# Platform Security Architecture — cryptography and keystore interface

Working draft

Generated by Doxygen 1.8.13

# Contents

# Chapter 1

# Module Index

## 1.1 Modules

Here is a list of all modules:

# Chapter 2

# Class Index

## 2.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

# Chapter 3

# File Index

## 3.1 File List

Here is a list of all documented files with brief descriptions:

# Chapter 4

# Module Documentation

## 4.1 Implementation-specific definitions

### Typedefs

- typedef _unsigned_integral_type_ psa_key_slot_t

  *Key slot number.*

### 4.1.1 Detailed Description

### 4.1.2 Typedef Documentation

#### 4.1.2.1 psa_key_slot_t

```
typedef _unsigned_integral_type_ psa_key_slot_t
```

Key slot number.

This type represents key slots. It must be an unsigned integral type. The choice of type is implementation-dependent. 0 is not a valid key slot number. The meaning of other values is implementation dependent.

At any given point in time, each key slot either contains a cryptographic object, or is empty. Key slots are persistent: once set, the cryptographic object remains in the key slot until explicitly destroyed.

## 4.2 Basic definitions

### Macros

- #define PSA_SUCCESS ((psa_status_t)0)
- #define PSA_ERROR_UNKNOWN_ERROR ((psa_status_t)1)
- #define PSA_ERROR_NOT_SUPPORTED ((psa_status_t)2)
- #define PSA_ERROR_NOT_PERMITTED ((psa_status_t)3)
- #define PSA_ERROR_BUFFER_TOO_SMALL ((psa_status_t)4)
- #define PSA_ERROR_OCCUPIED_SLOT ((psa_status_t)5)
- #define PSA_ERROR_EMPTY_SLOT ((psa_status_t)6)
- #define PSA_ERROR_BAD_STATE ((psa_status_t)7)
- #define PSA_ERROR_INVALID_ARGUMENT ((psa_status_t)8)
- #define PSA_ERROR_INSUFFICIENT_MEMORY ((psa_status_t)9)
- #define PSA_ERROR_INSUFFICIENT_STORAGE ((psa_status_t)10)
- #define PSA_ERROR_COMMUNICATION_FAILURE ((psa_status_t)11)
- #define PSA_ERROR_STORAGE_FAILURE ((psa_status_t)12)
- #define PSA_ERROR_HARDWARE_FAILURE ((psa_status_t)13)
- #define PSA_ERROR_TAMPERING_DETECTED ((psa_status_t)14)
- #define PSA_ERROR_INSUFFICIENT_ENTROPY ((psa_status_t)15)
- #define PSA_ERROR_INVALID_SIGNATURE ((psa_status_t)16)
- #define PSA_ERROR_INVALID_PADDING ((psa_status_t)17)
- #define PSA_ERROR_INSUFFICIENT_CAPACITY ((psa_status_t)18)
- #define **PSA_BITS_TO_BYTES**(bits) (((bits) + 7) / 8)
- #define **PSA_BYTES_TO_BITS**(bytes) ((bytes) ∗ 8)

### Typedefs

- typedef int32_t psa_status_t

  *Function return status.*

### Functions

- psa_status_t psa_crypto_init (void)

  *Library initialization.*

### 4.2.1 Detailed Description

### 4.2.2 Macro Definition Documentation

#### 4.2.2.1 PSA_ERROR_BAD_STATE

```
#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.

#### 4.2.2.2 PSA_ERROR_BUFFER_TOO_SMALL

```
#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.

#### 4.2.2.3 PSA_ERROR_COMMUNICATION_FAILURE

```
#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.

#### 4.2.2.4 PSA_ERROR_EMPTY_SLOT

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

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

If the slot number is invalid (i.e. the requested action could not be performed even after creating appropriate content in the slot), implementations shall return PSA_ERROR_INVALID_ARGUMENT instead.

#### 4.2.2.5 PSA_ERROR_HARDWARE_FAILURE

```
#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.

#### 4.2.2.6 PSA_ERROR_INSUFFICIENT_CAPACITY

```
#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.

#### 4.2.2.7 PSA_ERROR_INSUFFICIENT_ENTROPY

```
#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.

#### 4.2.2.8 PSA_ERROR_INSUFFICIENT_MEMORY

```
#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.

#### 4.2.2.9 PSA_ERROR_INSUFFICIENT_STORAGE

```
#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.

#### 4.2.2.10 PSA_ERROR_INVALID_ARGUMENT

```
#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.

#### 4.2.2.11 PSA_ERROR_INVALID_PADDING

```
#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.

#### 4.2.2.12 PSA_ERROR_INVALID_SIGNATURE

```
#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.

#### 4.2.2.13 PSA_ERROR_NOT_PERMITTED

```
#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.

#### 4.2.2.14 PSA_ERROR_NOT_SUPPORTED

```
#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.

#### 4.2.2.15 PSA_ERROR_OCCUPIED_SLOT

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

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

If the slot number is invalid (i.e. the requested action could not be performed even after erasing the slot's content), implementations shall return PSA_ERROR_INVALID_ARGUMENT instead.

#### 4.2.2.16 PSA_ERROR_STORAGE_FAILURE

```
#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 canont 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.

#### 4.2.2.17 PSA_ERROR_TAMPERING_DETECTED

```
#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_ERR↩
OR_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.

#### 4.2.2.18 PSA_ERROR_UNKNOWN_ERROR

```
#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.

#### 4.2.2.19 PSA_SUCCESS

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

The action was completed successfully.

### 4.2.3 Typedef Documentation

#### 4.2.3.1 psa_status_t

```
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.

### 4.2.4 Function Documentation

#### 4.2.4.1 psa_crypto_init()

```
psa_status_t psa_crypto_init (
            void  )
```

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.

**Return values**

| | |
|---|---|
| *PSA_SUCCESS* | |
| *PSA_ERROR_INSUFFICIENT_MEMORY* | |
| *PSA_ERROR_COMMUNICATION_FAILURE* | |
| *PSA_ERROR_HARDWARE_FAILURE* | |
| *PSA_ERROR_TAMPERING_DETECTED* | |
| *PSA_ERROR_INSUFFICIENT_ENTROPY* | |

## 4.3 Key and algorithm types

### Macros

- #define PSA_KEY_TYPE_NONE ((psa_key_type_t)0x00000000)
- #define PSA_KEY_TYPE_VENDOR_FLAG ((psa_key_type_t)0x80000000)
- #define **PSA_KEY_TYPE_CATEGORY_MASK** ((psa_key_type_t)0x7e000000)
- #define PSA_KEY_TYPE_RAW_DATA ((psa_key_type_t)0x02000000)
- #define **PSA_KEY_TYPE_CATEGORY_SYMMETRIC** ((psa_key_type_t)0x04000000)
- #define **PSA_KEY_TYPE_CATEGORY_ASYMMETRIC** ((psa_key_type_t)0x06000000)
- #define **PSA_KEY_TYPE_PAIR_FLAG** ((psa_key_type_t)0x01000000)
- #define PSA_KEY_TYPE_HMAC ((psa_key_type_t)0x02000001)
- #define PSA_KEY_TYPE_DERIVE ((psa_key_type_t)0x02000101)
- #define PSA_KEY_TYPE_AES ((psa_key_type_t)0x04000001)
- #define PSA_KEY_TYPE_DES ((psa_key_type_t)0x04000002)
- #define PSA_KEY_TYPE_CAMELLIA ((psa_key_type_t)0x04000003)
- #define PSA_KEY_TYPE_ARC4 ((psa_key_type_t)0x04000004)
- #define PSA_KEY_TYPE_RSA_PUBLIC_KEY ((psa_key_type_t)0x06010000)
- #define PSA_KEY_TYPE_RSA_KEYPAIR ((psa_key_type_t)0x07010000)
- #define PSA_KEY_TYPE_DSA_PUBLIC_KEY ((psa_key_type_t)0x06020000)
- #define PSA_KEY_TYPE_DSA_KEYPAIR ((psa_key_type_t)0x07020000)
- #define **PSA_KEY_TYPE_ECC_PUBLIC_KEY_BASE** ((psa_key_type_t)0x06030000)
- #define **PSA_KEY_TYPE_ECC_KEYPAIR_BASE** ((psa_key_type_t)0x07030000)
- #define **PSA_KEY_TYPE_ECC_CURVE_MASK** ((psa_key_type_t)0x0000ffff)
- #define PSA_KEY_TYPE_ECC_KEYPAIR(curve) (PSA_KEY_TYPE_ECC_KEYPAIR_BASE | (curve))
- #define PSA_KEY_TYPE_ECC_PUBLIC_KEY(curve) (PSA_KEY_TYPE_ECC_PUBLIC_KEY_BASE | (curve))
- #define PSA_KEY_TYPE_IS_VENDOR_DEFINED(type) (((type) & PSA_KEY_TYPE_VENDOR_FLAG) != 0)
- #define PSA_KEY_TYPE_IS_ASYMMETRIC(type) (((type) & PSA_KEY_TYPE_CATEGORY_MASK) == PSA_KEY_TYPE_CATEGORY_ASYMMETRIC)
- #define PSA_KEY_TYPE_IS_PUBLIC_KEY(type)
- #define PSA_KEY_TYPE_IS_KEYPAIR(type)
- #define PSA_KEY_TYPE_KEYPAIR_OF_PUBLIC_KEY(type) ((type) | PSA_KEY_TYPE_PAIR_FLAG)
- #define PSA_KEY_TYPE_PUBLIC_KEY_OF_KEYPAIR(type) ((type) & ~PSA_KEY_TYPE_PAIR_FLAG)
- #define PSA_KEY_TYPE_IS_RSA(type) (PSA_KEY_TYPE_PUBLIC_KEY_OF_KEYPAIR(type) == PSA_↩ KEY_TYPE_RSA_PUBLIC_KEY)
- #define PSA_KEY_TYPE_IS_ECC(type)
- #define **PSA_KEY_TYPE_IS_ECC_KEYPAIR**(type)
- #define **PSA_KEY_TYPE_IS_ECC_PUBLIC_KEY**(type)
- #define PSA_KEY_TYPE_GET_CURVE(type)
- #define **PSA_ECC_CURVE_SECT163K1** ((psa_ecc_curve_t) 0x0001)
- #define **PSA_ECC_CURVE_SECT163R1** ((psa_ecc_curve_t) 0x0002)
- #define **PSA_ECC_CURVE_SECT163R2** ((psa_ecc_curve_t) 0x0003)
- #define **PSA_ECC_CURVE_SECT193R1** ((psa_ecc_curve_t) 0x0004)
- #define **PSA_ECC_CURVE_SECT193R2** ((psa_ecc_curve_t) 0x0005)
- #define **PSA_ECC_CURVE_SECT233K1** ((psa_ecc_curve_t) 0x0006)
- #define **PSA_ECC_CURVE_SECT233R1** ((psa_ecc_curve_t) 0x0007)
- #define **PSA_ECC_CURVE_SECT239K1** ((psa_ecc_curve_t) 0x0008)
- #define **PSA_ECC_CURVE_SECT283K1** ((psa_ecc_curve_t) 0x0009)
- #define **PSA_ECC_CURVE_SECT283R1** ((psa_ecc_curve_t) 0x000a)
- #define **PSA_ECC_CURVE_SECT409K1** ((psa_ecc_curve_t) 0x000b)
- #define **PSA_ECC_CURVE_SECT409R1** ((psa_ecc_curve_t) 0x000c)
- #define **PSA_ECC_CURVE_SECT571K1** ((psa_ecc_curve_t) 0x000d)
- #define **PSA_ECC_CURVE_SECT571R1** ((psa_ecc_curve_t) 0x000e)

- #define **PSA_ECC_CURVE_SECP160K1** ((psa_ecc_curve_t) 0x000f)
- #define **PSA_ECC_CURVE_SECP160R1** ((psa_ecc_curve_t) 0x0010)
- #define **PSA_ECC_CURVE_SECP160R2** ((psa_ecc_curve_t) 0x0011)
- #define **PSA_ECC_CURVE_SECP192K1** ((psa_ecc_curve_t) 0x0012)
- #define **PSA_ECC_CURVE_SECP192R1** ((psa_ecc_curve_t) 0x0013)
- #define **PSA_ECC_CURVE_SECP224K1** ((psa_ecc_curve_t) 0x0014)
- #define **PSA_ECC_CURVE_SECP224R1** ((psa_ecc_curve_t) 0x0015)
- #define **PSA_ECC_CURVE_SECP256K1** ((psa_ecc_curve_t) 0x0016)
- #define **PSA_ECC_CURVE_SECP256R1** ((psa_ecc_curve_t) 0x0017)
- #define **PSA_ECC_CURVE_SECP384R1** ((psa_ecc_curve_t) 0x0018)
- #define **PSA_ECC_CURVE_SECP521R1** ((psa_ecc_curve_t) 0x0019)
- #define **PSA_ECC_CURVE_BRAINPOOL_P256R1** ((psa_ecc_curve_t) 0x001a)
- #define **PSA_ECC_CURVE_BRAINPOOL_P384R1** ((psa_ecc_curve_t) 0x001b)
- #define **PSA_ECC_CURVE_BRAINPOOL_P512R1** ((psa_ecc_curve_t) 0x001c)
- #define **PSA_ECC_CURVE_CURVE25519** ((psa_ecc_curve_t) 0x001d)
- #define **PSA_ECC_CURVE_CURVE448** ((psa_ecc_curve_t) 0x001e)
- #define **PSA_ECC_CURVE_FFDHE_2048** ((psa_ecc_curve_t) 0x0100)
- #define **PSA_ECC_CURVE_FFDHE_3072** ((psa_ecc_curve_t) 0x0101)
- #define **PSA_ECC_CURVE_FFDHE_4096** ((psa_ecc_curve_t) 0x0102)
- #define **PSA_ECC_CURVE_FFDHE_6144** ((psa_ecc_curve_t) 0x0103)
- #define **PSA_ECC_CURVE_FFDHE_8192** ((psa_ecc_curve_t) 0x0104)
- #define PSA_BLOCK_CIPHER_BLOCK_SIZE(type)
- #define **PSA_ALG_VENDOR_FLAG** ((psa_algorithm_t)0x80000000)
- #define **PSA_ALG_CATEGORY_MASK** ((psa_algorithm_t)0x7f000000)
- #define **PSA_ALG_CATEGORY_HASH** ((psa_algorithm_t)0x01000000)
- #define **PSA_ALG_CATEGORY_MAC** ((psa_algorithm_t)0x02000000)
- #define **PSA_ALG_CATEGORY_CIPHER** ((psa_algorithm_t)0x04000000)
- #define **PSA_ALG_CATEGORY_AEAD** ((psa_algorithm_t)0x06000000)
- #define **PSA_ALG_CATEGORY_SIGN** ((psa_algorithm_t)0x10000000)
- #define **PSA_ALG_CATEGORY_ASYMMETRIC_ENCRYPTION** ((psa_algorithm_t)0x12000000)
- #define **PSA_ALG_CATEGORY_KEY_AGREEMENT** ((psa_algorithm_t)0x22000000)
- #define **PSA_ALG_CATEGORY_KEY_DERIVATION** ((psa_algorithm_t)0x30000000)
- #define **PSA_ALG_IS_VENDOR_DEFINED**(alg) (((alg) & PSA_ALG_VENDOR_FLAG) != 0)
- #define PSA_ALG_IS_HASH(alg) (((alg) & PSA_ALG_CATEGORY_MASK) == PSA_ALG_CATEGORY_↩
HASH)
- #define PSA_ALG_IS_MAC(alg) (((alg) & PSA_ALG_CATEGORY_MASK) == PSA_ALG_CATEGORY_M↩
AC)
- #define PSA_ALG_IS_CIPHER(alg) (((alg) & PSA_ALG_CATEGORY_MASK) == PSA_ALG_CATEGOR↩
Y_CIPHER)
- #define PSA_ALG_IS_AEAD(alg) (((alg) & PSA_ALG_CATEGORY_MASK) == PSA_ALG_CATEGORY_↩
AEAD)
- #define PSA_ALG_IS_SIGN(alg) (((alg) & PSA_ALG_CATEGORY_MASK) == PSA_ALG_CATEGORY_S↩
IGN)
- #define PSA_ALG_IS_ASYMMETRIC_ENCRYPTION(alg) (((alg) & PSA_ALG_CATEGORY_MASK) == P↩
SA_ALG_CATEGORY_ASYMMETRIC_ENCRYPTION)
- #define PSA_ALG_IS_KEY_AGREEMENT(alg) (((alg) & PSA_ALG_CATEGORY_MASK) == PSA_ALG_↩
CATEGORY_KEY_AGREEMENT)
- #define PSA_ALG_IS_KEY_DERIVATION(alg) (((alg) & PSA_ALG_CATEGORY_MASK) == PSA_ALG_↩
CATEGORY_KEY_DERIVATION)
- #define **PSA_ALG_HASH_MASK** ((psa_algorithm_t)0x000000ff)
- #define **PSA_ALG_MD2** ((psa_algorithm_t)0x01000001)
- #define **PSA_ALG_MD4** ((psa_algorithm_t)0x01000002)
- #define **PSA_ALG_MD5** ((psa_algorithm_t)0x01000003)
- #define **PSA_ALG_RIPEMD160** ((psa_algorithm_t)0x01000004)
- #define **PSA_ALG_SHA_1** ((psa_algorithm_t)0x01000005)

- #define PSA_ALG_SHA_224 ((psa_algorithm_t)0x01000008)
- #define PSA_ALG_SHA_256 ((psa_algorithm_t)0x01000009)
- #define PSA_ALG_SHA_384 ((psa_algorithm_t)0x0100000a)
- #define PSA_ALG_SHA_512 ((psa_algorithm_t)0x0100000b)
- #define PSA_ALG_SHA_512_224 ((psa_algorithm_t)0x0100000c)
- #define PSA_ALG_SHA_512_256 ((psa_algorithm_t)0x0100000d)
- #define PSA_ALG_SHA3_224 ((psa_algorithm_t)0x01000010)
- #define PSA_ALG_SHA3_256 ((psa_algorithm_t)0x01000011)
- #define PSA_ALG_SHA3_384 ((psa_algorithm_t)0x01000012)
- #define PSA_ALG_SHA3_512 ((psa_algorithm_t)0x01000013)
- #define **PSA_ALG_MAC_SUBCATEGORY_MASK** ((psa_algorithm_t)0x00c00000)
- #define **PSA_ALG_HMAC_BASE** ((psa_algorithm_t)0x02800000)
- #define PSA_ALG_HMAC(hash_alg) (PSA_ALG_HMAC_BASE | ((hash_alg) & PSA_ALG_HASH_MASK))
- #define **PSA_ALG_HMAC_HASH**(hmac_alg) (PSA_ALG_CATEGORY_HASH | ((hmac_alg) & PSA_AL↩G_HASH_MASK))
- #define PSA_ALG_IS_HMAC(alg)
- #define **PSA_ALG_CIPHER_MAC_BASE** ((psa_algorithm_t)0x02c00000)
- #define **PSA_ALG_CBC_MAC** ((psa_algorithm_t)0x02c00001)
- #define **PSA_ALG_CMAC** ((psa_algorithm_t)0x02c00002)
- #define **PSA_ALG_GMAC** ((psa_algorithm_t)0x02c00003)
- #define PSA_ALG_IS_CIPHER_MAC(alg)
- #define **PSA_ALG_CIPHER_SUBCATEGORY_MASK** ((psa_algorithm_t)0x00c00000)
- #define **PSA_ALG_BLOCK_CIPHER_BASE** ((psa_algorithm_t)0x04000000)
- #define **PSA_ALG_BLOCK_CIPHER_MODE_MASK** ((psa_algorithm_t)0x000000ff)
- #define **PSA_ALG_BLOCK_CIPHER_PADDING_MASK** ((psa_algorithm_t)0x003f0000)
- #define PSA_ALG_BLOCK_CIPHER_PAD_NONE ((psa_algorithm_t)0x00000000)
- #define **PSA_ALG_BLOCK_CIPHER_PAD_PKCS7** ((psa_algorithm_t)0x00010000)
- #define PSA_ALG_IS_BLOCK_CIPHER(alg)
- #define PSA_ALG_CBC_BASE ((psa_algorithm_t)0x04000001)
- #define **PSA_ALG_CFB_BASE** ((psa_algorithm_t)0x04000002)
- #define **PSA_ALG_OFB_BASE** ((psa_algorithm_t)0x04000003)
- #define **PSA_ALG_XTS_BASE** ((psa_algorithm_t)0x04000004)
- #define **PSA_ALG_STREAM_CIPHER_BASE** ((psa_algorithm_t)0x04800000)
- #define PSA_ALG_CTR ((psa_algorithm_t)0x04800001)
- #define PSA_ALG_ARC4 ((psa_algorithm_t)0x04800002)
- #define PSA_ALG_IS_STREAM_CIPHER(alg)
- #define **PSA_ALG_CCM** ((psa_algorithm_t)0x06000001)
- #define **PSA_ALG_GCM** ((psa_algorithm_t)0x06000002)
- #define **PSA_ALG_RSA_PKCS1V15_SIGN_BASE** ((psa_algorithm_t)0x10020000)
- #define PSA_ALG_RSA_PKCS1V15_SIGN(hash_alg) (PSA_ALG_RSA_PKCS1V15_SIGN_BASE | ((hash_alg) & PSA_ALG_HASH_MASK))
- #define PSA_ALG_RSA_PKCS1V15_SIGN_RAW PSA_ALG_RSA_PKCS1V15_SIGN_BASE
- #define **PSA_ALG_IS_RSA_PKCS1V15_SIGN**(alg) (((alg) & ∼PSA_ALG_HASH_MASK) == PSA_ALG_↩RSA_PKCS1V15_SIGN_BASE)
- #define **PSA_ALG_RSA_PSS_BASE** ((psa_algorithm_t)0x10030000)
- #define PSA_ALG_RSA_PSS(hash_alg) (PSA_ALG_RSA_PSS_BASE | ((hash_alg) & PSA_ALG_HASH↩_MASK))
- #define **PSA_ALG_IS_RSA_PSS**(alg) (((alg) & ∼PSA_ALG_HASH_MASK) == PSA_ALG_RSA_PSS_B↩ASE)
- #define **PSA_ALG_DSA_BASE** ((psa_algorithm_t)0x10040000)
- #define PSA_ALG_DSA(hash_alg) (PSA_ALG_DSA_BASE | ((hash_alg) & PSA_ALG_HASH_MASK))
- #define **PSA_ALG_DETERMINISTIC_DSA_BASE** ((psa_algorithm_t)0x10050000)
- #define **PSA_ALG_DSA_DETERMINISTIC_FLAG** ((psa_algorithm_t)0x00010000)
- #define **PSA_ALG_DETERMINISTIC_DSA**(hash_alg) (PSA_ALG_DETERMINISTIC_DSA_BASE | ((hash_alg) & PSA_ALG_HASH_MASK))

- #define **PSA_ALG_IS_DSA**(alg)
- #define **PSA_ALG_DSA_IS_DETERMINISTIC**(alg) (((alg) & PSA_ALG_DSA_DETERMINISTIC_FLAG) != 0)
- #define **PSA_ALG_IS_DETERMINISTIC_DSA**(alg) (PSA_ALG_IS_DSA(alg) && PSA_ALG_DSA_IS_DE↩ TERMINISTIC(alg))
- #define **PSA_ALG_IS_RANDOMIZED_DSA**(alg) (PSA_ALG_IS_DSA(alg) && !PSA_ALG_DSA_IS_DET↩ ERMINISTIC(alg))
- #define **PSA_ALG_ECDSA_BASE** ((psa_algorithm_t)0x10060000)
- #define PSA_ALG_ECDSA(hash_alg) (PSA_ALG_ECDSA_BASE | ((hash_alg) & PSA_ALG_HASH_MA↩ SK))
- #define PSA_ALG_ECDSA_ANY PSA_ALG_ECDSA_BASE
- #define **PSA_ALG_DETERMINISTIC_ECDSA_BASE** ((psa_algorithm_t)0x10070000)
- #define PSA_ALG_DETERMINISTIC_ECDSA(hash_alg) (PSA_ALG_DETERMINISTIC_ECDSA_BASE | ((hash_alg) & PSA_ALG_HASH_MASK))
- #define **PSA_ALG_IS_ECDSA**(alg)
- #define **PSA_ALG_ECDSA_IS_DETERMINISTIC**(alg) (((alg) & PSA_ALG_DSA_DETERMINISTIC_FLAG) != 0)
- #define **PSA_ALG_IS_DETERMINISTIC_ECDSA**(alg) (PSA_ALG_IS_ECDSA(alg) && PSA_ALG_ECDS↩ A_IS_DETERMINISTIC(alg))
- #define **PSA_ALG_IS_RANDOMIZED_ECDSA**(alg) (PSA_ALG_IS_ECDSA(alg) && !PSA_ALG_ECDSA↩ _IS_DETERMINISTIC(alg))
- #define PSA_ALG_SIGN_GET_HASH(alg)
- #define PSA_ALG_RSA_PKCS1V15_CRYPT ((psa_algorithm_t)0x12020000)
- #define **PSA_ALG_RSA_OAEP_BASE** ((psa_algorithm_t)0x12030000)
- #define PSA_ALG_RSA_OAEP(hash_alg) (PSA_ALG_RSA_OAEP_BASE | ((hash_alg) & PSA_ALG_HA↩ SH_MASK))
- #define **PSA_ALG_IS_RSA_OAEP**(alg) (((alg) & ∼PSA_ALG_HASH_MASK) == PSA_ALG_RSA_OAEP↩ _BASE)
- #define **PSA_ALG_RSA_OAEP_GET_HASH**(alg)
- #define **PSA_ALG_HKDF_BASE** ((psa_algorithm_t)0x30000100)
- #define PSA_ALG_HKDF(hash_alg) (PSA_ALG_HKDF_BASE | ((hash_alg) & PSA_ALG_HASH_MASK))
- #define PSA_ALG_IS_HKDF(alg) (((alg) & ∼PSA_ALG_HASH_MASK) == PSA_ALG_HKDF_BASE)
- #define **PSA_ALG_HKDF_GET_HASH**(hkdf_alg) (PSA_ALG_CATEGORY_HASH | ((hkdf_alg) & PSA_↩ ALG_HASH_MASK))

## Typedefs

- typedef uint32_t psa_key_type_t

  *Encoding of a key type.*
- typedef uint16_t psa_ecc_curve_t
- typedef uint32_t psa_algorithm_t

  *Encoding of a cryptographic algorithm.*

### 4.3.1 Detailed Description

### 4.3.2 Macro Definition Documentation

#### 4.3.2.1 PSA_ALG_ARC4

```
#define PSA_ALG_ARC4 ((psa_algorithm_t)0x04800002)
```

The ARC4 stream cipher algorithm.

#### 4.3.2.2 PSA_ALG_BLOCK_CIPHER_PAD_NONE

```
#define PSA_ALG_BLOCK_CIPHER_PAD_NONE ((psa_algorithm_t)0x00000000)
```

Use a block cipher mode without padding.

This padding mode may only be used with messages whose lengths are a whole number of blocks for the chosen block cipher.

#### 4.3.2.3 PSA_ALG_CBC_BASE

```
#define PSA_ALG_CBC_BASE ((psa_algorithm_t)0x04000001)
```

The CBC block cipher mode.

#### 4.3.2.4 PSA_ALG_CTR

```
#define PSA_ALG_CTR ((psa_algorithm_t)0x04800001)
```

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).

#### 4.3.2.5 PSA_ALG_DETERMINISTIC_ECDSA

```
#define PSA_ALG_DETERMINISTIC_ECDSA(
              hash_alg ) (PSA_ALG_DETERMINISTIC_ECDSA_BASE | ((hash_alg) & PSA_ALG_HASH_MASK))
```

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_EC↩ DSA(`hash_alg`)) with the same private key are accepted. In other words, PSA_ALG_DETERMINISTIC_ECD↩ SA(`hash_alg`) differs from PSA_ALG_ECDSA(`hash_alg`) only for signature, not for verification.

**Parameters**

| | |
|---|---|
| *hash_alg* | A hash algorithm (`PSA_ALG_XXX` value such that PSA_ALG_IS_HASH(`hash_alg`) is true). |

**Returns**

The corresponding deterministic ECDSA signature algorithm.
Unspecified if `alg` is not a supported hash algorithm.

#### 4.3.2.6 PSA_ALG_DSA

```
#define PSA_ALG_DSA(
              hash_alg ) (PSA_ALG_DSA_BASE | ((hash_alg) & PSA_ALG_HASH_MASK))
```

DSA signature with hashing.

This is the signature scheme defined by FIPS 186-4, with a random per-message secret number (*k*).

**Parameters**

| | |
|---|---|
| *hash_alg* | A hash algorithm (`PSA_ALG_XXX` value such that PSA_ALG_IS_HASH(`hash_alg`) is true). |

**Returns**

The corresponding DSA signature algorithm.
Unspecified if `alg` is not a supported hash algorithm.

#### 4.3.2.7 PSA_ALG_ECDSA

```
#define PSA_ALG_ECDSA(
              hash_alg ) (PSA_ALG_ECDSA_BASE | ((hash_alg) & PSA_ALG_HASH_MASK))
```

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).

**Parameters**

| | |
|---|---|
| *hash_alg* | A hash algorithm (`PSA_ALG_XXX` value such that PSA_ALG_IS_HASH(`hash_alg`) is true). |

**Returns**

The corresponding ECDSA signature algorithm.
Unspecified if `alg` is not a supported hash algorithm.

### 4.3.2.8 PSA_ALG_ECDSA_ANY

```
#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.

### 4.3.2.9 PSA_ALG_HKDF

```
#define PSA_ALG_HKDF(
              hash_alg ) (PSA_ALG_HKDF_BASE | ((hash_alg) & PSA_ALG_HASH_MASK))
```

Macro to build an HKDF algorithm.

For example, `PSA_ALG_HKDF(PSA_ALG_SHA256)` is HKDF using HMAC-SHA-256.

**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.

### 4.3.2.10 PSA_ALG_HMAC

```
#define PSA_ALG_HMAC(
              hash_alg ) (PSA_ALG_HMAC_BASE | ((hash_alg) & PSA_ALG_HASH_MASK))
```

Macro to build an HMAC algorithm.

For example, PSA_ALG_HMAC(PSA_ALG_SHA_256) is HMAC-SHA-256.

**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.

#### 4.3.2.11 PSA_ALG_IS_AEAD

```
#define PSA_ALG_IS_AEAD(
              alg ) (((alg) & PSA_ALG_CATEGORY_MASK) == PSA_ALG_CATEGORY_AEAD)
```

Whether the specified algorithm is an authenticated encryption with associated data (AEAD) algorithm.

**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.

#### 4.3.2.12 PSA_ALG_IS_ASYMMETRIC_ENCRYPTION

```
#define PSA_ALG_IS_ASYMMETRIC_ENCRYPTION(
              alg ) (((alg) & PSA_ALG_CATEGORY_MASK) == PSA_ALG_CATEGORY_ASYMMETRIC_ENCRYPTI↩
ON)
```

Whether the specified algorithm is a public-key encryption algorithm.

**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.

#### 4.3.2.13 PSA_ALG_IS_BLOCK_CIPHER

```
#define PSA_ALG_IS_BLOCK_CIPHER(
              alg )
```

**Value:**

```
(((alg) & (PSA_ALG_CATEGORY_MASK | PSA_ALG_CIPHER_SUBCATEGORY_MASK)) == \
        PSA_ALG_BLOCK_CIPHER_BASE)
```

Whether the specified algorithm is a block cipher.

A block cipher is a symmetric cipher that encrypts or decrypts messages by chopping them into fixed-size blocks. Processing a message requires applying a *padding mode* to transform the message into one whose length is a whole number of blocks. To construct an algorithm identifier for a block cipher, apply a bitwise-or between the block cipher mode and the padding mode. For example, CBC with PKCS#7 padding is `PSA_ALG_CBC_BASE | PSA_ALG_BLOCK_CIPHER_PAD_PKCS7`.

The transformation applied to each block is determined by the key type. For example, to use AES-128-CBC-PKCS7, use the algorithm above with a key of type PSA_KEY_TYPE_AES and a length of 128 bits (16 bytes).

**Parameters**

| | |
|---|---|
| *alg* | An algorithm identifier (value of type psa_algorithm_t). |

**Returns**

1 if `alg` is a block 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.

#### 4.3.2.14 PSA_ALG_IS_CIPHER

```
#define PSA_ALG_IS_CIPHER(
              alg ) (((alg) & PSA_ALG_CATEGORY_MASK) == PSA_ALG_CATEGORY_CIPHER)
```

Whether the specified algorithm is a symmetric cipher algorithm.

**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.

#### 4.3.2.15 PSA_ALG_IS_CIPHER_MAC

```
#define PSA_ALG_IS_CIPHER_MAC(
              alg )
```

**Value:**

```
(((alg) & (PSA_ALG_CATEGORY_MASK | PSA_ALG_MAC_SUBCATEGORY_MASK)) == \
     PSA_ALG_CIPHER_MAC_BASE)
```

Whether the specified algorithm is a MAC algorithm based on a block cipher.

**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.

#### 4.3.2.16 PSA_ALG_IS_DSA

```
#define PSA_ALG_IS_DSA(
              alg )
```

**Value:**

```
(((alg) & ~PSA_ALG_HASH_MASK & ~PSA_ALG_DSA_DETERMINISTIC_FLAG) ==  \
     PSA_ALG_DSA_BASE)
```

#### 4.3.2.17 PSA_ALG_IS_ECDSA

```
#define PSA_ALG_IS_ECDSA(
              alg )
```

**Value:**

```
(((alg) & ~PSA_ALG_HASH_MASK & ~PSA_ALG_DSA_DETERMINISTIC_FLAG) ==  \
     PSA_ALG_ECDSA_BASE)
```

#### 4.3.2.18 PSA_ALG_IS_HASH

```
#define PSA_ALG_IS_HASH(
              alg )  (((alg) & PSA_ALG_CATEGORY_MASK) == PSA_ALG_CATEGORY_HASH)
```

Whether the specified algorithm is a hash algorithm.

**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.

### 4.3.2.19 PSA_ALG_IS_HKDF

```
#define PSA_ALG_IS_HKDF(
                alg ) (((alg) & ~PSA_ALG_HASH_MASK) == PSA_ALG_HKDF_BASE)
```

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.

**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.

### 4.3.2.20 PSA_ALG_IS_HMAC

```
#define PSA_ALG_IS_HMAC(
                alg )
```

**Value:**

```
(((alg) & (PSA_ALG_CATEGORY_MASK | PSA_ALG_MAC_SUBCATEGORY_MASK)) == \
     PSA_ALG_HMAC_BASE)
```

Whether the specified algorithm is an HMAC algorithm.

HMAC is a family of MAC algorithms that are based on a hash function.

**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.

#### 4.3.2.21 PSA_ALG_IS_KEY_AGREEMENT

```
#define PSA_ALG_IS_KEY_AGREEMENT(
              alg ) (((alg) & PSA_ALG_CATEGORY_MASK) == PSA_ALG_CATEGORY_KEY_AGREEMENT)
```

Whether the specified algorithm is a key agreement algorithm.

**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.

#### 4.3.2.22 PSA_ALG_IS_KEY_DERIVATION

```
#define PSA_ALG_IS_KEY_DERIVATION(
              alg ) (((alg) & PSA_ALG_CATEGORY_MASK) == PSA_ALG_CATEGORY_KEY_DERIVATION)
```

Whether the specified algorithm is a key derivation algorithm.

**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.

#### 4.3.2.23 PSA_ALG_IS_MAC

```
#define PSA_ALG_IS_MAC(
              alg ) (((alg) & PSA_ALG_CATEGORY_MASK) == PSA_ALG_CATEGORY_MAC)
```

Whether the specified algorithm is a MAC algorithm.

**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.

#### 4.3.2.24 PSA_ALG_IS_SIGN

```
#define PSA_ALG_IS_SIGN(
              alg ) (((alg) & PSA_ALG_CATEGORY_MASK) == PSA_ALG_CATEGORY_SIGN)
```

Whether the specified algorithm is a public-key signature algorithm.

**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.

#### 4.3.2.25 PSA_ALG_IS_STREAM_CIPHER

```
#define PSA_ALG_IS_STREAM_CIPHER(
              alg )
```

**Value:**

```
(((alg) & (PSA_ALG_CATEGORY_MASK | PSA_ALG_CIPHER_SUBCATEGORY_MASK)) == \
        PSA_ALG_STREAM_CIPHER_BASE)
```

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.

**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.

#### 4.3.2.26 PSA_ALG_RSA_OAEP

```
#define PSA_ALG_RSA_OAEP(
              hash_alg ) (PSA_ALG_RSA_OAEP_BASE | ((hash_alg) & PSA_ALG_HASH_MASK))
```

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.

**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.

#### 4.3.2.27 PSA_ALG_RSA_OAEP_GET_HASH

```
#define PSA_ALG_RSA_OAEP_GET_HASH(
              alg )
```

**Value:**

```
(PSA_ALG_IS_RSA_OAEP(alg) ?                              \
     ((alg) & PSA_ALG_HASH_MASK) | PSA_ALG_CATEGORY_HASH :     \
     0)
```

#### 4.3.2.28 PSA_ALG_RSA_PKCS1V15_CRYPT

```
#define PSA_ALG_RSA_PKCS1V15_CRYPT ((psa_algorithm_t)0x12020000)
```

RSA PKCS#1 v1.5 encryption.

#### 4.3.2.29 PSA_ALG_RSA_PKCS1V15_SIGN

```
#define PSA_ALG_RSA_PKCS1V15_SIGN(
              hash_alg ) (PSA_ALG_RSA_PKCS1V15_SIGN_BASE | ((hash_alg) & PSA_ALG_HASH_MASK))
```

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.

**Parameters**

| | |
|---|---|
| *hash_alg* | A hash algorithm (`PSA_ALG_XXX` value such that PSA_ALG_IS_HASH(`hash_alg`) is true). |

**Returns**

The corresponding RSA PKCS#1 v1.5 signature algorithm.
Unspecified if `alg` is not a supported hash algorithm.

#### 4.3.2.30 PSA_ALG_RSA_PKCS1V15_SIGN_RAW

```
#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.

#### 4.3.2.31 PSA_ALG_RSA_PSS

```
#define PSA_ALG_RSA_PSS(
             hash_alg ) (PSA_ALG_RSA_PSS_BASE | ((hash_alg) & PSA_ALG_HASH_MASK))
```

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.

**Parameters**

| | |
|---|---|
| *hash_alg* | A hash algorithm (`PSA_ALG_XXX` value such that PSA_ALG_IS_HASH(`hash_alg`) is true). |

**Returns**

The corresponding RSA PSS signature algorithm.
Unspecified if `alg` is not a supported hash algorithm.

#### 4.3.2.32 PSA_ALG_SHA3_224

```
#define PSA_ALG_SHA3_224 ((psa_algorithm_t)0x01000010)
```

SHA3-224

#### 4.3.2.33 PSA_ALG_SHA3_256

```
#define PSA_ALG_SHA3_256 ((psa_algorithm_t)0x01000011)
```

SHA3-256

#### 4.3.2.34 PSA_ALG_SHA3_384

```
#define PSA_ALG_SHA3_384 ((psa_algorithm_t)0x01000012)
```

SHA3-384

#### 4.3.2.35 PSA_ALG_SHA3_512

```
#define PSA_ALG_SHA3_512 ((psa_algorithm_t)0x01000013)
```

SHA3-512

#### 4.3.2.36 PSA_ALG_SHA_224

```
#define PSA_ALG_SHA_224 ((psa_algorithm_t)0x01000008)
```

SHA2-224

#### 4.3.2.37 PSA_ALG_SHA_256

```
#define PSA_ALG_SHA_256 ((psa_algorithm_t)0x01000009)
```

SHA2-256

#### 4.3.2.38 PSA_ALG_SHA_384

```
#define PSA_ALG_SHA_384 ((psa_algorithm_t)0x0100000a)
```

SHA2-384

#### 4.3.2.39 PSA_ALG_SHA_512

```
#define PSA_ALG_SHA_512 ((psa_algorithm_t)0x0100000b)
```

SHA2-512

#### 4.3.2.40 PSA_ALG_SHA_512_224

```
#define PSA_ALG_SHA_512_224 ((psa_algorithm_t)0x0100000c)
```

SHA2-512/224

#### 4.3.2.41 PSA_ALG_SHA_512_256

```
#define PSA_ALG_SHA_512_256 ((psa_algorithm_t)0x0100000d)
```

SHA2-512/256

#### 4.3.2.42 PSA_ALG_SIGN_GET_HASH

```
#define PSA_ALG_SIGN_GET_HASH(
                alg )
```

**Value:**

```
(PSA_ALG_IS_RSA_PSS(alg) || PSA_ALG_IS_RSA_PKCS1V15_SIGN(alg) ||    \
     PSA_ALG_IS_DSA(alg) || PSA_ALG_IS_ECDSA(alg) ?                    \
     ((alg) & PSA_ALG_HASH_MASK) == 0 ? /*"raw" algorithm*/ 0 :        \
     ((alg) & PSA_ALG_HASH_MASK) | PSA_ALG_CATEGORY_HASH :             \
     0)
```

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.

**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.

#### 4.3.2.43 PSA_BLOCK_CIPHER_BLOCK_SIZE

```
#define PSA_BLOCK_CIPHER_BLOCK_SIZE(
                type )
```

**Value:**

```
(                                                 \
        (type) == PSA_KEY_TYPE_AES ? 16 :             \
        (type) == PSA_KEY_TYPE_DES ? 8 :              \
        (type) == PSA_KEY_TYPE_CAMELLIA ? 16 :        \
        (type) == PSA_KEY_TYPE_ARC4 ? 1 :             \
        0)
```

The block size of a block cipher.

**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.

**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.
This macro returns a compile-time constant if its argument is one.

**Warning**

This macro may evaluate its argument multiple times.

#### 4.3.2.44 PSA_KEY_TYPE_AES

```
#define PSA_KEY_TYPE_AES ((psa_key_type_t)0x04000001)
```

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).

#### 4.3.2.45 PSA_KEY_TYPE_ARC4

```
#define PSA_KEY_TYPE_ARC4 ((psa_key_type_t)0x04000004)
```

Key for the RC4 stream cipher.

Note that RC4 is weak and deprecated and should only be used in legacy protocols.

#### 4.3.2.46 PSA_KEY_TYPE_CAMELLIA

```
#define PSA_KEY_TYPE_CAMELLIA ((psa_key_type_t)0x04000003)
```

Key for an cipher, AEAD or MAC algorithm based on the Camellia block cipher.

#### 4.3.2.47 PSA_KEY_TYPE_DERIVE

```
#define PSA_KEY_TYPE_DERIVE ((psa_key_type_t)0x02000101)
```

A secret for key derivation.

The key policy determines which key derivation algorithm the key can be used for.

#### 4.3.2.48 PSA_KEY_TYPE_DES

```
#define PSA_KEY_TYPE_DES ((psa_key_type_t)0x04000002)
```

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.

#### 4.3.2.49 PSA_KEY_TYPE_DSA_KEYPAIR

```
#define PSA_KEY_TYPE_DSA_KEYPAIR ((psa_key_type_t)0x07020000)
```

DSA key pair (private and public key).

#### 4.3.2.50 PSA_KEY_TYPE_DSA_PUBLIC_KEY

```
#define PSA_KEY_TYPE_DSA_PUBLIC_KEY ((psa_key_type_t)0x06020000)
```

DSA public key.

#### 4.3.2.51 PSA_KEY_TYPE_ECC_KEYPAIR

```
#define PSA_KEY_TYPE_ECC_KEYPAIR(
              curve ) (PSA_KEY_TYPE_ECC_KEYPAIR_BASE | (curve))
```

Elliptic curve key pair.

#### 4.3.2.52 PSA_KEY_TYPE_ECC_PUBLIC_KEY

```
#define PSA_KEY_TYPE_ECC_PUBLIC_KEY(
              curve ) (PSA_KEY_TYPE_ECC_PUBLIC_KEY_BASE | (curve))
```

Elliptic curve public key.

#### 4.3.2.53 PSA_KEY_TYPE_GET_CURVE

```
#define PSA_KEY_TYPE_GET_CURVE(
              type )
```

**Value:**

```
((psa_ecc_curve_t) (PSA_KEY_TYPE_IS_ECC(type) ?              \
                        ((type) & PSA_KEY_TYPE_ECC_CURVE_MASK) : \
                        0))
```

Extract the curve from an elliptic curve key type.

#### 4.3.2.54 PSA_KEY_TYPE_HMAC

```
#define PSA_KEY_TYPE_HMAC ((psa_key_type_t)0x02000001)
```

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.

#### 4.3.2.55 PSA_KEY_TYPE_IS_ASYMMETRIC

```
#define PSA_KEY_TYPE_IS_ASYMMETRIC(
              type ) (((type) & PSA_KEY_TYPE_CATEGORY_MASK) == PSA_KEY_TYPE_CATEGORY_ASYMMETR↩
IC)
```

Whether a key type is asymmetric: either a key pair or a public key.

#### 4.3.2.56 PSA_KEY_TYPE_IS_ECC

```
#define PSA_KEY_TYPE_IS_ECC(
              type )
```

**Value:**

```
((PSA_KEY_TYPE_PUBLIC_KEY_OF_KEYPAIR(type) &                          \
     ~PSA_KEY_TYPE_ECC_CURVE_MASK) == PSA_KEY_TYPE_ECC_PUBLIC_KEY_BASE)
```

Whether a key type is an elliptic curve key (pair or public-only).

#### 4.3.2.57 PSA_KEY_TYPE_IS_ECC_KEYPAIR

```
#define PSA_KEY_TYPE_IS_ECC_KEYPAIR(
              type )
```

**Value:**

```
(((type) & ~PSA_KEY_TYPE_ECC_CURVE_MASK) ==                           \
    PSA_KEY_TYPE_ECC_KEYPAIR_BASE)
```

#### 4.3.2.58 PSA_KEY_TYPE_IS_ECC_PUBLIC_KEY

```
#define PSA_KEY_TYPE_IS_ECC_PUBLIC_KEY(
              type )
```

**Value:**

```
(((type) & ~PSA_KEY_TYPE_ECC_CURVE_MASK) ==                           \
    PSA_KEY_TYPE_ECC_PUBLIC_KEY_BASE)
```

#### 4.3.2.59 PSA_KEY_TYPE_IS_KEYPAIR

```
#define PSA_KEY_TYPE_IS_KEYPAIR(
              type )
```

**Value:**

```
(((type) & (PSA_KEY_TYPE_CATEGORY_MASK | PSA_KEY_TYPE_PAIR_FLAG)) == \
     (PSA_KEY_TYPE_CATEGORY_ASYMMETRIC | PSA_KEY_TYPE_PAIR_FLAG))
```

Whether a key type is a key pair containing a private part and a public part.

#### 4.3.2.60 PSA_KEY_TYPE_IS_PUBLIC_KEY

```
#define PSA_KEY_TYPE_IS_PUBLIC_KEY(
              type )
```

**Value:**

```
(((type) & (PSA_KEY_TYPE_CATEGORY_MASK | PSA_KEY_TYPE_PAIR_FLAG)) == \
      PSA_KEY_TYPE_CATEGORY_ASYMMETRIC)
```

Whether a key type is the public part of a key pair.

#### 4.3.2.61 PSA_KEY_TYPE_IS_RSA

```
#define PSA_KEY_TYPE_IS_RSA(
              type ) (PSA_KEY_TYPE_PUBLIC_KEY_OF_KEYPAIR(type) == PSA_KEY_TYPE_RSA_PUBLIC_KEY)
```

Whether a key type is an RSA key (pair or public-only).

#### 4.3.2.62 PSA_KEY_TYPE_IS_VENDOR_DEFINED

```
#define PSA_KEY_TYPE_IS_VENDOR_DEFINED(
              type ) (((type) & PSA_KEY_TYPE_VENDOR_FLAG) != 0)
```

Whether a key type is vendor-defined.

#### 4.3.2.63 PSA_KEY_TYPE_KEYPAIR_OF_PUBLIC_KEY

```
#define PSA_KEY_TYPE_KEYPAIR_OF_PUBLIC_KEY(
              type ) ((type) | PSA_KEY_TYPE_PAIR_FLAG)
```

The key pair type corresponding to a public key type.

#### 4.3.2.64 PSA_KEY_TYPE_NONE

```
#define PSA_KEY_TYPE_NONE ((psa_key_type_t)0x00000000)
```

An invalid key type value.

Zero is not the encoding of any key type.

#### 4.3.2.65 PSA_KEY_TYPE_PUBLIC_KEY_OF_KEYPAIR

```
#define PSA_KEY_TYPE_PUBLIC_KEY_OF_KEYPAIR(
              type ) ((type) & ~PSA_KEY_TYPE_PAIR_FLAG)
```

The public key type corresponding to a key pair type.

#### 4.3.2.66 PSA_KEY_TYPE_RAW_DATA

```
#define PSA_KEY_TYPE_RAW_DATA ((psa_key_type_t)0x02000000)
```

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.

#### 4.3.2.67 PSA_KEY_TYPE_RSA_KEYPAIR

```
#define PSA_KEY_TYPE_RSA_KEYPAIR ((psa_key_type_t)0x07010000)
```

RSA key pair (private and public key).

#### 4.3.2.68 PSA_KEY_TYPE_RSA_PUBLIC_KEY

```
#define PSA_KEY_TYPE_RSA_PUBLIC_KEY ((psa_key_type_t)0x06010000)
```

RSA public key.

#### 4.3.2.69 PSA_KEY_TYPE_VENDOR_FLAG

```
#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.

## 4.3.3 Typedef Documentation

### 4.3.3.1 psa_algorithm_t

```
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.

### 4.3.3.2 psa_ecc_curve_t

```
typedef uint16_t psa_ecc_curve_t
```

The type of PSA elliptic curve identifiers.

## 4.4 Key management

### Functions

- psa_status_t psa_import_key (psa_key_slot_t key, psa_key_type_t type, const uint8_t ∗data, size_t data_↩ length)

  *Import a key in binary format.*
- psa_status_t psa_destroy_key (psa_key_slot_t key)

  *Destroy a key and restore the slot to its default state.*
- psa_status_t psa_get_key_information (psa_key_slot_t key, psa_key_type_t ∗type, size_t ∗bits)

  *Get basic metadata about a key.*
- psa_status_t psa_export_key (psa_key_slot_t key, uint8_t ∗data, size_t data_size, size_t ∗data_length)

  *Export a key in binary format.*
- psa_status_t psa_export_public_key (psa_key_slot_t key, uint8_t ∗data, size_t data_size, size_t ∗data_↩ length)

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

### 4.4.1 Detailed Description

### 4.4.2 Function Documentation

#### 4.4.2.1 psa_destroy_key()

```
psa_status_t psa_destroy_key (
            psa_key_slot_t key )
```

Destroy a key and restore the slot to its default state.

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. It returns the specified slot to its default state.

**Parameters**

| | |
|---|---|
| *key* | The key slot to erase. |

**Return values**

| | |
|---|---|
| *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_ARGUMENT* | The specified slot number does not designate a valid slot. |
| *PSA_ERROR_COMMUNICATION_FAILURE* | There was an failure in communication with the cryptoprocessor. The key material may still be present in the cryptoprocessor. |

**Return values**

| | |
|---|---|
| *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. |

### 4.4.2.2 psa_export_key()

```
psa_status_t psa_export_key (
             psa_key_slot_t key,
             uint8_t * data,
             size_t data_size,
             size_t * data_length )
```

Export a key in binary format.

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

If a key is created with psa_import_key() and then exported with this function, it is not guaranteed that the resulting data is identical: the implementation may choose a different representation of the same key if the format permits it.

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 representation defined by PKCS#1 (RFC 8017) as RSAPrivateKey.
- For RSA public keys (PSA_KEY_TYPE_RSA_PUBLIC_KEY), the format is the DER representation defined by RFC 5280 as SubjectPublicKeyInfo.

**Parameters**

| | | |
|---|---|---|
| | *key* | Slot whose content is to be exported. This must be an occupied key slot. |
| `out` | *data* | Buffer where the key data is to be written. |
| | *data_size* | Size of the `data` buffer in bytes. |
| `out` | *data_length* | On success, the number of bytes that make up the key data. |

**Return values**

| | |
|---|---|
| *PSA_SUCCESS* | |
| *PSA_ERROR_EMPTY_SLOT* | |

**Return values**

| | |
|---|---|
| *PSA_ERROR_NOT_PERMITTED* | |
| *PSA_ERROR_NOT_SUPPORTED* | |
| *PSA_ERROR_COMMUNICATION_FAILURE* | |
| *PSA_ERROR_HARDWARE_FAILURE* | |
| *PSA_ERROR_TAMPERING_DETECTED* | |

#### 4.4.2.3 psa_export_public_key()

```
psa_status_t psa_export_public_key (
             psa_key_slot_t key,
             uint8_t * data,
             size_t data_size,
             size_t * data_length )
```

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.

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

- For RSA keys (PSA_KEY_TYPE_RSA_KEYPAIR or PSA_KEY_TYPE_RSA_PUBLIC_KEY), the format is the DER representation of the public key defined by RFC 5280 as SubjectPublicKeyInfo.

**Parameters**

| | | |
|---|---|---|
| | *key* | Slot whose content is to be exported. This must be an occupied key slot. |
| `out` | *data* | Buffer where the key data is to be written. |
| | *data_size* | Size of the `data` buffer in bytes. |
| `out` | *data_length* | On success, the number of bytes that make up the key data. |

**Return values**

| | |
|---|---|
| *PSA_SUCCESS* | |
| *PSA_ERROR_EMPTY_SLOT* | |
| *PSA_ERROR_INVALID_ARGUMENT* | |
| *PSA_ERROR_COMMUNICATION_FAILURE* | |
| *PSA_ERROR_HARDWARE_FAILURE* | |
| *PSA_ERROR_TAMPERING_DETECTED* | |

#### 4.4.2.4 psa_get_key_information()

```
psa_status_t psa_get_key_information (
              psa_key_slot_t key,
              psa_key_type_t * type,
              size_t * bits )
```

Get basic metadata about a key.

**Parameters**

| | | |
|---|---|---|
| | *key* | Slot whose content is queried. This must be an occupied key slot. |
| out | *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. |
| out | *bits* | On success, the key size in bits. This may be a null pointer, in which case the key size is not written. |

**Return values**

| | |
|---|---|
| *PSA_SUCCESS* | |
| *PSA_ERROR_EMPTY_SLOT* | |
| *PSA_ERROR_COMMUNICATION_FAILURE* | |
| *PSA_ERROR_HARDWARE_FAILURE* | |
| *PSA_ERROR_TAMPERING_DETECTED* | |

### 4.4.2.5 psa_import_key()

```
psa_status_t psa_import_key (
              psa_key_slot_t key,
              psa_key_type_t type,
              const uint8_t * data,
              size_t data_length )
```

Import a key in binary format.

This function supports any output from psa_export_key(). Refer to the documentation of psa_export_key() for the format for each key type.

**Parameters**

| | | |
|---|---|---|
| | *key* | Slot where the key will be stored. This must be a valid slot for a key of the chosen type. It must be unoccupied. |
| | *type* | Key type (a `PSA_KEY_TYPE_XXX` value). |
| in | *data* | Buffer containing the key data. |
| | *data_length* | Size of the `data` buffer in bytes. |

**Return values**

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

**Return values**

| | |
|---|---|
| *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_HARDWARE_FAILURE* | |
| *PSA_ERROR_TAMPERING_DETECTED* | |

## 4.5 Key policies

### Macros

- #define PSA_KEY_USAGE_EXPORT ((psa_key_usage_t)0x00000001)
- #define PSA_KEY_USAGE_ENCRYPT ((psa_key_usage_t)0x00000100)
- #define PSA_KEY_USAGE_DECRYPT ((psa_key_usage_t)0x00000200)
- #define PSA_KEY_USAGE_SIGN ((psa_key_usage_t)0x00000400)
- #define PSA_KEY_USAGE_VERIFY ((psa_key_usage_t)0x00000800)
- #define PSA_KEY_USAGE_DERIVE ((psa_key_usage_t)0x00001000)

### Typedefs

- typedef uint32_t psa_key_usage_t

  *Encoding of permitted usage on a key.*
- typedef struct psa_key_policy_s psa_key_policy_t

### Functions

- void psa_key_policy_init (psa_key_policy_t ∗policy)

  *Initialize a key policy structure to a default that forbids all usage of the key.*
- void psa_key_policy_set_usage (psa_key_policy_t ∗policy, psa_key_usage_t usage, psa_algorithm_t alg)

  *Set the standard fields of a policy structure.*
- psa_key_usage_t psa_key_policy_get_usage (const psa_key_policy_t ∗policy)

  *Retrieve the usage field of a policy structure.*
- psa_algorithm_t psa_key_policy_get_algorithm (const psa_key_policy_t ∗policy)

  *Retrieve the algorithm field of a policy structure.*
- psa_status_t psa_set_key_policy (psa_key_slot_t key, const psa_key_policy_t ∗policy)

  *Set the usage policy on a key slot.*
- psa_status_t psa_get_key_policy (psa_key_slot_t key, psa_key_policy_t ∗policy)

  *Get the usage policy for a key slot.*

### 4.5.1 Detailed Description

### 4.5.2 Macro Definition Documentation

#### 4.5.2.1 PSA_KEY_USAGE_DECRYPT

```
#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.

#### 4.5.2.2 PSA_KEY_USAGE_DERIVE

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

Whether the key may be used to derive other keys.

#### 4.5.2.3 PSA_KEY_USAGE_ENCRYPT

```
#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.

#### 4.5.2.4 PSA_KEY_USAGE_EXPORT

```
#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.

#### 4.5.2.5 PSA_KEY_USAGE_SIGN

```
#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.

#### 4.5.2.6 PSA_KEY_USAGE_VERIFY

```
#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.

### 4.5.3 Typedef Documentation

#### 4.5.3.1 psa_key_policy_t

```
typedef struct psa_key_policy_s psa_key_policy_t
```

The type of the key policy data structure.

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.

### 4.5.4 Function Documentation

#### 4.5.4.1 psa_get_key_policy()

```
psa_status_t psa_get_key_policy (
             psa_key_slot_t key,
             psa_key_policy_t * policy )
```

Get the usage policy for a key slot.

**Parameters**

| | | |
|---|---|---|
| | *key* | The key slot whose policy is being queried. |
| out | *policy* | On success, the key's policy. |

**Return values**

| | |
|---|---|
| *PSA_SUCCESS* | |
| *PSA_ERROR_COMMUNICATION_FAILURE* | |
| *PSA_ERROR_HARDWARE_FAILURE* | |
| *PSA_ERROR_TAMPERING_DETECTED* | |

#### 4.5.4.2 psa_key_policy_get_algorithm()

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

Retrieve the algorithm field of a policy structure.

**Parameters**

| | | |
|---|---|---|
| in | *policy* | The policy object to query. |

**Returns**

The permitted algorithm for a key with this policy.

#### 4.5.4.3 psa_key_policy_get_usage()

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

Retrieve the usage field of a policy structure.

**Parameters**

| | | |
|---|---|---|
| in | *policy* | The policy object to query. |

**Returns**

The permitted uses for a key with this policy.

#### 4.5.4.4 psa_key_policy_init()

```
void psa_key_policy_init (
            psa_key_policy_t * policy )
```

Initialize a key policy structure to a default that forbids all usage of the key.

**Parameters**

| | | |
|---|---|---|
| out | *policy* | The policy object to initialize. |

#### 4.5.4.5 psa_key_policy_set_usage()

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

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().

**Parameters**

| | | |
|---|---|---|
| `out` | *policy* | The policy object to modify. |
| | *usage* | The permitted uses for the key. |
| | *alg* | The algorithm that the key may be used for. |

### 4.5.4.6 psa_set_key_policy()

```
psa_status_t psa_set_key_policy (
             psa_key_slot_t key,
             const psa_key_policy_t * policy )
```

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.

**Parameters**

| | | |
|---|---|---|
| | *key* | The key slot whose policy is to be changed. |
| `in` | *policy* | The policy object to query. |

**Return values**

| | |
|---|---|
| *PSA_SUCCESS* | |
| *PSA_ERROR_OCCUPIED_SLOT* | |
| *PSA_ERROR_NOT_SUPPORTED* | |
| *PSA_ERROR_INVALID_ARGUMENT* | |
| *PSA_ERROR_COMMUNICATION_FAILURE* | |
| *PSA_ERROR_HARDWARE_FAILURE* | |
| *PSA_ERROR_TAMPERING_DETECTED* | |

## 4.6 Key lifetime

### Macros

- #define PSA_KEY_LIFETIME_VOLATILE ((psa_key_lifetime_t)0x00000000)
- #define PSA_KEY_LIFETIME_PERSISTENT ((psa_key_lifetime_t)0x00000001)
- #define PSA_KEY_LIFETIME_WRITE_ONCE ((psa_key_lifetime_t)0x7fffffff)

### Typedefs

- typedef uint32_t psa_key_lifetime_t

### Functions

- psa_status_t psa_get_key_lifetime (psa_key_slot_t key, psa_key_lifetime_t ∗lifetime)

  *Retrieve the lifetime of a key slot.*
- psa_status_t psa_set_key_lifetime (psa_key_slot_t key, psa_key_lifetime_t lifetime)

  *Change the lifetime of a key slot.*

### 4.6.1 Detailed Description

### 4.6.2 Macro Definition Documentation

#### 4.6.2.1 PSA_KEY_LIFETIME_PERSISTENT

```
#define PSA_KEY_LIFETIME_PERSISTENT ((psa_key_lifetime_t)0x00000001)
```

A persistent key slot retains its content as long as it is not explicitly destroyed.

#### 4.6.2.2 PSA_KEY_LIFETIME_VOLATILE

```
#define PSA_KEY_LIFETIME_VOLATILE ((psa_key_lifetime_t)0x00000000)
```

A volatile key slot retains its content as long as the application is running. It is guaranteed to be erased on a power reset.

#### 4.6.2.3 PSA_KEY_LIFETIME_WRITE_ONCE

```
#define PSA_KEY_LIFETIME_WRITE_ONCE ((psa_key_lifetime_t)0x7fffffff)
```

A write-once key slot may not be modified once a key has been set. It will retain its content as long as the device remains operational.

### 4.6.3 Typedef Documentation

#### 4.6.3.1 psa_key_lifetime_t

```
typedef uint32_t psa_key_lifetime_t
```

Encoding of key lifetimes.

### 4.6.4 Function Documentation

#### 4.6.4.1 psa_get_key_lifetime()

```
psa_status_t psa_get_key_lifetime (
              psa_key_slot_t key,
              psa_key_lifetime_t * lifetime )
```

Retrieve the lifetime of a key slot.

The assignment of lifetimes to slots is implementation-dependent.

**Parameters**

| | | |
|---|---|---|
| | *key* | Slot to query. |
| out | *lifetime* | On success, the lifetime value. |

**Return values**

| | |
|---|---|
| *PSA_SUCCESS* | Success. |
| *PSA_ERROR_INVALID_ARGUMENT* | The key slot is invalid. |
| *PSA_ERROR_COMMUNICATION_FAILURE* | |
| *PSA_ERROR_HARDWARE_FAILURE* | |
| *PSA_ERROR_TAMPERING_DETECTED* | |

#### 4.6.4.2 psa_set_key_lifetime()

```
psa_status_t psa_set_key_lifetime (
              psa_key_slot_t key,
              psa_key_lifetime_t lifetime )
```

Change the lifetime of a key slot.

Whether the lifetime of a key slot can be changed at all, and if so whether the lifetime of an occupied key slot can be changed, is implementation-dependent.

#### Parameters

| | |
|---|---|
| *key* | Slot whose lifetime is to be changed. |
| *lifetime* | The lifetime value to set for the given key slot. |

#### Return values

| | |
|---|---|
| *PSA_SUCCESS* | Success. |
| *PSA_ERROR_INVALID_ARGUMENT* | The key slot is invalid, or the lifetime value is invalid. |
| *PSA_ERROR_NOT_SUPPORTED* | The implementation does not support the specified lifetime value, at least for the specified key slot. |
| *PSA_ERROR_OCCUPIED_SLOT* | The slot contains a key, and the implementation does not support changing the lifetime of an occupied slot. |
| *PSA_ERROR_COMMUNICATION_FAILURE* | |
| *PSA_ERROR_HARDWARE_FAILURE* | |
| *PSA_ERROR_TAMPERING_DETECTED* | |

## 4.7 Message digests

### Macros

- #define PSA_HASH_SIZE(alg)

### Typedefs

- typedef struct psa_hash_operation_s psa_hash_operation_t

### Functions

- psa_status_t psa_hash_setup (psa_hash_operation_t ∗operation, psa_algorithm_t alg)
- psa_status_t psa_hash_update (psa_hash_operation_t ∗operation, const uint8_t ∗input, size_t input_length)
- psa_status_t psa_hash_finish (psa_hash_operation_t ∗operation, uint8_t ∗hash, size_t hash_size, size_←
  t ∗hash_length)
- psa_status_t psa_hash_verify (psa_hash_operation_t ∗operation, const uint8_t ∗hash, size_t hash_length)
- psa_status_t psa_hash_abort (psa_hash_operation_t ∗operation)

### 4.7.1 Detailed Description

### 4.7.2 Macro Definition Documentation

#### 4.7.2.1 PSA_HASH_SIZE

```
#define PSA_HASH_SIZE(
               alg )
```

**Value:**

```
(                                                           \
        PSA_ALG_HMAC_HASH(alg) == PSA_ALG_MD2 ? 16 :            \
        PSA_ALG_HMAC_HASH(alg) == PSA_ALG_MD4 ? 16 :            \
        PSA_ALG_HMAC_HASH(alg) == PSA_ALG_MD5 ? 16 :            \
        PSA_ALG_HMAC_HASH(alg) == PSA_ALG_RIPEMD160 ? 20 :      \
        PSA_ALG_HMAC_HASH(alg) == PSA_ALG_SHA_1 ? 20 :          \
        PSA_ALG_HMAC_HASH(alg) == PSA_ALG_SHA_224 ? 28 :        \
        PSA_ALG_HMAC_HASH(alg) == PSA_ALG_SHA_256 ? 32 :        \
        PSA_ALG_HMAC_HASH(alg) == PSA_ALG_SHA_384 ? 48 :        \
        PSA_ALG_HMAC_HASH(alg) == PSA_ALG_SHA_512 ? 64 :        \
        PSA_ALG_HMAC_HASH(alg) == PSA_ALG_SHA_512_224 ? 28 :    \
        PSA_ALG_HMAC_HASH(alg) == PSA_ALG_SHA_512_256 ? 32 :    \
        PSA_ALG_HMAC_HASH(alg) == PSA_ALG_SHA3_224 ? 28 :       \
        PSA_ALG_HMAC_HASH(alg) == PSA_ALG_SHA3_256 ? 32 :       \
        PSA_ALG_HMAC_HASH(alg) == PSA_ALG_SHA3_384 ? 48 :       \
        PSA_ALG_HMAC_HASH(alg) == PSA_ALG_SHA3_512 ? 64 :       \
        0)
```

The size of the output of psa_hash_finish(), in bytes.

This is also the hash size that psa_hash_verify() expects.

**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.

## 4.7.3 Typedef Documentation

#### 4.7.3.1 psa_hash_operation_t

```
typedef struct psa_hash_operation_s psa_hash_operation_t
```

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

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.

## 4.7.4 Function Documentation

#### 4.7.4.1 psa_hash_abort()

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

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.

**Parameters**

| | | |
|---|---|---|
| `in,out` | *operation* | Initialized hash operation. |

**Return values**

| | |
|---|---|
| *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* | |

#### 4.7.4.2 psa_hash_finish()

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

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.

**Parameters**

| | | |
|---|---|---|
| `in,out` | *operation* | Active hash operation. |
| `out` | *hash* | Buffer where the hash is to be written. |
| | *hash_size* | Size of the `hash` buffer in bytes. |
| `out` | *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. |

**Return values**

| | |
|---|---|
| *PSA_SUCCESS* | Success. |
| *PSA_ERROR_BAD_STATE* | The operation state is not valid (not started, or already completed). |

**Return values**

| | |
|---|---|
| *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* | |

### 4.7.4.3 psa_hash_setup()

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

Start 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. Call psa_hash_setup() to specify the algorithm.
3. 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.
4. 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 with psa_hash_↩setup().

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().

**Parameters**

| | | |
|---|---|---|
| `out` | *operation* | The operation object to use. |
| | *alg* | The hash algorithm to compute (`PSA_ALG_XXX` value such that PSA_ALG_IS_HASH(`alg`) is true). |

**Return values**

| | |
|---|---|
| *PSA_SUCCESS* | Success. |
| *PSA_ERROR_NOT_SUPPORTED* | `alg` is not supported or is not a hash algorithm. |

**Return values**

| | |
|---|---|
| *PSA_ERROR_INSUFFICIENT_MEMORY* | |
| *PSA_ERROR_COMMUNICATION_FAILURE* | |
| *PSA_ERROR_HARDWARE_FAILURE* | |
| *PSA_ERROR_TAMPERING_DETECTED* | |

### 4.7.4.4 psa_hash_update()

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

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.

**Parameters**

| | | |
|---|---|---|
| `in,out` | *operation* | Active hash operation. |
| `in` | *input* | Buffer containing the message fragment to hash. |
| | *input_length* | Size of the `input` buffer in bytes. |

**Return values**

| | |
|---|---|
| *PSA_SUCCESS* | Success. |
| *PSA_ERROR_BAD_STATE* | The operation state is not valid (not started, or already completed). |
| *PSA_ERROR_INSUFFICIENT_MEMORY* | |
| *PSA_ERROR_COMMUNICATION_FAILURE* | |
| *PSA_ERROR_HARDWARE_FAILURE* | |
| *PSA_ERROR_TAMPERING_DETECTED* | |

### 4.7.4.5 psa_hash_verify()

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

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.

**Parameters**

| | | |
|---|---|---|
| `in,out` | *operation* | Active hash operation. |
| `in` | *hash* | Buffer containing the expected hash value. |
| | *hash_length* | Size of the `hash` buffer in bytes. |

**Return values**

| | |
|---|---|
| *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 started, or already completed). |
| *PSA_ERROR_INSUFFICIENT_MEMORY* | |
| *PSA_ERROR_COMMUNICATION_FAILURE* | |
| *PSA_ERROR_HARDWARE_FAILURE* | |
| *PSA_ERROR_TAMPERING_DETECTED* | |

## 4.8 Message authentication codes

### Typedefs

- typedef struct psa_mac_operation_s psa_mac_operation_t

### Functions

- psa_status_t psa_mac_sign_setup (psa_mac_operation_t ∗operation, psa_key_slot_t key, psa_algorithm_t alg)
- psa_status_t psa_mac_verify_setup (psa_mac_operation_t ∗operation, psa_key_slot_t key, psa_algorithm↩_t alg)
- psa_status_t psa_mac_update (psa_mac_operation_t ∗operation, const uint8_t ∗input, size_t input_length)
- psa_status_t psa_mac_sign_finish (psa_mac_operation_t ∗operation, uint8_t ∗mac, size_t mac_size, size_t ∗mac_length)
- psa_status_t psa_mac_verify_finish (psa_mac_operation_t ∗operation, const uint8_t ∗mac, size_t mac_↩length)
- psa_status_t psa_mac_abort (psa_mac_operation_t ∗operation)

### 4.8.1 Detailed Description

### 4.8.2 Typedef Documentation

#### 4.8.2.1 psa_mac_operation_t

```
typedef struct psa_mac_operation_s psa_mac_operation_t
```

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

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.

### 4.8.3 Function Documentation

#### 4.8.3.1 psa_mac_abort()

```
psa_status_t psa_mac_abort (
            psa_mac_operation_t ∗ operation )
```

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.

**Parameters**

| | | |
|---|---|---|
| in,out | *operation* | Initialized MAC operation. |

**Return values**

| | |
|---|---|
| *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* | |

#### 4.8.3.2 psa_mac_sign_finish()

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

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.

**Parameters**

| | | |
|---|---|---|
| in,out | *operation* | Active MAC operation. |
| out | *mac* | Buffer where the MAC value is to be written. |
| | *mac_size* | Size of the mac buffer in bytes. |
| out | *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. |

**Return values**

| | |
|---|---|
| *PSA_SUCCESS* | Success. |

**Return values**

| | |
|---|---|
| *PSA_ERROR_BAD_STATE* | The operation state is not valid (not started, 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* | |

### 4.8.3.3 psa_mac_sign_setup()

```
psa_status_t psa_mac_sign_setup (
             psa_mac_operation_t * operation,
             psa_key_slot_t key,
             psa_algorithm_t alg )
```

Start 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. Call psa_mac_sign_setup() to specify the algorithm and key. The key remains associated with the operation even if the content of the key slot changes.
3. 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.
4. 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 with psa_mac_sign↵_setup().

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().

**Parameters**

| | | |
|---|---|---|
| `out` | *operation* | The operation object to use. |
| | *key* | Slot containing 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). |

**Return values**

| | |
|---|---|
| *PSA_SUCCESS* | Success. |
| *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* | |

#### 4.8.3.4 psa_mac_update()

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

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.

**Parameters**

| | | |
|---|---|---|
| in,out | *operation* | Active MAC operation. |
| in | *input* | Buffer containing the message fragment to add to the MAC calculation. |
| | *input_length* | Size of the `input` buffer in bytes. |

**Return values**

| | |
|---|---|
| *PSA_SUCCESS* | Success. |
| *PSA_ERROR_BAD_STATE* | The operation state is not valid (not started, or already completed). |
| *PSA_ERROR_INSUFFICIENT_MEMORY* | |
| *PSA_ERROR_COMMUNICATION_FAILURE* | |
| *PSA_ERROR_HARDWARE_FAILURE* | |
| *PSA_ERROR_TAMPERING_DETECTED* | |

#### 4.8.3.5 psa_mac_verify_finish()

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

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.

**Parameters**

| | | |
|---|---|---|
| `in,out` | *operation* | Active MAC operation. |
| `in` | *mac* | Buffer containing the expected MAC value. |
| | *mac_length* | Size of the `mac` buffer in bytes. |

**Return values**

| | |
|---|---|
| *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 started, or already completed). |
| *PSA_ERROR_INSUFFICIENT_MEMORY* | |
| *PSA_ERROR_COMMUNICATION_FAILURE* | |
| *PSA_ERROR_HARDWARE_FAILURE* | |
| *PSA_ERROR_TAMPERING_DETECTED* | |

#### 4.8.3.6 psa_mac_verify_setup()

```
psa_status_t psa_mac_verify_setup (
            psa_mac_operation_t * operation,
            psa_key_slot_t key,
            psa_algorithm_t alg )
```

Start 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. Call psa_mac_verify_setup() to specify the algorithm and key. The key remains associated with the operation even if the content of the key slot changes.
3. 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.
4. 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 with psa_mac_verify↩_setup().

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().

**Parameters**

| | | |
|---|---|---|
| `out` | *operation* | The operation object to use. |
| | *key* | Slot containing 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). |

**Return values**

| | |
|---|---|
| *PSA_SUCCESS* | Success. |
| *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* | |

## 4.9 Symmetric ciphers

### Typedefs

- typedef struct psa_cipher_operation_s psa_cipher_operation_t

### Functions

- psa_status_t psa_cipher_encrypt_setup (psa_cipher_operation_t ∗operation, psa_key_slot_t key, psa_←algorithm_t alg)
- psa_status_t psa_cipher_decrypt_setup (psa_cipher_operation_t ∗operation, psa_key_slot_t key, psa_←algorithm_t alg)
- psa_status_t psa_cipher_generate_iv (psa_cipher_operation_t ∗operation, unsigned char ∗iv, size_t iv_size, size_t ∗iv_length)
- psa_status_t psa_cipher_set_iv (psa_cipher_operation_t ∗operation, const unsigned char ∗iv, size_t iv_←length)
- 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)
- psa_status_t psa_cipher_finish (psa_cipher_operation_t ∗operation, uint8_t ∗output, size_t output_size, size_t ∗output_length)
- psa_status_t psa_cipher_abort (psa_cipher_operation_t ∗operation)

### 4.9.1 Detailed Description

### 4.9.2 Typedef Documentation

#### 4.9.2.1 psa_cipher_operation_t

```
typedef struct psa_cipher_operation_s psa_cipher_operation_t
```

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

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.

### 4.9.3 Function Documentation

#### 4.9.3.1 psa_cipher_abort()

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

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.

**Parameters**

| | | |
|---|---|---|
| `in,out` | *operation* | Initialized cipher operation. |

**Return values**

| | |
|---|---|
| *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* | |

#### 4.9.3.2 psa_cipher_decrypt_setup()

```
psa_status_t psa_cipher_decrypt_setup (
             psa_cipher_operation_t * operation,
             psa_key_slot_t key,
             psa_algorithm_t alg )
```

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. Call psa_cipher_decrypt_setup() to specify the algorithm and key. The key remains associated with the operation even if the content of the key slot changes.

3. Call psa_cipher_update() 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.
4. Call psa_cipher_update() zero, one or more times, passing a fragment of the message each time.
5. Call psa_cipher_finish().

The application may call psa_cipher_abort() at any time after the operation has been initialized with psa_cipher_↩ decrypt_setup().

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 psa_cipher_update().
- A call to psa_cipher_finish() or psa_cipher_abort().

**Parameters**

| | | |
|---|---|---|
| `out` | *operation* | The operation object to use. |
| | *key* | Slot containing the key to use for the operation. |
| | *alg* | The cipher algorithm to compute (`PSA_ALG_XXX` value such that PSA_ALG_IS_CIPHER(`alg`) is true). |

**Return values**

| | |
|---|---|
| *PSA_SUCCESS* | Success. |
| *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* | |

#### 4.9.3.3 psa_cipher_encrypt_setup()

```
psa_status_t psa_cipher_encrypt_setup (
            psa_cipher_operation_t * operation,
            psa_key_slot_t key,
            psa_algorithm_t alg )
```

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. Call psa_cipher_encrypt_setup() to specify the algorithm and key. The key remains associated with the operation even if the content of the key slot changes.
3. 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.
4. Call psa_cipher_update() zero, one or more times, passing a fragment of the message each time.
5. Call psa_cipher_finish().

The application may call psa_cipher_abort() at any time after the operation has been initialized with psa_cipher_↩ encrypt_setup().

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 psa_cipher_generate_iv(), psa_cipher_set_iv() or psa_cipher_update().
- A call to psa_cipher_finish() or psa_cipher_abort().

#### Parameters

| | | |
|---|---|---|
| `out` | *operation* | The operation object to use. |
| | *key* | Slot containing the key to use for the operation. |
| | *alg* | The cipher algorithm to compute (`PSA_ALG_XXX` value such that PSA_ALG_IS_CIPHER(`alg`) is true). |

#### Return values

| | |
|---|---|
| *PSA_SUCCESS* | Success. |
| *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* | |

### 4.9.3.4 psa_cipher_finish()

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

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.

**Parameters**

| | | |
|---|---|---|
| `in,out` | *operation* | Active cipher operation. |
| `out` | *output* | Buffer where the output is to be written. |
| | *output_size* | Size of the `output` buffer in bytes. |
| `out` | *output_length* | On success, the number of bytes that make up the returned output. |

**Return values**

| | |
|---|---|
| *PSA_SUCCESS* | Success. |
| *PSA_ERROR_BAD_STATE* | The operation state is not valid (not started, 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* | |

#### 4.9.3.5 psa_cipher_generate_iv()

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

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.

**Parameters**

| | | |
|---|---|---|
| `in,out` | *operation* | Active cipher operation. |
| `out` | *iv* | Buffer where the generated IV is to be written. |
| | *iv_size* | Size of the `iv` buffer in bytes. |
| `out` | *iv_length* | On success, the number of bytes of the generated IV. |

**Return values**

| | |
|---:|---|
| *PSA_SUCCESS* | Success. |
| *PSA_ERROR_BAD_STATE* | The operation state is not valid (not started, 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* | |

#### 4.9.3.6 psa_cipher_set_iv()

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

Set the IV for a symmetric encryption or decryption operation.

This function sets the random 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.

**Parameters**

| | | |
|---|---|---|
| `in,out` | *operation* | Active cipher operation. |
| `in` | *iv* | Buffer containing the IV to use. |
| | *iv_length* | Size of the IV in bytes. |

**Return values**

| | |
|---:|---|
| *PSA_SUCCESS* | Success. |
| *PSA_ERROR_BAD_STATE* | The operation state is not valid (not started, 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* | |

#### 4.9.3.7 psa_cipher_update()

```
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 )
```

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.

**Parameters**

| | | |
|---|---|---|
| `in,out` | *operation* | Active cipher operation. |
| `in` | *input* | Buffer containing the message fragment to encrypt or decrypt. |
| | *input_length* | Size of the `input` buffer in bytes. |
| `out` | *output* | Buffer where the output is to be written. |
| | *output_size* | Size of the `output` buffer in bytes. |
| `out` | *output_length* | On success, the number of bytes that make up the returned output. |

**Return values**

| | |
|---|---|
| *PSA_SUCCESS* | Success. |
| *PSA_ERROR_BAD_STATE* | The operation state is not valid (not started, 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* | |

## 4.10 Authenticated encryption with associated data (AEAD)

### Macros

- #define PSA_AEAD_TAG_SIZE(alg)

### Functions

- psa_status_t psa_aead_encrypt (psa_key_slot_t key, 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)
- psa_status_t psa_aead_decrypt (psa_key_slot_t key, 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)

### 4.10.1 Detailed Description

### 4.10.2 Macro Definition Documentation

#### 4.10.2.1 PSA_AEAD_TAG_SIZE

```
#define PSA_AEAD_TAG_SIZE(
                alg )
```

**Value:**

```
((alg) == PSA_ALG_GCM ? 16 :            \
     (alg) == PSA_ALG_CCM ? 16 :            \
     0)
```

The tag size for an AEAD algorithm, in bytes.

**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.

### 4.10.3 Function Documentation

#### 4.10.3.1 psa_aead_decrypt()

```
psa_status_t psa_aead_decrypt (
             psa_key_slot_t key,
             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 )
```

Process an authenticated decryption operation.

**Parameters**

| | | |
|---|---|---|
| | *key* | Slot containing the key to use. |
| | *alg* | The AEAD algorithm to compute (`PSA_ALG_XXX` value such that PSA_ALG_IS_AEAD(`alg`) is true). |
| `in` | *nonce* | Nonce or IV to use. |
| | *nonce_length* | Size of the `nonce` buffer in bytes. |
| `in` | *additional_data* | Additional data that has been authenticated but not encrypted. |
| | *additional_data_length* | Size of `additional_data` in bytes. |
| `in` | *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. |
| `out` | *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`). |
| `out` | *plaintext_length* | On success, the size of the output in the **plaintext** buffer. |

**Return values**

| | |
|---|---|
| *PSA_SUCCESS* | Success. |
| *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* | |

#### 4.10.3.2 psa_aead_encrypt()

```
psa_status_t psa_aead_encrypt (
             psa_key_slot_t key,
             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 )
```

Process an authenticated encryption operation.

**Parameters**

| | | |
|---|---|---|
| | *key* | Slot containing the key to use. |
| | *alg* | The AEAD algorithm to compute (`PSA_ALG_XXX` value such that PSA_ALG_IS_AEAD(`alg`) is true). |
| `in` | *nonce* | Nonce or IV to use. |
| | *nonce_length* | Size of the `nonce` buffer in bytes. |
| `in` | *additional_data* | Additional data that will be authenticated but not encrypted. |
| | *additional_data_length* | Size of `additional_data` in bytes. |
| `in` | *plaintext* | Data that will be authenticated and encrypted. |
| | *plaintext_length* | Size of `plaintext` in bytes. |
| `out` | *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`). |
| `out` | *ciphertext_length* | On success, the size of the output in the **ciphertext** buffer. |

**Return values**

| | |
|---|---|
| *PSA_SUCCESS* | Success. |
| *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* | |

## 4.11 Asymmetric cryptography

### Macros

- #define PSA_ECDSA_SIGNATURE_SIZE(curve_bits) (PSA_BITS_TO_BYTES(curve_bits) ∗ 2)

    *ECDSA signature size for a given curve bit size.*
- #define **PSA_RSA_MINIMUM_PADDING_SIZE**(alg)

### Functions

- psa_status_t psa_asymmetric_sign (psa_key_slot_t key, psa_algorithm_t alg, const uint8_t ∗hash, size_←
t hash_length, uint8_t ∗signature, size_t signature_size, size_t ∗signature_length)

    *Sign a hash or short message with a private key.*
- psa_status_t psa_asymmetric_verify (psa_key_slot_t key, psa_algorithm_t alg, const uint8_t ∗hash, size_t hash_length, const uint8_t ∗signature, size_t signature_length)

    *Verify the signature a hash or short message using a public key.*
- psa_status_t psa_asymmetric_encrypt (psa_key_slot_t key, 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)

    *Encrypt a short message with a public key.*
- psa_status_t psa_asymmetric_decrypt (psa_key_slot_t key, 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)

    *Decrypt a short message with a private key.*

### 4.11.1 Detailed Description

### 4.11.2 Macro Definition Documentation

#### 4.11.2.1 PSA_ECDSA_SIGNATURE_SIZE

```
#define PSA_ECDSA_SIGNATURE_SIZE(
              curve_bits ) (PSA_BITS_TO_BYTES(curve_bits) * 2)
```

ECDSA signature size for a given curve bit size.

**Parameters**

| | |
|---|---|
| *curve_bits* | Curve size in bits. |

**Returns**

Signature size in bytes.

**Note**

This macro returns a compile-time constant if its argument is one.

#### 4.11.2.2 PSA_RSA_MINIMUM_PADDING_SIZE

```
#define PSA_RSA_MINIMUM_PADDING_SIZE(
              alg )
```

**Value:**

```
(PSA_ALG_IS_RSA_OAEP(alg) ?                                          \
     2 * PSA_HASH_FINAL_SIZE(PSA_ALG_RSA_OAEP_GET_HASH(alg)) + 1 :      \
     11 /*PKCS#1v1.5*/)
```

### 4.11.3 Function Documentation

#### 4.11.3.1 psa_asymmetric_decrypt()

```
psa_status_t psa_asymmetric_decrypt (
             psa_key_slot_t key,
             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 )
```

Decrypt a short message with a private key.

**Parameters**

| | | |
|---|---|---|
| | *key* | Key slot containing an asymmetric key pair. |
| | *alg* | An asymmetric encryption algorithm that is compatible with the type of `key`. |
| in | *input* | The message to decrypt. |
| | *input_length* | Size of the `input` buffer in bytes. |
| in | *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`. |

- For PSA_ALG_RSA_PKCS1V15_CRYPT, no salt is supported.

**Parameters**

| | | |
|---|---|---|
| | *salt_length* | Size of the `salt` buffer in bytes. If `salt` is `NULL`, pass 0. |
| `out` | *output* | Buffer where the decrypted message is to be written. |
| | *output_size* | Size of the `output` buffer in bytes. |
| `out` | *output_length* | On success, the number of bytes that make up the returned output. |

**Return values**

| | |
|---|---|
| *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_ASY↩ MMETRIC_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* | |

### 4.11.3.2 psa_asymmetric_encrypt()

```
psa_status_t psa_asymmetric_encrypt (
             psa_key_slot_t key,
             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 )
```

Encrypt a short message with a public key.

**Parameters**

| | | |
|---|---|---|
| | *key* | Key slot containing a public key or an asymmetric key pair. |
| | *alg* | An asymmetric encryption algorithm that is compatible with the type of `key`. |
| `in` | *input* | The message to encrypt. |
| | *input_length* | Size of the `input` buffer in bytes. |
| `in` | *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`. |

- For PSA_ALG_RSA_PKCS1V15_CRYPT, no salt is supported.

**Parameters**

| | | |
|---|---|---|
| | *salt_length* | Size of the `salt` buffer in bytes. If `salt` is `NULL`, pass 0. |
| `out` | *output* | Buffer where the encrypted message is to be written. |
| | *output_size* | Size of the `output` buffer in bytes. |
| `out` | *output_length* | On success, the number of bytes that make up the returned output. |

**Return values**

| | |
|---|---|
| *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_ASY↩ MMETRIC_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* | |

#### 4.11.3.3 psa_asymmetric_sign()

```
psa_status_t psa_asymmetric_sign (
             psa_key_slot_t key,
             psa_algorithm_t alg,
             const uint8_t * hash,
             size_t hash_length,
             uint8_t * signature,
             size_t signature_size,
             size_t * signature_length )
```

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.

**Parameters**

| | | |
|---|---|---|
| | *key* | Key slot containing an asymmetric key pair. |
| | *alg* | A signature algorithm that is compatible with the type of `key`. |
| `in` | *hash* | The hash or message to sign. |
| | *hash_length* | Size of the `hash` buffer in bytes. |
| `out` | *signature* | Buffer where the signature is to be written. |
| | *signature_size* | Size of the `signature` buffer in bytes. |
| `out` | *signature_length* | On success, the number of bytes that make up the returned signature value. |

**Return values**

| | |
|---:|---|
| *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* | |

### 4.11.3.4 psa_asymmetric_verify()

```
psa_status_t psa_asymmetric_verify (
              psa_key_slot_t key,
              psa_algorithm_t alg,
              const uint8_t * hash,
              size_t hash_length,
              const uint8_t * signature,
              size_t signature_length )
```

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.

**Parameters**

| | | |
|---|---|---|
| | *key* | Key slot containing a public key or an asymmetric key pair. |
| | *alg* | A signature algorithm that is compatible with the type of `key`. |
| `in` | *hash* | The hash or message whose signature is to be verified. |
| | *hash_length* | Size of the `hash` buffer in bytes. |
| `in` | *signature* | Buffer containing the signature to verify. |
| | *signature_length* | Size of the `signature` buffer in bytes. |

**Return values**

| | |
|---:|---|
| *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* | |

**Return values**

| | |
|---|---|
| *PSA_ERROR_COMMUNICATION_FAILURE* | |
| *PSA_ERROR_HARDWARE_FAILURE* | |
| *PSA_ERROR_TAMPERING_DETECTED* | |

## 4.12 Generators

### Macros

- #define PSA_CRYPTO_GENERATOR_INIT {0}

### Typedefs

- typedef struct psa_crypto_generator_s psa_crypto_generator_t

### Functions

- psa_status_t psa_get_generator_capacity (const psa_crypto_generator_t ∗generator, size_t ∗capacity)
- psa_status_t psa_generator_read (psa_crypto_generator_t ∗generator, uint8_t ∗output, size_t output_length)
- psa_status_t psa_generator_import_key (psa_key_slot_t key, psa_key_type_t type, size_t bits, psa_crypto←_generator_t ∗generator)
- psa_status_t psa_generator_abort (psa_crypto_generator_t ∗generator)

### 4.12.1 Detailed Description

### 4.12.2 Macro Definition Documentation

#### 4.12.2.1 PSA_CRYPTO_GENERATOR_INIT

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

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

### 4.12.3 Typedef Documentation

#### 4.12.3.1 psa_crypto_generator_t

```
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.

### 4.12.4 Function Documentation

#### 4.12.4.1 psa_generator_abort()

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

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_GE↩ NERATOR_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.

**Parameters**

| | | |
|---|---|---|
| in,out | *generator* | The generator to abort. |

**Return values**

| | |
|---|---|
| *PSA_SUCCESS* | |
| *PSA_ERROR_BAD_STATE* | |
| *PSA_ERROR_COMMUNICATION_FAILURE* | |
| *PSA_ERROR_HARDWARE_FAILURE* | |
| *PSA_ERROR_TAMPERING_DETECTED* | |

#### 4.12.4.2 psa_generator_import_key()

```
psa_status_t psa_generator_import_key (
            psa_key_slot_t key,
            psa_key_type_t type,
            size_t bits,
            psa_crypto_generator_t * generator )
```

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.

**Parameters**

| | | |
|---|---|---|
| | *key* | Slot where the key will be stored. This must be a valid slot for a key of the chosen type. It must be unoccupied. |
| | *type* | Key type (a `PSA_KEY_TYPE_XXX` value). This must be a symmetric key type. |
| | *bits* | Key size in bits. |
| `in,out` | *generator* | The generator object to read from. |

**Return values**

| | |
|---|---|
| *PSA_SUCCESS* | 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_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_ARGUMENT* | The key slot is invalid. |
| *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* | |

### 4.12.4.3 psa_generator_read()

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

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.

**Parameters**

| | | |
|---|---|---|
| `in,out` | *generator* | The generator object to read from. |
| `out` | *output* | Buffer where the generator output will be written. |
| | *output_length* | Number of bytes to output. |

**Return values**

| | |
|---|---|
| *PSA_SUCCESS* | |

**Return values**

| | |
|---|---|
| *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* | |

#### 4.12.4.4 psa_get_generator_capacity()

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

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*.

**Parameters**

| | | |
|---|---|---|
| `in` | *generator* | The generator to query. |
| `out` | *capacity* | On success, the capacity of the generator. |

**Return values**

| | |
|---|---|
| *PSA_SUCCESS* | |
| *PSA_ERROR_BAD_STATE* | |
| *PSA_ERROR_COMMUNICATION_FAILURE* | |

## 4.13 Key derivation

### Functions

- psa_status_t psa_key_derivation (psa_crypto_generator_t ∗generator, psa_key_slot_t key, psa_algorithm_t alg, const uint8_t ∗salt, size_t salt_length, const uint8_t ∗label, size_t label_length, size_t capacity)

### 4.13.1 Detailed Description

### 4.13.2 Function Documentation

#### 4.13.2.1 psa_key_derivation()

```
psa_status_t psa_key_derivation (
            psa_crypto_generator_t * generator,
            psa_key_slot_t key,
            psa_algorithm_t alg,
            const uint8_t * salt,
            size_t salt_length,
            const uint8_t * label,
            size_t label_length,
            size_t capacity )
```

Set up a key derivation operation.

A key derivation algorithm takes three inputs: a secret input `key` and two non-secret inputs `label` and p salt. The result of this function is a byte generator which can be used to produce keys and other cryptographic material.

The role of `label` and `salt` is as follows:

- For HKDF (PSA_ALG_HKDF), `salt` is the salt used in the "extract" step and `label` is the info string used in the "expand" step.

**Parameters**

| | | |
|---|---|---|
| in,out | *generator* | The generator object to set up. It must have been initialized to . |
| | *key* | Slot containing the secret key to use. |
| | *alg* | The key derivation algorithm to compute (`PSA_ALG_XXX` value such that PSA_ALG_IS_KEY_DERIVATION(`alg`) is true). |
| in | *salt* | Salt to use. |
| | *salt_length* | Size of the `salt` buffer in bytes. |
| in | *label* | Label to use. |
| | *label_length* | Size of the `label` buffer in bytes. |
| | *capacity* | The maximum number of bytes that the generator will be able to provide. |

**Return values**

| | |
|---|---|
| *PSA_SUCCESS* | Success. |
| *PSA_ERROR_EMPTY_SLOT* | |
| *PSA_ERROR_NOT_PERMITTED* | |
| *PSA_ERROR_INVALID_ARGUMENT* | `key` is not compatible with `alg`, or `capacity` is too large for the specified algorithm and 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* | |

## 4.14 Random generation

### Classes

- struct psa_generate_key_extra_rsa

### Functions

- psa_status_t psa_generate_random (uint8_t ∗output, size_t output_size)

  *Generate random bytes.*
- psa_status_t psa_generate_key (psa_key_slot_t key, psa_key_type_t type, size_t bits, const void ∗extra, size_t extra_size)

  *Generate a key or key pair.*

### 4.14.1 Detailed Description

### 4.14.2 Function Documentation

#### 4.14.2.1 psa_generate_key()

```
psa_status_t psa_generate_key (
            psa_key_slot_t key,
            psa_key_type_t type,
            size_t bits,
            const void ∗ extra,
            size_t extra_size )
```

Generate a key or key pair.

**Parameters**

| | | |
|---|---|---|
| | *key* | Slot where the key will be stored. This must be a valid slot for a key of the chosen type. It must be unoccupied. |
| | *type* | Key type (a `PSA_KEY_TYPE_XXX` value). |
| | *bits* | Key size in bits. |
| `in` | *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:<br>• For a symmetric key type (a type such that PSA_KEY_TYPE_IS_ASYMMETRIC(`type`) is false), `extra` must be `NULL`.<br>• For an elliptic curve key type (a type such that PSA_KEY_TYPE_IS_ECC(`type`) is false), `extra` must be `NULL`.<br>• 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. |
| | | |
| | *extra_size* | Size of the buffer that `extra` points to, in bytes. Note that if `extra` is `NULL` then `extra_size` must be zero. |

**Return values**

| | |
|---|---|
| *PSA_SUCCESS* | |
| *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* | |

#### 4.14.2.2 psa_generate_random()

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

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.

**Parameters**

| | | |
|---|---|---|
| out | *output* | Output buffer for the generated data. |
| | *output_size* | Number of bytes to generate and output. |

**Return values**

| | |
|---|---|
| *PSA_SUCCESS* | |
| *PSA_ERROR_NOT_SUPPORTED* | |
| *PSA_ERROR_INSUFFICIENT_ENTROPY* | |
| *PSA_ERROR_COMMUNICATION_FAILURE* | |
| *PSA_ERROR_HARDWARE_FAILURE* | |
| *PSA_ERROR_TAMPERING_DETECTED* | |

# Chapter 5

# Class Documentation

## 5.1 psa_generate_key_extra_rsa Struct Reference

```
#include <crypto.h>
```

### Public Attributes

- uint32_t e

### 5.1.1 Detailed 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().

### 5.1.2 Member Data Documentation

#### 5.1.2.1 e

```
uint32_t psa_generate_key_extra_rsa::e
```

Public exponent value. Default: 65537.

The documentation for this struct was generated from the following file:

- psa/crypto.h

# Chapter 6

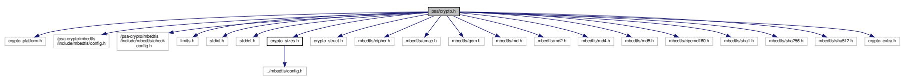
# File Documentation

## 6.1 psa/crypto.h File Reference

Platform Security Architecture cryptography module.

```
#include "crypto_platform.h"
#include <stddef.h>
#include "crypto_sizes.h"
#include "crypto_struct.h"
#include "crypto_extra.h"
```

Include dependency graph for crypto.h:



### Classes

- struct psa_generate_key_extra_rsa

### Macros

- #define PSA_SUCCESS ((psa_status_t)0)
- #define PSA_ERROR_UNKNOWN_ERROR ((psa_status_t)1)
- #define PSA_ERROR_NOT_SUPPORTED ((psa_status_t)2)
- #define PSA_ERROR_NOT_PERMITTED ((psa_status_t)3)
- #define PSA_ERROR_BUFFER_TOO_SMALL ((psa_status_t)4)
- #define PSA_ERROR_OCCUPIED_SLOT ((psa_status_t)5)
- #define PSA_ERROR_EMPTY_SLOT ((psa_status_t)6)
- #define PSA_ERROR_BAD_STATE ((psa_status_t)7)
- #define PSA_ERROR_INVALID_ARGUMENT ((psa_status_t)8)
- #define PSA_ERROR_INSUFFICIENT_MEMORY ((psa_status_t)9)
- #define PSA_ERROR_INSUFFICIENT_STORAGE ((psa_status_t)10)
- #define PSA_ERROR_COMMUNICATION_FAILURE ((psa_status_t)11)
- #define PSA_ERROR_STORAGE_FAILURE ((psa_status_t)12)

- #define PSA_ERROR_HARDWARE_FAILURE ((psa_status_t)13)
- #define PSA_ERROR_TAMPERING_DETECTED ((psa_status_t)14)
- #define PSA_ERROR_INSUFFICIENT_ENTROPY ((psa_status_t)15)
- #define PSA_ERROR_INVALID_SIGNATURE ((psa_status_t)16)
- #define PSA_ERROR_INVALID_PADDING ((psa_status_t)17)
- #define PSA_ERROR_INSUFFICIENT_CAPACITY ((psa_status_t)18)
- #define **PSA_BITS_TO_BYTES**(bits) (((bits) + 7) / 8)
- #define **PSA_BYTES_TO_BITS**(bytes) ((bytes) ∗ 8)
- #define PSA_KEY_TYPE_NONE ((psa_key_type_t)0x00000000)
- #define PSA_KEY_TYPE_VENDOR_FLAG ((psa_key_type_t)0x80000000)
- #define **PSA_KEY_TYPE_CATEGORY_MASK** ((psa_key_type_t)0x7e000000)
- #define PSA_KEY_TYPE_RAW_DATA ((psa_key_type_t)0x02000000)
- #define **PSA_KEY_TYPE_CATEGORY_SYMMETRIC** ((psa_key_type_t)0x04000000)
- #define **PSA_KEY_TYPE_CATEGORY_ASYMMETRIC** ((psa_key_type_t)0x06000000)
- #define **PSA_KEY_TYPE_PAIR_FLAG** ((psa_key_type_t)0x01000000)
- #define PSA_KEY_TYPE_HMAC ((psa_key_type_t)0x02000001)
- #define PSA_KEY_TYPE_DERIVE ((psa_key_type_t)0x02000101)
- #define PSA_KEY_TYPE_AES ((psa_key_type_t)0x04000001)
- #define PSA_KEY_TYPE_DES ((psa_key_type_t)0x04000002)
- #define PSA_KEY_TYPE_CAMELLIA ((psa_key_type_t)0x04000003)
- #define PSA_KEY_TYPE_ARC4 ((psa_key_type_t)0x04000004)
- #define PSA_KEY_TYPE_RSA_PUBLIC_KEY ((psa_key_type_t)0x06010000)
- #define PSA_KEY_TYPE_RSA_KEYPAIR ((psa_key_type_t)0x07010000)
- #define PSA_KEY_TYPE_DSA_PUBLIC_KEY ((psa_key_type_t)0x06020000)
- #define PSA_KEY_TYPE_DSA_KEYPAIR ((psa_key_type_t)0x07020000)
- #define **PSA_KEY_TYPE_ECC_PUBLIC_KEY_BASE** ((psa_key_type_t)0x06030000)
- #define **PSA_KEY_TYPE_ECC_KEYPAIR_BASE** ((psa_key_type_t)0x07030000)
- #define **PSA_KEY_TYPE_ECC_CURVE_MASK** ((psa_key_type_t)0x0000ffff)
- #define PSA_KEY_TYPE_ECC_KEYPAIR(curve) (PSA_KEY_TYPE_ECC_KEYPAIR_BASE | (curve))
- #define PSA_KEY_TYPE_ECC_PUBLIC_KEY(curve) (PSA_KEY_TYPE_ECC_PUBLIC_KEY_BASE | (curve))
- #define PSA_KEY_TYPE_IS_VENDOR_DEFINED(type) (((type) & PSA_KEY_TYPE_VENDOR_FLAG) != 0)
- #define PSA_KEY_TYPE_IS_ASYMMETRIC(type) (((type) & PSA_KEY_TYPE_CATEGORY_MASK) == PSA_KEY_TYPE_CATEGORY_ASYMMETRIC)
- #define PSA_KEY_TYPE_IS_PUBLIC_KEY(type)
- #define PSA_KEY_TYPE_IS_KEYPAIR(type)
- #define PSA_KEY_TYPE_KEYPAIR_OF_PUBLIC_KEY(type) ((type) | PSA_KEY_TYPE_PAIR_FLAG)
- #define PSA_KEY_TYPE_PUBLIC_KEY_OF_KEYPAIR(type) ((type) & ∼PSA_KEY_TYPE_PAIR_FLAG)
- #define PSA_KEY_TYPE_IS_RSA(type) (PSA_KEY_TYPE_PUBLIC_KEY_OF_KEYPAIR(type) == PSA_←KEY_TYPE_RSA_PUBLIC_KEY)
- #define PSA_KEY_TYPE_IS_ECC(type)
- #define **PSA_KEY_TYPE_IS_ECC_KEYPAIR**(type)
- #define **PSA_KEY_TYPE_IS_ECC_PUBLIC_KEY**(type)
- #define PSA_KEY_TYPE_GET_CURVE(type)
- #define **PSA_ECC_CURVE_SECT163K1** ((psa_ecc_curve_t) 0x0001)
- #define **PSA_ECC_CURVE_SECT163R1** ((psa_ecc_curve_t) 0x0002)
- #define **PSA_ECC_CURVE_SECT163R2** ((psa_ecc_curve_t) 0x0003)
- #define **PSA_ECC_CURVE_SECT193R1** ((psa_ecc_curve_t) 0x0004)
- #define **PSA_ECC_CURVE_SECT193R2** ((psa_ecc_curve_t) 0x0005)
- #define **PSA_ECC_CURVE_SECT233K1** ((psa_ecc_curve_t) 0x0006)
- #define **PSA_ECC_CURVE_SECT233R1** ((psa_ecc_curve_t) 0x0007)
- #define **PSA_ECC_CURVE_SECT239K1** ((psa_ecc_curve_t) 0x0008)
- #define **PSA_ECC_CURVE_SECT283K1** ((psa_ecc_curve_t) 0x0009)
- #define **PSA_ECC_CURVE_SECT283R1** ((psa_ecc_curve_t) 0x000a)
- #define **PSA_ECC_CURVE_SECT409K1** ((psa_ecc_curve_t) 0x000b)

- #define **PSA_ECC_CURVE_SECT409R1** ((psa_ecc_curve_t) 0x000c)
- #define **PSA_ECC_CURVE_SECT571K1** ((psa_ecc_curve_t) 0x000d)
- #define **PSA_ECC_CURVE_SECT571R1** ((psa_ecc_curve_t) 0x000e)
- #define **PSA_ECC_CURVE_SECP160K1** ((psa_ecc_curve_t) 0x000f)
- #define **PSA_ECC_CURVE_SECP160R1** ((psa_ecc_curve_t) 0x0010)
- #define **PSA_ECC_CURVE_SECP160R2** ((psa_ecc_curve_t) 0x0011)
- #define **PSA_ECC_CURVE_SECP192K1** ((psa_ecc_curve_t) 0x0012)
- #define **PSA_ECC_CURVE_SECP192R1** ((psa_ecc_curve_t) 0x0013)
- #define **PSA_ECC_CURVE_SECP224K1** ((psa_ecc_curve_t) 0x0014)
- #define **PSA_ECC_CURVE_SECP224R1** ((psa_ecc_curve_t) 0x0015)
- #define **PSA_ECC_CURVE_SECP256K1** ((psa_ecc_curve_t) 0x0016)
- #define **PSA_ECC_CURVE_SECP256R1** ((psa_ecc_curve_t) 0x0017)
- #define **PSA_ECC_CURVE_SECP384R1** ((psa_ecc_curve_t) 0x0018)
- #define **PSA_ECC_CURVE_SECP521R1** ((psa_ecc_curve_t) 0x0019)
- #define **PSA_ECC_CURVE_BRAINPOOL_P256R1** ((psa_ecc_curve_t) 0x001a)
- #define **PSA_ECC_CURVE_BRAINPOOL_P384R1** ((psa_ecc_curve_t) 0x001b)
- #define **PSA_ECC_CURVE_BRAINPOOL_P512R1** ((psa_ecc_curve_t) 0x001c)
- #define **PSA_ECC_CURVE_CURVE25519** ((psa_ecc_curve_t) 0x001d)
- #define **PSA_ECC_CURVE_CURVE448** ((psa_ecc_curve_t) 0x001e)
- #define **PSA_ECC_CURVE_FFDHE_2048** ((psa_ecc_curve_t) 0x0100)
- #define **PSA_ECC_CURVE_FFDHE_3072** ((psa_ecc_curve_t) 0x0101)
- #define **PSA_ECC_CURVE_FFDHE_4096** ((psa_ecc_curve_t) 0x0102)
- #define **PSA_ECC_CURVE_FFDHE_6144** ((psa_ecc_curve_t) 0x0103)
- #define **PSA_ECC_CURVE_FFDHE_8192** ((psa_ecc_curve_t) 0x0104)
- #define PSA_BLOCK_CIPHER_BLOCK_SIZE(type)
- #define **PSA_ALG_VENDOR_FLAG** ((psa_algorithm_t)0x80000000)
- #define **PSA_ALG_CATEGORY_MASK** ((psa_algorithm_t)0x7f000000)
- #define **PSA_ALG_CATEGORY_HASH** ((psa_algorithm_t)0x01000000)
- #define **PSA_ALG_CATEGORY_MAC** ((psa_algorithm_t)0x02000000)
- #define **PSA_ALG_CATEGORY_CIPHER** ((psa_algorithm_t)0x04000000)
- #define **PSA_ALG_CATEGORY_AEAD** ((psa_algorithm_t)0x06000000)
- #define **PSA_ALG_CATEGORY_SIGN** ((psa_algorithm_t)0x10000000)
- #define **PSA_ALG_CATEGORY_ASYMMETRIC_ENCRYPTION** ((psa_algorithm_t)0x12000000)
- #define **PSA_ALG_CATEGORY_KEY_AGREEMENT** ((psa_algorithm_t)0x22000000)
- #define **PSA_ALG_CATEGORY_KEY_DERIVATION** ((psa_algorithm_t)0x30000000)
- #define **PSA_ALG_IS_VENDOR_DEFINED**(alg) (((alg) & PSA_ALG_VENDOR_FLAG) != 0)
- #define PSA_ALG_IS_HASH(alg) (((alg) & PSA_ALG_CATEGORY_MASK) == PSA_ALG_CATEGORY_↩
  HASH)
- #define PSA_ALG_IS_MAC(alg) (((alg) & PSA_ALG_CATEGORY_MASK) == PSA_ALG_CATEGORY_M↩
  AC)
- #define PSA_ALG_IS_CIPHER(alg) (((alg) & PSA_ALG_CATEGORY_MASK) == PSA_ALG_CATEGOR↩
  Y_CIPHER)
- #define PSA_ALG_IS_AEAD(alg) (((alg) & PSA_ALG_CATEGORY_MASK) == PSA_ALG_CATEGORY_↩
  AEAD)
- #define PSA_ALG_IS_SIGN(alg) (((alg) & PSA_ALG_CATEGORY_MASK) == PSA_ALG_CATEGORY_S↩
  IGN)
- #define PSA_ALG_IS_ASYMMETRIC_ENCRYPTION(alg) (((alg) & PSA_ALG_CATEGORY_MASK) == P↩
  SA_ALG_CATEGORY_ASYMMETRIC_ENCRYPTION)
- #define PSA_ALG_IS_KEY_AGREEMENT(alg) (((alg) & PSA_ALG_CATEGORY_MASK) == PSA_ALG_↩
  CATEGORY_KEY_AGREEMENT)
- #define PSA_ALG_IS_KEY_DERIVATION(alg) (((alg) & PSA_ALG_CATEGORY_MASK) == PSA_ALG_↩
  CATEGORY_KEY_DERIVATION)
- #define **PSA_ALG_HASH_MASK** ((psa_algorithm_t)0x000000ff)
- #define **PSA_ALG_MD2** ((psa_algorithm_t)0x01000001)
- #define **PSA_ALG_MD4** ((psa_algorithm_t)0x01000002)

- #define **PSA_ALG_MD5** ((psa_algorithm_t)0x01000003)
- #define **PSA_ALG_RIPEMD160** ((psa_algorithm_t)0x01000004)
- #define **PSA_ALG_SHA_1** ((psa_algorithm_t)0x01000005)
- #define PSA_ALG_SHA_224 ((psa_algorithm_t)0x01000008)
- #define PSA_ALG_SHA_256 ((psa_algorithm_t)0x01000009)
- #define PSA_ALG_SHA_384 ((psa_algorithm_t)0x0100000a)
- #define PSA_ALG_SHA_512 ((psa_algorithm_t)0x0100000b)
- #define PSA_ALG_SHA_512_224 ((psa_algorithm_t)0x0100000c)
- #define PSA_ALG_SHA_512_256 ((psa_algorithm_t)0x0100000d)
- #define PSA_ALG_SHA3_224 ((psa_algorithm_t)0x01000010)
- #define PSA_ALG_SHA3_256 ((psa_algorithm_t)0x01000011)
- #define PSA_ALG_SHA3_384 ((psa_algorithm_t)0x01000012)
- #define PSA_ALG_SHA3_512 ((psa_algorithm_t)0x01000013)
- #define **PSA_ALG_MAC_SUBCATEGORY_MASK** ((psa_algorithm_t)0x00c00000)
- #define **PSA_ALG_HMAC_BASE** ((psa_algorithm_t)0x02800000)
- #define PSA_ALG_HMAC(hash_alg) (PSA_ALG_HMAC_BASE | ((hash_alg) & PSA_ALG_HASH_MASK))
- #define **PSA_ALG_HMAC_HASH**(hmac_alg) (PSA_ALG_CATEGORY_HASH | ((hmac_alg) & PSA_ALG↩ _HASH_MASK))
- #define PSA_ALG_IS_HMAC(alg)
- #define **PSA_ALG_CIPHER_MAC_BASE** ((psa_algorithm_t)0x02c00000)
- #define **PSA_ALG_CBC_MAC** ((psa_algorithm_t)0x02c00001)
- #define **PSA_ALG_CMAC** ((psa_algorithm_t)0x02c00002)
- #define **PSA_ALG_GMAC** ((psa_algorithm_t)0x02c00003)
- #define PSA_ALG_IS_CIPHER_MAC(alg)
- #define **PSA_ALG_CIPHER_SUBCATEGORY_MASK** ((psa_algorithm_t)0x00c00000)
- #define **PSA_ALG_BLOCK_CIPHER_BASE** ((psa_algorithm_t)0x04000000)
- #define **PSA_ALG_BLOCK_CIPHER_MODE_MASK** ((psa_algorithm_t)0x000000ff)
- #define **PSA_ALG_BLOCK_CIPHER_PADDING_MASK** ((psa_algorithm_t)0x003f0000)
- #define PSA_ALG_BLOCK_CIPHER_PAD_NONE ((psa_algorithm_t)0x00000000)
- #define **PSA_ALG_BLOCK_CIPHER_PAD_PKCS7** ((psa_algorithm_t)0x00010000)
- #define PSA_ALG_IS_BLOCK_CIPHER(alg)
- #define PSA_ALG_CBC_BASE ((psa_algorithm_t)0x04000001)
- #define **PSA_ALG_CFB_BASE** ((psa_algorithm_t)0x04000002)
- #define **PSA_ALG_OFB_BASE** ((psa_algorithm_t)0x04000003)
- #define **PSA_ALG_XTS_BASE** ((psa_algorithm_t)0x04000004)
- #define **PSA_ALG_STREAM_CIPHER_BASE** ((psa_algorithm_t)0x04800000)
- #define PSA_ALG_CTR ((psa_algorithm_t)0x04800001)
- #define PSA_ALG_ARC4 ((psa_algorithm_t)0x04800002)
- #define PSA_ALG_IS_STREAM_CIPHER(alg)
- #define **PSA_ALG_CCM** ((psa_algorithm_t)0x06000001)
- #define **PSA_ALG_GCM** ((psa_algorithm_t)0x06000002)
- #define **PSA_ALG_RSA_PKCS1V15_SIGN_BASE** ((psa_algorithm_t)0x10020000)
- #define PSA_ALG_RSA_PKCS1V15_SIGN(hash_alg) (PSA_ALG_RSA_PKCS1V15_SIGN_BASE | ((hash_alg) & PSA_ALG_HASH_MASK))
- #define PSA_ALG_RSA_PKCS1V15_SIGN_RAW PSA_ALG_RSA_PKCS1V15_SIGN_BASE
- #define **PSA_ALG_IS_RSA_PKCS1V15_SIGN**(alg) (((alg) & ∼PSA_ALG_HASH_MASK) == PSA_ALG_↩ RSA_PKCS1V15_SIGN_BASE)
- #define **PSA_ALG_RSA_PSS_BASE** ((psa_algorithm_t)0x10030000)
- #define PSA_ALG_RSA_PSS(hash_alg) (PSA_ALG_RSA_PSS_BASE | ((hash_alg) & PSA_ALG_HASH↩ _MASK))
- #define **PSA_ALG_IS_RSA_PSS**(alg) (((alg) & ∼PSA_ALG_HASH_MASK) == PSA_ALG_RSA_PSS_BA↩ SE)
- #define **PSA_ALG_DSA_BASE** ((psa_algorithm_t)0x10040000)
- #define PSA_ALG_DSA(hash_alg) (PSA_ALG_DSA_BASE | ((hash_alg) & PSA_ALG_HASH_MASK))
- #define **PSA_ALG_DETERMINISTIC_DSA_BASE** ((psa_algorithm_t)0x10050000)

- #define **PSA_ALG_DSA_DETERMINISTIC_FLAG** ((psa_algorithm_t)0x00010000)
- #define **PSA_ALG_DETERMINISTIC_DSA**(hash_alg) (PSA_ALG_DETERMINISTIC_DSA_BASE | ((hash↵_alg) & PSA_ALG_HASH_MASK))
- #define **PSA_ALG_IS_DSA**(alg)
- #define **PSA_ALG_DSA_IS_DETERMINISTIC**(alg) (((alg) & PSA_ALG_DSA_DETERMINISTIC_FLAG) != 0)
- #define **PSA_ALG_IS_DETERMINISTIC_DSA**(alg) (PSA_ALG_IS_DSA(alg) && PSA_ALG_DSA_IS_DE↵TERMINISTIC(alg))
- #define **PSA_ALG_IS_RANDOMIZED_DSA**(alg) (PSA_ALG_IS_DSA(alg) && !PSA_ALG_DSA_IS_DET↵ERMINISTIC(alg))
- #define **PSA_ALG_ECDSA_BASE** ((psa_algorithm_t)0x10060000)
- #define PSA_ALG_ECDSA(hash_alg) (PSA_ALG_ECDSA_BASE | ((hash_alg) & PSA_ALG_HASH_MA↵SK))
- #define PSA_ALG_ECDSA_ANY PSA_ALG_ECDSA_BASE
- #define **PSA_ALG_DETERMINISTIC_ECDSA_BASE** ((psa_algorithm_t)0x10070000)
- #define PSA_ALG_DETERMINISTIC_ECDSA(hash_alg) (PSA_ALG_DETERMINISTIC_ECDSA_BASE | ((hash_alg) & PSA_ALG_HASH_MASK))
- #define **PSA_ALG_IS_ECDSA**(alg)
- #define **PSA_ALG_ECDSA_IS_DETERMINISTIC**(alg) (((alg) & PSA_ALG_DSA_DETERMINISTIC_FLAG) != 0)
- #define **PSA_ALG_IS_DETERMINISTIC_ECDSA**(alg) (PSA_ALG_IS_ECDSA(alg) && PSA_ALG_ECDS↵A_IS_DETERMINISTIC(alg))
- #define **PSA_ALG_IS_RANDOMIZED_ECDSA**(alg) (PSA_ALG_IS_ECDSA(alg) && !PSA_ALG_ECDSA↵_IS_DETERMINISTIC(alg))
- #define PSA_ALG_SIGN_GET_HASH(alg)
- #define PSA_ALG_RSA_PKCS1V15_CRYPT ((psa_algorithm_t)0x12020000)
- #define **PSA_ALG_RSA_OAEP_BASE** ((psa_algorithm_t)0x12030000)
- #define PSA_ALG_RSA_OAEP(hash_alg) (PSA_ALG_RSA_OAEP_BASE | ((hash_alg) & PSA_ALG_HA↵SH_MASK))
- #define **PSA_ALG_IS_RSA_OAEP**(alg) (((alg) & ∼PSA_ALG_HASH_MASK) == PSA_ALG_RSA_OAEP↵_BASE)
- #define **PSA_ALG_RSA_OAEP_GET_HASH**(alg)
- #define **PSA_ALG_HKDF_BASE** ((psa_algorithm_t)0x30000100)
- #define PSA_ALG_HKDF(hash_alg) (PSA_ALG_HKDF_BASE | ((hash_alg) & PSA_ALG_HASH_MASK))
- #define PSA_ALG_IS_HKDF(alg) (((alg) & ∼PSA_ALG_HASH_MASK) == PSA_ALG_HKDF_BASE)
- #define **PSA_ALG_HKDF_GET_HASH**(hkdf_alg) (PSA_ALG_CATEGORY_HASH | ((hkdf_alg) & PSA_A↵LG_HASH_MASK))
- #define PSA_KEY_USAGE_EXPORT ((psa_key_usage_t)0x00000001)
- #define PSA_KEY_USAGE_ENCRYPT ((psa_key_usage_t)0x00000100)
- #define PSA_KEY_USAGE_DECRYPT ((psa_key_usage_t)0x00000200)
- #define PSA_KEY_USAGE_SIGN ((psa_key_usage_t)0x00000400)
- #define PSA_KEY_USAGE_VERIFY ((psa_key_usage_t)0x00000800)
- #define PSA_KEY_USAGE_DERIVE ((psa_key_usage_t)0x00001000)
- #define PSA_KEY_LIFETIME_VOLATILE ((psa_key_lifetime_t)0x00000000)
- #define PSA_KEY_LIFETIME_PERSISTENT ((psa_key_lifetime_t)0x00000001)
- #define PSA_KEY_LIFETIME_WRITE_ONCE ((psa_key_lifetime_t)0x7fffffff)
- #define PSA_HASH_SIZE(alg)
- #define PSA_AEAD_TAG_SIZE(alg)
- #define PSA_ECDSA_SIGNATURE_SIZE(curve_bits) (PSA_BITS_TO_BYTES(curve_bits) ∗ 2)

    *ECDSA signature size for a given curve bit size.*
- #define **PSA_RSA_MINIMUM_PADDING_SIZE**(alg)
- #define PSA_CRYPTO_GENERATOR_INIT {0}

## Typedefs

- typedef _unsigned_integral_type_ psa_key_slot_t

  *Key slot number.*
- typedef int32_t psa_status_t

  *Function return status.*
- typedef uint32_t psa_key_type_t

  *Encoding of a key type.*
- typedef uint16_t psa_ecc_curve_t
- typedef uint32_t psa_algorithm_t

  *Encoding of a cryptographic algorithm.*
- typedef uint32_t psa_key_usage_t

  *Encoding of permitted usage on a key.*
- typedef struct psa_key_policy_s psa_key_policy_t
- typedef uint32_t psa_key_lifetime_t
- typedef struct psa_hash_operation_s psa_hash_operation_t
- typedef struct psa_mac_operation_s psa_mac_operation_t
- typedef struct psa_cipher_operation_s psa_cipher_operation_t
- typedef struct psa_crypto_generator_s psa_crypto_generator_t

## Functions

- psa_status_t psa_crypto_init (void)

  *Library initialization.*
- psa_status_t psa_import_key (psa_key_slot_t key, psa_key_type_t type, const uint8_t ∗data, size_t data_↩ length)

  *Import a key in binary format.*
- psa_status_t psa_destroy_key (psa_key_slot_t key)

  *Destroy a key and restore the slot to its default state.*
- psa_status_t psa_get_key_information (psa_key_slot_t key, psa_key_type_t ∗type, size_t ∗bits)

  *Get basic metadata about a key.*
- psa_status_t psa_export_key (psa_key_slot_t key, uint8_t ∗data, size_t data_size, size_t ∗data_length)

  *Export a key in binary format.*
- psa_status_t psa_export_public_key (psa_key_slot_t key, uint8_t ∗data, size_t data_size, size_t ∗data_↩ length)

  *Export a public key or the public part of a key pair in binary format.*
- void psa_key_policy_init (psa_key_policy_t ∗policy)

  *Initialize a key policy structure to a default that forbids all usage of the key.*
- void psa_key_policy_set_usage (psa_key_policy_t ∗policy, psa_key_usage_t usage, psa_algorithm_t alg)

  *Set the standard fields of a policy structure.*
- psa_key_usage_t psa_key_policy_get_usage (const psa_key_policy_t ∗policy)

  *Retrieve the usage field of a policy structure.*
- psa_algorithm_t psa_key_policy_get_algorithm (const psa_key_policy_t ∗policy)

  *Retrieve the algorithm field of a policy structure.*
- psa_status_t psa_set_key_policy (psa_key_slot_t key, const psa_key_policy_t ∗policy)

  *Set the usage policy on a key slot.*
- psa_status_t psa_get_key_policy (psa_key_slot_t key, psa_key_policy_t ∗policy)

  *Get the usage policy for a key slot.*
- psa_status_t psa_get_key_lifetime (psa_key_slot_t key, psa_key_lifetime_t ∗lifetime)

  *Retrieve the lifetime of a key slot.*
- psa_status_t psa_set_key_lifetime (psa_key_slot_t key, psa_key_lifetime_t lifetime)

*Change the lifetime of a key slot.*

- psa_status_t psa_hash_setup (psa_hash_operation_t ∗operation, psa_algorithm_t alg)
- psa_status_t psa_hash_update (psa_hash_operation_t ∗operation, const uint8_t ∗input, size_t input_length)
- psa_status_t psa_hash_finish (psa_hash_operation_t ∗operation, uint8_t ∗hash, size_t hash_size, size_↩ t ∗hash_length)
- psa_status_t psa_hash_verify (psa_hash_operation_t ∗operation, const uint8_t ∗hash, size_t hash_length)
- psa_status_t psa_hash_abort (psa_hash_operation_t ∗operation)
- psa_status_t psa_mac_sign_setup (psa_mac_operation_t ∗operation, psa_key_slot_t key, psa_algorithm_t alg)
- psa_status_t psa_mac_verify_setup (psa_mac_operation_t ∗operation, psa_key_slot_t key, psa_algorithm↩ _t alg)
- psa_status_t psa_mac_update (psa_mac_operation_t ∗operation, const uint8_t ∗input, size_t input_length)
- psa_status_t psa_mac_sign_finish (psa_mac_operation_t ∗operation, uint8_t ∗mac, size_t mac_size, size_t ∗mac_length)
- psa_status_t psa_mac_verify_finish (psa_mac_operation_t ∗operation, const uint8_t ∗mac, size_t mac_↩ length)
- psa_status_t psa_mac_abort (psa_mac_operation_t ∗operation)
- psa_status_t psa_cipher_encrypt_setup (psa_cipher_operation_t ∗operation, psa_key_slot_t key, psa_↩ algorithm_t alg)
- psa_status_t psa_cipher_decrypt_setup (psa_cipher_operation_t ∗operation, psa_key_slot_t key, psa_↩ algorithm_t alg)
- psa_status_t psa_cipher_generate_iv (psa_cipher_operation_t ∗operation, unsigned char ∗iv, size_t iv_size, size_t ∗iv_length)
- psa_status_t psa_cipher_set_iv (psa_cipher_operation_t ∗operation, const unsigned char ∗iv, size_t iv_↩ length)
- 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)
- psa_status_t psa_cipher_finish (psa_cipher_operation_t ∗operation, uint8_t ∗output, size_t output_size, size_t ∗output_length)
- psa_status_t psa_cipher_abort (psa_cipher_operation_t ∗operation)
- psa_status_t psa_aead_encrypt (psa_key_slot_t key, 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)
- psa_status_t psa_aead_decrypt (psa_key_slot_t key, 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)
- psa_status_t psa_asymmetric_sign (psa_key_slot_t key, psa_algorithm_t alg, const uint8_t ∗hash, size_↩ t hash_length, uint8_t ∗signature, size_t signature_size, size_t ∗signature_length)

  *Sign a hash or short message with a private key.*
- psa_status_t psa_asymmetric_verify (psa_key_slot_t key, psa_algorithm_t alg, const uint8_t ∗hash, size_t hash_length, const uint8_t ∗signature, size_t signature_length)

  *Verify the signature a hash or short message using a public key.*
- psa_status_t psa_asymmetric_encrypt (psa_key_slot_t key, 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)

  *Encrypt a short message with a public key.*
- psa_status_t psa_asymmetric_decrypt (psa_key_slot_t key, 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)

  *Decrypt a short message with a private key.*
- psa_status_t psa_get_generator_capacity (const psa_crypto_generator_t ∗generator, size_t ∗capacity)
- psa_status_t psa_generator_read (psa_crypto_generator_t ∗generator, uint8_t ∗output, size_t output_length)
- psa_status_t psa_generator_import_key (psa_key_slot_t key, psa_key_type_t type, size_t bits, psa_crypto↩ _generator_t ∗generator)
- psa_status_t psa_generator_abort (psa_crypto_generator_t ∗generator)

- psa_status_t psa_key_derivation (psa_crypto_generator_t ∗generator, psa_key_slot_t key, psa_algorithm_t alg, const uint8_t ∗salt, size_t salt_length, const uint8_t ∗label, size_t label_length, size_t capacity)
- psa_status_t psa_generate_random (uint8_t ∗output, size_t output_size)

  *Generate random bytes.*
- psa_status_t psa_generate_key (psa_key_slot_t key, psa_key_type_t type, size_t bits, const void ∗extra, size_t extra_size)

  *Generate a key or key pair.*

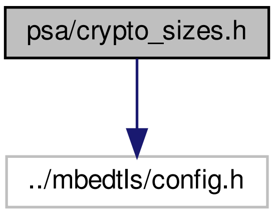
### 6.1.1 Detailed Description

Platform Security Architecture cryptography module.

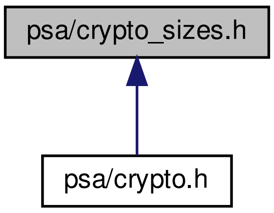
## 6.2 psa/crypto_sizes.h File Reference

PSA cryptography module: Mbed TLS buffer size macros.

```
#include "../mbedtls/config.h"
```
Include dependency graph for crypto_sizes.h:



This graph shows which files directly or indirectly include this file:



## Macros

- #define PSA_HASH_MAX_SIZE 64
- #define **PSA_HMAC_MAX_HASH_BLOCK_SIZE** 128
- #define PSA_MAC_MAX_SIZE PSA_HASH_MAX_SIZE
- #define **PSA_VENDOR_RSA_MAX_KEY_BITS** 4096
- #define **PSA_VENDOR_ECC_MAX_CURVE_BITS** 521
- #define PSA_ASYMMETRIC_SIGNATURE_MAX_SIZE
- #define PSA_MAC_FINAL_SIZE(key_type, key_bits, alg)
- #define PSA_AEAD_ENCRYPT_OUTPUT_SIZE(alg, plaintext_length)
- #define PSA_AEAD_DECRYPT_OUTPUT_SIZE(alg, ciphertext_length)
- #define PSA_ASYMMETRIC_SIGN_OUTPUT_SIZE(key_type, key_bits, alg)
- #define PSA_ASYMMETRIC_ENCRYPT_OUTPUT_SIZE(key_type, key_bits, alg)
- #define PSA_ASYMMETRIC_DECRYPT_OUTPUT_SIZE(key_type, key_bits, alg)

## 6.2.1 Detailed Description

PSA cryptography module: Mbed TLS buffer size macros.

### Note

This file may not be included directly. Applications must include psa/crypto.h.

This file contains the definitions of macros that are useful to compute buffer sizes. The signatures and semantics of these macros are standardized, but the definitions are not, because they depend on the available algorithms and, in some cases, on permitted tolerances on buffer sizes.

In implementations with isolation between the application and the cryptography module, implementers should take care to ensure that the definitions that are exposed to applications match what the module implements.

Macros that compute sizes whose values do not depend on the implementation are in crypto.h.

## 6.2.2 Macro Definition Documentation

### 6.2.2.1 PSA_AEAD_DECRYPT_OUTPUT_SIZE

```
#define PSA_AEAD_DECRYPT_OUTPUT_SIZE(
                alg,
                ciphertext_length )
```

**Value:**

```
(PSA_AEAD_TAG_SIZE(alg) != 0 ?                    \
     (plaintext_length) - PSA_AEAD_TAG_SIZE(alg) :      \
     0)
```

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.

**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.

### 6.2.2.2 PSA_AEAD_ENCRYPT_OUTPUT_SIZE

```
#define PSA_AEAD_ENCRYPT_OUTPUT_SIZE(
              alg,
              plaintext_length )
```

**Value:**

```
(PSA_AEAD_TAG_SIZE(alg) != 0 ?                              \
     (plaintext_length) + PSA_AEAD_TAG_SIZE(alg) :              \
     0)
```

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.

**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.

### 6.2.2.3 PSA_ASYMMETRIC_DECRYPT_OUTPUT_SIZE

```
#define PSA_ASYMMETRIC_DECRYPT_OUTPUT_SIZE(
              key_type,
              key_bits,
              alg )
```

**Value:**

```
(PSA_KEY_TYPE_IS_RSA(key_type) ?                                    \
     PSA_BITS_TO_BYTES(key_bits) - PSA_RSA_MINIMUM_PADDING_SIZE(alg) :  \
     0)
```

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.

**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.

#### 6.2.2.4 PSA_ASYMMETRIC_ENCRYPT_OUTPUT_SIZE

```
#define PSA_ASYMMETRIC_ENCRYPT_OUTPUT_SIZE(
             key_type,
             key_bits,
             alg )
```

**Value:**

```
(PSA_KEY_TYPE_IS_RSA(key_type) ?                                    \
     ((void)alg, PSA_BITS_TO_BYTES(key_bits)) :                         \
     0)
```

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.

**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.

#### 6.2.2.5 PSA_ASYMMETRIC_SIGN_OUTPUT_SIZE

```
#define PSA_ASYMMETRIC_SIGN_OUTPUT_SIZE(
              key_type,
              key_bits,
              alg )
```

**Value:**

```
(PSA_KEY_TYPE_IS_RSA(key_type) ? ((void)alg, PSA_BITS_TO_BYTES(key_bits)) : \
     PSA_KEY_TYPE_IS_ECC(key_type) ? PSA_ECDSA_SIGNATURE_SIZE(
      key_bits) : \
     ((void)alg, 0))
```

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.

**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.

#### 6.2.2.6 PSA_ASYMMETRIC_SIGNATURE_MAX_SIZE

```
#define PSA_ASYMMETRIC_SIGNATURE_MAX_SIZE
```

**Value:**

```
PSA_BITS_TO_BYTES(                                                    \
        PSA_VENDOR_RSA_MAX_KEY_BITS > PSA_VENDOR_ECC_MAX_CURVE_BITS ?   \
        PSA_VENDOR_RSA_MAX_KEY_BITS :                                   \
        PSA_VENDOR_ECC_MAX_CURVE_BITS                                   \
        )
```

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.

#### 6.2.2.7 PSA_HASH_MAX_SIZE

```
#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.

#### 6.2.2.8 PSA_MAC_FINAL_SIZE

```
#define PSA_MAC_FINAL_SIZE(
                key_type,
                key_bits,
                alg )
```

**Value:**

```
(PSA_ALG_IS_HMAC(alg) ? PSA_HASH_SIZE(PSA_ALG_HMAC_HASH(alg)) : \
     PSA_ALG_IS_BLOCK_CIPHER_MAC(alg) ? PSA_BLOCK_CIPHER_BLOCK_SIZE(key_type) :
      \
     0)
```

The size of the output of psa_mac_sign_finish(), in bytes.

This is also the MAC size that psa_mac_verify_finish() expects.

**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.

#### 6.2.2.9 PSA_MAC_MAX_SIZE

```
#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.

# Index