constraint. If this parameter is non-null then the resulting key will conform to this policy in addition to the source policy and the policy already present on the target slot. If this parameter is null then the function behaves in the same way as if it was the target policy, i.e. only the source and target policies apply.

Returns: `psa_status_t`

PSA_SUCCESS

PSA_ERROR_INVALID_HANDLE

PSA_ERROR_OCCUPIED_SLOT `target` already contains key material.

PSA_ERROR_EMPTY_SLOT `source` does not contain key material.

PSA_ERROR_INVALID_ARGUMENT The policy constraints on the source, on the target and `constraints` are incompatible.

PSA_ERROR_NOT_PERMITTED The source key is not exportable and its lifetime does not allow copying it to the target's lifetime.

PSA_ERROR_INSUFFICIENT_MEMORY

PSA_ERROR_INSUFFICIENT_STORAGE

PSA_ERROR_COMMUNICATION_FAILURE

PSA_ERROR_HARDWARE_FAILURE

PSA_ERROR_TAMPERING_DETECTED

Description: Make a copy of a key.

Copy key material from one location to another.

This function is primarily useful to copy a key from one lifetime to another. The target key retains its lifetime and location.

In an implementation where slots have different ownerships, this function may be used to share a key with a different party, subject to implementation-defined restrictions on key sharing. In this case `constraint` would typically prevent the recipient from exporting the key.

The resulting key may only be used in a way that conforms to all three of: the policy of the source key, the policy previously set on the target, and the `constraint` parameter passed when calling this function.

- The usage flags on the resulting key are the bitwise-and of the usage flags on the source policy, the previously-set target policy and the policy constraint.
- If all three policies allow the same algorithm or wildcard-based algorithm policy, the resulting key has the same algorithm policy.
- If one of the policies allows an algorithm and all the other policies either allow the same algorithm or a wildcard-based algorithm policy that includes this algorithm, the resulting key allows the same algorithm.

The effect of this function on implementation-defined metadata is implementation-defined.

CHAPTER

THIRTEEN

MESSAGE DIGESTS

psa_hash_operation_t (type)

```
typedef struct psa_hash_operation_s psa_hash_operation_t;
```

The type of the state data structure for multipart hash operations.

Before calling any function on a hash operation object, the application must initialize it by any of the following means:

- Set the structure to all-bits-zero, for example:

```
psa_hash_operation_t operation;
memset(&operation, 0, sizeof(operation));
```

- Initialize the structure to logical zero values, for example:

```
psa_hash_operation_t operation = {0};
```

- Initialize the structure to the initializer *PSA_HASH_OPERATION_INIT*, for example:

```
psa_hash_operation_t operation = PSA_HASH_OPERATION_INIT;
```

- Assign the result of the function psa_hash_operation_init() to the structure, for example:

```
psa_hash_operation_t operation;
operation = psa_hash_operation_init();
```

This is an implementation-defined `struct`. Applications should not make any assumptions about the content of this structure except as directed by the documentation of a specific implementation.

PSA_HASH_OPERATION_INIT (macro)

```
#define PSA_HASH_OPERATION_INIT {0}
```

This macro returns a suitable initializer for a hash operation object of type *psa_hash_operation_t*.

psa_hash_compute (function)

```
psa_status_t psa_hash_compute(psa_algorithm_t alg,
                              const uint8_t *input,
                              size_t input_length,
                              uint8_t *hash,
```

```
                              size_t hash_size,
                              size_t *hash_length);
```

Parameters:

alg The hash algorithm to compute (PSA_ALG_XXX value such that *PSA_ALG_IS_HASH*(alg) is true).

input Buffer containing the message to hash.

input_length Size of the input buffer in bytes.

hash Buffer where the hash is to be written.

hash_size Size of the hash buffer in bytes.

hash_length On success, the number of bytes that make up the hash value. This is always *PSA_HASH_SIZE*(alg).

Returns: psa_status_t

PSA_SUCCESS Success.

PSA_ERROR_NOT_SUPPORTED alg is not supported or is not a hash algorithm.

PSA_ERROR_INSUFFICIENT_MEMORY

PSA_ERROR_COMMUNICATION_FAILURE

PSA_ERROR_HARDWARE_FAILURE

PSA_ERROR_TAMPERING_DETECTED

Description: Calculate the hash (digest) of a message.

Note: To verify the hash of a message against an expected value, use *psa_hash_compare()* instead.

psa_hash_compare (function)

```
psa_status_t psa_hash_compare(psa_algorithm_t alg,
                              const uint8_t *input,
                              size_t input_length,
                              const uint8_t *hash,
                              const size_t hash_length);
```

Parameters:

alg The hash algorithm to compute (PSA_ALG_XXX value such that *PSA_ALG_IS_HASH*(alg) is true).

input Buffer containing the message to hash.

input_length Size of the input buffer in bytes.

hash Buffer containing the expected hash value.

hash_length Size of the hash buffer in bytes.

Returns: psa_status_t

PSA_SUCCESS The expected hash is identical to the actual hash of the input.

PSA_ERROR_INVALID_SIGNATURE The hash of the message was calculated successfully, but it differs from the expected hash.

PSA_ERROR_NOT_SUPPORTED `alg` is not supported or is not a hash algorithm.

PSA_ERROR_INSUFFICIENT_MEMORY

PSA_ERROR_COMMUNICATION_FAILURE

PSA_ERROR_HARDWARE_FAILURE

PSA_ERROR_TAMPERING_DETECTED

Description: Calculate the hash (digest) of a message and compare it with a reference value.

psa_hash_operation_init (function)

```
psa_hash_operation_t psa_hash_operation_init(void);
```

Returns: `psa_hash_operation_t` **Description:** Return an initial value for a hash operation object.

psa_hash_setup (function)

```
psa_status_t psa_hash_setup(psa_hash_operation_t *operation,
                            psa_algorithm_t alg);
```

Parameters:

operation The operation object to set up. It must have been initialized as per the documentation for *psa_hash_operation_t* and not yet in use.

alg The hash algorithm to compute (`PSA_ALG_XXX` value such that *PSA_ALG_IS_HASH*(`alg`) is true).

Returns: `psa_status_t`

PSA_SUCCESS Success.

PSA_ERROR_NOT_SUPPORTED `alg` is not supported or is not a hash algorithm.

PSA_ERROR_INSUFFICIENT_MEMORY

PSA_ERROR_COMMUNICATION_FAILURE

PSA_ERROR_HARDWARE_FAILURE

PSA_ERROR_TAMPERING_DETECTED

Description: Set up a multipart hash operation.

The sequence of operations to calculate a hash (message digest) is as follows:

1. Allocate an operation object which will be passed to all the functions listed here.
2. Initialize the operation object with one of the methods described in the documentation for *psa_hash_operation_t*, e.g. PSA_HASH_OPERATION_INIT.
3. Call *psa_hash_setup()* to specify the algorithm.
4. Call *psa_hash_update()* zero, one or more times, passing a fragment of the message each time. The hash that is calculated is the hash of the concatenation of these messages in order.
5. To calculate the hash, call *psa_hash_finish()*. To compare the hash with an expected value, call *psa_hash_verify()*.

The application may call *psa_hash_abort()* at any time after the operation has been initialized.

After a successful call to *psa_hash_setup()*, the application must eventually terminate the operation. The following events terminate an operation:

- A failed call to *psa_hash_update()*.
- A call to *psa_hash_finish()*, *psa_hash_verify()* or *psa_hash_abort()*.

psa_hash_update (function)

```
psa_status_t psa_hash_update(psa_hash_operation_t *operation,
                             const uint8_t *input,
                             size_t input_length);
```

Parameters:

operation Active hash operation.

input Buffer containing the message fragment to hash.

input_length Size of the input buffer in bytes.

Returns: psa_status_t

PSA_SUCCESS Success.

PSA_ERROR_BAD_STATE The operation state is not valid (not set up, or already completed).

PSA_ERROR_INSUFFICIENT_MEMORY

PSA_ERROR_COMMUNICATION_FAILURE

PSA_ERROR_HARDWARE_FAILURE

PSA_ERROR_TAMPERING_DETECTED

Description: Add a message fragment to a multipart hash operation.

The application must call *psa_hash_setup()* before calling this function.

If this function returns an error status, the operation becomes inactive.

psa_hash_finish (function)

```
psa_status_t psa_hash_finish(psa_hash_operation_t *operation,
                             uint8_t *hash,
                             size_t hash_size,
                             size_t *hash_length);
```

Parameters:

operation Active hash operation.

hash Buffer where the hash is to be written.

hash_size Size of the hash buffer in bytes.

hash_length On success, the number of bytes that make up the hash value. This is always *PSA_HASH_SIZE*(alg) where alg is the hash algorithm that is calculated.

Returns: psa_status_t

PSA_SUCCESS Success.

PSA_ERROR_BAD_STATE The operation state is not valid (not set up, or already completed).

PSA_ERROR_BUFFER_TOO_SMALL The size of the `hash` buffer is too small. You can determine a sufficient buffer size by calling *PSA_HASH_SIZE*(`alg`) where `alg` is the hash algorithm that is calculated.

PSA_ERROR_INSUFFICIENT_MEMORY

PSA_ERROR_COMMUNICATION_FAILURE

PSA_ERROR_HARDWARE_FAILURE

PSA_ERROR_TAMPERING_DETECTED

Description: Finish the calculation of the hash of a message.

The application must call *psa_hash_setup()* before calling this function. This function calculates the hash of the message formed by concatenating the inputs passed to preceding calls to *psa_hash_update()*.

When this function returns, the operation becomes inactive.

> **Warning:** Applications should not call this function if they expect a specific value for the hash. Call *psa_hash_verify()* instead. Beware that comparing integrity or authenticity data such as hash values with a function such as `memcmp` is risky because the time taken by the comparison may leak information about the hashed data which could allow an attacker to guess a valid hash and thereby bypass security controls.

psa_hash_verify (function)

```
psa_status_t psa_hash_verify(psa_hash_operation_t *operation,
                             const uint8_t *hash,
                             size_t hash_length);
```

Parameters:

operation Active hash operation.

hash Buffer containing the expected hash value.

hash_length Size of the `hash` buffer in bytes.

Returns: `psa_status_t`

PSA_SUCCESS The expected hash is identical to the actual hash of the message.

PSA_ERROR_INVALID_SIGNATURE The hash of the message was calculated successfully, but it differs from the expected hash.

PSA_ERROR_BAD_STATE The operation state is not valid (not set up, or already completed).

PSA_ERROR_INSUFFICIENT_MEMORY

PSA_ERROR_COMMUNICATION_FAILURE

PSA_ERROR_HARDWARE_FAILURE

PSA_ERROR_TAMPERING_DETECTED

Description: Finish the calculation of the hash of a message and compare it with an expected value.

The application must call *psa_hash_setup()* before calling this function. This function calculates the hash of the message formed by concatenating the inputs passed to preceding calls to *psa_hash_update()*. It then compares the calculated hash with the expected hash passed as a parameter to this function.

When this function returns, the operation becomes inactive.

Note: Implementations shall make the best effort to ensure that the comparison between the actual hash and the expected hash is performed in constant time.

psa_hash_abort (function)

```
psa_status_t psa_hash_abort(psa_hash_operation_t *operation);
```

Parameters:

operation Initialized hash operation.

Returns: psa_status_t

PSA_SUCCESS

PSA_ERROR_BAD_STATE operation is not an active hash operation.

PSA_ERROR_COMMUNICATION_FAILURE

PSA_ERROR_HARDWARE_FAILURE

PSA_ERROR_TAMPERING_DETECTED

Description: Abort a hash operation.

Aborting an operation frees all associated resources except for the operation structure itself. Once aborted, the operation object can be reused for another operation by calling *psa_hash_setup()* again.

You may call this function any time after the operation object has been initialized by any of the following methods:

- A call to *psa_hash_setup()*, whether it succeeds or not.
- Initializing the struct to all-bits-zero.
- Initializing the struct to logical zeros, e.g. psa_hash_operation_t operation = {0}.

In particular, calling *psa_hash_abort()* after the operation has been terminated by a call to *psa_hash_abort()*, *psa_hash_finish()* or *psa_hash_verify()* is safe and has no effect.

psa_hash_clone (function)

```
psa_status_t psa_hash_clone(const psa_hash_operation_t *source_operation,
                            psa_hash_operation_t *target_operation);
```

Parameters:

source_operation The active hash operation to clone.

target_operation The operation object to set up. It must be initialized but not active.

Returns: psa_status_t

PSA_SUCCESS

PSA_ERROR_BAD_STATE source_operation is not an active hash operation.

PSA_ERROR_BAD_STATE target_operation is active.

PSA_ERROR_COMMUNICATION_FAILURE

PSA_ERROR_HARDWARE_FAILURE

PSA_ERROR_TAMPERING_DETECTED

Description: Clone a hash operation.

This function copies the state of an ongoing hash operation to a new operation object. In other words, this function is equivalent to calling *psa_hash_setup()* on `target_operation` with the same algorithm that `source_operation` was set up for, then *psa_hash_update()* on `target_operation` with the same input that that was passed to `source_operation`. After this function returns, the two objects are independent, i.e. subsequent calls involving one of the objects do not affect the other object.

CHAPTER

FOURTEEN

MESSAGE AUTHENTICATION CODES

psa_mac_operation_t (type)

```
typedef struct psa_mac_operation_s psa_mac_operation_t;
```

The type of the state data structure for multipart MAC operations.

Before calling any function on a MAC operation object, the application must initialize it by any of the following means:

- Set the structure to all-bits-zero, for example:

```
psa_mac_operation_t operation;
memset(&operation, 0, sizeof(operation));
```

- Initialize the structure to logical zero values, for example:

```
psa_mac_operation_t operation = {0};
```

- Initialize the structure to the initializer *PSA_MAC_OPERATION_INIT*, for example:

```
psa_mac_operation_t operation = PSA_MAC_OPERATION_INIT;
```

- Assign the result of the function psa_mac_operation_init() to the structure, for example:

```
psa_mac_operation_t operation;
operation = psa_mac_operation_init();
```

This is an implementation-defined `struct`. Applications should not make any assumptions about the content of this structure except as directed by the documentation of a specific implementation.

PSA_MAC_OPERATION_INIT (macro)

```
#define PSA_MAC_OPERATION_INIT {0}
```

This macro returns a suitable initializer for a MAC operation object of type *psa_mac_operation_t*.

psa_mac_compute (function)

```
psa_status_t psa_mac_compute(psa_key_handle_t handle,
                             psa_algorithm_t alg,
                             const uint8_t *input,
                             size_t input_length,
                             uint8_t *mac,
                             size_t mac_size,
                             size_t *mac_length);
```

Parameters:

handle Handle to the key to use for the operation.

alg The MAC algorithm to compute (PSA_ALG_XXX value such that *PSA_ALG_IS_MAC*(alg) is true).

input Buffer containing the input message.

input_length Size of the input buffer in bytes.

mac Buffer where the MAC value is to be written.

mac_size Size of the mac buffer in bytes.

mac_length On success, the number of bytes that make up the MAC value.

Returns: psa_status_t

PSA_SUCCESS Success.

PSA_ERROR_INVALID_HANDLE

PSA_ERROR_EMPTY_SLOT

PSA_ERROR_NOT_PERMITTED

PSA_ERROR_INVALID_ARGUMENT key is not compatible with alg.

PSA_ERROR_NOT_SUPPORTED alg is not supported or is not a MAC algorithm.

PSA_ERROR_INSUFFICIENT_MEMORY

PSA_ERROR_COMMUNICATION_FAILURE

PSA_ERROR_HARDWARE_FAILURE

PSA_ERROR_TAMPERING_DETECTED

PSA_ERROR_BAD_STATE The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

Description: Calculate the MAC (message authentication code) of a message.

Note: To verify the MAC of a message against an expected value, use *psa_mac_verify()* instead. Beware that comparing integrity or authenticity data such as MAC values with a function such as memcmp is risky because the time taken by the comparison may leak information about the MAC value which could allow an attacker to guess a valid MAC and thereby bypass security controls.

---

psa_mac_verify (function)

```
psa_status_t psa_mac_verify(psa_key_handle_t handle,
                            psa_algorithm_t alg,
                            const uint8_t *input,
                            size_t input_length,
                            const uint8_t *mac,
                            const size_t mac_length);
```

Parameters:

handle Handle to the key to use for the operation.

alg The MAC algorithm to compute (PSA_ALG_XXX value such that *PSA_ALG_IS_MAC*(alg) is true).

input Buffer containing the input message.

input_length Size of the input buffer in bytes.

mac Buffer containing the expected MAC value.

mac_length Size of the mac buffer in bytes.

Returns: psa_status_t

PSA_SUCCESS The expected MAC is identical to the actual MAC of the input.

PSA_ERROR_INVALID_SIGNATURE The MAC of the message was calculated successfully, but it differs from the expected value.

PSA_ERROR_INVALID_HANDLE

PSA_ERROR_EMPTY_SLOT

PSA_ERROR_NOT_PERMITTED

PSA_ERROR_INVALID_ARGUMENT key is not compatible with alg.

PSA_ERROR_NOT_SUPPORTED alg is not supported or is not a MAC algorithm.

PSA_ERROR_INSUFFICIENT_MEMORY

PSA_ERROR_COMMUNICATION_FAILURE

PSA_ERROR_HARDWARE_FAILURE

PSA_ERROR_TAMPERING_DETECTED

Description: Calculate the MAC of a message and compare it with a reference value.

psa_mac_operation_init (function)

```
psa_mac_operation_t psa_mac_operation_init(void);
```

Returns: psa_mac_operation_t **Description:** Return an initial value for a MAC operation object.

psa_mac_sign_setup (function)

```
psa_status_t psa_mac_sign_setup(psa_mac_operation_t *operation,
                                psa_key_handle_t handle,
                                psa_algorithm_t alg);
```

Parameters:

operation The operation object to set up. It must have been initialized as per the documentation for *psa_mac_operation_t* and not yet in use.

handle Handle to the key to use for the operation. It must remain valid until the operation terminates.

alg The MAC algorithm to compute (`PSA_ALG_XXX` value such that *PSA_ALG_IS_MAC*(`alg`) is true).

Returns: `psa_status_t`

PSA_SUCCESS Success.

PSA_ERROR_INVALID_HANDLE

PSA_ERROR_EMPTY_SLOT

PSA_ERROR_NOT_PERMITTED

PSA_ERROR_INVALID_ARGUMENT `key` is not compatible with `alg`.

PSA_ERROR_NOT_SUPPORTED `alg` is not supported or is not a MAC algorithm.

PSA_ERROR_INSUFFICIENT_MEMORY

PSA_ERROR_COMMUNICATION_FAILURE

PSA_ERROR_HARDWARE_FAILURE

PSA_ERROR_TAMPERING_DETECTED

PSA_ERROR_BAD_STATE The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

Description: Set up a multipart MAC calculation operation.

This function sets up the calculation of the MAC (message authentication code) of a byte string. To verify the MAC of a message against an expected value, use *psa_mac_verify_setup()* instead.

The sequence of operations to calculate a MAC is as follows:

1. Allocate an operation object which will be passed to all the functions listed here.
2. Initialize the operation object with one of the methods described in the documentation for *psa_mac_operation_t*, e.g. PSA_MAC_OPERATION_INIT.
3. Call *psa_mac_sign_setup()* to specify the algorithm and key.
4. Call *psa_mac_update()* zero, one or more times, passing a fragment of the message each time. The MAC that is calculated is the MAC of the concatenation of these messages in order.
5. At the end of the message, call *psa_mac_sign_finish()* to finish calculating the MAC value and retrieve it.

The application may call *psa_mac_abort()* at any time after the operation has been initialized.

After a successful call to *psa_mac_sign_setup()*, the application must eventually terminate the operation through one of the following methods:

- A failed call to *psa_mac_update()*.
- A call to *psa_mac_sign_finish()* or *psa_mac_abort()*.

psa_mac_verify_setup (function)

```
psa_status_t psa_mac_verify_setup(psa_mac_operation_t *operation,
                                  psa_key_handle_t handle,
                                  psa_algorithm_t alg);
```

Parameters:

operation The operation object to set up. It must have been initialized as per the documentation for *psa_mac_operation_t* and not yet in use.

handle Handle to the key to use for the operation. It must remain valid until the operation terminates.

alg The MAC algorithm to compute (PSA_ALG_XXX value such that *PSA_ALG_IS_MAC*(alg) is true).

Returns: psa_status_t

PSA_SUCCESS Success.

PSA_ERROR_INVALID_HANDLE

PSA_ERROR_EMPTY_SLOT

PSA_ERROR_NOT_PERMITTED

PSA_ERROR_INVALID_ARGUMENT key is not compatible with alg.

PSA_ERROR_NOT_SUPPORTED alg is not supported or is not a MAC algorithm.

PSA_ERROR_INSUFFICIENT_MEMORY

PSA_ERROR_COMMUNICATION_FAILURE

PSA_ERROR_HARDWARE_FAILURE

PSA_ERROR_TAMPERING_DETECTED

PSA_ERROR_BAD_STATE The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

Description: Set up a multipart MAC verification operation.

This function sets up the verification of the MAC (message authentication code) of a byte string against an expected value.

The sequence of operations to verify a MAC is as follows:

1. Allocate an operation object which will be passed to all the functions listed here.
2. Initialize the operation object with one of the methods described in the documentation for *psa_mac_operation_t*, e.g. PSA_MAC_OPERATION_INIT.
3. Call *psa_mac_verify_setup()* to specify the algorithm and key.
4. Call *psa_mac_update()* zero, one or more times, passing a fragment of the message each time. The MAC that is calculated is the MAC of the concatenation of these messages in order.
5. At the end of the message, call *psa_mac_verify_finish()* to finish calculating the actual MAC of the message and verify it against the expected value.

The application may call *psa_mac_abort()* at any time after the operation has been initialized.

After a successful call to *psa_mac_verify_setup()*, the application must eventually terminate the operation through one of the following methods:

- A failed call to *psa_mac_update()*.
- A call to *psa_mac_verify_finish()* or *psa_mac_abort()*.

psa_mac_update (function)

```
psa_status_t psa_mac_update(psa_mac_operation_t *operation,
                            const uint8_t *input,
                            size_t input_length);
```

Parameters:

operation Active MAC operation.

input Buffer containing the message fragment to add to the MAC calculation.

input_length Size of the `input` buffer in bytes.

Returns: `psa_status_t`

PSA_SUCCESS Success.

PSA_ERROR_BAD_STATE The operation state is not valid (not set up, or already completed).

PSA_ERROR_INSUFFICIENT_MEMORY

PSA_ERROR_COMMUNICATION_FAILURE

PSA_ERROR_HARDWARE_FAILURE

PSA_ERROR_TAMPERING_DETECTED

Description: Add a message fragment to a multipart MAC operation.

The application must call *psa_mac_sign_setup()* or *psa_mac_verify_setup()* before calling this function.

If this function returns an error status, the operation becomes inactive.

psa_mac_sign_finish (function)

```
psa_status_t psa_mac_sign_finish(psa_mac_operation_t *operation,
                                 uint8_t *mac,
                                 size_t mac_size,
                                 size_t *mac_length);
```

Parameters:

operation Active MAC operation.

mac Buffer where the MAC value is to be written.

mac_size Size of the `mac` buffer in bytes.

mac_length On success, the number of bytes that make up the MAC value. This is always *PSA_MAC_FINAL_SIZE*(`key_type`, `key_bits`, `alg`) where `key_type` and `key_bits` are the type and bit-size respectively of the key and `alg` is the MAC algorithm that is calculated.

Returns: `psa_status_t`

PSA_SUCCESS Success.

PSA_ERROR_BAD_STATE The operation state is not valid (not set up, or already completed).

PSA_ERROR_BUFFER_TOO_SMALL The size of the `mac` buffer is too small. You can determine a sufficient buffer size by calling *PSA_MAC_FINAL_SIZE()*.

PSA_ERROR_INSUFFICIENT_MEMORY

PSA_ERROR_COMMUNICATION_FAILURE

PSA_ERROR_HARDWARE_FAILURE

PSA_ERROR_TAMPERING_DETECTED

Description: Finish the calculation of the MAC of a message.

The application must call *psa_mac_sign_setup()* before calling this function. This function calculates the MAC of the message formed by concatenating the inputs passed to preceding calls to *psa_mac_update()*.

When this function returns, the operation becomes inactive.

> **Warning:** Applications should not call this function if they expect a specific value for the MAC. Call *psa_mac_verify_finish()* instead. Beware that comparing integrity or authenticity data such as MAC values with a function such as `memcmp` is risky because the time taken by the comparison may leak information about the MAC value which could allow an attacker to guess a valid MAC and thereby bypass security controls.

psa_mac_verify_finish (function)

```
psa_status_t psa_mac_verify_finish(psa_mac_operation_t *operation,
                                   const uint8_t *mac,
                                   size_t mac_length);
```

Parameters:

operation Active MAC operation.

mac Buffer containing the expected MAC value.

mac_length Size of the `mac` buffer in bytes.

Returns: `psa_status_t`

PSA_SUCCESS The expected MAC is identical to the actual MAC of the message.

PSA_ERROR_INVALID_SIGNATURE The MAC of the message was calculated successfully, but it differs from the expected MAC.

PSA_ERROR_BAD_STATE The operation state is not valid (not set up, or already completed).

PSA_ERROR_INSUFFICIENT_MEMORY

PSA_ERROR_COMMUNICATION_FAILURE

PSA_ERROR_HARDWARE_FAILURE

PSA_ERROR_TAMPERING_DETECTED

Description: Finish the calculation of the MAC of a message and compare it with an expected value.

The application must call *psa_mac_verify_setup()* before calling this function. This function calculates the MAC of the message formed by concatenating the inputs passed to preceding calls to *psa_mac_update()*. It then compares the calculated MAC with the expected MAC passed as a parameter to this function.

When this function returns, the operation becomes inactive.

Note: Implementations shall make the best effort to ensure that the comparison between the actual MAC and the expected MAC is performed in constant time.

`psa_mac_abort` (function)

```
psa_status_t psa_mac_abort(psa_mac_operation_t *operation);
```

Parameters:

`operation` Initialized MAC operation.

Returns: `psa_status_t`

`PSA_SUCCESS`

`PSA_ERROR_BAD_STATE` `operation` is not an active MAC operation.

`PSA_ERROR_COMMUNICATION_FAILURE`

`PSA_ERROR_HARDWARE_FAILURE`

`PSA_ERROR_TAMPERING_DETECTED`

Description: Abort a MAC operation.

Aborting an operation frees all associated resources except for the `operation` structure itself. Once aborted, the operation object can be reused for another operation by calling *`psa_mac_sign_setup()`* or *`psa_mac_verify_setup()`* again.

You may call this function any time after the operation object has been initialized by any of the following methods:

- A call to *`psa_mac_sign_setup()`* or *`psa_mac_verify_setup()`*, whether it succeeds or not.
- Initializing the `struct` to all-bits-zero.
- Initializing the `struct` to logical zeros, e.g. `psa_mac_operation_t operation = {0}`.

In particular, calling *`psa_mac_abort()`* after the operation has been terminated by a call to *`psa_mac_abort()`*, *`psa_mac_sign_finish()`* or *`psa_mac_verify_finish()`* is safe and has no effect.

CHAPTER

FIFTEEN

SYMMETRIC CIPHERS

psa_cipher_operation_t (type)

```
typedef struct psa_cipher_operation_s psa_cipher_operation_t;
```

The type of the state data structure for multipart cipher operations.

Before calling any function on a cipher operation object, the application must initialize it by any of the following means:

- Set the structure to all-bits-zero, for example:

```
psa_cipher_operation_t operation;
memset(&operation, 0, sizeof(operation));
```

- Initialize the structure to logical zero values, for example:

```
psa_cipher_operation_t operation = {0};
```

- Initialize the structure to the initializer *PSA_CIPHER_OPERATION_INIT*, for example:

```
psa_cipher_operation_t operation = PSA_CIPHER_OPERATION_INIT;
```

- Assign the result of the function psa_cipher_operation_init() to the structure, for example:

```
psa_cipher_operation_t operation;
operation = psa_cipher_operation_init();
```

This is an implementation-defined `struct`. Applications should not make any assumptions about the content of this structure except as directed by the documentation of a specific implementation.

PSA_CIPHER_OPERATION_INIT (macro)

```
#define PSA_CIPHER_OPERATION_INIT {0}
```

This macro returns a suitable initializer for a cipher operation object of type *psa_cipher_operation_t*.

psa_cipher_encrypt (function)

```
psa_status_t psa_cipher_encrypt(psa_key_handle_t handle,
                                psa_algorithm_t alg,
                                const uint8_t *input,
                                size_t input_length,
                                uint8_t *output,
                                size_t output_size,
                                size_t *output_length);
```

Parameters:

handle Handle to the key to use for the operation. It must remain valid until the operation terminates.

alg The cipher algorithm to compute (`PSA_ALG_XXX` value such that *PSA_ALG_IS_CIPHER*(`alg`) is true).

input Buffer containing the message to encrypt.

input_length Size of the `input` buffer in bytes.

output Buffer where the output is to be written. The output contains the IV followed by the ciphertext proper.

output_size Size of the `output` buffer in bytes.

output_length On success, the number of bytes that make up the output.

Returns: `psa_status_t`

PSA_SUCCESS Success.

PSA_ERROR_INVALID_HANDLE

PSA_ERROR_EMPTY_SLOT

PSA_ERROR_NOT_PERMITTED

PSA_ERROR_INVALID_ARGUMENT `key` is not compatible with `alg`.

PSA_ERROR_NOT_SUPPORTED `alg` is not supported or is not a cipher algorithm.

PSA_ERROR_BUFFER_TOO_SMALL

PSA_ERROR_INSUFFICIENT_MEMORY

PSA_ERROR_COMMUNICATION_FAILURE

PSA_ERROR_HARDWARE_FAILURE

PSA_ERROR_TAMPERING_DETECTED

Description: Encrypt a message using a symmetric cipher.

This function encrypts a message with a random IV (initialization vector).

psa_cipher_decrypt (function)

```
psa_status_t psa_cipher_decrypt(psa_key_handle_t handle,
                                psa_algorithm_t alg,
                                const uint8_t *input,
                                size_t input_length,
                                uint8_t *output,
                                size_t output_size,
                                size_t *output_length);
```

Parameters:

handle Handle to the key to use for the operation. It must remain valid until the operation terminates.

alg The cipher algorithm to compute (PSA_ALG_XXX value such that *PSA_ALG_IS_CIPHER*(alg) is true).

input Buffer containing the message to decrypt. This consists of the IV followed by the ciphertext proper.

input_length Size of the input buffer in bytes.

output Buffer where the plaintext is to be written.

output_size Size of the output buffer in bytes.

output_length On success, the number of bytes that make up the output.

Returns: psa_status_t

PSA_SUCCESS Success.

PSA_ERROR_INVALID_HANDLE

PSA_ERROR_EMPTY_SLOT

PSA_ERROR_NOT_PERMITTED

PSA_ERROR_INVALID_ARGUMENT key is not compatible with alg.

PSA_ERROR_NOT_SUPPORTED alg is not supported or is not a cipher algorithm.

PSA_ERROR_BUFFER_TOO_SMALL

PSA_ERROR_INSUFFICIENT_MEMORY

PSA_ERROR_COMMUNICATION_FAILURE

PSA_ERROR_HARDWARE_FAILURE

PSA_ERROR_TAMPERING_DETECTED

Description: Decrypt a message using a symmetric cipher.

This function decrypts a message encrypted with a symmetric cipher.

psa_cipher_operation_init (function)

```
psa_cipher_operation_t psa_cipher_operation_init(void);
```

Returns: psa_cipher_operation_t **Description:** Return an initial value for a cipher operation object.

psa_cipher_encrypt_setup (function)

```
psa_status_t psa_cipher_encrypt_setup(psa_cipher_operation_t *operation,
                                      psa_key_handle_t handle,
                                      psa_algorithm_t alg);
```

Parameters:

operation The operation object to set up. It must have been initialized as per the documentation for *psa_cipher_operation_t* and not yet in use.

handle Handle to the key to use for the operation. It must remain valid until the operation terminates.

alg The cipher algorithm to compute (PSA_ALG_XXX value such that *PSA_ALG_IS_CIPHER*(alg) is true).

Returns: `psa_status_t`

PSA_SUCCESS Success.

PSA_ERROR_INVALID_HANDLE

PSA_ERROR_EMPTY_SLOT

PSA_ERROR_NOT_PERMITTED

PSA_ERROR_INVALID_ARGUMENT `key` is not compatible with `alg`.

PSA_ERROR_NOT_SUPPORTED `alg` is not supported or is not a cipher algorithm.

PSA_ERROR_INSUFFICIENT_MEMORY

PSA_ERROR_COMMUNICATION_FAILURE

PSA_ERROR_HARDWARE_FAILURE

PSA_ERROR_TAMPERING_DETECTED

PSA_ERROR_BAD_STATE The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

Description: Set the key for a multipart symmetric encryption operation.

The sequence of operations to encrypt a message with a symmetric cipher is as follows:

1. Allocate an operation object which will be passed to all the functions listed here.
2. Initialize the operation object with one of the methods described in the documentation for *psa_cipher_operation_t*, e.g. PSA_CIPHER_OPERATION_INIT.
3. Call *psa_cipher_encrypt_setup()* to specify the algorithm and key.
4. Call either *psa_cipher_generate_iv()* or *psa_cipher_set_iv()* to generate or set the IV (initialization vector). You should use *psa_cipher_generate_iv()* unless the protocol you are implementing requires a specific IV value.
5. Call *psa_cipher_update()* zero, one or more times, passing a fragment of the message each time.
6. Call *psa_cipher_finish()*.

The application may call *psa_cipher_abort()* at any time after the operation has been initialized.

After a successful call to *psa_cipher_encrypt_setup()*, the application must eventually terminate the operation. The following events terminate an operation:

- A failed call to any of the `psa_cipher_xxx` functions.
- A call to *psa_cipher_finish()* or *psa_cipher_abort()*.

psa_cipher_decrypt_setup (function)

```
psa_status_t psa_cipher_decrypt_setup(psa_cipher_operation_t *operation,
                                      psa_key_handle_t handle,
                                      psa_algorithm_t alg);
```

Parameters:

operation The operation object to set up. It must have been initialized as per the documentation for *psa_cipher_operation_t* and not yet in use.

handle Handle to the key to use for the operation. It must remain valid until the operation terminates.

alg The cipher algorithm to compute (PSA_ALG_XXX value such that *PSA_ALG_IS_CIPHER*(alg) is true).

Returns: psa_status_t

PSA_SUCCESS Success.

PSA_ERROR_INVALID_HANDLE

PSA_ERROR_EMPTY_SLOT

PSA_ERROR_NOT_PERMITTED

PSA_ERROR_INVALID_ARGUMENT key is not compatible with alg.

PSA_ERROR_NOT_SUPPORTED alg is not supported or is not a cipher algorithm.

PSA_ERROR_INSUFFICIENT_MEMORY

PSA_ERROR_COMMUNICATION_FAILURE

PSA_ERROR_HARDWARE_FAILURE

PSA_ERROR_TAMPERING_DETECTED

PSA_ERROR_BAD_STATE The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

Description: Set the key for a multipart symmetric decryption operation.

The sequence of operations to decrypt a message with a symmetric cipher is as follows:

1. Allocate an operation object which will be passed to all the functions listed here.
2. Initialize the operation object with one of the methods described in the documentation for *psa_cipher_operation_t*, e.g. PSA_CIPHER_OPERATION_INIT.
3. Call *psa_cipher_decrypt_setup()* to specify the algorithm and key.
4. Call *psa_cipher_set_iv()* with the IV (initialization vector) for the decryption. If the IV is prepended to the ciphertext, you can call *psa_cipher_update()* on a buffer containing the IV followed by the beginning of the message.
5. Call *psa_cipher_update()* zero, one or more times, passing a fragment of the message each time.
6. Call *psa_cipher_finish()*.

The application may call *psa_cipher_abort()* at any time after the operation has been initialized.

After a successful call to *psa_cipher_decrypt_setup()*, the application must eventually terminate the operation. The following events terminate an operation:

- A failed call to any of the psa_cipher_xxx functions.
- A call to *psa_cipher_finish()* or *psa_cipher_abort()*.

psa_cipher_generate_iv (function)

```
psa_status_t psa_cipher_generate_iv(psa_cipher_operation_t *operation,
                                    unsigned char *iv,
                                    size_t iv_size,
                                    size_t *iv_length);
```

Parameters:

operation Active cipher operation.

iv Buffer where the generated IV is to be written.

iv_size Size of the `iv` buffer in bytes.

iv_length On success, the number of bytes of the generated IV.

Returns: `psa_status_t`

PSA_SUCCESS Success.

PSA_ERROR_BAD_STATE The operation state is not valid (not set up, or IV already set).

PSA_ERROR_BUFFER_TOO_SMALL The size of the `iv` buffer is too small.

PSA_ERROR_INSUFFICIENT_MEMORY

PSA_ERROR_COMMUNICATION_FAILURE

PSA_ERROR_HARDWARE_FAILURE

PSA_ERROR_TAMPERING_DETECTED

Description: Generate an IV for a symmetric encryption operation.

This function generates a random IV (initialization vector), nonce or initial counter value for the encryption operation as appropriate for the chosen algorithm, key type and key size.

The application must call *psa_cipher_encrypt_setup()* before calling this function.

If this function returns an error status, the operation becomes inactive.

psa_cipher_set_iv (function)

```
psa_status_t psa_cipher_set_iv(psa_cipher_operation_t *operation,
                               const unsigned char *iv,
                               size_t iv_length);
```

Parameters:

operation Active cipher operation.

iv Buffer containing the IV to use.

iv_length Size of the IV in bytes.

Returns: `psa_status_t`

PSA_SUCCESS Success.

PSA_ERROR_BAD_STATE The operation state is not valid (not set up, or IV already set).

PSA_ERROR_INVALID_ARGUMENT The size of `iv` is not acceptable for the chosen algorithm, or the chosen algorithm does not use an IV.

PSA_ERROR_INSUFFICIENT_MEMORY

PSA_ERROR_COMMUNICATION_FAILURE

PSA_ERROR_HARDWARE_FAILURE

PSA_ERROR_TAMPERING_DETECTED

Description: Set the IV for a symmetric encryption or decryption operation.

This function sets the IV (initialization vector), nonce or initial counter value for the encryption or decryption operation.

The application must call *psa_cipher_encrypt_setup()* before calling this function.

If this function returns an error status, the operation becomes inactive.

Note: When encrypting, applications should use *psa_cipher_generate_iv()* instead of this function, unless implementing a protocol that requires a non-random IV.

---

psa_cipher_update (function)

```
psa_status_t psa_cipher_update(psa_cipher_operation_t *operation,
                               const uint8_t *input,
                               size_t input_length,
                               unsigned char *output,
                               size_t output_size,
                               size_t *output_length);
```

Parameters:

operation Active cipher operation.

input Buffer containing the message fragment to encrypt or decrypt.

input_length Size of the `input` buffer in bytes.

output Buffer where the output is to be written.

output_size Size of the `output` buffer in bytes.

output_length On success, the number of bytes that make up the returned output.

Returns: `psa_status_t`

PSA_SUCCESS Success.

PSA_ERROR_BAD_STATE The operation state is not valid (not set up, IV required but not set, or already completed).

PSA_ERROR_BUFFER_TOO_SMALL The size of the `output` buffer is too small.

PSA_ERROR_INSUFFICIENT_MEMORY

PSA_ERROR_COMMUNICATION_FAILURE

PSA_ERROR_HARDWARE_FAILURE

PSA_ERROR_TAMPERING_DETECTED

Description: Encrypt or decrypt a message fragment in an active cipher operation.

Before calling this function, you must:

1. Call either *psa_cipher_encrypt_setup()* or *psa_cipher_decrypt_setup()*. The choice of setup function determines whether this function encrypts or decrypts its input.
2. If the algorithm requires an IV, call *psa_cipher_generate_iv()* (recommended when encrypting) or *psa_cipher_set_iv()*.

If this function returns an error status, the operation becomes inactive.

psa_cipher_finish (function)

```
psa_status_t psa_cipher_finish(psa_cipher_operation_t *operation,
                               uint8_t *output,
                               size_t output_size,
                               size_t *output_length);
```

Parameters:

operation Active cipher operation.

output Buffer where the output is to be written.

output_size Size of the `output` buffer in bytes.

output_length On success, the number of bytes that make up the returned output.

Returns: `psa_status_t`

PSA_SUCCESS Success.

PSA_ERROR_BAD_STATE The operation state is not valid (not set up, IV required but not set, or already completed).

PSA_ERROR_BUFFER_TOO_SMALL The size of the `output` buffer is too small.

PSA_ERROR_INSUFFICIENT_MEMORY

PSA_ERROR_COMMUNICATION_FAILURE

PSA_ERROR_HARDWARE_FAILURE

PSA_ERROR_TAMPERING_DETECTED

Description: Finish encrypting or decrypting a message in a cipher operation.

The application must call *`psa_cipher_encrypt_setup()`* or *`psa_cipher_decrypt_setup()`* before calling this function. The choice of setup function determines whether this function encrypts or decrypts its input.

This function finishes the encryption or decryption of the message formed by concatenating the inputs passed to preceding calls to *`psa_cipher_update()`*.

When this function returns, the operation becomes inactive.

psa_cipher_abort (function)

```
psa_status_t psa_cipher_abort(psa_cipher_operation_t *operation);
```

Parameters:

operation Initialized cipher operation.

Returns: `psa_status_t`

PSA_SUCCESS

PSA_ERROR_BAD_STATE `operation` is not an active cipher operation.

PSA_ERROR_COMMUNICATION_FAILURE

PSA_ERROR_HARDWARE_FAILURE

PSA_ERROR_TAMPERING_DETECTED

Description: Abort a cipher operation.

Aborting an operation frees all associated resources except for the `operation` structure itself. Once aborted, the operation object can be reused for another operation by calling *psa_cipher_encrypt_setup()* or *psa_cipher_decrypt_setup()* again.

You may call this function any time after the operation object has been initialized by any of the following methods:

- A call to *psa_cipher_encrypt_setup()* or *psa_cipher_decrypt_setup()*, whether it succeeds or not.
- Initializing the `struct` to all-bits-zero.
- Initializing the `struct` to logical zeros, e.g. `psa_cipher_operation_t operation = {0}`.

In particular, calling *psa_cipher_abort()* after the operation has been terminated by a call to *psa_cipher_abort()* or *psa_cipher_finish()* is safe and has no effect.

CHAPTER

SIXTEEN

AUTHENTICATED ENCRYPTION WITH ASSOCIATED DATA (AEAD)

psa_aead_operation_t (type)

```
typedef struct psa_aead_operation_s psa_aead_operation_t;
```

The type of the state data structure for multipart AEAD operations.

Before calling any function on an AEAD operation object, the application must initialize it by any of the following means:

- Set the structure to all-bits-zero, for example:

```
psa_aead_operation_t operation;
memset(&operation, 0, sizeof(operation));
```

- Initialize the structure to logical zero values, for example:

```
psa_aead_operation_t operation = {0};
```

- Initialize the structure to the initializer *PSA_AEAD_OPERATION_INIT*, for example:

```
psa_aead_operation_t operation = PSA_AEAD_OPERATION_INIT;
```

- Assign the result of the function psa_aead_operation_init() to the structure, for example:

```
psa_aead_operation_t operation;
operation = psa_aead_operation_init();
```

This is an implementation-defined `struct`. Applications should not make any assumptions about the content of this structure except as directed by the documentation of a specific implementation.

PSA_AEAD_OPERATION_INIT (macro)

```
#define PSA_AEAD_OPERATION_INIT  {0}
```

This macro returns a suitable initializer for an AEAD operation object of type *psa_aead_operation_t*.

psa_aead_encrypt (function)

```
psa_status_t psa_aead_encrypt(psa_key_handle_t handle,
                              psa_algorithm_t alg,
                              const uint8_t *nonce,
                              size_t nonce_length,
                              const uint8_t *additional_data,
                              size_t additional_data_length,
                              const uint8_t *plaintext,
                              size_t plaintext_length,
                              uint8_t *ciphertext,
                              size_t ciphertext_size,
                              size_t *ciphertext_length);
```

Parameters:

`handle` Handle to the key to use for the operation.

`alg` The AEAD algorithm to compute (`PSA_ALG_XXX` value such that *`PSA_ALG_IS_AEAD`*(`alg`) is true).

`nonce` Nonce or IV to use.

`nonce_length` Size of the `nonce` buffer in bytes.

`additional_data` Additional data that will be authenticated but not encrypted.

`additional_data_length` Size of `additional_data` in bytes.

`plaintext` Data that will be authenticated and encrypted.

`plaintext_length` Size of `plaintext` in bytes.

`ciphertext` Output buffer for the authenticated and encrypted data. The additional data is not part of this output. For algorithms where the encrypted data and the authentication tag are defined as separate outputs, the authentication tag is appended to the encrypted data.

`ciphertext_size` Size of the `ciphertext` buffer in bytes. This must be at least *`PSA_AEAD_ENCRYPT_OUTPUT_SIZE`*(`alg`, `plaintext_length`).

`ciphertext_length` On success, the size of the output in the `ciphertext` buffer.

Returns: `psa_status_t`

`PSA_SUCCESS` Success.

`PSA_ERROR_INVALID_HANDLE`

`PSA_ERROR_EMPTY_SLOT`

`PSA_ERROR_NOT_PERMITTED`

`PSA_ERROR_INVALID_ARGUMENT` `key` is not compatible with `alg`.

`PSA_ERROR_NOT_SUPPORTED` `alg` is not supported or is not an AEAD algorithm.

`PSA_ERROR_INSUFFICIENT_MEMORY`

`PSA_ERROR_COMMUNICATION_FAILURE`

`PSA_ERROR_HARDWARE_FAILURE`

`PSA_ERROR_TAMPERING_DETECTED`

`PSA_ERROR_BAD_STATE` The library has not been previously initialized by *`psa_crypto_init()`*. It is implementation-dependent whether a failure to initialize results in this error code.

Description: Process an authenticated encryption operation.

psa_aead_decrypt (function)

```
psa_status_t psa_aead_decrypt(psa_key_handle_t handle,
                              psa_algorithm_t alg,
                              const uint8_t *nonce,
                              size_t nonce_length,
                              const uint8_t *additional_data,
                              size_t additional_data_length,
                              const uint8_t *ciphertext,
                              size_t ciphertext_length,
                              uint8_t *plaintext,
                              size_t plaintext_size,
                              size_t *plaintext_length);
```

Parameters:

handle Handle to the key to use for the operation.

alg The AEAD algorithm to compute (`PSA_ALG_XXX` value such that *PSA_ALG_IS_AEAD*(`alg`) is true).

nonce Nonce or IV to use.

nonce_length Size of the `nonce` buffer in bytes.

additional_data Additional data that has been authenticated but not encrypted.

additional_data_length Size of `additional_data` in bytes.

ciphertext Data that has been authenticated and encrypted. For algorithms where the encrypted data and the authentication tag are defined as separate inputs, the buffer must contain the encrypted data followed by the authentication tag.

ciphertext_length Size of `ciphertext` in bytes.

plaintext Output buffer for the decrypted data.

plaintext_size Size of the `plaintext` buffer in bytes. This must be at least *PSA_AEAD_DECRYPT_OUTPUT_SIZE*(`alg`, `ciphertext_length`).

plaintext_length On success, the size of the output in the `plaintext` buffer.

Returns: `psa_status_t`

PSA_SUCCESS Success.

PSA_ERROR_INVALID_HANDLE

PSA_ERROR_EMPTY_SLOT

PSA_ERROR_INVALID_SIGNATURE The ciphertext is not authentic.

PSA_ERROR_NOT_PERMITTED

PSA_ERROR_INVALID_ARGUMENT `key` is not compatible with `alg`.

PSA_ERROR_NOT_SUPPORTED `alg` is not supported or is not an AEAD algorithm.

PSA_ERROR_INSUFFICIENT_MEMORY

PSA_ERROR_COMMUNICATION_FAILURE

PSA_ERROR_HARDWARE_FAILURE

PSA_ERROR_TAMPERING_DETECTED

PSA_ERROR_BAD_STATE The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

Description: Process an authenticated decryption operation.

psa_aead_operation_init (function)

```
psa_aead_operation_t psa_aead_operation_init(void);
```

Returns: `psa_aead_operation_t` **Description:** Return an initial value for an AEAD operation object.

psa_aead_encrypt_setup (function)

```
psa_status_t psa_aead_encrypt_setup(psa_aead_operation_t *operation,
                                     psa_key_handle_t handle,
                                     psa_algorithm_t alg);
```

Parameters:

operation The operation object to set up. It must have been initialized as per the documentation for *psa_aead_operation_t* and not yet in use.

handle Handle to the key to use for the operation. It must remain valid until the operation terminates.

alg The AEAD algorithm to compute (`PSA_ALG_XXX` value such that *PSA_ALG_IS_AEAD*(`alg`) is true).

Returns: `psa_status_t`

PSA_SUCCESS Success.

PSA_ERROR_INVALID_HANDLE

PSA_ERROR_EMPTY_SLOT

PSA_ERROR_NOT_PERMITTED

PSA_ERROR_INVALID_ARGUMENT `key` is not compatible with `alg`.

PSA_ERROR_NOT_SUPPORTED `alg` is not supported or is not an AEAD algorithm.

PSA_ERROR_INSUFFICIENT_MEMORY

PSA_ERROR_COMMUNICATION_FAILURE

PSA_ERROR_HARDWARE_FAILURE

PSA_ERROR_TAMPERING_DETECTED

PSA_ERROR_BAD_STATE The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

Description: Set the key for a multipart authenticated encryption operation.

The sequence of operations to encrypt a message with authentication is as follows:

1. Allocate an operation object which will be passed to all the functions listed here.
2. Initialize the operation object with one of the methods described in the documentation for *psa_aead_operation_t*, e.g. PSA_AEAD_OPERATION_INIT.
3. Call *psa_aead_encrypt_setup()* to specify the algorithm and key.

4. If needed, call *psa_aead_set_lengths()* to specify the length of the inputs to the subsequent calls to *psa_aead_update_ad()* and *psa_aead_update()*. See the documentation of *psa_aead_set_lengths()* for details.
5. Call either *psa_aead_generate_nonce()* or *psa_aead_set_nonce()* to generate or set the nonce. You should use *psa_aead_generate_nonce()* unless the protocol you are implementing requires a specific nonce value.
6. Call *psa_aead_update_ad()* zero, one or more times, passing a fragment of the non-encrypted additional authenticated data each time.
7. Call *psa_aead_update()* zero, one or more times, passing a fragment of the message to encrypt each time.
8. Call *psa_aead_finish()*.

The application may call *psa_aead_abort()* at any time after the operation has been initialized.

After a successful call to *psa_aead_encrypt_setup()*, the application must eventually terminate the operation. The following events terminate an operation:

- A failed call to any of the `psa_aead_xxx` functions.
- A call to *psa_aead_finish()*, *psa_aead_verify()* or *psa_aead_abort()*.

`psa_aead_decrypt_setup` (function)

```
psa_status_t psa_aead_decrypt_setup(psa_aead_operation_t *operation,
                                    psa_key_handle_t handle,
                                    psa_algorithm_t alg);
```

Parameters:

`operation` The operation object to set up. It must have been initialized as per the documentation for *psa_aead_operation_t* and not yet in use.

`handle` Handle to the key to use for the operation. It must remain valid until the operation terminates.

`alg` The AEAD algorithm to compute (`PSA_ALG_XXX` value such that *PSA_ALG_IS_AEAD*(`alg`) is true).

Returns: `psa_status_t`

`PSA_SUCCESS` Success.

`PSA_ERROR_INVALID_HANDLE`

`PSA_ERROR_EMPTY_SLOT`

`PSA_ERROR_NOT_PERMITTED`

`PSA_ERROR_INVALID_ARGUMENT` `key` is not compatible with `alg`.

`PSA_ERROR_NOT_SUPPORTED` `alg` is not supported or is not an AEAD algorithm.

`PSA_ERROR_INSUFFICIENT_MEMORY`

`PSA_ERROR_COMMUNICATION_FAILURE`

`PSA_ERROR_HARDWARE_FAILURE`

`PSA_ERROR_TAMPERING_DETECTED`

`PSA_ERROR_BAD_STATE` The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

Description: Set the key for a multipart authenticated decryption operation.

The sequence of operations to decrypt a message with authentication is as follows:

1. Allocate an operation object which will be passed to all the functions listed here.
2. Initialize the operation object with one of the methods described in the documentation for *psa_aead_operation_t*, e.g. PSA_AEAD_OPERATION_INIT.
3. Call *psa_aead_decrypt_setup()* to specify the algorithm and key.
4. If needed, call *psa_aead_set_lengths()* to specify the length of the inputs to the subsequent calls to *psa_aead_update_ad()* and *psa_aead_update()*. See the documentation of *psa_aead_set_lengths()* for details.
5. Call *psa_aead_set_nonce()* with the nonce for the decryption.
6. Call *psa_aead_update_ad()* zero, one or more times, passing a fragment of the non-encrypted additional authenticated data each time.
7. Call *psa_aead_update()* zero, one or more times, passing a fragment of the ciphertext to decrypt each time.
8. Call *psa_aead_verify()*.

The application may call *psa_aead_abort()* at any time after the operation has been initialized.

After a successful call to *psa_aead_decrypt_setup()*, the application must eventually terminate the operation. The following events terminate an operation:

- A failed call to any of the `psa_aead_xxx` functions.
- A call to *psa_aead_finish()*, *psa_aead_verify()* or *psa_aead_abort()*.

psa_aead_generate_nonce (function)

```
psa_status_t psa_aead_generate_nonce(psa_aead_operation_t *operation,
                                     unsigned char *nonce,
                                     size_t nonce_size,
                                     size_t *nonce_length);
```

Parameters:

operation Active AEAD operation.

nonce Buffer where the generated nonce is to be written.

nonce_size Size of the `nonce` buffer in bytes.

nonce_length On success, the number of bytes of the generated nonce.

Returns: `psa_status_t`

PSA_SUCCESS Success.

PSA_ERROR_BAD_STATE The operation state is not valid (not set up, or nonce already set).

PSA_ERROR_BUFFER_TOO_SMALL The size of the `nonce` buffer is too small.

PSA_ERROR_INSUFFICIENT_MEMORY

PSA_ERROR_COMMUNICATION_FAILURE

PSA_ERROR_HARDWARE_FAILURE

PSA_ERROR_TAMPERING_DETECTED

Description: Generate a random nonce for an authenticated encryption operation.

This function generates a random nonce for the authenticated encryption operation with an appropriate size for the chosen algorithm, key type and key size.

The application must call *psa_aead_encrypt_setup()* before calling this function.

If this function returns an error status, the operation becomes inactive.

psa_aead_set_nonce (function)

```
psa_status_t psa_aead_set_nonce(psa_aead_operation_t *operation,
                                const unsigned char *nonce,
                                size_t nonce_length);
```

Parameters:

operation Active AEAD operation.

nonce Buffer containing the nonce to use.

nonce_length Size of the nonce in bytes.

Returns: `psa_status_t`

PSA_SUCCESS Success.

PSA_ERROR_BAD_STATE The operation state is not valid (not set up, or nonce already set).

PSA_ERROR_INVALID_ARGUMENT The size of `nonce` is not acceptable for the chosen algorithm.

PSA_ERROR_INSUFFICIENT_MEMORY

PSA_ERROR_COMMUNICATION_FAILURE

PSA_ERROR_HARDWARE_FAILURE

PSA_ERROR_TAMPERING_DETECTED

Description: Set the nonce for an authenticated encryption or decryption operation.

This function sets the nonce for the authenticated encryption or decryption operation.

The application must call *psa_aead_encrypt_setup()* before calling this function.

If this function returns an error status, the operation becomes inactive.

Note: When encrypting, applications should use *psa_aead_generate_nonce()* instead of this function, unless implementing a protocol that requires a non-random IV.

psa_aead_set_lengths (function)

```
psa_status_t psa_aead_set_lengths(psa_aead_operation_t *operation,
                                  size_t ad_length,
                                  size_t plaintext_length);
```

Parameters:

operation Active AEAD operation.

ad_length Size of the non-encrypted additional authenticated data in bytes.

plaintext_length Size of the plaintext to encrypt in bytes.

Returns: psa_status_t

PSA_SUCCESS Success.

PSA_ERROR_BAD_STATE The operation state is not valid (not set up, already completed, or *psa_aead_update_ad()* or *psa_aead_update()* already called).

PSA_ERROR_INVALID_ARGUMENT At least one of the lengths is not acceptable for the chosen algorithm.

PSA_ERROR_INSUFFICIENT_MEMORY

PSA_ERROR_COMMUNICATION_FAILURE

PSA_ERROR_HARDWARE_FAILURE

PSA_ERROR_TAMPERING_DETECTED

Description: Declare the lengths of the message and additional data for AEAD.

The application must call this function before calling *psa_aead_update_ad()* or *psa_aead_update()* if the algorithm for the operation requires it. If the algorithm does not require it, calling this function is optional, but if this function is called then the implementation must enforce the lengths.

You may call this function before or after setting the nonce with *psa_aead_set_nonce()* or *psa_aead_generate_nonce()*.

- For *PSA_ALG_CCM*, calling this function is required.
- For the other AEAD algorithms defined in this specification, calling this function is not required.
- For vendor-defined algorithm, refer to the vendor documentation.

psa_aead_update_ad (function)

```
psa_status_t psa_aead_update_ad(psa_aead_operation_t *operation,
                                const uint8_t *input,
                                size_t input_length);
```

Parameters:

operation Active AEAD operation.

input Buffer containing the fragment of additional data.

input_length Size of the input buffer in bytes.

Returns: psa_status_t

PSA_SUCCESS Success.

PSA_ERROR_BAD_STATE The operation state is not valid (not set up, nonce not set, *psa_aead_update()* already called, or operation already completed).

PSA_ERROR_INVALID_ARGUMENT The total input length overflows the additional data length that was previously specified with *psa_aead_set_lengths()*.

PSA_ERROR_INSUFFICIENT_MEMORY

PSA_ERROR_COMMUNICATION_FAILURE

PSA_ERROR_HARDWARE_FAILURE

PSA_ERROR_TAMPERING_DETECTED

Description: Pass additional data to an active AEAD operation.

Additional data is authenticated, but not encrypted.

You may call this function multiple times to pass successive fragments of the additional data. You may not call this function after passing data to encrypt or decrypt with *psa_aead_update()*.

Before calling this function, you must:

1. Call either *psa_aead_encrypt_setup()* or *psa_aead_decrypt_setup()*.
2. Set the nonce with *psa_aead_generate_nonce()* or *psa_aead_set_nonce()*.

If this function returns an error status, the operation becomes inactive.

> **Warning:** When decrypting, until *psa_aead_verify()* has returned *PSA_SUCCESS*, there is no guarantee that the input is valid. Therefore, until you have called *psa_aead_verify()* and it has returned *PSA_SUCCESS*, treat the input as untrusted and prepare to undo any action that depends on the input if *psa_aead_verify()* returns an error status.

psa_aead_update (function)

```
psa_status_t psa_aead_update(psa_aead_operation_t *operation,
                             const uint8_t *input,
                             size_t input_length,
                             unsigned char *output,
                             size_t output_size,
                             size_t *output_length);
```

Parameters:

operation Active AEAD operation.

input Buffer containing the message fragment to encrypt or decrypt.

input_length Size of the `input` buffer in bytes.

output Buffer where the output is to be written.

output_size Size of the `output` buffer in bytes.

output_length On success, the number of bytes that make up the returned output.

Returns: `psa_status_t`

PSA_SUCCESS Success.

PSA_ERROR_BAD_STATE The operation state is not valid (not set up, nonce not set or already completed).

PSA_ERROR_BUFFER_TOO_SMALL The size of the `output` buffer is too small.

PSA_ERROR_INVALID_ARGUMENT The total length of input to *psa_aead_update_ad()* so far is less than the additional data length that was previously specified with *psa_aead_set_lengths()*.

PSA_ERROR_INVALID_ARGUMENT The total input length overflows the plaintext length that was previously specified with *psa_aead_set_lengths()*.

PSA_ERROR_INSUFFICIENT_MEMORY

PSA_ERROR_COMMUNICATION_FAILURE

PSA_ERROR_HARDWARE_FAILURE

PSA_ERROR_TAMPERING_DETECTED

Description: Encrypt or decrypt a message fragment in an active AEAD operation.

Before calling this function, you must:

1. Call either *psa_aead_encrypt_setup()* or *psa_aead_decrypt_setup()*. The choice of setup function determines whether this function encrypts or decrypts its input.
2. Set the nonce with *psa_aead_generate_nonce()* or *psa_aead_set_nonce()*.
3. Call *psa_aead_update_ad()* to pass all the additional data.

If this function returns an error status, the operation becomes inactive.

> **Warning:** When decrypting, until *psa_aead_verify()* has returned *PSA_SUCCESS*, there is no guarantee that the input is valid. Therefore, until you have called *psa_aead_verify()* and it has returned *PSA_SUCCESS*:
> - Do not use the output in any way other than storing it in a confidential location. If you take any action that depends on the tentative decrypted data, this action will need to be undone if the input turns out not to be valid. Furthermore, if an adversary can observe that this action took place (for example through timing), they may be able to use this fact as an oracle to decrypt any message encrypted with the same key.
> - In particular, do not copy the output anywhere but to a memory or storage space that you have exclusive access to.

psa_aead_finish (function)

```
psa_status_t psa_aead_finish(psa_aead_operation_t *operation,
                             uint8_t *ciphertext,
                             size_t ciphertext_size,
                             size_t *ciphertext_length,
                             uint8_t *tag,
                             size_t tag_size,
                             size_t *tag_length);
```

Parameters:

operation Active AEAD operation.

ciphertext Buffer where the last part of the ciphertext is to be written.

ciphertext_size Size of the `ciphertext` buffer in bytes.

ciphertext_length On success, the number of bytes of returned ciphertext.

tag Buffer where the authentication tag is to be written.

tag_size Size of the `tag` buffer in bytes.

tag_length On success, the number of bytes that make up the returned tag.

Returns: `psa_status_t`

PSA_SUCCESS Success.

PSA_ERROR_BAD_STATE The operation state is not valid (not set up, nonce not set, decryption, or already completed).

PSA_ERROR_BUFFER_TOO_SMALL The size of the `output` buffer is too small.

PSA_ERROR_INVALID_ARGUMENT The total length of input to *psa_aead_update_ad()* so far is less than the additional data length that was previously specified with *psa_aead_set_lengths()*.

PSA_ERROR_INVALID_ARGUMENT The total length of input to *psa_aead_update()* so far is less than the plaintext length that was previously specified with *psa_aead_set_lengths()*.

PSA_ERROR_INSUFFICIENT_MEMORY

PSA_ERROR_COMMUNICATION_FAILURE

PSA_ERROR_HARDWARE_FAILURE

PSA_ERROR_TAMPERING_DETECTED

Description: Finish encrypting a message in an AEAD operation.

The operation must have been set up with *psa_aead_encrypt_setup()*.

This function finishes the authentication of the additional data formed by concatenating the inputs passed to preceding calls to *psa_aead_update_ad()* with the plaintext formed by concatenating the inputs passed to preceding calls to *psa_aead_update()*.

This function has two output buffers:

- `ciphertext` contains trailing ciphertext that was buffered from preceding calls to *psa_aead_update()*. For all standard AEAD algorithms, *psa_aead_update()* does not buffer any output and therefore `ciphertext` will not contain any output and can be a 0-sized buffer.
- `tag` contains the authentication tag. Its length is always *PSA_AEAD_TAG_LENGTH*(`alg`) where `alg` is the AEAD algorithm that the operation performs.

When this function returns, the operation becomes inactive.

psa_aead_verify (function)

```
psa_status_t psa_aead_verify(psa_aead_operation_t *operation,
                             const uint8_t *tag,
                             size_t tag_length);
```

Parameters:

operation Active AEAD operation.

tag Buffer containing the authentication tag.

tag_length Size of the `tag` buffer in bytes.

Returns: `psa_status_t`

PSA_SUCCESS Success.

PSA_ERROR_BAD_STATE The operation state is not valid (not set up, nonce not set, encryption, or already completed).

PSA_ERROR_BUFFER_TOO_SMALL The size of the `output` buffer is too small.

PSA_ERROR_INVALID_ARGUMENT The total length of input to *psa_aead_update_ad()* so far is less than the additional data length that was previously specified with *psa_aead_set_lengths()*.

PSA_ERROR_INVALID_ARGUMENT The total length of input to *psa_aead_update()* so far is less than the plaintext length that was previously specified with *psa_aead_set_lengths()*.

PSA_ERROR_INSUFFICIENT_MEMORY

PSA_ERROR_COMMUNICATION_FAILURE

PSA_ERROR_HARDWARE_FAILURE

PSA_ERROR_TAMPERING_DETECTED

Description: Finish authenticating and decrypting a message in an AEAD operation.

The operation must have been set up with *psa_aead_decrypt_setup()*.

This function finishes the authentication of the additional data formed by concatenating the inputs passed to preceding calls to *psa_aead_update_ad()* with the ciphertext formed by concatenating the inputs passed to preceding calls to *psa_aead_update()*.

When this function returns, the operation becomes inactive.

psa_aead_abort (function)

```
psa_status_t psa_aead_abort(psa_aead_operation_t *operation);
```

Parameters:

operation Initialized AEAD operation.

Returns: `psa_status_t`

PSA_SUCCESS

PSA_ERROR_BAD_STATE `operation` is not an active AEAD operation.

PSA_ERROR_COMMUNICATION_FAILURE

PSA_ERROR_HARDWARE_FAILURE

PSA_ERROR_TAMPERING_DETECTED

Description: Abort an AEAD operation.

Aborting an operation frees all associated resources except for the `operation` structure itself. Once aborted, the operation object can be reused for another operation by calling *psa_aead_encrypt_setup()* or *psa_aead_decrypt_setup()* again.

You may call this function any time after the operation object has been initialized by any of the following methods:

- A call to *psa_aead_encrypt_setup()* or *psa_aead_decrypt_setup()*, whether it succeeds or not.
- Initializing the `struct` to all-bits-zero.
- Initializing the `struct` to logical zeros, e.g. `psa_aead_operation_t operation = {0}`.

In particular, calling *psa_aead_abort()* after the operation has been terminated by a call to *psa_aead_abort()* or *psa_aead_finish()* is safe and has no effect.

CHAPTER

SEVENTEEN

ASYMMETRIC CRYPTOGRAPHY

psa_asymmetric_sign (function)

```
psa_status_t psa_asymmetric_sign(psa_key_handle_t handle,
                                 psa_algorithm_t alg,
                                 const uint8_t *hash,
                                 size_t hash_length,
                                 uint8_t *signature,
                                 size_t signature_size,
                                 size_t *signature_length);
```

Parameters:

handle Handle to the key to use for the operation. It must be an asymmetric key pair.

alg A signature algorithm that is compatible with the type of `key`.

hash The hash or message to sign.

hash_length Size of the `hash` buffer in bytes.

signature Buffer where the signature is to be written.

signature_size Size of the `signature` buffer in bytes.

signature_length On success, the number of bytes that make up the returned signature value.

Returns: `psa_status_t`

PSA_SUCCESS

PSA_ERROR_BUFFER_TOO_SMALL The size of the `signature` buffer is too small. You can determine a sufficient buffer size by calling *PSA_ASYMMETRIC_SIGN_OUTPUT_SIZE*(`key_type`, `key_bits`, `alg`) where `key_type` and `key_bits` are the type and bit-size respectively of `key`.

PSA_ERROR_NOT_SUPPORTED

PSA_ERROR_INVALID_ARGUMENT

PSA_ERROR_INSUFFICIENT_MEMORY

PSA_ERROR_COMMUNICATION_FAILURE

PSA_ERROR_HARDWARE_FAILURE

PSA_ERROR_TAMPERING_DETECTED

PSA_ERROR_INSUFFICIENT_ENTROPY

PSA_ERROR_BAD_STATE The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

Description: Sign a hash or short message with a private key.

Note that to perform a hash-and-sign signature algorithm, you must first calculate the hash by calling *psa_hash_setup()*, *psa_hash_update()* and *psa_hash_finish()*. Then pass the resulting hash as the `hash` parameter to this function. You can use *PSA_ALG_SIGN_GET_HASH*(`alg`) to determine the hash algorithm to use.

psa_asymmetric_verify (function)

```
psa_status_t psa_asymmetric_verify(psa_key_handle_t handle,
                                   psa_algorithm_t alg,
                                   const uint8_t *hash,
                                   size_t hash_length,
                                   const uint8_t *signature,
                                   size_t signature_length);
```

Parameters:

handle Handle to the key to use for the operation. It must be a public key or an asymmetric key pair.

alg A signature algorithm that is compatible with the type of `key`.

hash The hash or message whose signature is to be verified.

hash_length Size of the `hash` buffer in bytes.

signature Buffer containing the signature to verify.

signature_length Size of the `signature` buffer in bytes.

Returns: `psa_status_t`

PSA_SUCCESS The signature is valid.

PSA_ERROR_INVALID_SIGNATURE The calculation was perfomed successfully, but the passed signature is not a valid signature.

PSA_ERROR_NOT_SUPPORTED

PSA_ERROR_INVALID_ARGUMENT

PSA_ERROR_INSUFFICIENT_MEMORY

PSA_ERROR_COMMUNICATION_FAILURE

PSA_ERROR_HARDWARE_FAILURE

PSA_ERROR_TAMPERING_DETECTED

PSA_ERROR_BAD_STATE The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

Description: Verify the signature a hash or short message using a public key.

Note that to perform a hash-and-sign signature algorithm, you must first calculate the hash by calling *psa_hash_setup()*, *psa_hash_update()* and *psa_hash_finish()*. Then pass the resulting hash as the `hash` parameter to this function. You can use *PSA_ALG_SIGN_GET_HASH*(`alg`) to determine the hash algorithm to use.

psa_asymmetric_encrypt (function)

```
psa_status_t psa_asymmetric_encrypt(psa_key_handle_t handle,
                                    psa_algorithm_t alg,
                                    const uint8_t *input,
                                    size_t input_length,
                                    const uint8_t *salt,
                                    size_t salt_length,
                                    uint8_t *output,
                                    size_t output_size,
                                    size_t *output_length);
```

Parameters:

handle Handle to the key to use for the operation. It must be a public key or an asymmetric key pair.

alg An asymmetric encryption algorithm that is compatible with the type of `key`.

input The message to encrypt.

input_length Size of the `input` buffer in bytes.

salt A salt or label, if supported by the encryption algorithm. If the algorithm does not support a salt, pass `NULL`. If the algorithm supports an optional salt and you do not want to pass a salt, pass `NULL`.

salt_length Size of the `salt` buffer in bytes. If `salt` is `NULL`, pass 0.

output Buffer where the encrypted message is to be written.

output_size Size of the `output` buffer in bytes.

output_length On success, the number of bytes that make up the returned output.

Returns: `psa_status_t`

PSA_SUCCESS

PSA_ERROR_BUFFER_TOO_SMALL The size of the `output` buffer is too small. You can determine a sufficient buffer size by calling *PSA_ASYMMETRIC_ENCRYPT_OUTPUT_SIZE*(`key_type`, `key_bits`, `alg`) where `key_type` and `key_bits` are the type and bit-size respectively of `key`.

PSA_ERROR_NOT_SUPPORTED

PSA_ERROR_INVALID_ARGUMENT

PSA_ERROR_INSUFFICIENT_MEMORY

PSA_ERROR_COMMUNICATION_FAILURE

PSA_ERROR_HARDWARE_FAILURE

PSA_ERROR_TAMPERING_DETECTED

PSA_ERROR_INSUFFICIENT_ENTROPY

PSA_ERROR_BAD_STATE The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

Description: Encrypt a short message with a public key.

- For *PSA_ALG_RSA_PKCS1V15_CRYPT*, no salt is supported.

psa_asymmetric_decrypt (function)

```
psa_status_t psa_asymmetric_decrypt(psa_key_handle_t handle,
                                    psa_algorithm_t alg,
                                    const uint8_t *input,
                                    size_t input_length,
                                    const uint8_t *salt,
                                    size_t salt_length,
                                    uint8_t *output,
                                    size_t output_size,
                                    size_t *output_length);
```

Parameters:

handle Handle to the key to use for the operation. It must be an asymmetric key pair.

alg An asymmetric encryption algorithm that is compatible with the type of `key`.

input The message to decrypt.

input_length Size of the `input` buffer in bytes.

salt A salt or label, if supported by the encryption algorithm. If the algorithm does not support a salt, pass `NULL`. If the algorithm supports an optional salt and you do not want to pass a salt, pass `NULL`.

salt_length Size of the `salt` buffer in bytes. If `salt` is `NULL`, pass 0.

output Buffer where the decrypted message is to be written.

output_size Size of the `output` buffer in bytes.

output_length On success, the number of bytes that make up the returned output.

Returns: `psa_status_t`

PSA_SUCCESS

PSA_ERROR_BUFFER_TOO_SMALL The size of the `output` buffer is too small. You can determine a sufficient buffer size by calling *PSA_ASYMMETRIC_DECRYPT_OUTPUT_SIZE*(`key_type`, `key_bits`, `alg`) where `key_type` and `key_bits` are the type and bit-size respectively of `key`.

PSA_ERROR_NOT_SUPPORTED

PSA_ERROR_INVALID_ARGUMENT

PSA_ERROR_INSUFFICIENT_MEMORY

PSA_ERROR_COMMUNICATION_FAILURE

PSA_ERROR_HARDWARE_FAILURE

PSA_ERROR_TAMPERING_DETECTED

PSA_ERROR_INSUFFICIENT_ENTROPY

PSA_ERROR_INVALID_PADDING

PSA_ERROR_BAD_STATE The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

Description: Decrypt a short message with a private key.

- For *PSA_ALG_RSA_PKCS1V15_CRYPT*, no salt is supported.

CHAPTER

EIGHTEEN

GENERATORS

psa_crypto_generator_t (type)

```
typedef struct psa_crypto_generator_s psa_crypto_generator_t;
```

The type of the state data structure for generators.

Before calling any function on a generator, the application must initialize it by any of the following means:

- Set the structure to all-bits-zero, for example:

```
psa_crypto_generator_t generator;
memset(&generator, 0, sizeof(generator));
```

- Initialize the structure to logical zero values, for example:

```
psa_crypto_generator_t generator = {0};
```

- Initialize the structure to the initializer `PSA_CRYPTO_GENERATOR_INIT`, for example:

```
psa_crypto_generator_t generator = PSA_CRYPTO_GENERATOR_INIT;
```

- Assign the result of the function psa_crypto_generator_init() to the structure, for example:

```
psa_crypto_generator_t generator;
generator = psa_crypto_generator_init();
```

This is an implementation-defined `struct`. Applications should not make any assumptions about the content of this structure except as directed by the documentation of a specific implementation.

PSA_CRYPTO_GENERATOR_INIT (macro)

```
#define PSA_CRYPTO_GENERATOR_INIT {0}
```

This macro returns a suitable initializer for a generator object of type `psa_crypto_generator_t`.

PSA_GENERATOR_UNBRIDLED_CAPACITY (macro)

```
#define PSA_GENERATOR_UNBRIDLED_CAPACITY ((size_t)(-1))
```

Use the maximum possible capacity for a generator.

Use this value as the capacity argument when setting up a generator to indicate that the generator should have the maximum possible capacity. The value of the maximum possible capacity depends on the generator algorithm.

psa_crypto_generator_init (function)

```
psa_crypto_generator_t psa_crypto_generator_init(void);
```

Returns: `psa_crypto_generator_t` **Description:** Return an initial value for a generator object.

psa_get_generator_capacity (function)

```
psa_status_t psa_get_generator_capacity(const psa_crypto_generator_t *generator,
                                        size_t *capacity);
```

Parameters:

generator The generator to query.

capacity On success, the capacity of the generator.

Returns: `psa_status_t`

PSA_SUCCESS

PSA_ERROR_BAD_STATE

PSA_ERROR_COMMUNICATION_FAILURE

Description: Retrieve the current capacity of a generator.

The capacity of a generator is the maximum number of bytes that it can return. Reading *N* bytes from a generator reduces its capacity by *N*.

psa_set_generator_capacity (function)

```
psa_status_t psa_set_generator_capacity(psa_crypto_generator_t *generator,
                                        size_t capacity);
```

Parameters:

generator The generator object to modify.

capacity The new capacity of the generator. It must be less or equal to the generator's current capacity.

Returns: `psa_status_t`

PSA_SUCCESS

PSA_ERROR_INVALID_ARGUMENT `capacity` is larger than the generator's current capacity.

PSA_ERROR_BAD_STATE

PSA_ERROR_COMMUNICATION_FAILURE

Description: Set the maximum capacity of a generator.

`psa_generator_read` (function)

```
psa_status_t psa_generator_read(psa_crypto_generator_t *generator,
                                uint8_t *output,
                                size_t output_length);
```

Parameters:

`generator` The generator object to read from.

`output` Buffer where the generator output will be written.

`output_length` Number of bytes to output.

Returns: `psa_status_t`

`PSA_SUCCESS`

`PSA_ERROR_INSUFFICIENT_CAPACITY` There were fewer than `output_length` bytes in the generator. Note that in this case, no output is written to the output buffer. The generator's capacity is set to 0, thus subsequent calls to this function will not succeed, even with a smaller output buffer.

`PSA_ERROR_BAD_STATE`

`PSA_ERROR_INSUFFICIENT_MEMORY`

`PSA_ERROR_COMMUNICATION_FAILURE`

`PSA_ERROR_HARDWARE_FAILURE`

`PSA_ERROR_TAMPERING_DETECTED`

Description: Read some data from a generator.

This function reads and returns a sequence of bytes from a generator. The data that is read is discarded from the generator. The generator's capacity is decreased by the number of bytes read.

`psa_generator_import_key` (function)

```
psa_status_t psa_generator_import_key(psa_key_handle_t handle,
                                      psa_key_type_t type,
                                      size_t bits,
                                      psa_crypto_generator_t *generator);
```

Parameters:

`handle` Handle to the slot where the key will be stored. It must have been obtained by calling *`psa_allocate_key()`* or *`psa_create_key()`* and must not contain key material yet.

`type` Key type (a `PSA_KEY_TYPE_XXX` value). This must be a symmetric key type.

`bits` Key size in bits.

`generator` The generator object to read from.

Returns: `psa_status_t`

`PSA_SUCCESS` Success. If the key is persistent, the key material and the key's metadata have been saved to persistent storage.

`PSA_ERROR_INSUFFICIENT_CAPACITY` There were fewer than `output_length` bytes in the generator. Note that in this case, no output is written to the output buffer. The generator's capacity is set to 0, thus subsequent calls to this function will not succeed, even with a smaller output buffer.

PSA_ERROR_NOT_SUPPORTED The key type or key size is not supported, either by the implementation in general or in this particular slot.

PSA_ERROR_BAD_STATE

PSA_ERROR_INVALID_HANDLE

PSA_ERROR_OCCUPIED_SLOT There is already a key in the specified slot.

PSA_ERROR_INSUFFICIENT_MEMORY

PSA_ERROR_INSUFFICIENT_STORAGE

PSA_ERROR_COMMUNICATION_FAILURE

PSA_ERROR_HARDWARE_FAILURE

PSA_ERROR_TAMPERING_DETECTED

PSA_ERROR_BAD_STATE The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

Description: Create a symmetric key from data read from a generator.

This function reads a sequence of bytes from a generator and imports these bytes as a key. The data that is read is discarded from the generator. The generator's capacity is decreased by the number of bytes read.

This function is equivalent to calling *psa_generator_read* and passing the resulting output to *psa_import_key*, but if the implementation provides an isolation boundary then the key material is not exposed outside the isolation boundary.

psa_generator_abort (function)

```
psa_status_t psa_generator_abort(psa_crypto_generator_t *generator);
```

Parameters:

generator The generator to abort.

Returns: `psa_status_t`

PSA_SUCCESS

PSA_ERROR_BAD_STATE

PSA_ERROR_COMMUNICATION_FAILURE

PSA_ERROR_HARDWARE_FAILURE

PSA_ERROR_TAMPERING_DETECTED

Description: Abort a generator.

Once a generator has been aborted, its capacity is zero. Aborting a generator frees all associated resources except for the `generator` structure itself.

This function may be called at any time as long as the generator object has been initialized to *PSA_CRYPTO_GENERATOR_INIT*, to psa_crypto_generator_init() or a zero value. In particular, it is valid to call *psa_generator_abort()* twice, or to call *psa_generator_abort()* on a generator that has not been set up.

Once aborted, the generator object may be called.

CHAPTER

NINETEEN

KEY DERIVATION

psa_key_derivation_step_t (type)

```
typedef uint16_t psa_key_derivation_step_t;
```

Encoding of the step of a key derivation.

PSA_KDF_STEP_SECRET (macro)

```
#define PSA_KDF_STEP_SECRET ((psa_key_derivation_step_t)0x0101)
```

A secret input for key derivation.

This must be a key of type *PSA_KEY_TYPE_DERIVE*.

PSA_KDF_STEP_LABEL (macro)

```
#define PSA_KDF_STEP_LABEL ((psa_key_derivation_step_t)0x0201)
```

A label for key derivation.

This must be a direct input.

PSA_KDF_STEP_SALT (macro)

```
#define PSA_KDF_STEP_SALT ((psa_key_derivation_step_t)0x0202)
```

A salt for key derivation.

This must be a direct input.

PSA_KDF_STEP_INFO (macro)

```
#define PSA_KDF_STEP_INFO ((psa_key_derivation_step_t)0x0203)
```

An information string for key derivation.

This must be a direct input.

psa_key_derivation_setup (function)

```
psa_status_t psa_key_derivation_setup(psa_crypto_generator_t *generator,
                                      psa_algorithm_t alg);
```

Parameters:

generator The generator object to set up. It must have been initialized but not set up yet.

alg The key derivation algorithm to compute (PSA_ALG_XXX value such that *PSA_ALG_IS_KEY_DERIVATION*(alg) is true).

Returns: psa_status_t

PSA_SUCCESS Success.

PSA_ERROR_INVALID_ARGUMENT alg is not a key derivation algorithm.

PSA_ERROR_NOT_SUPPORTED alg is not supported or is not a key derivation algorithm.

PSA_ERROR_INSUFFICIENT_MEMORY

PSA_ERROR_COMMUNICATION_FAILURE

PSA_ERROR_HARDWARE_FAILURE

PSA_ERROR_TAMPERING_DETECTED

PSA_ERROR_BAD_STATE

Description: Set up a key derivation operation.

A key derivation algorithm takes some inputs and uses them to create a byte generator which can be used to produce keys and other cryptographic material.

To use a generator for key derivation:

- Start with an initialized object of type *psa_crypto_generator_t*.
- Call *psa_key_derivation_setup()* to select the algorithm.
- Provide the inputs for the key derivation by calling *psa_key_derivation_input_bytes()* or *psa_key_derivation_input_key()* as appropriate. Which inputs are needed, in what order, and whether they may be keys and if so of what type depends on the algorithm.
- Optionally set the generator's maximum capacity with *psa_set_generator_capacity()*. You may do this before, in the middle of or after providing inputs. For some algorithms, this step is mandatory because the output depends on the maximum capacity.
- Generate output with *psa_generator_read()* or *psa_generator_import_key()*. Successive calls to these functions use successive output bytes from the generator.
- Clean up the generator object with *psa_generator_abort()*.

psa_key_derivation_input_bytes (function)

```
psa_status_t psa_key_derivation_input_bytes(psa_crypto_generator_t *generator,
                                            psa_key_derivation_step_t step,
                                            const uint8_t *data,
                                            size_t data_length);
```

Parameters:

generator The generator object to use. It must have been set up with *psa_key_derivation_setup()* and must not have produced any output yet.

step Which step the input data is for.

data Input data to use.

data_length Size of the `data` buffer in bytes.

Returns: `psa_status_t`

PSA_SUCCESS Success.

PSA_ERROR_INVALID_ARGUMENT `step` is not compatible with the generator's algorithm.

PSA_ERROR_INVALID_ARGUMENT `step` does not allow direct inputs.

PSA_ERROR_INSUFFICIENT_MEMORY

PSA_ERROR_COMMUNICATION_FAILURE

PSA_ERROR_HARDWARE_FAILURE

PSA_ERROR_TAMPERING_DETECTED

PSA_ERROR_BAD_STATE The value of `step` is not valid given the state of `generator`.

PSA_ERROR_BAD_STATE The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

Description: Provide an input for key derivation or key agreement.

Which inputs are required and in what order depends on the algorithm. Refer to the documentation of each key derivation or key agreement algorithm for information.

This function passes direct inputs. Some inputs must be passed as keys using *psa_key_derivation_input_key()* instead of this function. Refer to the documentation of individual step types for information.

psa_key_derivation_input_key (function)

```
psa_status_t psa_key_derivation_input_key(psa_crypto_generator_t *generator,
                                          psa_key_derivation_step_t step,
                                          psa_key_handle_t handle);
```

Parameters:

generator The generator object to use. It must have been set up with *psa_key_derivation_setup()* and must not have produced any output yet.

step Which step the input data is for.

handle Handle to the key. It must have an appropriate type for `step` and must allow the usage *PSA_KEY_USAGE_DERIVE*.

Returns: `psa_status_t`

PSA_SUCCESS Success.

PSA_ERROR_INVALID_HANDLE

PSA_ERROR_EMPTY_SLOT

PSA_ERROR_NOT_PERMITTED

PSA_ERROR_INVALID_ARGUMENT `step` is not compatible with the generator's algorithm.

PSA_ERROR_INVALID_ARGUMENT `step` does not allow key inputs.

PSA_ERROR_INSUFFICIENT_MEMORY

PSA_ERROR_COMMUNICATION_FAILURE

PSA_ERROR_HARDWARE_FAILURE

PSA_ERROR_TAMPERING_DETECTED

PSA_ERROR_BAD_STATE The value of `step` is not valid given the state of `generator`.

PSA_ERROR_BAD_STATE The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

Description: Provide an input for key derivation in the form of a key.

Which inputs are required and in what order depends on the algorithm. Refer to the documentation of each key derivation or key agreement algorithm for information.

This function passes key inputs. Some inputs must be passed as keys of the appropriate type using this function, while others must be passed as direct inputs using *psa_key_derivation_input_bytes()*. Refer to the documentation of individual step types for information.

psa_key_agreement (function)

```
psa_status_t psa_key_agreement(psa_crypto_generator_t *generator,
                               psa_key_derivation_step_t step,
                               psa_key_handle_t private_key,
                               const uint8_t *peer_key,
                               size_t peer_key_length);
```

Parameters:

generator The generator object to use. It must have been set up with *psa_key_derivation_setup()* with a key agreement and derivation algorithm `alg` (`PSA_ALG_XXX` value such that *PSA_ALG_IS_KEY_AGREEMENT*(`alg`) is true and *PSA_ALG_IS_RAW_KEY_AGREEMENT*(`alg`) is false). The generator must be ready for an input of the type given by `step`.

step Which step the input data is for.

private_key Handle to the private key to use.

peer_key Public key of the peer. The peer key must be in the same format that *psa_import_key()* accepts for the public key type corresponding to the type of private_key. That is, this function performs the equivalent of *psa_import_key*(`internal_public_key_handle`, *PSA_KEY_TYPE_PUBLIC_KEY_OF_KEYPAIR*(`private_key_type`), `peer_key`, `peer_key_length`) where `private_key_type` is the type of `private_key`. For example, for EC keys, this means that peer_key is interpreted as a point on the curve that the private key is on. The standard formats for public keys are documented in the documentation of *psa_export_public_key()*.

peer_key_length Size of `peer_key` in bytes.

Returns: `psa_status_t`

PSA_SUCCESS Success.

PSA_ERROR_INVALID_HANDLE

PSA_ERROR_EMPTY_SLOT

PSA_ERROR_NOT_PERMITTED

PSA_ERROR_INVALID_ARGUMENT `private_key` is not compatible with `alg`, or `peer_key` is not valid for `alg` or not compatible with `private_key`.

PSA_ERROR_NOT_SUPPORTED `alg` is not supported or is not a key derivation algorithm.

PSA_ERROR_INSUFFICIENT_MEMORY

PSA_ERROR_COMMUNICATION_FAILURE

PSA_ERROR_HARDWARE_FAILURE

PSA_ERROR_TAMPERING_DETECTED

Description: Perform a key agreement and use the shared secret as input to a key derivation.

A key agreement algorithm takes two inputs: a private key `private_key` a public key `peer_key`. The result of this function is passed as input to a key derivation. The output of this key derivation can be extracted by reading from the resulting generator to produce keys and other cryptographic material.

`psa_key_agreement_raw_shared_secret` (function)

```
psa_status_t psa_key_agreement_raw_shared_secret(psa_algorithm_t alg,
                                                 psa_key_handle_t private_key,
                                                 const uint8_t *peer_key,
                                                 size_t peer_key_length,
                                                 uint8_t *output,
                                                 size_t output_size,
                                                 size_t *output_length);
```

Parameters:

alg The key agreement algorithm to compute (`PSA_ALG_XXX` value such that *`PSA_ALG_IS_RAW_KEY_AGREEMENT`*(`alg`) is true).

private_key Handle to the private key to use.

peer_key Public key of the peer. It must be in the same format that *`psa_import_key()`* accepts. The standard formats for public keys are documented in the documentation of *`psa_export_public_key()`*.

peer_key_length Size of `peer_key` in bytes.

output Buffer where the decrypted message is to be written.

output_size Size of the `output` buffer in bytes.

output_length On success, the number of bytes that make up the returned output.

Returns: `psa_status_t`

PSA_SUCCESS Success.

PSA_ERROR_INVALID_HANDLE

PSA_ERROR_EMPTY_SLOT

PSA_ERROR_NOT_PERMITTED

PSA_ERROR_INVALID_ARGUMENT `alg` is not a key agreement algorithm

PSA_ERROR_INVALID_ARGUMENT `private_key` is not compatible with `alg`, or `peer_key` is not valid for `alg` or not compatible with `private_key`.

PSA_ERROR_NOT_SUPPORTED `alg` is not a supported key agreement algorithm.

PSA_ERROR_INSUFFICIENT_MEMORY

PSA_ERROR_COMMUNICATION_FAILURE

PSA_ERROR_HARDWARE_FAILURE

PSA_ERROR_TAMPERING_DETECTED

Description: Perform a key agreement and use the shared secret as input to a key derivation.

A key agreement algorithm takes two inputs: a private key `private_key` a public key `peer_key`.

> **Warning:** The raw result of a key agreement algorithm such as finite-field Diffie-Hellman or elliptic curve Diffie-Hellman has biases and should not be used directly as key material. It should instead be passed as input to a key derivation algorithm. To chain a key agreement with a key derivation, use *`psa_key_agreement()`* and other functions from the key derivation and generator interface.

CHAPTER

TWENTY

RANDOM GENERATION

psa_generate_key_extra_rsa (struct)

```
struct psa_generate_key_extra_rsa {
    uint32_t e;
};
```

Fields:

e Public exponent value. Default: 65537.

Description: Extra parameters for RSA key generation.

You may pass a pointer to a structure of this type as the `extra` parameter to *psa_generate_key()*.

psa_generate_random (function)

```
psa_status_t psa_generate_random(uint8_t *output,
                                 size_t output_size);
```

Parameters:

output Output buffer for the generated data.

output_size Number of bytes to generate and output.

Returns: `psa_status_t`

PSA_SUCCESS

PSA_ERROR_NOT_SUPPORTED

PSA_ERROR_INSUFFICIENT_ENTROPY

PSA_ERROR_COMMUNICATION_FAILURE

PSA_ERROR_HARDWARE_FAILURE

PSA_ERROR_TAMPERING_DETECTED

PSA_ERROR_BAD_STATE The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

Description: Generate random bytes.

> **Warning:** This function **can** fail! Callers MUST check the return status and MUST NOT use the content of the output buffer if the return status is not *PSA_SUCCESS*.

Note: To generate a key, use *psa_generate_key()* instead.

psa_generate_key (function)

```
psa_status_t psa_generate_key(psa_key_handle_t handle,
                              psa_key_type_t type,
                              size_t bits,
                              const void *extra,
                              size_t extra_size);
```

Parameters:

handle Handle to the slot where the key will be stored. It must have been obtained by calling *psa_allocate_key()* or *psa_create_key()* and must not contain key material yet.

type Key type (a PSA_KEY_TYPE_XXX value).

bits Key size in bits.

extra Extra parameters for key generation. The interpretation of this parameter depends on type. All types support NULL to use default parameters. Implementation that support the generation of vendor-specific key types that allow extra parameters shall document the format of these extra parameters and the default values. For standard parameters, the meaning of extra is as follows:

- For a symmetric key type (a type such that *PSA_KEY_TYPE_IS_ASYMMETRIC*(type) is false), extra must be NULL.
- For an elliptic curve key type (a type such that *PSA_KEY_TYPE_IS_ECC*(type) is false), extra must be NULL.
- For an RSA key (type is *PSA_KEY_TYPE_RSA_KEYPAIR*), extra is an optional *psa_generate_key_extra_rsa* structure specifying the public exponent. The default public exponent used when extra is NULL is 65537.
- For an DSA key (type is *PSA_KEY_TYPE_DSA_KEYPAIR*), extra is an optional structure specifying the key domain parameters. The key domain parameters can also be provided by *psa_set_key_domain_parameters()*, which documents the format of the structure.
- For a DH key (type is *PSA_KEY_TYPE_DH_KEYPAIR*), the extra is an optional structure specifying the key domain parameters. The key domain parameters can also be provided by *psa_set_key_domain_parameters()*, which documents the format of the structure.

extra_size Size of the buffer that extra points to, in bytes. Note that if extra is NULL then extra_size must be zero.

Returns: psa_status_t

PSA_SUCCESS Success. If the key is persistent, the key material and the key's metadata have been saved to persistent storage.

PSA_ERROR_INVALID_HANDLE

PSA_ERROR_OCCUPIED_SLOT There is already a key in the specified slot.

PSA_ERROR_NOT_SUPPORTED

PSA_ERROR_INVALID_ARGUMENT

PSA_ERROR_INSUFFICIENT_MEMORY

PSA_ERROR_INSUFFICIENT_ENTROPY

PSA_ERROR_COMMUNICATION_FAILURE

PSA_ERROR_HARDWARE_FAILURE

PSA_ERROR_TAMPERING_DETECTED

PSA_ERROR_BAD_STATE The library has not been previously initialized by *psa_crypto_init()*. It is implementation-dependent whether a failure to initialize results in this error code.

Description: Generate a key or key pair.

CHAPTER

TWENTYONE

ERROR CODES

psa_status_t (type)

```
typedef int32_t psa_status_t;
```

Function return status.

This is either *PSA_SUCCESS* (which is zero), indicating success, or a nonzero value indicating that an error occurred. Errors are encoded as one of the PSA_ERROR_xxx values defined here.

PSA_SUCCESS (macro)

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

The action was completed successfully.

PSA_ERROR_UNKNOWN_ERROR (macro)

```
#define PSA_ERROR_UNKNOWN_ERROR  ((psa_status_t)1)
```

An error occurred that does not correspond to any defined failure cause.

Implementations may use this error code if none of the other standard error codes are applicable.

PSA_ERROR_NOT_SUPPORTED (macro)

```
#define PSA_ERROR_NOT_SUPPORTED  ((psa_status_t)2)
```

The requested operation or a parameter is not supported by this implementation.

Implementations should return this error code when an enumeration parameter such as a key type, algorithm, etc. is not recognized. If a combination of parameters is recognized and identified as not valid, return *PSA_ERROR_INVALID_ARGUMENT* instead.

PSA_ERROR_NOT_PERMITTED (macro)

```
#define PSA_ERROR_NOT_PERMITTED ((psa_status_t)3)
```

The requested action is denied by a policy.

Implementations should return this error code when the 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, it is unspecified whether the function returns *PSA_ERROR_NOT_PERMITTED*, *PSA_ERROR_NOT_SUPPORTED* or *PSA_ERROR_INVALID_ARGUMENT*.

PSA_ERROR_BUFFER_TOO_SMALL (macro)

```
#define PSA_ERROR_BUFFER_TOO_SMALL ((psa_status_t)4)
```

An output buffer is too small.

Applications can call the `PSA_xxx_SIZE` macro listed in the function description to determine a sufficient buffer size.

Implementations should preferably return this error code only in cases when performing the operation with a larger output buffer would succeed. However implementations may return this error if a function has invalid or unsupported parameters in addition to the parameters that determine the necessary output buffer size.

PSA_ERROR_OCCUPIED_SLOT (macro)

```
#define PSA_ERROR_OCCUPIED_SLOT ((psa_status_t)5)
```

A slot is occupied, but must be empty to carry out the requested action.

If a handle is invalid, it does not designate an occupied slot. The error for an invalid handle is *PSA_ERROR_INVALID_HANDLE*.

PSA_ERROR_EMPTY_SLOT (macro)

```
#define PSA_ERROR_EMPTY_SLOT ((psa_status_t)6)
```

A slot is empty, but must be occupied to carry out the requested action.

If a handle is invalid, it does not designate an empty slot. The error for an invalid handle is *PSA_ERROR_INVALID_HANDLE*.

PSA_ERROR_BAD_STATE (macro)

```
#define PSA_ERROR_BAD_STATE ((psa_status_t)7)
```

The requested action cannot be performed in the current state.

Multipart operations return this error when one of the functions is called out of sequence. Refer to the function descriptions for permitted sequencing of functions.

Implementations shall not return this error code to indicate that a key slot is occupied when it needs to be free or vice versa, but shall return *PSA_ERROR_OCCUPIED_SLOT* or *PSA_ERROR_EMPTY_SLOT* as applicable.

PSA_ERROR_INVALID_ARGUMENT (macro)

```
#define PSA_ERROR_INVALID_ARGUMENT  ((psa_status_t)8)
```

The parameters passed to the function are invalid.

Implementations may return this error any time a parameter or combination of parameters are recognized as invalid.

Implementations shall not return this error code to indicate that a key slot is occupied when it needs to be free or vice versa, but shall return *PSA_ERROR_OCCUPIED_SLOT* or *PSA_ERROR_EMPTY_SLOT* as applicable.

Implementation shall not return this error code to indicate that a key handle is invalid, but shall return *PSA_ERROR_INVALID_HANDLE* instead.

PSA_ERROR_INSUFFICIENT_MEMORY (macro)

```
#define PSA_ERROR_INSUFFICIENT_MEMORY  ((psa_status_t)9)
```

There is not enough runtime memory.

If the action is carried out across multiple security realms, this error can refer to available memory in any of the security realms.

PSA_ERROR_INSUFFICIENT_STORAGE (macro)

```
#define PSA_ERROR_INSUFFICIENT_STORAGE  ((psa_status_t)10)
```

There is not enough persistent storage.

Functions that modify the key storage return this error code if there is insufficient storage space on the host media. In addition, many functions that do not otherwise access storage may 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_COMMUNICATION_FAILURE (macro)

```
#define PSA_ERROR_COMMUNICATION_FAILURE  ((psa_status_t)11)
```

There was a communication failure inside the implementation.

This can indicate a communication failure between the application and an external cryptoprocessor or between the cryptoprocessor and an external volatile or persistent memory. A communication failure may be transient or permanent depending on the cause.

> **Warning:** If a function returns this error, it is undetermined whether the requested action has completed or not. Implementations should return *PSA_SUCCESS* on successful completion whenver possible, however functions may return *PSA_ERROR_COMMUNICATION_FAILURE* if the requested action was completed successfully in an external cryptoprocessor but there was a breakdown of communication before the cryptoprocessor could report the status to the application.

PSA_ERROR_STORAGE_FAILURE (macro)

```
#define PSA_ERROR_STORAGE_FAILURE  ((psa_status_t)12)
```

There was a storage failure that may have led to data loss.

This error indicates that some persistent storage is corrupted. It should not be used for a corruption of volatile memory (use *PSA_ERROR_TAMPERING_DETECTED*), for a communication error between the cryptoprocessor and its external storage (use *PSA_ERROR_COMMUNICATION_FAILURE*), or when the storage is in a valid state but is full (use *PSA_ERROR_INSUFFICIENT_STORAGE*).

Note that 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.

When a storage failure occurs, it is no longer possible to ensure the global integrity of the keystore. Depending on the global integrity guarantees offered by the implementation, access to other data may or may not fail even if the data is still readable but its integrity cannot be guaranteed.

Implementations 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 (macro)

```
#define PSA_ERROR_HARDWARE_FAILURE  ((psa_status_t)13)
```

A hardware failure was detected.

A hardware failure may be transient or permanent depending on the cause.

PSA_ERROR_TAMPERING_DETECTED (macro)

```
#define PSA_ERROR_TAMPERING_DETECTED  ((psa_status_t)14)
```

A tampering attempt was detected.

If an application receives this error code, there is no guarantee that previously accessed or computed data was correct and remains confidential. Applications should not perform any security function and should enter a safe failure state.

Implementations may return this error code if they detect an invalid state that cannot happen during normal operation and that indicates that the implementation's security guarantees no longer hold. Depending on the implementation architecture and on its security and safety goals, the implementation may forcibly terminate the application.

This error code is intended as a last resort when a security breach is detected and it is unsure whether the keystore data is still protected. Implementations shall only return this error code to report an alarm from a tampering detector, to indicate that the confidentiality of stored data can no longer be guaranteed, or to indicate that the integrity of previously returned data is now considered compromised. Implementations shall not use this error code to indicate a hardware failure that merely makes it impossible to perform the requested operation (use *PSA_ERROR_COMMUNICATION_FAILURE*, *PSA_ERROR_STORAGE_FAILURE*, *PSA_ERROR_HARDWARE_FAILURE*, *PSA_ERROR_INSUFFICIENT_ENTROPY* or other applicable error code instead).

This error indicates an attack against the application. Implementations shall not return this error code as a consequence of the behavior of the application itself.

PSA_ERROR_INSUFFICIENT_ENTROPY (macro)

```
#define PSA_ERROR_INSUFFICIENT_ENTROPY  ((psa_status_t)15)
```

There is not enough entropy to generate random data needed for the requested action.

This error indicates a failure of a hardware random generator. Application writers should note that this error can be returned not only by functions whose purpose is to generate random data, such as key, IV or nonce generation, but also by functions that execute an algorithm with a randomized result, as well as functions that use randomization of intermediate computations as a countermeasure to certain attacks.

Implementations should avoid returning this error after *psa_crypto_init()* has succeeded. Implementations should generate sufficient entropy during initialization and subsequently use a cryptographically secure pseudorandom generator (PRNG). However implementations may return this error at any time if a policy requires the PRNG to be reseeded during normal operation.

PSA_ERROR_INVALID_SIGNATURE (macro)

```
#define PSA_ERROR_INVALID_SIGNATURE  ((psa_status_t)16)
```

The signature, MAC or hash is incorrect.

Verification functions return this error if the verification calculations completed successfully, and the value to be verified was determined to be incorrect.

If the value to verify has an invalid size, implementations may return either *PSA_ERROR_INVALID_ARGUMENT* or *PSA_ERROR_INVALID_SIGNATURE*.

PSA_ERROR_INVALID_PADDING (macro)

```
#define PSA_ERROR_INVALID_PADDING  ((psa_status_t)17)
```

The decrypted padding is incorrect.

> **Warning:** In some protocols, when decrypting data, it is essential that the behavior of the application does not depend on whether the padding is correct, down to precise timing. Applications should prefer protocols that use authenticated encryption rather than plain encryption. If the application must perform a decryption of unauthenticated data, the application writer should take care not to reveal whether the padding is invalid.

Implementations should strive to make valid and invalid padding as close as possible to indistinguishable to an external observer. In particular, the timing of a decryption operation should not depend on the validity of the padding.

PSA_ERROR_INSUFFICIENT_CAPACITY (macro)

```
#define PSA_ERROR_INSUFFICIENT_CAPACITY  ((psa_status_t)18)
```

The generator has insufficient capacity left.

Once a function returns this error, attempts to read from the generator will always return this error.

PSA_ERROR_INVALID_HANDLE (macro)

```
#define PSA_ERROR_INVALID_HANDLE  ((psa_status_t)19)
```

The key handle is not valid.

CHAPTER

TWENTYTWO

KEY AND ALGORITHM TYPES

psa_key_type_t (type)

```
typedef uint32_t psa_key_type_t;
```

Encoding of a key type.

psa_ecc_curve_t (type)

```
typedef uint16_t psa_ecc_curve_t;
```

The type of PSA elliptic curve identifiers.

psa_algorithm_t (type)

```
typedef uint32_t psa_algorithm_t;
```

Encoding of a cryptographic algorithm.

For algorithms that can be applied to multiple key types, this type does not encode the key type. For example, for symmetric ciphers based on a block cipher, *psa_algorithm_t* encodes the block cipher mode and the padding mode while the block cipher itself is encoded via *psa_key_type_t*.

PSA_KEY_TYPE_NONE (macro)

```
#define PSA_KEY_TYPE_NONE  ((psa_key_type_t)0x00000000)
```

An invalid key type value.

Zero is not the encoding of any key type.

PSA_KEY_TYPE_VENDOR_FLAG (macro)

```
#define PSA_KEY_TYPE_VENDOR_FLAG  ((psa_key_type_t)0x80000000)
```

Vendor-defined flag

Key types defined by this standard will never have the *PSA_KEY_TYPE_VENDOR_FLAG* bit set. Vendors who define additional key types must use an encoding with the *PSA_KEY_TYPE_VENDOR_FLAG* bit set and should respect the bitwise structure used by standard encodings whenever practical.

PSA_KEY_TYPE_CATEGORY_MASK (macro)

```
#define PSA_KEY_TYPE_CATEGORY_MASK  ((psa_key_type_t)0x70000000)
```

PSA_KEY_TYPE_CATEGORY_SYMMETRIC (macro)

```
#define PSA_KEY_TYPE_CATEGORY_SYMMETRIC  ((psa_key_type_t)0x40000000)
```

PSA_KEY_TYPE_CATEGORY_RAW (macro)

```
#define PSA_KEY_TYPE_CATEGORY_RAW  ((psa_key_type_t)0x50000000)
```

PSA_KEY_TYPE_CATEGORY_PUBLIC_KEY (macro)

```
#define PSA_KEY_TYPE_CATEGORY_PUBLIC_KEY  ((psa_key_type_t)0x60000000)
```

PSA_KEY_TYPE_CATEGORY_KEY_PAIR (macro)

```
#define PSA_KEY_TYPE_CATEGORY_KEY_PAIR  ((psa_key_type_t)0x70000000)
```

PSA_KEY_TYPE_CATEGORY_FLAG_PAIR (macro)

```
#define PSA_KEY_TYPE_CATEGORY_FLAG_PAIR  ((psa_key_type_t)0x10000000)
```

PSA_KEY_TYPE_IS_VENDOR_DEFINED (macro)

```
#define PSA_KEY_TYPE_IS_VENDOR_DEFINED( type) \
    (((type) & PSA_KEY_TYPE_VENDOR_FLAG) != 0)
```

Parameters:

type

Description: Whether a key type is vendor-defined.

PSA_KEY_TYPE_IS_UNSTRUCTURED (macro)

```
#define PSA_KEY_TYPE_IS_UNSTRUCTURED( type) \
    (((type) & PSA_KEY_TYPE_CATEGORY_MASK & ~(psa_key_type_t)0x10000000) == PSA_KEY_TYPE_CATEGORY_SYM
```

Parameters:

type

Description: Whether a key type is an unstructured array of bytes.

This encompasses both symmetric keys and non-key data.

PSA_KEY_TYPE_IS_ASYMMETRIC (macro)

```
#define PSA_KEY_TYPE_IS_ASYMMETRIC( type) \
    (((type) & PSA_KEY_TYPE_CATEGORY_MASK & ~PSA_KEY_TYPE_CATEGORY_FLAG_PAIR) == PSA_KEY_TYPE_CATEGOR
```

Parameters:

type

Description: Whether a key type is asymmetric: either a key pair or a public key.

PSA_KEY_TYPE_IS_PUBLIC_KEY (macro)

```
#define PSA_KEY_TYPE_IS_PUBLIC_KEY( type) \
    (((type) & PSA_KEY_TYPE_CATEGORY_MASK) == PSA_KEY_TYPE_CATEGORY_PUBLIC_KEY)
```

Parameters:

type

Description: Whether a key type is the public part of a key pair.

PSA_KEY_TYPE_IS_KEYPAIR (macro)

```
#define PSA_KEY_TYPE_IS_KEYPAIR( type) \
    (((type) & PSA_KEY_TYPE_CATEGORY_MASK) == PSA_KEY_TYPE_CATEGORY_KEY_PAIR)
```

Parameters:

type

Description: Whether a key type is a key pair containing a private part and a public part.

PSA_KEY_TYPE_KEYPAIR_OF_PUBLIC_KEY (macro)

```
#define PSA_KEY_TYPE_KEYPAIR_OF_PUBLIC_KEY( type) \
    ((type) | PSA_KEY_TYPE_CATEGORY_FLAG_PAIR)
```

Parameters:

type A public key type or key pair type.

Returns:

The corresponding key pair type. If `type` is not a public key or a key pair, the return value is undefined. **Description:** The key pair type corresponding to a public key type.

You may also pass a key pair type as `type`, it will be left unchanged.

PSA_KEY_TYPE_PUBLIC_KEY_OF_KEYPAIR (macro)

```
#define PSA_KEY_TYPE_PUBLIC_KEY_OF_KEYPAIR( type) \
    ((type) & ~PSA_KEY_TYPE_CATEGORY_FLAG_PAIR)
```

Parameters:

type A public key type or key pair type.

Returns:

The corresponding public key type. If `type` is not a public key or a key pair, the return value is undefined. **Description:** The public key type corresponding to a key pair type.

You may also pass a key pair type as `type`, it will be left unchanged.

PSA_KEY_TYPE_RAW_DATA (macro)

```
#define PSA_KEY_TYPE_RAW_DATA  ((psa_key_type_t)0x50000001)
```

Raw data.

A "key" of this type cannot be used for any cryptographic operation. Applications may use this type to store arbitrary data in the keystore.

PSA_KEY_TYPE_HMAC (macro)

```
#define PSA_KEY_TYPE_HMAC  ((psa_key_type_t)0x51000000)
```

HMAC key.

The key policy determines which underlying hash algorithm the key can be used for.

HMAC keys should generally have the same size as the underlying hash. This size can be calculated with *PSA_HASH_SIZE*(`alg`) where `alg` is the HMAC algorithm or the underlying hash algorithm.

PSA_KEY_TYPE_DERIVE (macro)

```
#define PSA_KEY_TYPE_DERIVE  ((psa_key_type_t)0x52000000)
```

A secret for key derivation.

The key policy determines which key derivation algorithm the key can be used for.

PSA_KEY_TYPE_AES (macro)

```
#define PSA_KEY_TYPE_AES   ((psa_key_type_t)0x40000001)
```

Key for an cipher, AEAD or MAC algorithm based on the AES block cipher.

The size of the key can be 16 bytes (AES-128), 24 bytes (AES-192) or 32 bytes (AES-256).

PSA_KEY_TYPE_DES (macro)

```
#define PSA_KEY_TYPE_DES   ((psa_key_type_t)0x40000002)
```

Key for a cipher or MAC algorithm based on DES or 3DES (Triple-DES).

The size of the key can be 8 bytes (single DES), 16 bytes (2-key 3DES) or 24 bytes (3-key 3DES).

Note that single DES and 2-key 3DES are weak and strongly deprecated and should only be used to decrypt legacy data. 3-key 3DES is weak and deprecated and should only be used in legacy protocols.

PSA_KEY_TYPE_CAMELLIA (macro)

```
#define PSA_KEY_TYPE_CAMELLIA   ((psa_key_type_t)0x40000003)
```

Key for an cipher, AEAD or MAC algorithm based on the Camellia block cipher.

PSA_KEY_TYPE_ARC4 (macro)

```
#define PSA_KEY_TYPE_ARC4   ((psa_key_type_t)0x40000004)
```

Key for the RC4 stream cipher.

Note that RC4 is weak and deprecated and should only be used in legacy protocols.

PSA_KEY_TYPE_RSA_PUBLIC_KEY (macro)

```
#define PSA_KEY_TYPE_RSA_PUBLIC_KEY   ((psa_key_type_t)0x60010000)
```

RSA public key.

PSA_KEY_TYPE_RSA_KEYPAIR (macro)

```
#define PSA_KEY_TYPE_RSA_KEYPAIR   ((psa_key_type_t)0x70010000)
```

RSA key pair (private and public key).

PSA_KEY_TYPE_IS_RSA (macro)

```
#define PSA_KEY_TYPE_IS_RSA( type) \
    (PSA_KEY_TYPE_PUBLIC_KEY_OF_KEYPAIR(type) == PSA_KEY_TYPE_RSA_PUBLIC_KEY)
```

Parameters:

type

Description: Whether a key type is an RSA key (pair or public-only).

PSA_KEY_TYPE_DSA_PUBLIC_KEY (macro)

```
#define PSA_KEY_TYPE_DSA_PUBLIC_KEY  ((psa_key_type_t)0x60020000)
```

DSA public key.

PSA_KEY_TYPE_DSA_KEYPAIR (macro)

```
#define PSA_KEY_TYPE_DSA_KEYPAIR  ((psa_key_type_t)0x70020000)
```

DSA key pair (private and public key).

PSA_KEY_TYPE_IS_DSA (macro)

```
#define PSA_KEY_TYPE_IS_DSA( type) \
    (PSA_KEY_TYPE_PUBLIC_KEY_OF_KEYPAIR(type) == PSA_KEY_TYPE_DSA_PUBLIC_KEY)
```

Parameters:

type

Description: Whether a key type is an DSA key (pair or public-only).

PSA_KEY_TYPE_ECC_PUBLIC_KEY_BASE (macro)

```
#define PSA_KEY_TYPE_ECC_PUBLIC_KEY_BASE  ((psa_key_type_t)0x60030000)
```

PSA_KEY_TYPE_ECC_KEYPAIR_BASE (macro)

```
#define PSA_KEY_TYPE_ECC_KEYPAIR_BASE  ((psa_key_type_t)0x70030000)
```

PSA_KEY_TYPE_ECC_CURVE_MASK (macro)

```
#define PSA_KEY_TYPE_ECC_CURVE_MASK  ((psa_key_type_t)0x0000ffff)
```

PSA_KEY_TYPE_ECC_KEYPAIR (macro)

```
#define PSA_KEY_TYPE_ECC_KEYPAIR( curve) \
    (PSA_KEY_TYPE_ECC_KEYPAIR_BASE | (curve))
```

Parameters:

curve

Description: Elliptic curve key pair.

PSA_KEY_TYPE_ECC_PUBLIC_KEY (macro)

```
#define PSA_KEY_TYPE_ECC_PUBLIC_KEY( curve) \
    (PSA_KEY_TYPE_ECC_PUBLIC_KEY_BASE | (curve))
```

Parameters:

curve

Description: Elliptic curve public key.

PSA_KEY_TYPE_IS_ECC (macro)

```
#define PSA_KEY_TYPE_IS_ECC( type) \
    ((PSA_KEY_TYPE_PUBLIC_KEY_OF_KEYPAIR(type) & ~PSA_KEY_TYPE_ECC_CURVE_MASK) == PSA_KEY_TYPE_ECC_PU
```

Parameters:

type

Description: Whether a key type is an elliptic curve key (pair or public-only).

PSA_KEY_TYPE_IS_ECC_KEYPAIR (macro)

```
#define PSA_KEY_TYPE_IS_ECC_KEYPAIR( type) \
    (((type) & ~PSA_KEY_TYPE_ECC_CURVE_MASK) == PSA_KEY_TYPE_ECC_KEYPAIR_BASE)
```

Parameters:

type

Description: Whether a key type is an elliptic curve key pair.

PSA_KEY_TYPE_IS_ECC_PUBLIC_KEY (macro)

```
#define PSA_KEY_TYPE_IS_ECC_PUBLIC_KEY( type) \
    (((type) & ~PSA_KEY_TYPE_ECC_CURVE_MASK) == PSA_KEY_TYPE_ECC_PUBLIC_KEY_BASE)
```

Parameters:

`type`

Description: Whether a key type is an elliptic curve public key.

`PSA_KEY_TYPE_GET_CURVE` (macro)

```
#define PSA_KEY_TYPE_GET_CURVE( type) \
    ((psa_ecc_curve_t) (PSA_KEY_TYPE_IS_ECC(type) ? ((type) & PSA_KEY_TYPE_ECC_CURVE_MASK) : 0))
```

Parameters:

`type`

Description: Extract the curve from an elliptic curve key type.

`PSA_ECC_CURVE_SECT163K1` (macro)

```
#define PSA_ECC_CURVE_SECT163K1  ((psa_ecc_curve_t) 0x0001)
```

`PSA_ECC_CURVE_SECT163R1` (macro)

```
#define PSA_ECC_CURVE_SECT163R1  ((psa_ecc_curve_t) 0x0002)
```

`PSA_ECC_CURVE_SECT163R2` (macro)

```
#define PSA_ECC_CURVE_SECT163R2  ((psa_ecc_curve_t) 0x0003)
```

`PSA_ECC_CURVE_SECT193R1` (macro)

```
#define PSA_ECC_CURVE_SECT193R1  ((psa_ecc_curve_t) 0x0004)
```

`PSA_ECC_CURVE_SECT193R2` (macro)

```
#define PSA_ECC_CURVE_SECT193R2  ((psa_ecc_curve_t) 0x0005)
```

`PSA_ECC_CURVE_SECT233K1` (macro)

```
#define PSA_ECC_CURVE_SECT233K1  ((psa_ecc_curve_t) 0x0006)
```

`PSA_ECC_CURVE_SECT233R1` (macro)

```
#define PSA_ECC_CURVE_SECT233R1  ((psa_ecc_curve_t) 0x0007)
```

`PSA_ECC_CURVE_SECT239K1` (macro)

```
#define PSA_ECC_CURVE_SECT239K1  ((psa_ecc_curve_t) 0x0008)
```

`PSA_ECC_CURVE_SECT283K1` (macro)

```
#define PSA_ECC_CURVE_SECT283K1  ((psa_ecc_curve_t) 0x0009)
```

`PSA_ECC_CURVE_SECT283R1` (macro)

```
#define PSA_ECC_CURVE_SECT283R1  ((psa_ecc_curve_t) 0x000a)
```

`PSA_ECC_CURVE_SECT409K1` (macro)

```
#define PSA_ECC_CURVE_SECT409K1  ((psa_ecc_curve_t) 0x000b)
```

`PSA_ECC_CURVE_SECT409R1` (macro)

```
#define PSA_ECC_CURVE_SECT409R1  ((psa_ecc_curve_t) 0x000c)
```

`PSA_ECC_CURVE_SECT571K1` (macro)

```
#define PSA_ECC_CURVE_SECT571K1  ((psa_ecc_curve_t) 0x000d)
```

`PSA_ECC_CURVE_SECT571R1` (macro)

```
#define PSA_ECC_CURVE_SECT571R1  ((psa_ecc_curve_t) 0x000e)
```

`PSA_ECC_CURVE_SECP160K1` (macro)

```
#define PSA_ECC_CURVE_SECP160K1  ((psa_ecc_curve_t) 0x000f)
```

PSA_ECC_CURVE_SECP160R1 (macro)

```
#define PSA_ECC_CURVE_SECP160R1  ((psa_ecc_curve_t) 0x0010)
```

PSA_ECC_CURVE_SECP160R2 (macro)

```
#define PSA_ECC_CURVE_SECP160R2  ((psa_ecc_curve_t) 0x0011)
```

PSA_ECC_CURVE_SECP192K1 (macro)

```
#define PSA_ECC_CURVE_SECP192K1  ((psa_ecc_curve_t) 0x0012)
```

PSA_ECC_CURVE_SECP192R1 (macro)

```
#define PSA_ECC_CURVE_SECP192R1  ((psa_ecc_curve_t) 0x0013)
```

PSA_ECC_CURVE_SECP224K1 (macro)

```
#define PSA_ECC_CURVE_SECP224K1  ((psa_ecc_curve_t) 0x0014)
```

PSA_ECC_CURVE_SECP224R1 (macro)

```
#define PSA_ECC_CURVE_SECP224R1  ((psa_ecc_curve_t) 0x0015)
```

PSA_ECC_CURVE_SECP256K1 (macro)

```
#define PSA_ECC_CURVE_SECP256K1  ((psa_ecc_curve_t) 0x0016)
```

PSA_ECC_CURVE_SECP256R1 (macro)

```
#define PSA_ECC_CURVE_SECP256R1  ((psa_ecc_curve_t) 0x0017)
```

PSA_ECC_CURVE_SECP384R1 (macro)

```
#define PSA_ECC_CURVE_SECP384R1  ((psa_ecc_curve_t) 0x0018)
```

PSA_ECC_CURVE_SECP521R1 (macro)

```
#define PSA_ECC_CURVE_SECP521R1  ((psa_ecc_curve_t) 0x0019)
```

PSA_ECC_CURVE_BRAINPOOL_P256R1 (macro)

```
#define PSA_ECC_CURVE_BRAINPOOL_P256R1  ((psa_ecc_curve_t) 0x001a)
```

PSA_ECC_CURVE_BRAINPOOL_P384R1 (macro)

```
#define PSA_ECC_CURVE_BRAINPOOL_P384R1  ((psa_ecc_curve_t) 0x001b)
```

PSA_ECC_CURVE_BRAINPOOL_P512R1 (macro)

```
#define PSA_ECC_CURVE_BRAINPOOL_P512R1  ((psa_ecc_curve_t) 0x001c)
```

PSA_ECC_CURVE_CURVE25519 (macro)

```
#define PSA_ECC_CURVE_CURVE25519  ((psa_ecc_curve_t) 0x001d)
```

PSA_ECC_CURVE_CURVE448 (macro)

```
#define PSA_ECC_CURVE_CURVE448  ((psa_ecc_curve_t) 0x001e)
```

PSA_KEY_TYPE_DH_PUBLIC_KEY (macro)

```
#define PSA_KEY_TYPE_DH_PUBLIC_KEY  ((psa_key_type_t)0x60040000)
```

Diffie-Hellman key exchange public key.

PSA_KEY_TYPE_DH_KEYPAIR (macro)

```
#define PSA_KEY_TYPE_DH_KEYPAIR  ((psa_key_type_t)0x70040000)
```

Diffie-Hellman key exchange key pair (private and public key).

PSA_KEY_TYPE_IS_DH (macro)

```
#define PSA_KEY_TYPE_IS_DH( type) \
    (PSA_KEY_TYPE_PUBLIC_KEY_OF_KEYPAIR(type) == PSA_KEY_TYPE_DH_PUBLIC_KEY)
```

Parameters:

type

Description: Whether a key type is a Diffie-Hellman key exchange key (pair or public-only).

PSA_BLOCK_CIPHER_BLOCK_SIZE (macro)

```
#define PSA_BLOCK_CIPHER_BLOCK_SIZE( type) \
    ( (type) == PSA_KEY_TYPE_AES ? 16 : (type) == PSA_KEY_TYPE_DES ? 8 : (type) == PSA_KEY_TYPE_CAMEL
```

Parameters:

type A cipher key type (value of type *psa_key_type_t*).

Returns:

The block size for a block cipher, or 1 for a stream cipher. The return value is undefined if `type` is not a supported cipher key type. **Description:** The block size of a block cipher.

Note: It is possible to build stream cipher algorithms on top of a block cipher, for example CTR mode (*PSA_ALG_CTR*). This macro only takes the key type into account, so it cannot be used to determine the size of the data that *psa_cipher_update()* might buffer for future processing in general.

Note: This macro returns a compile-time constant if its argument is one.

Warning: This macro may evaluate its argument multiple times.

PSA_ALG_VENDOR_FLAG (macro)

```
#define PSA_ALG_VENDOR_FLAG  ((psa_algorithm_t)0x80000000)
```

PSA_ALG_CATEGORY_MASK (macro)

```
#define PSA_ALG_CATEGORY_MASK  ((psa_algorithm_t)0x7f000000)
```

PSA_ALG_CATEGORY_HASH (macro)

```
#define PSA_ALG_CATEGORY_HASH  ((psa_algorithm_t)0x01000000)
```

PSA_ALG_CATEGORY_MAC (macro)

```
#define PSA_ALG_CATEGORY_MAC  ((psa_algorithm_t)0x02000000)
```

PSA_ALG_CATEGORY_CIPHER (macro)

```
#define PSA_ALG_CATEGORY_CIPHER  ((psa_algorithm_t)0x04000000)
```

PSA_ALG_CATEGORY_AEAD (macro)

```
#define PSA_ALG_CATEGORY_AEAD  ((psa_algorithm_t)0x06000000)
```

PSA_ALG_CATEGORY_SIGN (macro)

```
#define PSA_ALG_CATEGORY_SIGN  ((psa_algorithm_t)0x10000000)
```

PSA_ALG_CATEGORY_ASYMMETRIC_ENCRYPTION (macro)

```
#define PSA_ALG_CATEGORY_ASYMMETRIC_ENCRYPTION  ((psa_algorithm_t)0x12000000)
```

PSA_ALG_CATEGORY_KEY_DERIVATION (macro)

```
#define PSA_ALG_CATEGORY_KEY_DERIVATION  ((psa_algorithm_t)0x20000000)
```

PSA_ALG_CATEGORY_KEY_AGREEMENT (macro)

```
#define PSA_ALG_CATEGORY_KEY_AGREEMENT  ((psa_algorithm_t)0x30000000)
```

PSA_ALG_IS_VENDOR_DEFINED (macro)

```
#define PSA_ALG_IS_VENDOR_DEFINED( alg)  (((alg) & PSA_ALG_VENDOR_FLAG) != 0)
```

Parameters:

`alg`

Description:

PSA_ALG_IS_HASH (macro)

```
#define PSA_ALG_IS_HASH( alg) \
    (((alg) & PSA_ALG_CATEGORY_MASK) == PSA_ALG_CATEGORY_HASH)
```

Parameters:

alg An algorithm identifier (value of type *psa_algorithm_t*).

Returns:

1 if alg is a hash algorithm, 0 otherwise. This macro may return either 0 or 1 if alg is not a supported algorithm identifier. **Description:** Whether the specified algorithm is a hash algorithm.

PSA_ALG_IS_MAC (macro)

```
#define PSA_ALG_IS_MAC( alg) \
    (((alg) & PSA_ALG_CATEGORY_MASK) == PSA_ALG_CATEGORY_MAC)
```

Parameters:

alg An algorithm identifier (value of type *psa_algorithm_t*).

Returns:

1 if alg is a MAC algorithm, 0 otherwise. This macro may return either 0 or 1 if alg is not a supported algorithm identifier. **Description:** Whether the specified algorithm is a MAC algorithm.

PSA_ALG_IS_CIPHER (macro)

```
#define PSA_ALG_IS_CIPHER( alg) \
    (((alg) & PSA_ALG_CATEGORY_MASK) == PSA_ALG_CATEGORY_CIPHER)
```

Parameters:

alg An algorithm identifier (value of type *psa_algorithm_t*).

Returns:

1 if alg is a symmetric cipher algorithm, 0 otherwise. This macro may return either 0 or 1 if alg is not a supported algorithm identifier. **Description:** Whether the specified algorithm is a symmetric cipher algorithm.

PSA_ALG_IS_AEAD (macro)

```
#define PSA_ALG_IS_AEAD( alg) \
    (((alg) & PSA_ALG_CATEGORY_MASK) == PSA_ALG_CATEGORY_AEAD)
```

Parameters:

alg An algorithm identifier (value of type *psa_algorithm_t*).

Returns:

1 if alg is an AEAD algorithm, 0 otherwise. This macro may return either 0 or 1 if alg is not a supported algorithm identifier. **Description:** Whether the specified algorithm is an authenticated encryption with associated data (AEAD) algorithm.

PSA_ALG_IS_SIGN (macro)

```
#define PSA_ALG_IS_SIGN( alg) \
    (((alg) & PSA_ALG_CATEGORY_MASK) == PSA_ALG_CATEGORY_SIGN)
```

Parameters:

alg An algorithm identifier (value of type *psa_algorithm_t*).

Returns:

1 if alg is a public-key signature algorithm, 0 otherwise. This macro may return either 0 or 1 if alg is not a supported algorithm identifier. **Description:** Whether the specified algorithm is a public-key signature algorithm.

PSA_ALG_IS_ASYMMETRIC_ENCRYPTION (macro)

```
#define PSA_ALG_IS_ASYMMETRIC_ENCRYPTION( alg) \
    (((alg) & PSA_ALG_CATEGORY_MASK) == PSA_ALG_CATEGORY_ASYMMETRIC_ENCRYPTION)
```

Parameters:

alg An algorithm identifier (value of type *psa_algorithm_t*).

Returns:

1 if alg is a public-key encryption algorithm, 0 otherwise. This macro may return either 0 or 1 if alg is not a supported algorithm identifier. **Description:** Whether the specified algorithm is a public-key encryption algorithm.

PSA_ALG_IS_KEY_AGREEMENT (macro)

```
#define PSA_ALG_IS_KEY_AGREEMENT( alg) \
    (((alg) & PSA_ALG_CATEGORY_MASK) == PSA_ALG_CATEGORY_KEY_AGREEMENT)
```

Parameters:

alg An algorithm identifier (value of type *psa_algorithm_t*).

Returns:

1 if alg is a key agreement algorithm, 0 otherwise. This macro may return either 0 or 1 if alg is not a supported algorithm identifier. **Description:** Whether the specified algorithm is a key agreement algorithm.

PSA_ALG_IS_KEY_DERIVATION (macro)

```
#define PSA_ALG_IS_KEY_DERIVATION( alg) \
    (((alg) & PSA_ALG_CATEGORY_MASK) == PSA_ALG_CATEGORY_KEY_DERIVATION)
```

Parameters:

alg An algorithm identifier (value of type *psa_algorithm_t*).

Returns:

1 if alg is a key derivation algorithm, 0 otherwise. This macro may return either 0 or 1 if alg is not a supported algorithm identifier. **Description:** Whether the specified algorithm is a key derivation algorithm.

PSA_ALG_HASH_MASK (macro)

```
#define PSA_ALG_HASH_MASK ((psa_algorithm_t)0x000000ff)
```

PSA_ALG_MD2 (macro)

```
#define PSA_ALG_MD2 ((psa_algorithm_t)0x01000001)
```

PSA_ALG_MD4 (macro)

```
#define PSA_ALG_MD4 ((psa_algorithm_t)0x01000002)
```

PSA_ALG_MD5 (macro)

```
#define PSA_ALG_MD5 ((psa_algorithm_t)0x01000003)
```

PSA_ALG_RIPEMD160 (macro)

```
#define PSA_ALG_RIPEMD160 ((psa_algorithm_t)0x01000004)
```

PSA_ALG_SHA_1 (macro)

```
#define PSA_ALG_SHA_1 ((psa_algorithm_t)0x01000005)
```

PSA_ALG_SHA_224 (macro)

```
#define PSA_ALG_SHA_224 ((psa_algorithm_t)0x01000008)
```

SHA2-224

PSA_ALG_SHA_256 (macro)

```
#define PSA_ALG_SHA_256 ((psa_algorithm_t)0x01000009)
```

SHA2-256

PSA_ALG_SHA_384 (macro)

```
#define PSA_ALG_SHA_384  ((psa_algorithm_t)0x0100000a)
```

SHA2-384

PSA_ALG_SHA_512 (macro)

```
#define PSA_ALG_SHA_512  ((psa_algorithm_t)0x0100000b)
```

SHA2-512

PSA_ALG_SHA_512_224 (macro)

```
#define PSA_ALG_SHA_512_224  ((psa_algorithm_t)0x0100000c)
```

SHA2-512/224

PSA_ALG_SHA_512_256 (macro)

```
#define PSA_ALG_SHA_512_256  ((psa_algorithm_t)0x0100000d)
```

SHA2-512/256

PSA_ALG_SHA3_224 (macro)

```
#define PSA_ALG_SHA3_224  ((psa_algorithm_t)0x01000010)
```

SHA3-224

PSA_ALG_SHA3_256 (macro)

```
#define PSA_ALG_SHA3_256  ((psa_algorithm_t)0x01000011)
```

SHA3-256

PSA_ALG_SHA3_384 (macro)

```
#define PSA_ALG_SHA3_384  ((psa_algorithm_t)0x01000012)
```

SHA3-384

PSA_ALG_SHA3_512 (macro)

```
#define PSA_ALG_SHA3_512  ((psa_algorithm_t)0x01000013)
```

SHA3-512

PSA_ALG_ANY_HASH (macro)

```
#define PSA_ALG_ANY_HASH  ((psa_algorithm_t)0x010000ff)
```

Allow any hash algorithm.

This value may only be used to form the algorithm usage field of a policy for a signature algorithm that is parametrized by a hash. That is, suppose that `PSA_xxx_SIGNATURE` is one of the following macros:

- *PSA_ALG_RSA_PKCS1V15_SIGN*, *PSA_ALG_RSA_PSS*,
- *PSA_ALG_DSA*, *PSA_ALG_DETERMINISTIC_DSA*,
- *PSA_ALG_ECDSA*, *PSA_ALG_DETERMINISTIC_ECDSA*. Then you may create a key as follows:
- Set the key usage field using *PSA_ALG_ANY_HASH*, for example:

```
psa_key_policy_set_usage(&policy,
                         PSA_KEY_USAGE_SIGN, //or PSA_KEY_USAGE_VERIFY
                         PSA_xxx_SIGNATURE(PSA_ALG_ANY_HASH));
psa_set_key_policy(handle, &policy);
```

- Import or generate key material.
- Call *psa_asymmetric_sign()* or *psa_asymmetric_verify()*, passing an algorithm built from `PSA_xxx_SIGNATURE` and a specific hash. Each call to sign or verify a message may use a different hash.

```
psa_asymmetric_sign(handle, PSA_xxx_SIGNATURE(PSA_ALG_SHA_256), ...);
psa_asymmetric_sign(handle, PSA_xxx_SIGNATURE(PSA_ALG_SHA_512), ...);
psa_asymmetric_sign(handle, PSA_xxx_SIGNATURE(PSA_ALG_SHA3_256), ...);
```

This value may not be used to build other algorithms that are parametrized over a hash. For any valid use of this macro to build an algorithm `\p alg`, *PSA_ALG_IS_HASH_AND_SIGN*(`alg`) is true.

This value may not be used to build an algorithm specification to perform an operation. It is only valid to build policies.

PSA_ALG_MAC_SUBCATEGORY_MASK (macro)

```
#define PSA_ALG_MAC_SUBCATEGORY_MASK  ((psa_algorithm_t)0x00c00000)
```

PSA_ALG_HMAC_BASE (macro)

```
#define PSA_ALG_HMAC_BASE  ((psa_algorithm_t)0x02800000)
```

PSA_ALG_HMAC (macro)

```
#define PSA_ALG_HMAC( hash_alg) \
    (PSA_ALG_HMAC_BASE | ((hash_alg) & PSA_ALG_HASH_MASK))
```

Parameters:

hash_alg A hash algorithm (PSA_ALG_XXX value such that *PSA_ALG_IS_HASH*(hash_alg) is true).

Returns:

The corresponding HMAC algorithm.

Unspecified if alg is not a supported hash algorithm. **Description:** Macro to build an HMAC algorithm.

For example, *PSA_ALG_HMAC*(*PSA_ALG_SHA_256*) is HMAC-SHA-256.

PSA_ALG_HMAC_GET_HASH (macro)

```
#define PSA_ALG_HMAC_GET_HASH( hmac_alg) \
    (PSA_ALG_CATEGORY_HASH | ((hmac_alg) & PSA_ALG_HASH_MASK))
```

Parameters:

hmac_alg

Description:

PSA_ALG_IS_HMAC (macro)

```
#define PSA_ALG_IS_HMAC( alg) \
    (((alg) & (PSA_ALG_CATEGORY_MASK | PSA_ALG_MAC_SUBCATEGORY_MASK)) == PSA_ALG_HMAC_BASE)
```

Parameters:

alg An algorithm identifier (value of type *psa_algorithm_t*).

Returns:

1 if alg is an HMAC algorithm, 0 otherwise. This macro may return either 0 or 1 if alg is not a supported algorithm identifier. **Description:** Whether the specified algorithm is an HMAC algorithm.

HMAC is a family of MAC algorithms that are based on a hash function.

PSA_ALG_MAC_TRUNCATION_MASK (macro)

```
#define PSA_ALG_MAC_TRUNCATION_MASK  ((psa_algorithm_t)0x00003f00)
```

PSA_MAC_TRUNCATION_OFFSET (macro)

```
#define PSA_MAC_TRUNCATION_OFFSET  8
```

PSA_ALG_TRUNCATED_MAC (macro)

```
#define PSA_ALG_TRUNCATED_MAC( alg,  mac_length) \
    (((alg) & ~PSA_ALG_MAC_TRUNCATION_MASK) | ((mac_length) << PSA_MAC_TRUNCATION_OFFSET & PSA_ALG_MA
```

Parameters:

alg A MAC algorithm identifier (value of type *psa_algorithm_t* such that *PSA_ALG_IS_MAC*(alg) is true). This may be a truncated or untruncated MAC algorithm.

mac_length Desired length of the truncated MAC in bytes. This must be at most the full length of the MAC and must be at least an implementation-specified minimum. The implementation-specified minimum shall not be zero.

Returns:

The corresponding MAC algorithm with the specified length.

Unspecified if alg is not a supported MAC algorithm or if mac_length is too small or too large for the specified MAC algorithm. **Description:** Macro to build a truncated MAC algorithm.

A truncated MAC algorithm is identical to the corresponding MAC algorithm except that the MAC value for the truncated algorithm consists of only the first mac_length bytes of the MAC value for the untruncated algorithm.

Note: This macro may allow constructing algorithm identifiers that are not valid, either because the specified length is larger than the untruncated MAC or because the specified length is smaller than permitted by the implementation.

Note: It is implementation-defined whether a truncated MAC that is truncated to the same length as the MAC of the untruncated algorithm is considered identical to the untruncated algorithm for policy comparison purposes.

PSA_ALG_FULL_LENGTH_MAC (macro)

```
#define PSA_ALG_FULL_LENGTH_MAC( alg)   ((alg) & ~PSA_ALG_MAC_TRUNCATION_MASK)
```

Parameters:

alg A MAC algorithm identifier (value of type *psa_algorithm_t* such that *PSA_ALG_IS_MAC*(alg) is true). This may be a truncated or untruncated MAC algorithm.

Returns:

The corresponding base MAC algorithm.

Unspecified if alg is not a supported MAC algorithm. **Description:** Macro to build the base MAC algorithm corresponding to a truncated MAC algorithm.

PSA_MAC_TRUNCATED_LENGTH (macro)

```
#define PSA_MAC_TRUNCATED_LENGTH( alg) \
    (((alg) & PSA_ALG_MAC_TRUNCATION_MASK) >> PSA_MAC_TRUNCATION_OFFSET)
```

Parameters:

alg A MAC algorithm identifier (value of type *psa_algorithm_t* such that *PSA_ALG_IS_MAC*(alg) is true).

Returns:

Length of the truncated MAC in bytes.

0 if alg is a non-truncated MAC algorithm.

Unspecified if alg is not a supported MAC algorithm. **Description:** Length to which a MAC algorithm is truncated.

PSA_ALG_CIPHER_MAC_BASE (macro)

```
#define PSA_ALG_CIPHER_MAC_BASE  ((psa_algorithm_t)0x02c00000)
```

PSA_ALG_CBC_MAC (macro)

```
#define PSA_ALG_CBC_MAC  ((psa_algorithm_t)0x02c00001)
```

PSA_ALG_CMAC (macro)

```
#define PSA_ALG_CMAC  ((psa_algorithm_t)0x02c00002)
```

PSA_ALG_GMAC (macro)

```
#define PSA_ALG_GMAC  ((psa_algorithm_t)0x02c00003)
```

PSA_ALG_IS_BLOCK_CIPHER_MAC (macro)

```
#define PSA_ALG_IS_BLOCK_CIPHER_MAC( alg) \
    (((alg) & (PSA_ALG_CATEGORY_MASK | PSA_ALG_MAC_SUBCATEGORY_MASK)) == PSA_ALG_CIPHER_MAC_BASE)
```

Parameters:

alg An algorithm identifier (value of type *psa_algorithm_t*).

Returns:

1 if alg is a MAC algorithm based on a block cipher, 0 otherwise. This macro may return either 0 or 1 if alg is not a supported algorithm identifier. **Description:** Whether the specified algorithm is a MAC algorithm based on a block cipher.

PSA_ALG_CIPHER_STREAM_FLAG (macro)

```
#define PSA_ALG_CIPHER_STREAM_FLAG  ((psa_algorithm_t)0x00800000)
```

PSA_ALG_CIPHER_FROM_BLOCK_FLAG (macro)

```
#define PSA_ALG_CIPHER_FROM_BLOCK_FLAG  ((psa_algorithm_t)0x00400000)
```

PSA_ALG_IS_STREAM_CIPHER (macro)

```
#define PSA_ALG_IS_STREAM_CIPHER( alg) \
    (((alg) & (PSA_ALG_CATEGORY_MASK | PSA_ALG_CIPHER_STREAM_FLAG)) == (PSA_ALG_CATEGORY_CIPHER | PSA
```

Parameters:

alg An algorithm identifier (value of type *psa_algorithm_t*).

Returns:

1 if `alg` is a stream cipher algorithm, 0 otherwise. This macro may return either 0 or 1 if `alg` is not a supported algorithm identifier or if it is not a symmetric cipher algorithm. **Description:** Whether the specified algorithm is a stream cipher.

A stream cipher is a symmetric cipher that encrypts or decrypts messages by applying a bitwise-xor with a stream of bytes that is generated from a key.

PSA_ALG_ARC4 (macro)

```
#define PSA_ALG_ARC4  ((psa_algorithm_t)0x04800001)
```

The ARC4 stream cipher algorithm.

PSA_ALG_CTR (macro)

```
#define PSA_ALG_CTR  ((psa_algorithm_t)0x04c00001)
```

The CTR stream cipher mode.

CTR is a stream cipher which is built from a block cipher. The underlying block cipher is determined by the key type. For example, to use AES-128-CTR, use this algorithm with a key of type *PSA_KEY_TYPE_AES* and a length of 128 bits (16 bytes).

PSA_ALG_CFB (macro)

```
#define PSA_ALG_CFB  ((psa_algorithm_t)0x04c00002)
```

PSA_ALG_OFB (macro)

```
#define PSA_ALG_OFB  ((psa_algorithm_t)0x04c00003)
```

PSA_ALG_XTS (macro)

```
#define PSA_ALG_XTS   ((psa_algorithm_t)0x044000ff)
```

The XTS cipher mode.

XTS is a cipher mode which is built from a block cipher. It requires at least one full block of input, but beyond this minimum the input does not need to be a whole number of blocks.

PSA_ALG_CBC_NO_PADDING (macro)

```
#define PSA_ALG_CBC_NO_PADDING   ((psa_algorithm_t)0x04600100)
```

The CBC block cipher chaining mode, with no padding.

The underlying block cipher is determined by the key type.

This symmetric cipher mode can only be used with messages whose lengths are whole number of blocks for the chosen block cipher.

PSA_ALG_CBC_PKCS7 (macro)

```
#define PSA_ALG_CBC_PKCS7   ((psa_algorithm_t)0x04600101)
```

The CBC block cipher chaining mode with PKCS#7 padding.

The underlying block cipher is determined by the key type.

This is the padding method defined by PKCS#7 (RFC 2315) §10.3.

PSA_ALG_CCM (macro)

```
#define PSA_ALG_CCM   ((psa_algorithm_t)0x06001001)
```

The CCM authenticated encryption algorithm.

PSA_ALG_GCM (macro)

```
#define PSA_ALG_GCM   ((psa_algorithm_t)0x06001002)
```

The GCM authenticated encryption algorithm.

PSA_ALG_AEAD_TAG_LENGTH_MASK (macro)

```
#define PSA_ALG_AEAD_TAG_LENGTH_MASK   ((psa_algorithm_t)0x00003f00)
```

PSA_AEAD_TAG_LENGTH_OFFSET (macro)

```
#define PSA_AEAD_TAG_LENGTH_OFFSET  8
```

PSA_ALG_AEAD_WITH_TAG_LENGTH (macro)

```
#define PSA_ALG_AEAD_WITH_TAG_LENGTH( alg,  tag_length) \
    (((alg) & ~PSA_ALG_AEAD_TAG_LENGTH_MASK) | ((tag_length) << PSA_AEAD_TAG_LENGTH_OFFSET & PSA_ALG_
```

Parameters:

alg A AEAD algorithm identifier (value of type *psa_algorithm_t* such that *PSA_ALG_IS_AEAD*(alg) is true).

tag_length Desired length of the authentication tag in bytes.

Returns:

The corresponding AEAD algorithm with the specified length.

Unspecified if alg is not a supported AEAD algorithm or if tag_length is not valid for the specified AEAD algorithm. **Description:** Macro to build a shortened AEAD algorithm.

A shortened AEAD algorithm is similar to the corresponding AEAD algorithm, but has an authentication tag that consists of fewer bytes. Depending on the algorithm, the tag length may affect the calculation of the ciphertext.

PSA_ALG_AEAD_WITH_DEFAULT_TAG_LENGTH (macro)

```
#define PSA_ALG_AEAD_WITH_DEFAULT_TAG_LENGTH( alg) \
    ( PSA__ALG_AEAD_WITH_DEFAULT_TAG_LENGTH__CASE(alg, PSA_ALG_CCM) PSA__ALG_AEAD_WITH_DEFAULT_TAG_LE
```

Parameters:

alg An AEAD algorithm (PSA_ALG_XXX value such that *PSA_ALG_IS_AEAD*(alg) is true).

Returns:

The corresponding AEAD algorithm with the default tag length for that algorithm. **Description:** Calculate the corresponding AEAD algorithm with the default tag length.

PSA__ALG_AEAD_WITH_DEFAULT_TAG_LENGTH__CASE (macro)

```
#define PSA__ALG_AEAD_WITH_DEFAULT_TAG_LENGTH__CASE( alg,  ref) \
    PSA_ALG_AEAD_WITH_TAG_LENGTH(alg, 0) == PSA_ALG_AEAD_WITH_TAG_LENGTH(ref, 0) ? ref :
```

Parameters:

alg

ref

Description:

PSA_ALG_RSA_PKCS1V15_SIGN_BASE (macro)

```
#define PSA_ALG_RSA_PKCS1V15_SIGN_BASE  ((psa_algorithm_t)0x10020000)
```

PSA_ALG_RSA_PKCS1V15_SIGN (macro)

```
#define PSA_ALG_RSA_PKCS1V15_SIGN( hash_alg) \
    (PSA_ALG_RSA_PKCS1V15_SIGN_BASE | ((hash_alg) & PSA_ALG_HASH_MASK))
```

Parameters:

hash_alg A hash algorithm (PSA_ALG_XXX value such that *PSA_ALG_IS_HASH*(hash_alg) is true). This includes *PSA_ALG_ANY_HASH* when specifying the algorithm in a usage policy.

Returns:

The corresponding RSA PKCS#1 v1.5 signature algorithm.

Unspecified if alg is not a supported hash algorithm. **Description:** RSA PKCS#1 v1.5 signature with hashing.

This is the signature scheme defined by RFC 8017 (PKCS#1: RSA Cryptography Specifications) under the name RSASSA-PKCS1-v1_5.

PSA_ALG_RSA_PKCS1V15_SIGN_RAW (macro)

```
#define PSA_ALG_RSA_PKCS1V15_SIGN_RAW  PSA_ALG_RSA_PKCS1V15_SIGN_BASE
```

Raw PKCS#1 v1.5 signature.

The input to this algorithm is the DigestInfo structure used by RFC 8017 (PKCS#1: RSA Cryptography Specifications), §9.2 steps 3–6.

PSA_ALG_IS_RSA_PKCS1V15_SIGN (macro)

```
#define PSA_ALG_IS_RSA_PKCS1V15_SIGN( alg) \
    (((alg) & ~PSA_ALG_HASH_MASK) == PSA_ALG_RSA_PKCS1V15_SIGN_BASE)
```

Parameters:

alg

Description:

PSA_ALG_RSA_PSS_BASE (macro)

```
#define PSA_ALG_RSA_PSS_BASE  ((psa_algorithm_t)0x10030000)
```

PSA_ALG_RSA_PSS (macro)

```
#define PSA_ALG_RSA_PSS( hash_alg) \
    (PSA_ALG_RSA_PSS_BASE | ((hash_alg) & PSA_ALG_HASH_MASK))
```

Parameters:

hash_alg A hash algorithm (PSA_ALG_XXX value such that *PSA_ALG_IS_HASH*(hash_alg) is true). This includes *PSA_ALG_ANY_HASH* when specifying the algorithm in a usage policy.

Returns:

The corresponding RSA PSS signature algorithm.

Unspecified if alg is not a supported hash algorithm. **Description:** RSA PSS signature with hashing.

This is the signature scheme defined by RFC 8017 (PKCS#1: RSA Cryptography Specifications) under the name RSASSA-PSS, with the message generation function MGF1, and with a salt length equal to the length of the hash. The specified hash algorithm is used to hash the input message, to create the salted hash, and for the mask generation.

PSA_ALG_IS_RSA_PSS (macro)

```
#define PSA_ALG_IS_RSA_PSS( alg) \
    (((alg) & ~PSA_ALG_HASH_MASK) == PSA_ALG_RSA_PSS_BASE)
```

Parameters:

alg

Description:

PSA_ALG_DSA_BASE (macro)

```
#define PSA_ALG_DSA_BASE  ((psa_algorithm_t)0x10040000)
```

PSA_ALG_DSA (macro)

```
#define PSA_ALG_DSA( hash_alg) \
    (PSA_ALG_DSA_BASE | ((hash_alg) & PSA_ALG_HASH_MASK))
```

Parameters:

hash_alg A hash algorithm (PSA_ALG_XXX value such that *PSA_ALG_IS_HASH*(hash_alg) is true). This includes *PSA_ALG_ANY_HASH* when specifying the algorithm in a usage policy.

Returns:

The corresponding DSA signature algorithm.

Unspecified if alg is not a supported hash algorithm. **Description:** DSA signature with hashing.

This is the signature scheme defined by FIPS 186-4, with a random per-message secret number (*k*).

PSA_ALG_DETERMINISTIC_DSA_BASE (macro)

```
#define PSA_ALG_DETERMINISTIC_DSA_BASE  ((psa_algorithm_t)0x10050000)
```

PSA_ALG_DSA_DETERMINISTIC_FLAG (macro)

```
#define PSA_ALG_DSA_DETERMINISTIC_FLAG  ((psa_algorithm_t)0x00010000)
```

PSA_ALG_DETERMINISTIC_DSA (macro)

```
#define PSA_ALG_DETERMINISTIC_DSA( hash_alg) \
    (PSA_ALG_DETERMINISTIC_DSA_BASE | ((hash_alg) & PSA_ALG_HASH_MASK))
```

Parameters:

hash_alg A hash algorithm (`PSA_ALG_XXX` value such that *PSA_ALG_IS_HASH*(`hash_alg`) is true). This includes *PSA_ALG_ANY_HASH* when specifying the algorithm in a usage policy.

Returns:

The corresponding DSA signature algorithm.

Unspecified if `alg` is not a supported hash algorithm. **Description:** Deterministic DSA signature with hashing.

This is the deterministic variant defined by RFC 6979 of the signature scheme defined by FIPS 186-4.

PSA_ALG_IS_DSA (macro)

```
#define PSA_ALG_IS_DSA( alg) \
    (((alg) & ~PSA_ALG_HASH_MASK & ~PSA_ALG_DSA_DETERMINISTIC_FLAG) == PSA_ALG_DSA_BASE)
```

Parameters:

alg

Description:

PSA_ALG_DSA_IS_DETERMINISTIC (macro)

```
#define PSA_ALG_DSA_IS_DETERMINISTIC( alg) \
    (((alg) & PSA_ALG_DSA_DETERMINISTIC_FLAG) != 0)
```

Parameters:

alg

Description:

PSA_ALG_IS_DETERMINISTIC_DSA (macro)

```
#define PSA_ALG_IS_DETERMINISTIC_DSA( alg) \
    (PSA_ALG_IS_DSA(alg) && PSA_ALG_DSA_IS_DETERMINISTIC(alg))
```

Parameters:

alg

Description:

PSA_ALG_IS_RANDOMIZED_DSA (macro)

```
#define PSA_ALG_IS_RANDOMIZED_DSA( alg) \
    (PSA_ALG_IS_DSA(alg) && !PSA_ALG_DSA_IS_DETERMINISTIC(alg))
```

Parameters:

alg

Description:

PSA_ALG_ECDSA_BASE (macro)

```
#define PSA_ALG_ECDSA_BASE  ((psa_algorithm_t)0x10060000)
```

PSA_ALG_ECDSA (macro)

```
#define PSA_ALG_ECDSA( hash_alg) \
    (PSA_ALG_ECDSA_BASE | ((hash_alg) & PSA_ALG_HASH_MASK))
```

Parameters:

hash_alg A hash algorithm (`PSA_ALG_XXX` value such that *PSA_ALG_IS_HASH*(`hash_alg`) is true). This includes *PSA_ALG_ANY_HASH* when specifying the algorithm in a usage policy.

Returns:

The corresponding ECDSA signature algorithm.

Unspecified if `alg` is not a supported hash algorithm. **Description:** ECDSA signature with hashing.

This is the ECDSA signature scheme defined by ANSI X9.62, with a random per-message secret number (k).

The representation of the signature as a byte string consists of the concatentation of the signature values r and s. Each of r and s is encoded as an N-octet string, where N is the length of the base point of the curve in octets. Each value is represented in big-endian order (most significant octet first).

PSA_ALG_ECDSA_ANY (macro)

```
#define PSA_ALG_ECDSA_ANY  PSA_ALG_ECDSA_BASE
```

ECDSA signature without hashing.

This is the same signature scheme as *PSA_ALG_ECDSA()*, but without specifying a hash algorithm. This algorithm may only be used to sign or verify a sequence of bytes that should be an already-calculated hash. Note that the input is padded with zeros on the left or truncated on the left as required to fit the curve size.

PSA_ALG_DETERMINISTIC_ECDSA_BASE (macro)

```
#define PSA_ALG_DETERMINISTIC_ECDSA_BASE  ((psa_algorithm_t)0x10070000)
```

PSA_ALG_DETERMINISTIC_ECDSA (macro)

```
#define PSA_ALG_DETERMINISTIC_ECDSA( hash_alg) \
    (PSA_ALG_DETERMINISTIC_ECDSA_BASE | ((hash_alg) & PSA_ALG_HASH_MASK))
```

Parameters:

hash_alg A hash algorithm (`PSA_ALG_XXX` value such that *PSA_ALG_IS_HASH*(`hash_alg`) is true). This includes *PSA_ALG_ANY_HASH* when specifying the algorithm in a usage policy.

Returns:

The corresponding deterministic ECDSA signature algorithm.

Unspecified if `alg` is not a supported hash algorithm. **Description:** Deterministic ECDSA signature with hashing.

This is the deterministic ECDSA signature scheme defined by RFC 6979.

The representation of a signature is the same as with *PSA_ALG_ECDSA()*.

Note that when this algorithm is used for verification, signatures made with randomized ECDSA (*PSA_ALG_ECDSA*(`hash_alg`)) with the same private key are accepted. In other words, *PSA_ALG_DETERMINISTIC_ECDSA*(`hash_alg`) differs from *PSA_ALG_ECDSA*(`hash_alg`) only for signature, not for verification.

PSA_ALG_IS_ECDSA (macro)

```
#define PSA_ALG_IS_ECDSA( alg) \
    (((alg) & ~PSA_ALG_HASH_MASK & ~PSA_ALG_DSA_DETERMINISTIC_FLAG) == PSA_ALG_ECDSA_BASE)
```

Parameters:

alg

Description:

PSA_ALG_ECDSA_IS_DETERMINISTIC (macro)

```
#define PSA_ALG_ECDSA_IS_DETERMINISTIC( alg) \
    (((alg) & PSA_ALG_DSA_DETERMINISTIC_FLAG) != 0)
```

Parameters:

alg

Description:

PSA_ALG_IS_DETERMINISTIC_ECDSA (macro)

```
#define PSA_ALG_IS_DETERMINISTIC_ECDSA( alg) \
    (PSA_ALG_IS_ECDSA(alg) && PSA_ALG_ECDSA_IS_DETERMINISTIC(alg))
```

Parameters:

alg

Description:

PSA_ALG_IS_RANDOMIZED_ECDSA (macro)

```
#define PSA_ALG_IS_RANDOMIZED_ECDSA( alg) \
    (PSA_ALG_IS_ECDSA(alg) && !PSA_ALG_ECDSA_IS_DETERMINISTIC(alg))
```

Parameters:

alg

Description:

PSA_ALG_IS_HASH_AND_SIGN (macro)

```
#define PSA_ALG_IS_HASH_AND_SIGN( alg) \
    (PSA_ALG_IS_RSA_PSS(alg) || PSA_ALG_IS_RSA_PKCS1V15_SIGN(alg) || PSA_ALG_IS_DSA(alg) || PSA_ALG_I
```

Parameters:

alg An algorithm identifier (value of type *psa_algorithm_t*).

Returns:

1 if alg is a hash-and-sign algorithm, 0 otherwise. This macro may return either 0 or 1 if alg is not a supported algorithm identifier. **Description:** Whether the specified algorithm is a hash-and-sign algorithm.

Hash-and-sign algorithms are public-key signature algorithms structured in two parts: first the calculation of a hash in a way that does not depend on the key, then the calculation of a signature from the hash value and the key.

PSA_ALG_SIGN_GET_HASH (macro)

```
#define PSA_ALG_SIGN_GET_HASH( alg) \
    (PSA_ALG_IS_HASH_AND_SIGN(alg) ? ((alg) & PSA_ALG_HASH_MASK) == 0 ? /*"raw" algorithm*/ 0 : ((alg
```

Parameters:

alg A signature algorithm (PSA_ALG_XXX value such that *PSA_ALG_IS_SIGN*(alg) is true).

Returns:

The underlying hash algorithm if `alg` is a hash-and-sign algorithm.

0 if `alg` is a signature algorithm that does not follow the hash-and-sign structure.

Unspecified if `alg` is not a signature algorithm or if it is not supported by the implementation. **Description:** Get the hash used by a hash-and-sign signature algorithm.

A hash-and-sign algorithm is a signature algorithm which is composed of two phases: first a hashing phase which does not use the key and produces a hash of the input message, then a signing phase which only uses the hash and the key and not the message itself.

PSA_ALG_RSA_PKCS1V15_CRYPT (macro)

```
#define PSA_ALG_RSA_PKCS1V15_CRYPT  ((psa_algorithm_t)0x12020000)
```

RSA PKCS#1 v1.5 encryption.

PSA_ALG_RSA_OAEP_BASE (macro)

```
#define PSA_ALG_RSA_OAEP_BASE  ((psa_algorithm_t)0x12030000)
```

PSA_ALG_RSA_OAEP (macro)

```
#define PSA_ALG_RSA_OAEP( hash_alg) \
    (PSA_ALG_RSA_OAEP_BASE | ((hash_alg) & PSA_ALG_HASH_MASK))
```

Parameters:

hash_alg The hash algorithm (`PSA_ALG_XXX` value such that *PSA_ALG_IS_HASH*(`hash_alg`) is true) to use for MGF1.

Returns:

The corresponding RSA OAEP signature algorithm.

Unspecified if `alg` is not a supported hash algorithm. **Description:** RSA OAEP encryption.

This is the encryption scheme defined by RFC 8017 (PKCS#1: RSA Cryptography Specifications) under the name RSAES-OAEP, with the message generation function MGF1.

PSA_ALG_IS_RSA_OAEP (macro)

```
#define PSA_ALG_IS_RSA_OAEP( alg) \
    (((alg) & ~PSA_ALG_HASH_MASK) == PSA_ALG_RSA_OAEP_BASE)
```

Parameters:

alg

Description:

PSA_ALG_RSA_OAEP_GET_HASH (macro)

```
#define PSA_ALG_RSA_OAEP_GET_HASH( alg) \
    (PSA_ALG_IS_RSA_OAEP(alg) ? ((alg) & PSA_ALG_HASH_MASK) | PSA_ALG_CATEGORY_HASH : 0)
```

Parameters:

alg

Description:

PSA_ALG_HKDF_BASE (macro)

```
#define PSA_ALG_HKDF_BASE  ((psa_algorithm_t)0x20000100)
```

PSA_ALG_HKDF (macro)

```
#define PSA_ALG_HKDF( hash_alg) \
    (PSA_ALG_HKDF_BASE | ((hash_alg) & PSA_ALG_HASH_MASK))
```

Parameters:

hash_alg A hash algorithm (PSA_ALG_XXX value such that *PSA_ALG_IS_HASH*(hash_alg) is true).

Returns:

The corresponding HKDF algorithm.

Unspecified if alg is not a supported hash algorithm. **Description:** Macro to build an HKDF algorithm.

For example, PSA_ALG_HKDF(PSA_ALG_SHA256) is HKDF using HMAC-SHA-256.

This key derivation algorithm uses the following inputs:

- *PSA_KDF_STEP_SALT* is the salt used in the "extract" step. It is optional; if omitted, the derivation uses an empty salt.
- *PSA_KDF_STEP_SECRET* is the secret key used in the "extract" step.
- *PSA_KDF_STEP_INFO* is the info string used in the "expand" step. You must pass *PSA_KDF_STEP_SALT* before *PSA_KDF_STEP_SECRET*. You may pass *PSA_KDF_STEP_INFO* at any time after steup and before starting to generate output.

PSA_ALG_IS_HKDF (macro)

```
#define PSA_ALG_IS_HKDF( alg) \
    (((alg) & ~PSA_ALG_HASH_MASK) == PSA_ALG_HKDF_BASE)
```

Parameters:

alg An algorithm identifier (value of type *psa_algorithm_t*).

Returns:

1 if alg is an HKDF algorithm, 0 otherwise. This macro may return either 0 or 1 if alg is not a supported key derivation algorithm identifier. **Description:** Whether the specified algorithm is an HKDF algorithm.

HKDF is a family of key derivation algorithms that are based on a hash function and the HMAC construction.

PSA_ALG_HKDF_GET_HASH (macro)

```
#define PSA_ALG_HKDF_GET_HASH( hkdf_alg) \
    (PSA_ALG_CATEGORY_HASH | ((hkdf_alg) & PSA_ALG_HASH_MASK))
```

Parameters:

hkdf_alg

Description:

PSA_ALG_TLS12_PRF_BASE (macro)

```
#define PSA_ALG_TLS12_PRF_BASE  ((psa_algorithm_t)0x20000200)
```

PSA_ALG_TLS12_PRF (macro)

```
#define PSA_ALG_TLS12_PRF( hash_alg) \
    (PSA_ALG_TLS12_PRF_BASE | ((hash_alg) & PSA_ALG_HASH_MASK))
```

Parameters:

hash_alg A hash algorithm (`PSA_ALG_XXX` value such that *PSA_ALG_IS_HASH*(`hash_alg`) is true).

Returns:

The corresponding TLS-1.2 PRF algorithm.

Unspecified if `alg` is not a supported hash algorithm. **Description:** Macro to build a TLS-1.2 PRF algorithm.

TLS 1.2 uses a custom pseudorandom function (PRF) for key schedule, specified in Section 5 of RFC 5246. It is based on HMAC and can be used with either SHA-256 or SHA-384.

For the application to TLS-1.2, the salt and label arguments passed to psa_key_derivation() are what's called 'seed' and 'label' in RFC 5246, respectively. For example, for TLS key expansion, the salt is the concatenation of ServerHello.Random + ClientHello.Random, while the label is "key expansion".

For example, `PSA_ALG_TLS12_PRF(PSA_ALG_SHA256)` represents the TLS 1.2 PRF using HMAC-SHA-256.

PSA_ALG_IS_TLS12_PRF (macro)

```
#define PSA_ALG_IS_TLS12_PRF( alg) \
    (((alg) & ~PSA_ALG_HASH_MASK) == PSA_ALG_TLS12_PRF_BASE)
```

Parameters:

alg An algorithm identifier (value of type *psa_algorithm_t*).

Returns:

1 if `alg` is a TLS-1.2 PRF algorithm, 0 otherwise. This macro may return either 0 or 1 if `alg` is not a supported key derivation algorithm identifier. **Description:** Whether the specified algorithm is a TLS-1.2 PRF algorithm.

PSA_ALG_TLS12_PRF_GET_HASH (macro)

```
#define PSA_ALG_TLS12_PRF_GET_HASH( hkdf_alg) \
    (PSA_ALG_CATEGORY_HASH | ((hkdf_alg) & PSA_ALG_HASH_MASK))
```

Parameters:

hkdf_alg

Description:

PSA_ALG_TLS12_PSK_TO_MS_BASE (macro)

```
#define PSA_ALG_TLS12_PSK_TO_MS_BASE  ((psa_algorithm_t)0x20000300)
```

PSA_ALG_TLS12_PSK_TO_MS (macro)

```
#define PSA_ALG_TLS12_PSK_TO_MS( hash_alg) \
    (PSA_ALG_TLS12_PSK_TO_MS_BASE | ((hash_alg) & PSA_ALG_HASH_MASK))
```

Parameters:

hash_alg A hash algorithm (`PSA_ALG_XXX` value such that *PSA_ALG_IS_HASH*(`hash_alg`) is true).

Returns:

The corresponding TLS-1.2 PSK to MS algorithm.

Unspecified if `alg` is not a supported hash algorithm. **Description:** Macro to build a TLS-1.2 PSK-to-MasterSecret algorithm.

In a pure-PSK handshake in TLS 1.2, the master secret is derived from the PreSharedKey (PSK) through the application of padding (RFC 4279, Section 2) and the TLS-1.2 PRF (RFC 5246, Section 5). The latter is based on HMAC and can be used with either SHA-256 or SHA-384.

For the application to TLS-1.2, the salt passed to psa_key_derivation() (and forwarded to the TLS-1.2 PRF) is the concatenation of the ClientHello.Random + ServerHello.Random, while the label is "master secret" or "extended master secret".

For example, `PSA_ALG_TLS12_PSK_TO_MS(PSA_ALG_SHA256)` represents the TLS-1.2 PSK to MasterSecret derivation PRF using HMAC-SHA-256.

PSA_ALG_IS_TLS12_PSK_TO_MS (macro)

```
#define PSA_ALG_IS_TLS12_PSK_TO_MS( alg) \
    (((alg) & ~PSA_ALG_HASH_MASK) == PSA_ALG_TLS12_PSK_TO_MS_BASE)
```

Parameters:

alg An algorithm identifier (value of type *psa_algorithm_t*).

Returns:

1 if `alg` is a TLS-1.2 PSK to MS algorithm, 0 otherwise. This macro may return either 0 or 1 if `alg` is not a supported key derivation algorithm identifier. **Description:** Whether the specified algorithm is a TLS-1.2 PSK to MS algorithm.

PSA_ALG_TLS12_PSK_TO_MS_GET_HASH (macro)

```
#define PSA_ALG_TLS12_PSK_TO_MS_GET_HASH( hkdf_alg) \
    (PSA_ALG_CATEGORY_HASH | ((hkdf_alg) & PSA_ALG_HASH_MASK))
```

Parameters:

hkdf_alg

Description:

PSA_ALG_KEY_DERIVATION_MASK (macro)

```
#define PSA_ALG_KEY_DERIVATION_MASK  ((psa_algorithm_t)0x080fffff)
```

PSA_ALG_KEY_AGREEMENT_MASK (macro)

```
#define PSA_ALG_KEY_AGREEMENT_MASK  ((psa_algorithm_t)0x10f00000)
```

PSA_ALG_KEY_AGREEMENT (macro)

```
#define PSA_ALG_KEY_AGREEMENT( ka_alg,  kdf_alg)  ((ka_alg) | (kdf_alg))
```

Parameters:

ka_alg A key agreement algorithm (PSA_ALG_XXX value such that *PSA_ALG_IS_KEY_AGREEMENT*(ka_alg) is true).

kdf_alg A key derivation algorithm (PSA_ALG_XXX value such that *PSA_ALG_IS_KEY_DERIVATION*(kdf_alg) is true).

Returns:

The corresponding key agreement and derivation algorithm.

Unspecified if ka_alg is not a supported key agreement algorithm or kdf_alg is not a supported key derivation algorithm. **Description:** Macro to build a combined algorithm that chains a key agreement with a key derivation.

PSA_ALG_KEY_AGREEMENT_GET_KDF (macro)

```
#define PSA_ALG_KEY_AGREEMENT_GET_KDF( alg) \
    (((alg) & PSA_ALG_KEY_DERIVATION_MASK) | PSA_ALG_CATEGORY_KEY_DERIVATION)
```

Parameters:

alg

Description:

PSA_ALG_KEY_AGREEMENT_GET_BASE (macro)

```
#define PSA_ALG_KEY_AGREEMENT_GET_BASE( alg) \
    (((alg) & PSA_ALG_KEY_AGREEMENT_MASK) | PSA_ALG_CATEGORY_KEY_AGREEMENT)
```

Parameters:

alg

Description:

PSA_ALG_IS_RAW_KEY_AGREEMENT (macro)

```
#define PSA_ALG_IS_RAW_KEY_AGREEMENT( alg) \
    (PSA_ALG_IS_KEY_AGREEMENT(alg) && PSA_ALG_KEY_AGREEMENT_GET_KDF(alg) == PSA_ALG_CATEGORY_KEY_DERI
```

Parameters:

alg An algorithm identifier (value of type *psa_algorithm_t*).

Returns:

1 if `alg` is a raw key agreement algorithm, 0 otherwise. This macro may return either 0 or 1 if `alg` is not a supported algorithm identifier. **Description:** Whether the specified algorithm is a raw key agreement algorithm.

A raw key agreement algorithm is one that does not specify a key derivation function. Usually, raw key agreement algorithms are constructed directly with a `PSA_ALG_xxx` macro while non-raw key agreement algorithms are constructed with *PSA_ALG_KEY_AGREEMENT()*.

PSA_ALG_IS_KEY_DERIVATION_OR_AGREEMENT (macro)

```
#define PSA_ALG_IS_KEY_DERIVATION_OR_AGREEMENT( alg) \
    ((PSA_ALG_IS_KEY_DERIVATION(alg) || PSA_ALG_IS_KEY_AGREEMENT(alg)))
```

Parameters:

alg

Description:

PSA_ALG_FFDH (macro)

```
#define PSA_ALG_FFDH  ((psa_algorithm_t)0x30100000)
```

The finite-field Diffie-Hellman (DH) key agreement algorithm.

The shared secret produced by key agreement and passed as input to the derivation or selection algorithm `kdf_alg` is the shared secret `g^{ab}` in big-endian format. It is `ceiling(m / 8)` bytes long where `m` is the size of the prime `p` in bits.

`PSA_ALG_IS_FFDH` (macro)

```
#define PSA_ALG_IS_FFDH( alg) \
    (PSA_ALG_KEY_AGREEMENT_GET_BASE(alg) == PSA_ALG_FFDH)
```

Parameters:

alg An algorithm identifier (value of type *psa_algorithm_t*).

Returns:

1 if `alg` is a finite field Diffie-Hellman algorithm, 0 otherwise. This macro may return either 0 or 1 if `alg` is not a supported key agreement algorithm identifier. **Description:** Whether the specified algorithm is a finite field Diffie-Hellman algorithm.

This includes every supported key selection or key agreement algorithm for the output of the Diffie-Hellman calculation.

`PSA_ALG_ECDH` (macro)

```
#define PSA_ALG_ECDH  ((psa_algorithm_t)0x30200000)
```

The elliptic curve Diffie-Hellman (ECDH) key agreement algorithm.

The shared secret produced by key agreement is the x-coordinate of the shared secret point. It is always `ceiling(m / 8)` bytes long where `m` is the bit size associated with the curve, i.e. the bit size of the order of the curve's coordinate field. When `m` is not a multiple of 8, the byte containing the most significant bit of the shared secret is padded with zero bits. The byte order is either little-endian or big-endian depending on the curve type.

- For Montgomery curves (curve types `PSA_ECC_CURVE_CURVEXXX`), the shared secret is the x-coordinate of `d_A Q_B = d_B Q_A` in little-endian byte order. The bit size is 448 for Curve448 and 255 for Curve25519.
- For Weierstrass curves over prime fields (curve types `PSA_ECC_CURVE_SECPXXX` and `PSA_ECC_CURVE_BRAINPOOL_PXXX`), the shared secret is the x-coordinate of `d_A Q_B = d_B Q_A` in big-endian byte order. The bit size is `m = ceiling(log_2(p))` for the field `F_p`.
- For Weierstrass curves over binary fields (curve types `PSA_ECC_CURVE_SECTXXX`), the shared secret is the x-coordinate of `d_A Q_B = d_B Q_A` in big-endian byte order. The bit size is `m` for the field `F_{2^m}`.

`PSA_ALG_IS_ECDH` (macro)

```
#define PSA_ALG_IS_ECDH( alg) \
    (PSA_ALG_KEY_AGREEMENT_GET_BASE(alg) == PSA_ALG_ECDH)
```

Parameters:

alg An algorithm identifier (value of type *psa_algorithm_t*).

Returns:

1 if `alg` is an elliptic curve Diffie-Hellman algorithm, 0 otherwise. This macro may return either 0 or 1 if `alg` is not a supported key agreement algorithm identifier. **Description:** Whether the specified algorithm is an elliptic curve Diffie-Hellman algorithm.

This includes every supported key selection or key agreement algorithm for the output of the Diffie-Hellman calculation.

PSA_ALG_IS_WILDCARD (macro)

```
#define PSA_ALG_IS_WILDCARD( alg) \
    (PSA_ALG_IS_HASH_AND_SIGN(alg) ? PSA_ALG_SIGN_GET_HASH(alg) == PSA_ALG_ANY_HASH : (alg) == PSA_AL
```

Parameters:

alg An algorithm identifier (value of type *psa_algorithm_t*).

Returns:

1 if `alg` is a wildcard algorithm encoding.

0 if `alg` is a non-wildcard algorithm encoding (suitable for an operation).

This macro may return either 0 or 1 if `alg` is not a supported algorithm identifier. **Description:** Whether the specified algorithm encoding is a wildcard.

Wildcard values may only be used to set the usage algorithm field in a policy, not to perform an operation.

CHAPTER

TWENTYTHREE

KEY LIFETIMES

psa_key_lifetime_t (type)

```
typedef uint32_t psa_key_lifetime_t;
```

Encoding of key lifetimes.

psa_key_id_t (type)

```
typedef uint32_t psa_key_id_t;
```

Encoding of identifiers of persistent keys.

PSA_KEY_LIFETIME_VOLATILE (macro)

```
#define PSA_KEY_LIFETIME_VOLATILE  ((psa_key_lifetime_t)0x00000000)
```

A volatile key only exists as long as the handle to it is not closed. The key material is guaranteed to be erased on a power reset.

PSA_KEY_LIFETIME_PERSISTENT (macro)

```
#define PSA_KEY_LIFETIME_PERSISTENT  ((psa_key_lifetime_t)0x00000001)
```

The default storage area for persistent keys.

A persistent key remains in storage until it is explicitly destroyed or until the corresponding storage area is wiped. This specification does not define any mechanism to wipe a storage area, but implementations may provide their own mechanism (for example to perform a factory reset, to prepare for device refurbishment, or to uninstall an application).

This lifetime value is the default storage area for the calling application. Implementations may offer other storage areas designated by other lifetime values as implementation-specific extensions.

CHAPTER

TWENTYFOUR

OTHER DEFINITIONS

PSA_BITS_TO_BYTES (macro)

```
#define PSA_BITS_TO_BYTES( bits)   (((bits) + 7) / 8)
```

Parameters:

bits

Description:

PSA_BYTES_TO_BITS (macro)

```
#define PSA_BYTES_TO_BITS( bytes)   ((bytes) * 8)
```

Parameters:

bytes

Description:

PSA_HASH_SIZE (macro)

```
#define PSA_HASH_SIZE( alg) \
    ( PSA_ALG_HMAC_GET_HASH(alg) == PSA_ALG_MD2 ? 16 : PSA_ALG_HMAC_GET_HASH(alg) == PSA_ALG_MD4 ? 16
```

Parameters:

alg A hash algorithm (PSA_ALG_XXX value such that *PSA_ALG_IS_HASH*(alg) is true), or an HMAC algorithm (*PSA_ALG_HMAC*(hash_alg) where hash_alg is a hash algorithm).

Returns:

The hash size for the specified hash algorithm. If the hash algorithm is not recognized, return 0. An implementation may return either 0 or the correct size for a hash algorithm that it recognizes, but does not support. **Description:** The size of the output of *psa_hash_finish()*, in bytes.

This is also the hash size that *psa_hash_verify()* expects.

PSA_HASH_MAX_SIZE (macro)

```
#define PSA_HASH_MAX_SIZE 64
```

Maximum size of a hash.

This macro must expand to a compile-time constant integer. This value should be the maximum size of a hash supported by the implementation, in bytes, and must be no smaller than this maximum.

PSA_HMAC_MAX_HASH_BLOCK_SIZE (macro)

```
#define PSA_HMAC_MAX_HASH_BLOCK_SIZE 128
```

PSA_MAC_MAX_SIZE (macro)

```
#define PSA_MAC_MAX_SIZE PSA_HASH_MAX_SIZE
```

Maximum size of a MAC.

This macro must expand to a compile-time constant integer. This value should be the maximum size of a MAC supported by the implementation, in bytes, and must be no smaller than this maximum.

PSA_AEAD_TAG_LENGTH (macro)

```
#define PSA_AEAD_TAG_LENGTH( alg) \
    (PSA_ALG_IS_AEAD(alg) ? (((alg) & PSA_ALG_AEAD_TAG_LENGTH_MASK) >> PSA_AEAD_TAG_LENGTH_OFFSET) :
```

Parameters:

alg An AEAD algorithm (PSA_ALG_XXX value such that *PSA_ALG_IS_AEAD*(alg) is true).

Returns:

The tag size for the specified algorithm. If the AEAD algorithm does not have an identified tag that can be distinguished from the rest of the ciphertext, return 0. If the AEAD algorithm is not recognized, return 0. An implementation may return either 0 or a correct size for an AEAD algorithm that it recognizes, but does not support. **Description:** The tag size for an AEAD algorithm, in bytes.

PSA_VENDOR_RSA_MAX_KEY_BITS (macro)

```
#define PSA_VENDOR_RSA_MAX_KEY_BITS 4096
```

PSA_VENDOR_ECC_MAX_CURVE_BITS (macro)

```
#define PSA_VENDOR_ECC_MAX_CURVE_BITS 521
```

PSA_ALG_TLS12_PSK_TO_MS_MAX_PSK_LEN (macro)

```
#define PSA_ALG_TLS12_PSK_TO_MS_MAX_PSK_LEN  128
```

This macro returns the maximum length of the PSK supported by the TLS-1.2 PSK-to-MS key derivation.

Quoting RFC 4279, Sect 5.3: TLS implementations supporting these ciphersuites MUST support arbitrary PSK identities up to 128 octets in length, and arbitrary PSKs up to 64 octets in length. Supporting longer identities and keys is RECOMMENDED.

Therefore, no implementation should define a value smaller than 64 for *PSA_ALG_TLS12_PSK_TO_MS_MAX_PSK_LEN*.

PSA_ASYMMETRIC_SIGNATURE_MAX_SIZE (macro)

```
#define PSA_ASYMMETRIC_SIGNATURE_MAX_SIZE \
    PSA_BITS_TO_BYTES( PSA_VENDOR_RSA_MAX_KEY_BITS > PSA_VENDOR_ECC_MAX_CURVE_BITS ? PSA_VENDOR_RSA_M
```

Maximum size of an asymmetric signature.

This macro must expand to a compile-time constant integer. This value should be the maximum size of a MAC supported by the implementation, in bytes, and must be no smaller than this maximum.

PSA_MAX_BLOCK_CIPHER_BLOCK_SIZE (macro)

```
#define PSA_MAX_BLOCK_CIPHER_BLOCK_SIZE  16
```

The maximum size of a block cipher supported by the implementation.

PSA_MAC_FINAL_SIZE (macro)

```
#define PSA_MAC_FINAL_SIZE( key_type,  key_bits,  alg) \
    ((alg) & PSA_ALG_MAC_TRUNCATION_MASK ? PSA_MAC_TRUNCATED_LENGTH(alg) : PSA_ALG_IS_HMAC(alg) ? PSA
```

Parameters:

key_type The type of the MAC key.

key_bits The size of the MAC key in bits.

alg A MAC algorithm (PSA_ALG_XXX value such that *PSA_ALG_IS_MAC*(alg) is true).

Returns:

The MAC size for the specified algorithm with the specified key parameters.

0 if the MAC algorithm is not recognized.

Either 0 or the correct size for a MAC algorithm that the implementation recognizes, but does not support.

Unspecified if the key parameters are not consistent with the algorithm. **Description:** The size of the output of *psa_mac_sign_finish()*, in bytes.

This is also the MAC size that *psa_mac_verify_finish()* expects.

PSA_AEAD_ENCRYPT_OUTPUT_SIZE (macro)

```
#define PSA_AEAD_ENCRYPT_OUTPUT_SIZE( alg,  plaintext_length) \
    (PSA_AEAD_TAG_LENGTH(alg) != 0 ? (plaintext_length) + PSA_AEAD_TAG_LENGTH(alg) : 0)
```

Parameters:

alg An AEAD algorithm (PSA_ALG_XXX value such that *PSA_ALG_IS_AEAD*(alg) is true).

plaintext_length Size of the plaintext in bytes.

Returns:

The AEAD ciphertext size for the specified algorithm. If the AEAD algorithm is not recognized, return 0. An implementation may return either 0 or a correct size for an AEAD algorithm that it recognizes, but does not support. **Description:** The maximum size of the output of *psa_aead_encrypt()*, in bytes.

If the size of the ciphertext buffer is at least this large, it is guaranteed that *psa_aead_encrypt()* will not fail due to an insufficient buffer size. Depending on the algorithm, the actual size of the ciphertext may be smaller.

PSA_AEAD_FINISH_OUTPUT_SIZE (macro)

```
#define PSA_AEAD_FINISH_OUTPUT_SIZE( alg)   ((size_t)0)
```

Parameters:

alg An AEAD algorithm (PSA_ALG_XXX value such that *PSA_ALG_IS_AEAD*(alg) is true).

Returns:

The maximum trailing ciphertext size for the specified algorithm. If the AEAD algorithm is not recognized, return 0. An implementation may return either 0 or a correct size for an AEAD algorithm that it recognizes, but does not support. **Description:** The maximum size of the output of *psa_aead_finish()*, in bytes.

If the size of the ciphertext buffer is at least this large, it is guaranteed that *psa_aead_finish()* will not fail due to an insufficient buffer size. Depending on the algorithm, the actual size of the ciphertext may be smaller.

PSA_AEAD_DECRYPT_OUTPUT_SIZE (macro)

```
#define PSA_AEAD_DECRYPT_OUTPUT_SIZE( alg,  ciphertext_length) \
    (PSA_AEAD_TAG_LENGTH(alg) != 0 ? (plaintext_length) - PSA_AEAD_TAG_LENGTH(alg) : 0)
```

Parameters:

alg An AEAD algorithm (PSA_ALG_XXX value such that *PSA_ALG_IS_AEAD*(alg) is true).

ciphertext_length Size of the plaintext in bytes.

Returns:

The AEAD ciphertext size for the specified algorithm. If the AEAD algorithm is not recognized, return 0. An implementation may return either 0 or a correct size for an AEAD algorithm that it recognizes, but does not support. **Description:** The maximum size of the output of *psa_aead_decrypt()*, in bytes.

If the size of the plaintext buffer is at least this large, it is guaranteed that *psa_aead_decrypt()* will not fail due to an insufficient buffer size. Depending on the algorithm, the actual size of the plaintext may be smaller.

PSA_RSA_MINIMUM_PADDING_SIZE (macro)

```
#define PSA_RSA_MINIMUM_PADDING_SIZE( alg) \
    (PSA_ALG_IS_RSA_OAEP(alg) ? 2 * PSA_HASH_FINAL_SIZE(PSA_ALG_RSA_OAEP_GET_HASH(alg)) + 1 : 11 /*PK
```

Parameters:

alg

Description:

PSA_ECDSA_SIGNATURE_SIZE (macro)

```
#define PSA_ECDSA_SIGNATURE_SIZE( curve_bits) \
    (PSA_BITS_TO_BYTES(curve_bits) * 2)
```

Parameters:

curve_bits Curve size in bits.

Returns:

Signature size in bytes. **Description:** ECDSA signature size for a given curve bit size.

Note: This macro returns a compile-time constant if its argument is one.

PSA_ASYMMETRIC_SIGN_OUTPUT_SIZE (macro)

```
#define PSA_ASYMMETRIC_SIGN_OUTPUT_SIZE( key_type,  key_bits,  alg) \
    (PSA_KEY_TYPE_IS_RSA(key_type) ? ((void)alg, PSA_BITS_TO_BYTES(key_bits)) : PSA_KEY_TYPE_IS_ECC(k
```

Parameters:

key_type An asymmetric key type (this may indifferently be a key pair type or a public key type).

key_bits The size of the key in bits.

alg The signature algorithm.

Returns:

If the parameters are valid and supported, return a buffer size in bytes that guarantees that *psa_asymmetric_sign()* will not fail with *PSA_ERROR_BUFFER_TOO_SMALL*. If the parameters are a valid combination that is not supported by the implementation, this macro either shall return either a sensible size or 0. If the parameters are not valid, the return value is unspecified. **Description:** Safe signature buffer size for *psa_asymmetric_sign()*.

This macro returns a safe buffer size for a signature using a key of the specified type and size, with the specified algorithm. Note that the actual size of the signature may be smaller (some algorithms produce a variable-size signature).

Warning: This function may call its arguments multiple times or zero times, so you should not pass arguments that contain side effects.

PSA_ASYMMETRIC_ENCRYPT_OUTPUT_SIZE (macro)

```
#define PSA_ASYMMETRIC_ENCRYPT_OUTPUT_SIZE( key_type,  key_bits,  alg) \
    (PSA_KEY_TYPE_IS_RSA(key_type) ? ((void)alg, PSA_BITS_TO_BYTES(key_bits)) : 0)
```

Parameters:

key_type An asymmetric key type (this may indifferently be a key pair type or a public key type).

key_bits The size of the key in bits.

alg The signature algorithm.

Returns:

If the parameters are valid and supported, return a buffer size in bytes that guarantees that *psa_asymmetric_encrypt()* will not fail with *PSA_ERROR_BUFFER_TOO_SMALL*. If the parameters are a valid combination that is not supported by the implementation, this macro either shall return either a sensible size or 0. If the parameters are not valid, the return value is unspecified. **Description:** Safe output buffer size for *psa_asymmetric_encrypt()*.

This macro returns a safe buffer size for a ciphertext produced using a key of the specified type and size, with the specified algorithm. Note that the actual size of the ciphertext may be smaller, depending on the algorithm.

> **Warning:** This function may call its arguments multiple times or zero times, so you should not pass arguments that contain side effects.

PSA_ASYMMETRIC_DECRYPT_OUTPUT_SIZE (macro)

```
#define PSA_ASYMMETRIC_DECRYPT_OUTPUT_SIZE( key_type,  key_bits,  alg) \
    (PSA_KEY_TYPE_IS_RSA(key_type) ? PSA_BITS_TO_BYTES(key_bits) - PSA_RSA_MINIMUM_PADDING_SIZE(alg)
```

Parameters:

key_type An asymmetric key type (this may indifferently be a key pair type or a public key type).

key_bits The size of the key in bits.

alg The signature algorithm.

Returns:

If the parameters are valid and supported, return a buffer size in bytes that guarantees that *psa_asymmetric_decrypt()* will not fail with *PSA_ERROR_BUFFER_TOO_SMALL*. If the parameters are a valid combination that is not supported by the implementation, this macro either shall return either a sensible size or 0. If the parameters are not valid, the return value is unspecified. **Description:** Safe output buffer size for *psa_asymmetric_decrypt()*.

This macro returns a safe buffer size for a ciphertext produced using a key of the specified type and size, with the specified algorithm. Note that the actual size of the ciphertext may be smaller, depending on the algorithm.

> **Warning:** This function may call its arguments multiple times or zero times, so you should not pass arguments that contain side effects.

PSA_KEY_EXPORT_ASN1_INTEGER_MAX_SIZE (macro)

```
#define PSA_KEY_EXPORT_ASN1_INTEGER_MAX_SIZE( bits)   ((bits) / 8 + 5)
```

Parameters:

bits

Description:

PSA_KEY_EXPORT_RSA_PUBLIC_KEY_MAX_SIZE (macro)

```
#define PSA_KEY_EXPORT_RSA_PUBLIC_KEY_MAX_SIZE( key_bits) \
    (PSA_KEY_EXPORT_ASN1_INTEGER_MAX_SIZE(key_bits) + 36)
```

Parameters:

key_bits

Description:

PSA_KEY_EXPORT_RSA_KEYPAIR_MAX_SIZE (macro)

```
#define PSA_KEY_EXPORT_RSA_KEYPAIR_MAX_SIZE( key_bits) \
    (9 * PSA_KEY_EXPORT_ASN1_INTEGER_MAX_SIZE((key_bits) / 2 + 1) + 14)
```

Parameters:

key_bits

Description:

PSA_KEY_EXPORT_DSA_PUBLIC_KEY_MAX_SIZE (macro)

```
#define PSA_KEY_EXPORT_DSA_PUBLIC_KEY_MAX_SIZE( key_bits) \
    (PSA_KEY_EXPORT_ASN1_INTEGER_MAX_SIZE(key_bits) * 3 + 59)
```

Parameters:

key_bits

Description:

PSA_KEY_EXPORT_DSA_KEYPAIR_MAX_SIZE (macro)

```
#define PSA_KEY_EXPORT_DSA_KEYPAIR_MAX_SIZE( key_bits) \
    (PSA_KEY_EXPORT_ASN1_INTEGER_MAX_SIZE(key_bits) * 3 + 75)
```

Parameters:

key_bits

Description:

PSA_KEY_EXPORT_ECC_PUBLIC_KEY_MAX_SIZE (macro)

```
#define PSA_KEY_EXPORT_ECC_PUBLIC_KEY_MAX_SIZE( key_bits) \
    (2 * PSA_BITS_TO_BYTES(key_bits) + 36)
```

Parameters:

key_bits

Description:

PSA_KEY_EXPORT_ECC_KEYPAIR_MAX_SIZE (macro)

```
#define PSA_KEY_EXPORT_ECC_KEYPAIR_MAX_SIZE( key_bits) \
    (PSA_BITS_TO_BYTES(key_bits))
```

Parameters:

key_bits

Description:

PSA_KEY_EXPORT_MAX_SIZE (macro)

```
#define PSA_KEY_EXPORT_MAX_SIZE( key_type,  key_bits) \
    (PSA_KEY_TYPE_IS_UNSTRUCTURED(key_type) ? PSA_BITS_TO_BYTES(key_bits) : (key_type) == PSA_KEY_TYP
```

Parameters:

key_type A supported key type.

key_bits The size of the key in bits.

Returns:

If the parameters are valid and supported, return a buffer size in bytes that guarantees that *psa_asymmetric_sign()* will not fail with *PSA_ERROR_BUFFER_TOO_SMALL*. If the parameters are a valid combination that is not supported by the implementation, this macro either shall return either a sensible size or 0. If the parameters are not valid, the return value is unspecified. **Description:** Safe output buffer size for *psa_export_key()* or *psa_export_public_key()*.

This macro returns a compile-time constant if its arguments are compile-time constants.

> **Warning:** This function may call its arguments multiple times or zero times, so you should not pass arguments that contain side effects.

The following code illustrates how to allocate enough memory to export a key by querying the key type and size at runtime.

```
psa_key_type_t key_type;
size_t key_bits;
psa_status_t status;
status = psa_get_key_information(key, &key_type, &key_bits);
if (status != PSA_SUCCESS) handle_error(...);
size_t buffer_size = PSA_KEY_EXPORT_MAX_SIZE(key_type, key_bits);
unsigned char *buffer = malloc(buffer_size);
```

```
if (buffer != NULL) handle_error(...);
size_t buffer_length;
status = psa_export_key(key, buffer, buffer_size, &buffer_length);
if (status != PSA_SUCCESS) handle_error(...);
```

For *psa_export_public_key()*, calculate the buffer size from the public key type. You can use the macro *PSA_KEY_TYPE_PUBLIC_KEY_OF_KEYPAIR* to convert a key pair type to the corresponding public key type.

```
psa_key_type_t key_type;
size_t key_bits;
psa_status_t status;
status = psa_get_key_information(key, &key_type, &key_bits);
if (status != PSA_SUCCESS) handle_error(...);
psa_key_type_t public_key_type = PSA_KEY_TYPE_PUBLIC_KEY_OF_KEYPAIR(key_type);
size_t buffer_size = PSA_KEY_EXPORT_MAX_SIZE(public_key_type, key_bits);
unsigned char *buffer = malloc(buffer_size);
if (buffer != NULL) handle_error(...);
size_t buffer_length;
status = psa_export_public_key(key, buffer, buffer_size, &buffer_length);
if (status != PSA_SUCCESS) handle_error(...);
```

CHAPTER

TWENTYFIVE

DOCUMENT HISTORY

Date	Changes
2019-01-21	*Release 1.0 beta 1*
2019-02-08	• Remove obsolete definition `PSA_ALG_IS_KEY_SELECTION`. • *`psa_key_agreement`*: document `alg` parameter. • *`PSA_AEAD_FINISH_OUTPUT_SIZE`*: remove spurious paraameter `plaintext_length`.
2019-02-08	Document formatting improvements
2019-02-22	*Release 1.0 beta 2*