Platform Security Architecture — cryptography and keystore interface

beta 1 — 2019-01-21

Generated by Doxygen 1.8.11

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_handle_t

  Key handle.

4.1.1 Detailed Description

4.1.2 Typedef Documentation

4.1.2.1 typedef _unsigned_integral_type_ **psa_key_handle_t**

Key handle.

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

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

4.2 Library initialization

Functions

- psa_status_t psa_crypto_init (void)

  Library initialization.

4.2.1 Detailed Description

4.2.2 Function Documentation

4.2.2.1 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.

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

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 policies

Macros

- #define PSA_KEY_POLICY_INIT {0}
- #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 struct psa_key_policy_s psa_key_policy_t
- typedef uint32_t psa_key_usage_t

  Encoding of permitted usage on a key.

Functions

- 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_handle_t handle, const psa_key_policy_t ∗policy)

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

  Get the usage policy for a key slot.

4.3.1 Detailed Description

4.3.2 Macro Definition Documentation

4.3.2.1 #define PSA_KEY_POLICY_INIT {0}

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

4.3.2.2 #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.

4.3.2.3 #define PSA_KEY_USAGE_DERIVE ((psa_key_usage_t)0x00001000)

Whether the key may be used to derive other keys.

4.3.2.4 #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.

4.3.2.5 #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.

4.3.2.6 #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.

4.3.2.7 #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.

4.3.3 Typedef Documentation

4.3.3.1 typedef struct psa_key_policy_s psa_key_policy_t

The type of the key policy data structure.

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

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

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

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

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

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

```
psa_key_policy_t policy = PSA_KEY_POLICY_INIT;
```

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

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

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

4.3.4 Function Documentation

4.3.4.1 psa_status_t psa_get_key_policy (psa_key_handle_t *handle*, psa_key_policy_t ∗ *policy*)

Get the usage policy for a key slot.

Parameters

	handle	Handle to the key slot whose policy is being queried.
`out`	*policy*	On success, the key's policy.

Return values

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

4.3.4.2 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.

4.3.4.3 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.

4.3.4.4 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

`in,out`	*policy*	The key policy to modify. It must have been initialized as per the documentation for psa_key_policy_t.
	usage	The permitted uses for the key.
	alg	The algorithm that the key may be used for.

4.3.4.5 psa_status_t psa_set_key_policy (psa_key_handle_t *handle*, 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

	handle	Handle to the key whose policy is to be changed.
`in`	*policy*	The policy object to query.

Return values

PSA_SUCCESS	Success. If the key is persistent, it is implementation-defined whether the policy has been saved to persistent storage. Implementations may defer saving the policy until the key material is created.
PSA_ERROR_INVALID_HANDLE	
PSA_ERROR_OCCUPIED_SLOT	
PSA_ERROR_NOT_SUPPORTED	
PSA_ERROR_INVALID_ARGUMENT	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_TAMPERING_DETECTED	
PSA_ERROR_BAD_STATE	The library has not been previously initialized by psa_crypto_init(). It is implementation-dependent whether a failure to initialize results in this error code.

4.4 Key management

Functions

- psa_status_t psa_get_key_lifetime (psa_key_handle_t handle, psa_key_lifetime_t ∗lifetime)

  Retrieve the lifetime of an open key.
- psa_status_t psa_allocate_key (psa_key_handle_t ∗handle)
- psa_status_t psa_open_key (psa_key_lifetime_t lifetime, psa_key_id_t id, psa_key_handle_t ∗handle)
- psa_status_t psa_create_key (psa_key_lifetime_t lifetime, psa_key_id_t id, psa_key_handle_t ∗handle)
- psa_status_t psa_close_key (psa_key_handle_t handle)

4.4.1 Detailed Description

4.4.2 Function Documentation

4.4.2.1 psa_status_t psa_allocate_key (psa_key_handle_t ∗ *handle*)

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

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

Parameters

`out`	*handle*	On success, a handle to a volatile key slot.

Return values

PSA_SUCCESS	Success. The application can now use the value of `*handle` to access the newly allocated key slot.
PSA_ERROR_INSUFFICIENT_MEMORY	There was not enough memory, or the maximum number of key slots has been reached.

4.4.2.2 psa_status_t psa_close_key (psa_key_handle_t *handle*)

Close a key handle.

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

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

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

Parameters

handle	The key handle to close.

Return values

PSA_SUCCESS	
PSA_ERROR_INVALID_HANDLE	
PSA_ERROR_COMMUNICATION_FAILURE	

4.4.2.3 psa_status_t psa_create_key (psa_key_lifetime_t *lifetime,* psa_key_id_t *id,* psa_key_handle_t ∗ *handle*)

Create a new persistent key slot.

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

Parameters

	lifetime	The lifetime of the key. This designates a storage area where the key material is stored. This must not be PSA_KEY_LIFETIME_VOLATILE.
	id	The persistent identifier of the key.
`out`	*handle*	On success, a handle to the newly created key slot. When key material is later created in this key slot, it will be saved to the specified persistent location.

Return values

PSA_SUCCESS	Success. The application can now use the value of `*handle` to access the newly allocated key slot.
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_INSUFFICIENT_STORAGE	
PSA_ERROR_OCCUPIED_SLOT	There is already a key with the identifier `id` in the storage area designated by `lifetime`.
PSA_ERROR_INVALID_ARGUMENT	`lifetime` is invalid, for example PSA_KEY_LIFETIME_VOLATILE.
PSA_ERROR_INVALID_ARGUMENT	`id` is invalid for the specified lifetime.
PSA_ERROR_NOT_SUPPORTED	`lifetime` is not supported.
PSA_ERROR_NOT_PERMITTED	`lifetime` is valid, but the application does not have the permission to create a key there.

4.4.2.4 psa_status_t psa_get_key_lifetime (psa_key_handle_t *handle,* psa_key_lifetime_t ∗ *lifetime*)

Retrieve the lifetime of an open key.

Parameters

	handle	Handle to query.
`out`	*lifetime*	On success, the lifetime value.

Return values

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

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

Open a handle to an existing persistent key.

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

Parameters

	lifetime	The lifetime of the key. This designates a storage area where the key material is stored. This must not be PSA_KEY_LIFETIME_VOLATILE.
	id	The persistent identifier of the key.
`out`	*handle*	On success, a handle to a key slot which contains the data and metadata loaded from the specified persistent location.

Return values

PSA_SUCCESS	Success. The application can now use the value of `*handle` to access the newly allocated key slot.
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_EMPTY_SLOT	
PSA_ERROR_INVALID_ARGUMENT	`lifetime` is invalid, for example PSA_KEY_LIFETIME_VOLATILE.
PSA_ERROR_INVALID_ARGUMENT	`id` is invalid for the specified lifetime.
PSA_ERROR_NOT_SUPPORTED	`lifetime` is not supported.
PSA_ERROR_NOT_PERMITTED	The specified key exists, but the application does not have the permission to access it. Note that this specification does not define any way to create such a key, but it may be possible through implementation-specific means.

4.5 Key import and export

Functions

- psa_status_t psa_import_key (psa_key_handle_t handle, 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_handle_t handle)

  Destroy a key.
- psa_status_t psa_get_key_information (psa_key_handle_t handle, psa_key_type_t ∗type, size_t ∗bits)

  Get basic metadata about a key.
- psa_status_t psa_set_key_domain_parameters (psa_key_handle_t handle, psa_key_type_t type, const uint8_t ∗data, size_t data_length)

  Set domain parameters for a key.
- psa_status_t psa_get_key_domain_parameters (psa_key_handle_t handle, uint8_t ∗data, size_t data_size, size_t ∗data_length)

  Get domain parameters for a key.
- psa_status_t psa_export_key (psa_key_handle_t handle, 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_handle_t handle, 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.
- psa_status_t psa_copy_key (psa_key_handle_t source_handle, psa_key_handle_t target_handle, const psa_key_policy_t ∗constraint)

4.5.1 Detailed Description

4.5.2 Function Documentation

4.5.2.1 psa_status_t psa_copy_key (psa_key_handle_t *source_handle,* psa_key_handle_t *target_handle,* const psa_key_policy_t ∗ *constraint*)

Make a copy of a key.

Copy key material from one location to another.

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

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

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

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

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

Parameters

	source_handle	The key to copy. It must be a handle to an occupied slot.
	target_handle	A handle to the target slot. It must not contain key material yet.
`in`	*constraint*	An optional policy constraint. If this parameter is non-null then the resulting key will conform to this policy in addition to the source policy and the policy already present on the target slot. If this parameter is null then the function behaves in the same way as if it was the target policy, i.e. only the source and target policies apply.

Return values

PSA_SUCCESS	
PSA_ERROR_INVALID_HANDLE	
PSA_ERROR_OCCUPIED_SLOT	`target` already contains key material.
PSA_ERROR_EMPTY_SLOT	`source` does not contain key material.
PSA_ERROR_INVALID_ARGUMENT	The policy constraints on the source, on the target and `constraints` are incompatible.
PSA_ERROR_NOT_PERMITTED	The source key is not exportable and its lifetime does not allow copying it to the target's lifetime.
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_INSUFFICIENT_STORAGE	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_TAMPERING_DETECTED	

4.5.2.2 psa_status_t psa_destroy_key (psa_key_handle_t *handle*)

Destroy a key.

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

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

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

Parameters

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

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

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

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

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

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

Parameters

	handle	Handle to the key to export.
`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_INVALID_HANDLE	
PSA_ERROR_EMPTY_SLOT	
PSA_ERROR_NOT_PERMITTED	
PSA_ERROR_NOT_SUPPORTED	
PSA_ERROR_BUFFER_TOO_SMALL	The size of the `data` buffer is too small. You can determine a sufficient buffer size by calling PSA_KEY_EXPORT_MAX_SIZE(`type`, `bits`) where `type` is the key type and `bits` is the key size in bits.
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_TAMPERING_DETECTED	
PSA_ERROR_BAD_STATE	The library has not been previously initialized by psa_crypto_init(). It is implementation-dependent whether a failure to initialize results in this error code.

4.5.2.4 psa_status_t psa_export_public_key (psa_key_handle_t *handle,* 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.

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

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

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

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

- For elliptic curve public keys (key types for which PSA_KEY_TYPE_IS_ECC_PUBLIC_KEY is true), the format is the uncompressed representation defined by SEC1 §2.3.3 as the content of an ECPoint. Let `m` be the bit size associated with the curve, i.e. the bit size of `q` for a curve over `F_q`. The representation consists of:
  - The byte 0x04;
  - `x_P` as a `ceiling(m/8)`-byte string, big-endian;

  - `y_P` as a `ceiling(m/8)`-byte string, big-endian.

- For DSA public keys (PSA_KEY_TYPE_DSA_PUBLIC_KEY), the format is the representation of the public key `y = g^x mod p` as a big-endian byte string. The length of the byte string is the length of the base prime `p` in bytes.
- For Diffie-Hellman key exchange public keys (PSA_KEY_TYPE_DH_PUBLIC_KEY), the format is the representation of the public key `y = g^x mod p` as a big-endian byte string. The length of the byte string is the length of the base prime `p` in bytes.

Parameters

	handle	Handle to the key to export.
`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_INVALID_HANDLE	
PSA_ERROR_EMPTY_SLOT	
PSA_ERROR_INVALID_ARGUMENT	The key is neither a public key nor a key pair.
PSA_ERROR_NOT_SUPPORTED	
PSA_ERROR_BUFFER_TOO_SMALL	The size of the `data` buffer is too small. You can determine a sufficient buffer size by calling PSA_KEY_EXPORT_MAX_S↩ IZE(PSA_KEY_TYPE_PUBLIC_KEY_OF_KEYPAIR(`type`), `bits`) where `type` is the key type and `bits` is the key size in bits.
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_TAMPERING_DETECTED	
PSA_ERROR_BAD_STATE	The library has not been previously initialized by psa_crypto_init(). It is implementation-dependent whether a failure to initialize results in this error code.

4.5.2.5 psa_status_t psa_get_key_domain_parameters (psa_key_handle_t *handle,* uint8_t ∗ *data,* size_t *data_size,* size_t ∗ *data_length*)

Get domain parameters for a key.

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

Parameters

	handle	Handle to the key to get domain parameters from.
`out`	*data*	On success, the key domain parameters.
	data_size	Size of the `data` buffer in bytes.
`out`	*data_length*	On success, the number of bytes that make up the key domain parameters data.

Return values

PSA_SUCCESS	
PSA_ERROR_INVALID_HANDLE	
PSA_ERROR_EMPTY_SLOT	There is no key in the specified slot.
PSA_ERROR_INVALID_ARGUMENT	
PSA_ERROR_NOT_SUPPORTED	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_TAMPERING_DETECTED	
PSA_ERROR_BAD_STATE	The library has not been previously initialized by psa_crypto_init(). It is implementation-dependent whether a failure to initialize results in this error code.

4.5.2.6 psa_status_t psa_get_key_information (psa_key_handle_t *handle*, psa_key_type_t ∗ *type*, size_t ∗ *bits*)

Get basic metadata about a key.

Parameters

	handle	Handle to the key slot to query.
`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_INVALID_HANDLE	
PSA_ERROR_EMPTY_SLOT	The handle is to a key slot which does not contain key material yet.
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_TAMPERING_DETECTED	
PSA_ERROR_BAD_STATE	The library has not been previously initialized by psa_crypto_init(). It is implementation-dependent whether a failure to initialize results in this error code.

4.5.2.7 psa_status_t psa_import_key (psa_key_handle_t *handle*, 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_public_key() for the format of public keys and to the documentation of psa_export_key() for the format for other key types.

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

Parameters

	handle	Handle to the slot where the key will be stored. It must have been obtained by calling psa_allocate_key() or psa_create_key() and must not contain key material yet.
	type	Key type (a `PSA_KEY_TYPE_XXX` value). On a successful import, the key slot will contain a key of this type.
`in`	*data*	Buffer containing the key data. The content of this buffer is interpreted according to `type`. It must contain the format described in the documentation of psa_export_key() or psa_export_public_key() for the chosen type.
	data_length	Size of the `data` buffer in bytes.

Return values

PSA_SUCCESS	Success. If the key is persistent, the key material and the key's metadata have been saved to persistent storage.
PSA_ERROR_INVALID_HANDLE	
PSA_ERROR_NOT_SUPPORTED	The key type or key size is not supported, either by the implementation in general or in this particular slot.
PSA_ERROR_INVALID_ARGUMENT	The key slot is invalid, or the key data is not correctly formatted.
PSA_ERROR_OCCUPIED_SLOT	There is already a key in the specified slot.
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_INSUFFICIENT_STORAGE	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_STORAGE_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_TAMPERING_DETECTED	
PSA_ERROR_BAD_STATE	The library has not been previously initialized by psa_crypto_init(). It is implementation-dependent whether a failure to initialize results in this error code.

4.5.2.8 psa_status_t psa_set_key_domain_parameters (psa_key_handle_t *handle*, psa_key_type_t *type*, const uint8_t ∗ *data*, size_t *data_length*)

Set domain parameters for a key.

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

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

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

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

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

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

Parameters

	handle	Handle to the slot where the key will be stored. This must be a valid slot for a key of the chosen type: it must have been obtained by calling psa_allocate_key() or psa_create_key() with the correct `type` and with a maximum size that is compatible with `data`. It must not contain key material yet.
	type	Key type (a `PSA_KEY_TYPE_XXX` value). When subsequently creating key material into `handle`, the type must be compatible.
`in`	*data*	Buffer containing the key domain parameters. The content of this buffer is interpreted according to `type`. of psa_export_key() or psa_export_public_key() for the chosen type.
	data_length	Size of the `data` buffer in bytes.

Return values

PSA_SUCCESS	
PSA_ERROR_INVALID_HANDLE	
PSA_ERROR_OCCUPIED_SLOT	There is already a key in the specified slot.
PSA_ERROR_INVALID_ARGUMENT	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_TAMPERING_DETECTED	
PSA_ERROR_BAD_STATE	The library has not been previously initialized by psa_crypto_init(). It is implementation-dependent whether a failure to initialize results in this error code.

4.6 Message digests

Macros

- #define PSA_HASH_OPERATION_INIT {0}

Typedefs

- typedef struct psa_hash_operation_s psa_hash_operation_t

Functions

- psa_status_t psa_hash_compute (psa_algorithm_t alg, const uint8_t ∗input, size_t input_length, uint8_↩
t ∗hash, size_t hash_size, size_t ∗hash_length)
- psa_status_t psa_hash_compare (psa_algorithm_t alg, const uint8_t ∗input, size_t input_length, const uint8_t ∗hash, const size_t hash_length)
- 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_hash_clone (const psa_hash_operation_t ∗source_operation, psa_hash_operation_↩
t ∗target_operation)

4.6.1 Detailed Description

4.6.2 Macro Definition Documentation

4.6.2.1 #define PSA_HASH_OPERATION_INIT {0}

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

4.6.3 Typedef Documentation

4.6.3.1 typedef struct psa_hash_operation_s psa_hash_operation_t

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

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

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

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

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

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

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

```
psa_hash_operation_t operation = PSA_HASH_OPERATION_INIT;
```

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

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

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

4.6.4 Function Documentation

4.6.4.1 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	

4.6.4.2 psa_status_t psa_hash_clone (const psa_hash_operation_t * *source_operation,* psa_hash_operation_t * *target_operation*)

Clone a hash operation.

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

Parameters

`in`	*source_operation*	The active hash operation to clone.
`in,out`	*target_operation*	The operation object to set up. It must be initialized but not active.

Return values

PSA_SUCCESS	
PSA_ERROR_BAD_STATE	`source_operation` is not an active hash operation.
PSA_ERROR_BAD_STATE	`target_operation` is active.
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_TAMPERING_DETECTED	

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

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

Parameters

	alg	The hash algorithm to compute (`PSA_ALG_XXX` value such that PSA_ALG_IS_HASH(`alg`) is true).
`in`	*input*	Buffer containing the message to hash.
	input_length	Size of the `input` buffer in bytes.
`out`	*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 input.
PSA_ERROR_INVALID_SIGNATURE	The hash of the message was calculated successfully, but it differs from the expected hash.
PSA_ERROR_NOT_SUPPORTED	`alg` is not supported or is not a hash algorithm.
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_TAMPERING_DETECTED	

4.6.4.4 psa_status_t psa_hash_compute (psa_algorithm_t *alg,* const uint8_t * *input,* size_t *input_length,* uint8_t * *hash,* size_t *hash_size,* size_t * *hash_length*)

Calculate the hash (digest) of a message.

Note

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

Parameters

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

Parameters

in	*input*	Buffer containing the message to hash.
	input_length	Size of the `input` buffer in bytes.
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_NOT_SUPPORTED	`alg` is not supported or is not a hash algorithm.
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_TAMPERING_DETECTED	

4.6.4.5 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 set up, 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	

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

Set up a multipart hash operation.

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

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

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

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

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

Parameters

`in,out`	*operation*	The operation object to set up. It must have been initialized as per the documentation for psa_hash_operation_t and not yet in use.
	alg	The hash algorithm to compute (`PSA_ALG_XXX` value such that PSA_ALG_IS_HASH(`alg`) is true).

Return values

PSA_SUCCESS	Success.
PSA_ERROR_NOT_SUPPORTED	`alg` is not supported or is not a hash algorithm.
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	

Return values

PSA_ERROR_TAMPERING_DETECTED	

4.6.4.7 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 set up, or already completed).
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_TAMPERING_DETECTED	

4.6.4.8 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 set up, or already completed).
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_TAMPERING_DETECTED	

4.7 Message authentication codes

Macros

- #define PSA_MAC_OPERATION_INIT {0}

Typedefs

- typedef struct psa_mac_operation_s psa_mac_operation_t

Functions

- psa_status_t psa_mac_compute (psa_key_handle_t handle, psa_algorithm_t alg, const uint8_t ∗input, size↩_t input_length, uint8_t ∗mac, size_t mac_size, size_t ∗mac_length)
- psa_status_t psa_mac_verify (psa_key_handle_t handle, psa_algorithm_t alg, const uint8_t ∗input, size_t input_length, const uint8_t ∗mac, const size_t mac_length)
- psa_status_t psa_mac_sign_setup (psa_mac_operation_t ∗operation, psa_key_handle_t handle, psa_↩algorithm_t alg)
- psa_status_t psa_mac_verify_setup (psa_mac_operation_t ∗operation, psa_key_handle_t handle, 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)

4.7.1 Detailed Description

4.7.2 Macro Definition Documentation

4.7.2.1 #define PSA_MAC_OPERATION_INIT {0}

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

4.7.3 Typedef Documentation

4.7.3.1 typedef struct psa_mac_operation_s psa_mac_operation_t

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

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

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

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

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

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

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

```
psa_mac_operation_t operation = PSA_MAC_OPERATION_INIT;
```

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

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

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

4.7.4 Function Documentation

4.7.4.1 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	

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

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

Note

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

Parameters

	handle	Handle to the key to use for the operation.
	alg	The MAC algorithm to compute (`PSA_ALG_XXX` value such that PSA_ALG_IS_MAC(alg) is true).
`in`	*input*	Buffer containing the input message.
	input_length	Size of the `input` buffer in bytes.
`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_HASH_SIZE(`alg`) where `alg` is the hash algorithm that is calculated.

Return values

PSA_SUCCESS	Success.
PSA_ERROR_INVALID_HANDLE	
PSA_ERROR_EMPTY_SLOT	
PSA_ERROR_NOT_PERMITTED	
PSA_ERROR_INVALID_ARGUMENT	`key` is not compatible with `alg`.
PSA_ERROR_NOT_SUPPORTED	`alg` is not supported or is not a MAC algorithm.
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_TAMPERING_DETECTED	
PSA_ERROR_BAD_STATE	The library has not been previously initialized by psa_crypto_init(). It is implementation-dependent whether a failure to initialize results in this error code.

4.7.4.3 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.
PSA_ERROR_BAD_STATE	The operation state is not valid (not set up, or already completed).
PSA_ERROR_BUFFER_TOO_SMALL	The size of the `mac` buffer is too small. You can determine a sufficient buffer size by calling PSA_MAC_FINAL_SIZE().
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_TAMPERING_DETECTED	

4.7.4.4 psa_status_t psa_mac_sign_setup (psa_mac_operation_t ∗ *operation,* psa_key_handle_t *handle,* psa_algorithm_t *alg*)

Set up a multipart MAC calculation operation.

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

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

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

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

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

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

Parameters

`in,out`	*operation*	The operation object to set up. It must have been initialized as per the documentation for psa_mac_operation_t and not yet in use.
	handle	Handle to the key to use for the operation. It must remain valid until the operation terminates.
	alg	The MAC algorithm to compute (`PSA_ALG_XXX` value such that PSA_ALG_IS_MAC(alg) is true).

Return values

PSA_SUCCESS	Success.
PSA_ERROR_INVALID_HANDLE	
PSA_ERROR_EMPTY_SLOT	
PSA_ERROR_NOT_PERMITTED	
PSA_ERROR_INVALID_ARGUMENT	`key` is not compatible with `alg`.
PSA_ERROR_NOT_SUPPORTED	`alg` is not supported or is not a MAC algorithm.
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_TAMPERING_DETECTED	
PSA_ERROR_BAD_STATE	The library has not been previously initialized by psa_crypto_init(). It is implementation-dependent whether a failure to initialize results in this error code.

4.7.4.5 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 set up, or already completed).
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_TAMPERING_DETECTED	

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

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

Parameters

	handle	Handle to the key to use for the operation.
	alg	The MAC algorithm to compute (`PSA_ALG_XXX` value such that PSA_ALG_IS_MAC(alg) is true).
`in`	*input*	Buffer containing the input message.
	input_length	Size of the `input` buffer in bytes.
`out`	*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 input.
PSA_ERROR_INVALID_SIGNATURE	The MAC of the message was calculated successfully, but it differs from the expected value.
PSA_ERROR_INVALID_HANDLE	
PSA_ERROR_EMPTY_SLOT	
PSA_ERROR_NOT_PERMITTED	
PSA_ERROR_INVALID_ARGUMENT	`key` is not compatible with `alg`.
PSA_ERROR_NOT_SUPPORTED	`alg` is not supported or is not a MAC algorithm.
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_TAMPERING_DETECTED	

4.7.4.7 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 set up, or already completed).
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_TAMPERING_DETECTED	

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

Set up a multipart MAC verification operation.

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

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

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

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

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

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

Parameters

`in,out`	*operation*	The operation object to set up. It must have been initialized as per the documentation for psa_mac_operation_t and not yet in use.
	handle	Handle to the key to use for the operation. It must remain valid until the operation terminates.
	alg	The MAC algorithm to compute (`PSA_ALG_XXX` value such that PSA_ALG_IS_MAC(`alg`) is true).

Return values

PSA_SUCCESS	Success.
PSA_ERROR_INVALID_HANDLE	
PSA_ERROR_EMPTY_SLOT	
PSA_ERROR_NOT_PERMITTED	
PSA_ERROR_INVALID_ARGUMENT	`key` is not compatible with `alg`.
PSA_ERROR_NOT_SUPPORTED	`alg` is not supported or is not a MAC algorithm.
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_TAMPERING_DETECTED	
PSA_ERROR_BAD_STATE	The library has not been previously initialized by psa_crypto_init(). It is implementation-dependent whether a failure to initialize results in this error code.

4.8 Symmetric ciphers

Macros

- #define PSA_CIPHER_OPERATION_INIT {0}

Typedefs

- typedef struct psa_cipher_operation_s psa_cipher_operation_t

Functions

- psa_status_t psa_cipher_encrypt (psa_key_handle_t handle, psa_algorithm_t alg, const uint8_t ∗input, size_t input_length, uint8_t ∗output, size_t output_size, size_t ∗output_length)
- psa_status_t psa_cipher_decrypt (psa_key_handle_t handle, psa_algorithm_t alg, const uint8_t ∗input, size_t input_length, uint8_t ∗output, size_t output_size, size_t ∗output_length)
- psa_status_t psa_cipher_encrypt_setup (psa_cipher_operation_t ∗operation, psa_key_handle_t handle, psa_algorithm_t alg)
- psa_status_t psa_cipher_decrypt_setup (psa_cipher_operation_t ∗operation, psa_key_handle_t handle, 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)

4.8.1 Detailed Description

4.8.2 Macro Definition Documentation

4.8.2.1 #define PSA_CIPHER_OPERATION_INIT {0}

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

4.8.3 Typedef Documentation

4.8.3.1 typedef struct psa_cipher_operation_s psa_cipher_operation_t

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

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

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

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

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

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

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

```
psa_cipher_operation_t operation = PSA_CIPHER_OPERATION_INIT;
```

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

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

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

4.8.4 Function Documentation

4.8.4.1 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	

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

Decrypt a message using a symmetric cipher.

This function decrypts a message encrypted with a symmetric cipher.

Parameters

	handle	Handle to the key to use for the operation. It must remain valid until the operation terminates.
	alg	The cipher algorithm to compute (`PSA_ALG_XXX` value such that PSA_ALG_IS_CIPHER(`alg`) is true).
`in`	*input*	Buffer containing the message to decrypt. This consists of the IV followed by the ciphertext proper.
	input_length	Size of the `input` buffer in bytes.
`out`	*output*	Buffer where the plaintext 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 output.

Return values

PSA_SUCCESS	Success.
PSA_ERROR_INVALID_HANDLE	
PSA_ERROR_EMPTY_SLOT	
PSA_ERROR_NOT_PERMITTED	
PSA_ERROR_INVALID_ARGUMENT	`key` is not compatible with `alg`.
PSA_ERROR_NOT_SUPPORTED	`alg` is not supported or is not a cipher algorithm.
PSA_ERROR_BUFFER_TOO_SMALL	
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_TAMPERING_DETECTED	

4.8.4.3 psa_status_t psa_cipher_decrypt_setup (psa_cipher_operation_t * *operation,* psa_key_handle_t *handle,* 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. Initialize the operation object with one of the methods described in the documentation for psa_cipher_↩ operation_t, e.g. PSA_CIPHER_OPERATION_INIT.
3. Call psa_cipher_decrypt_setup() to specify the algorithm and key.
4. Call psa_cipher_set_iv() with the IV (initialization vector) for the decryption. If the IV is prepended to the ciphertext, you can call psa_cipher_update() on a buffer containing the IV followed by the beginning of the message.
5. Call psa_cipher_update() zero, one or more times, passing a fragment of the message each time.
6. Call psa_cipher_finish().

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

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

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

Parameters

`in,out`	*operation*	The operation object to set up. It must have been initialized as per the documentation for psa_cipher_operation_t and not yet in use.
	handle	Handle to the key to use for the operation. It must remain valid until the operation terminates.
	alg	The cipher algorithm to compute (`PSA_ALG_XXX` value such that PSA_ALG_IS_CIPHER(`alg`) is true).

Return values

PSA_SUCCESS	Success.
PSA_ERROR_INVALID_HANDLE	
PSA_ERROR_EMPTY_SLOT	
PSA_ERROR_NOT_PERMITTED	
PSA_ERROR_INVALID_ARGUMENT	`key` is not compatible with `alg`.
PSA_ERROR_NOT_SUPPORTED	`alg` is not supported or is not a cipher algorithm.
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_TAMPERING_DETECTED	

Return values

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

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

Encrypt a message using a symmetric cipher.

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

Parameters

	handle	Handle to the key to use for the operation. It must remain valid until the operation terminates.
	alg	The cipher algorithm to compute (`PSA_ALG_XXX` value such that PSA_ALG_IS_CIPHER(`alg`) is true).
`in`	*input*	Buffer containing the message to encrypt.
	input_length	Size of the `input` buffer in bytes.
`out`	*output*	Buffer where the output is to be written. The output contains the IV followed by the ciphertext proper.
	output_size	Size of the `output` buffer in bytes.
`out`	*output_length*	On success, the number of bytes that make up the output.

Return values

PSA_SUCCESS	Success.
PSA_ERROR_INVALID_HANDLE	
PSA_ERROR_EMPTY_SLOT	
PSA_ERROR_NOT_PERMITTED	
PSA_ERROR_INVALID_ARGUMENT	`key` is not compatible with `alg`.
PSA_ERROR_NOT_SUPPORTED	`alg` is not supported or is not a cipher algorithm.
PSA_ERROR_BUFFER_TOO_SMALL	
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_TAMPERING_DETECTED	

4.8.4.5 psa_status_t psa_cipher_encrypt_setup (psa_cipher_operation_t ∗ *operation*, psa_key_handle_t *handle*, 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. Initialize the operation object with one of the methods described in the documentation for psa_cipher_↩ operation_t, e.g. PSA_CIPHER_OPERATION_INIT.
3. Call psa_cipher_encrypt_setup() to specify the algorithm and key.
4. Call either psa_cipher_generate_iv() or psa_cipher_set_iv() to generate or set the IV (initialization vector). You should use psa_cipher_generate_iv() unless the protocol you are implementing requires a specific IV value.
5. Call psa_cipher_update() zero, one or more times, passing a fragment of the message each time.
6. Call psa_cipher_finish().

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

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

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

Parameters

`in,out`	*operation*	The operation object to set up. It must have been initialized as per the documentation for psa_cipher_operation_t and not yet in use.
	handle	Handle to the key to use for the operation. It must remain valid until the operation terminates.
	alg	The cipher algorithm to compute (`PSA_ALG_XXX` value such that PSA_ALG_IS_CIPHER(`alg`) is true).

Return values

PSA_SUCCESS	Success.
PSA_ERROR_INVALID_HANDLE	
PSA_ERROR_EMPTY_SLOT	
PSA_ERROR_NOT_PERMITTED	
PSA_ERROR_INVALID_ARGUMENT	`key` is not compatible with `alg`.
PSA_ERROR_NOT_SUPPORTED	`alg` is not supported or is not a cipher algorithm.
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_TAMPERING_DETECTED	
PSA_ERROR_BAD_STATE	The library has not been previously initialized by psa_crypto_init(). It is implementation-dependent whether a failure to initialize results in this error code.

4.8.4.6 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 set up, IV required but not set, or already completed).
PSA_ERROR_BUFFER_TOO_SMALL	The size of the `output` buffer is too small.
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_TAMPERING_DETECTED	

4.8.4.7 **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 set up, or IV already set).
PSA_ERROR_BUFFER_TOO_SMALL	The size of the `iv` buffer is too small.

Return values

PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_TAMPERING_DETECTED	

4.8.4.8 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 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 set up, or IV already set).
PSA_ERROR_INVALID_ARGUMENT	The size of `iv` is not acceptable for the chosen algorithm, or the chosen algorithm does not use an IV.
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_TAMPERING_DETECTED	

4.8.4.9 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 set up, IV required but not set, or already completed).
PSA_ERROR_BUFFER_TOO_SMALL	The size of the `output` buffer is too small.
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_TAMPERING_DETECTED	

4.9 Authenticated encryption with associated data (AEAD)

Macros

- #define PSA_AEAD_OPERATION_INIT {0}

Typedefs

- typedef struct psa_aead_operation_s psa_aead_operation_t

Functions

- psa_status_t psa_aead_encrypt (psa_key_handle_t handle, psa_algorithm_t alg, const uint8_t ∗nonce, size_t nonce_length, const uint8_t ∗additional_data, size_t additional_data_length, const uint8_t ∗plaintext, size_t plaintext_length, uint8_t ∗ciphertext, size_t ciphertext_size, size_t ∗ciphertext_length)
- psa_status_t psa_aead_decrypt (psa_key_handle_t handle, psa_algorithm_t alg, const uint8_t ∗nonce, size_t nonce_length, const uint8_t ∗additional_data, size_t additional_data_length, const uint8_t ∗ciphertext, size_t ciphertext_length, uint8_t ∗plaintext, size_t plaintext_size, size_t ∗plaintext_length)
- psa_status_t psa_aead_encrypt_setup (psa_aead_operation_t ∗operation, psa_key_handle_t handle, psa←_algorithm_t alg)
- psa_status_t psa_aead_decrypt_setup (psa_aead_operation_t ∗operation, psa_key_handle_t handle, psa←_algorithm_t alg)
- psa_status_t psa_aead_generate_nonce (psa_aead_operation_t ∗operation, unsigned char ∗nonce, size_t nonce_size, size_t ∗nonce_length)
- psa_status_t psa_aead_set_nonce (psa_aead_operation_t ∗operation, const unsigned char ∗nonce, size_t nonce_length)
- psa_status_t psa_aead_set_lengths (psa_aead_operation_t ∗operation, size_t ad_length, size_t plaintext←_length)
- psa_status_t psa_aead_update_ad (psa_aead_operation_t ∗operation, const uint8_t ∗input, size_t input_←length)
- psa_status_t psa_aead_update (psa_aead_operation_t ∗operation, const uint8_t ∗input, size_t input_length, unsigned char ∗output, size_t output_size, size_t ∗output_length)
- psa_status_t psa_aead_finish (psa_aead_operation_t ∗operation, uint8_t ∗ciphertext, size_t ciphertext_size, size_t ∗ciphertext_length, uint8_t ∗tag, size_t tag_size, size_t ∗tag_length)
- psa_status_t psa_aead_verify (psa_aead_operation_t ∗operation, const uint8_t ∗tag, size_t tag_length)
- psa_status_t psa_aead_abort (psa_aead_operation_t ∗operation)

4.9.1 Detailed Description

4.9.2 Macro Definition Documentation

4.9.2.1 #define PSA_AEAD_OPERATION_INIT {0}

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

4.9.3 Typedef Documentation

4.9.3.1 typedef struct psa_aead_operation_s psa_aead_operation_t

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

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

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

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

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

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

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

```
psa_aead_operation_t operation = PSA_AEAD_OPERATION_INIT;
```

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

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

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

4.9.4 Function Documentation

4.9.4.1 psa_status_t psa_aead_abort (psa_aead_operation_t ∗ *operation*)

Abort an AEAD operation.

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

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

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

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

Parameters

`in,out`	*operation*	Initialized AEAD operation.

Return values

PSA_SUCCESS	
PSA_ERROR_BAD_STATE	`operation` is not an active AEAD operation.
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_TAMPERING_DETECTED	

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

Process an authenticated decryption operation.

Parameters

	handle	Handle to the key to use for the operation.
	alg	The AEAD algorithm to compute (`PSA_ALG_XXX` value such that PSA_ALG_IS_AEAD(`alg`) is true).
`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_INVALID_HANDLE	
PSA_ERROR_EMPTY_SLOT	
PSA_ERROR_INVALID_SIGNATURE	The ciphertext is not authentic.
PSA_ERROR_NOT_PERMITTED	
PSA_ERROR_INVALID_ARGUMENT	`key` is not compatible with `alg`.
PSA_ERROR_NOT_SUPPORTED	`alg` is not supported or is not an AEAD algorithm.
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_TAMPERING_DETECTED	

Return values

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

4.9.4.3 psa_status_t psa_aead_decrypt_setup (psa_aead_operation_t ∗ *operation,* psa_key_handle_t *handle,* psa_algorithm_t *alg*)

Set the key for a multipart authenticated decryption operation.

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

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

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

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

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

Parameters

`in,out`	*operation*	The operation object to set up. It must have been initialized as per the documentation for psa_aead_operation_t and not yet in use.
	handle	Handle to the key to use for the operation. It must remain valid until the operation terminates.
	alg	The AEAD algorithm to compute (`PSA_ALG_XXX` value such that PSA_ALG_IS_AEAD(`alg`) is true).

Return values

PSA_SUCCESS	Success.
PSA_ERROR_INVALID_HANDLE	

Return values

PSA_ERROR_EMPTY_SLOT	
PSA_ERROR_NOT_PERMITTED	
PSA_ERROR_INVALID_ARGUMENT	`key` is not compatible with `alg`.
PSA_ERROR_NOT_SUPPORTED	`alg` is not supported or is not an AEAD algorithm.
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_TAMPERING_DETECTED	
PSA_ERROR_BAD_STATE	The library has not been previously initialized by psa_crypto_init(). It is implementation-dependent whether a failure to initialize results in this error code.

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

Process an authenticated encryption operation.

Parameters

	handle	Handle to the key to use for the operation.
	alg	The AEAD algorithm to compute (`PSA_ALG_XXX` value such that PSA_ALG_IS_AEAD(`alg`) is true).
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_INVALID_HANDLE	
PSA_ERROR_EMPTY_SLOT	
PSA_ERROR_NOT_PERMITTED	
PSA_ERROR_INVALID_ARGUMENT	`key` is not compatible with `alg`.
PSA_ERROR_NOT_SUPPORTED	`alg` is not supported or is not an AEAD algorithm.
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	

Return values

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

4.9.4.5 psa_status_t psa_aead_encrypt_setup (psa_aead_operation_t ∗ *operation*, psa_key_handle_t *handle*, psa_algorithm_t *alg*)

Set the key for a multipart authenticated encryption operation.

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

1. Allocate an operation object which will be passed to all the functions listed here.
2. Initialize the operation object with one of the methods described in the documentation for psa_aead_↩ operation_t, e.g. PSA_AEAD_OPERATION_INIT.
3. Call psa_aead_encrypt_setup() to specify the algorithm and key.
4. If needed, call psa_aead_set_lengths() to specify the length of the inputs to the subsequent calls to psa_↩ aead_update_ad() and psa_aead_update(). See the documentation of psa_aead_set_lengths() for details.
5. Call either psa_aead_generate_nonce() or psa_aead_set_nonce() to generate or set the nonce. You should use psa_aead_generate_nonce() unless the protocol you are implementing requires a specific nonce value.
6. Call psa_aead_update_ad() zero, one or more times, passing a fragment of the non-encrypted additional authenticated data each time.
7. Call psa_aead_update() zero, one or more times, passing a fragment of the message to encrypt each time.
8. Call psa_aead_finish().

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

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

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

Parameters

`in,out`	*operation*	The operation object to set up. It must have been initialized as per the documentation for psa_aead_operation_t and not yet in use.
	handle	Handle to the key to use for the operation. It must remain valid until the operation terminates.
	alg	The AEAD algorithm to compute (`PSA_ALG_XXX` value such that PSA_ALG_IS_AEAD(`alg`) is true).

Return values

PSA_SUCCESS	Success.
PSA_ERROR_INVALID_HANDLE	
PSA_ERROR_EMPTY_SLOT	
PSA_ERROR_NOT_PERMITTED	
PSA_ERROR_INVALID_ARGUMENT	`key` is not compatible with `alg`.
PSA_ERROR_NOT_SUPPORTED	`alg` is not supported or is not an AEAD algorithm.
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_TAMPERING_DETECTED	
PSA_ERROR_BAD_STATE	The library has not been previously initialized by psa_crypto_init(). It is implementation-dependent whether a failure to initialize results in this error code.

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

Finish encrypting a message in an AEAD operation.

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

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

This function has two output buffers:

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

When this function returns, the operation becomes inactive.

Parameters

`in,out`	*operation*	Active AEAD operation.
`out`	*ciphertext*	Buffer where the last part of the ciphertext is to be written.
	ciphertext_size	Size of the `ciphertext` buffer in bytes.
`out`	*ciphertext_length*	On success, the number of bytes of returned ciphertext.
`out`	*tag*	Buffer where the authentication tag is to be written.
	tag_size	Size of the `tag` buffer in bytes.
`out`	*tag_length*	On success, the number of bytes that make up the returned tag.

Return values

PSA_SUCCESS	Success.
PSA_ERROR_BAD_STATE	The operation state is not valid (not set up, nonce not set, decryption, or already completed).
PSA_ERROR_BUFFER_TOO_SMALL	The size of the `output` buffer is too small.
PSA_ERROR_INVALID_ARGUMENT	The total length of input to psa_aead_update_ad() so far is less than the additional data length that was previously specified with psa_aead_set_lengths().
PSA_ERROR_INVALID_ARGUMENT	The total length of input to psa_aead_update() so far is less than the plaintext length that was previously specified with psa_aead_set_lengths().
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_TAMPERING_DETECTED	

4.9.4.7 psa_status_t psa_aead_generate_nonce (psa_aead_operation_t ∗ *operation,* unsigned char ∗ *nonce,* size_t *nonce_size,* size_t ∗ *nonce_length*)

Generate a random nonce for an authenticated encryption operation.

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

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

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

Parameters

`in,out`	*operation*	Active AEAD operation.
`out`	*nonce*	Buffer where the generated nonce is to be written.
	nonce_size	Size of the `nonce` buffer in bytes.
`out`	*nonce_length*	On success, the number of bytes of the generated nonce.

Return values

PSA_SUCCESS	Success.
PSA_ERROR_BAD_STATE	The operation state is not valid (not set up, or nonce already set).
PSA_ERROR_BUFFER_TOO_SMALL	The size of the `nonce` buffer is too small.
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_TAMPERING_DETECTED	

4.9.4.8 psa_status_t psa_aead_set_lengths (psa_aead_operation_t ∗ *operation,* size_t *ad_length,* size_t *plaintext_length*)

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

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

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

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

Parameters

in,out	*operation*	Active AEAD operation.
	ad_length	Size of the non-encrypted additional authenticated data in bytes.
	plaintext_length	Size of the plaintext to encrypt in bytes.

Return values

PSA_SUCCESS	Success.
PSA_ERROR_BAD_STATE	The operation state is not valid (not set up, already completed, or psa_aead_update_ad() or psa_aead_update() already called).
PSA_ERROR_INVALID_ARGUMENT	At least one of the lengths is not acceptable for the chosen algorithm.
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_TAMPERING_DETECTED	

4.9.4.9 psa_status_t psa_aead_set_nonce (psa_aead_operation_t ∗ *operation,* const unsigned char ∗ *nonce,* size_t *nonce_length*)

Set the nonce for an authenticated encryption or decryption operation.

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

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

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

Note

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

Parameters

in,out	*operation*	Active AEAD operation.
in	*nonce*	Buffer containing the nonce to use.
	nonce_length	Size of the nonce in bytes.

Return values

PSA_SUCCESS	Success.
PSA_ERROR_BAD_STATE	The operation state is not valid (not set up, or nonce already set).
PSA_ERROR_INVALID_ARGUMENT	The size of `nonce` is not acceptable for the chosen algorithm.
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_TAMPERING_DETECTED	

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

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

Before calling this function, you must:

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

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

Warning

When decrypting, until psa_aead_verify() has returned PSA_SUCCESS, there is no guarantee that the input is valid. Therefore, until you have called psa_aead_verify() and it has returned PSA_SUCCESS:

- Do not use the output in any way other than storing it in a confidential location. If you take any action that depends on the tentative decrypted data, this action will need to be undone if the input turns out not to be valid. Furthermore, if an adversary can observe that this action took place (for example through timing), they may be able to use this fact as an oracle to decrypt any message encrypted with the same key.
- In particular, do not copy the output anywhere but to a memory or storage space that you have exclusive access to.

Parameters

in,out	*operation*	Active AEAD 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 set up, nonce not set or already completed).
PSA_ERROR_BUFFER_TOO_SMALL	The size of the `output` buffer is too small.
PSA_ERROR_INVALID_ARGUMENT	The total length of input to psa_aead_update_ad() so far is less than the additional data length that was previously specified with psa_aead_set_lengths().
PSA_ERROR_INVALID_ARGUMENT	The total input length overflows the plaintext length that was previously specified with psa_aead_set_lengths().
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_TAMPERING_DETECTED	

4.9.4.11 psa_status_t psa_aead_update_ad (psa_aead_operation_t ∗ *operation,* const uint8_t ∗ *input,* size_t *input_length*)

Pass additional data to an active AEAD operation.

Additional data is authenticated, but not encrypted.

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

Before calling this function, you must:

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

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

Warning

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

Parameters

`in,out`	*operation*	Active AEAD operation.
`in`	*input*	Buffer containing the fragment of additional data.
	input_length	Size of the `input` buffer in bytes.

Return values

PSA_SUCCESS	Success.

Return values

PSA_ERROR_BAD_STATE	The operation state is not valid (not set up, nonce not set, psa_aead_update() already called, or operation already completed).
PSA_ERROR_INVALID_ARGUMENT	The total input length overflows the additional data length that was previously specified with psa_aead_set_lengths().
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_TAMPERING_DETECTED	

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

Finish authenticating and decrypting a message in an AEAD operation.

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

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

When this function returns, the operation becomes inactive.

Parameters

`in,out`	*operation*	Active AEAD operation.
`in`	*tag*	Buffer containing the authentication tag.
	tag_length	Size of the `tag` buffer in bytes.

Return values

PSA_SUCCESS	Success.
PSA_ERROR_BAD_STATE	The operation state is not valid (not set up, nonce not set, encryption, or already completed).
PSA_ERROR_BUFFER_TOO_SMALL	The size of the `output` buffer is too small.
PSA_ERROR_INVALID_ARGUMENT	The total length of input to psa_aead_update_ad() so far is less than the additional data length that was previously specified with psa_aead_set_lengths().
PSA_ERROR_INVALID_ARGUMENT	The total length of input to psa_aead_update() so far is less than the plaintext length that was previously specified with psa_aead_set_lengths().
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_TAMPERING_DETECTED	

4.10 Asymmetric cryptography

Functions

- psa_status_t psa_asymmetric_sign (psa_key_handle_t handle, psa_algorithm_t alg, const uint8_t ∗hash, size_t hash_length, uint8_t ∗signature, size_t signature_size, size_t ∗signature_length)

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

  Verify the signature a hash or short message using a public key.
- psa_status_t psa_asymmetric_encrypt (psa_key_handle_t handle, psa_algorithm_t alg, const uint8_t ∗input, size_t input_length, const uint8_t ∗salt, size_t salt_length, uint8_t ∗output, size_t output_size, size_↩ t ∗output_length)

  Encrypt a short message with a public key.
- psa_status_t psa_asymmetric_decrypt (psa_key_handle_t handle, psa_algorithm_t alg, const uint8_t ∗input, size_t input_length, const uint8_t ∗salt, size_t salt_length, uint8_t ∗output, size_t output_size, size_↩ t ∗output_length)

  Decrypt a short message with a private key.

4.10.1 Detailed Description

4.10.2 Function Documentation

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

Decrypt a short message with a private key.

Parameters

	handle	Handle to the key to use for the operation. It must be an asymmetric key pair.
	alg	An asymmetric encryption algorithm that is compatible with the type of `key`.
`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	
PSA_ERROR_BAD_STATE	The library has not been previously initialized by psa_crypto_init(). It is implementation-dependent whether a failure to initialize results in this error code.

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

Encrypt a short message with a public key.

Parameters

	handle	Handle to the key to use for the operation. It must be a public key or an asymmetric key pair.
	alg	An asymmetric encryption algorithm that is compatible with the type of `key`.
`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_ASYMMETRIC_ENCRYPT_OUTPUT_SIZE(`key_type`, `key_bits`, `alg`) where `key_type` and `key_bits` are the type and bit-size respectively of `key`.
PSA_ERROR_NOT_SUPPORTED	
PSA_ERROR_INVALID_ARGUMENT	
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_TAMPERING_DETECTED	
PSA_ERROR_INSUFFICIENT_ENTROPY	
PSA_ERROR_BAD_STATE	The library has not been previously initialized by psa_crypto_init(). It is implementation-dependent whether a failure to initialize results in this error code.

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

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

	handle	Handle to the key to use for the operation. It must be an asymmetric key pair.
	alg	A signature algorithm that is compatible with the type of `key`.
`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	

Return values

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

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

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

	handle	Handle to the key to use for the operation. It must be a public key or an asymmetric key pair.
	alg	A signature algorithm that is compatible with the type of `key`.
`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	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_TAMPERING_DETECTED	
PSA_ERROR_BAD_STATE	The library has not been previously initialized by psa_crypto_init(). It is implementation-dependent whether a failure to initialize results in this error code.

4.11 Generators

Macros

- #define PSA_CRYPTO_GENERATOR_INIT {0}
- #define PSA_GENERATOR_UNBRIDLED_CAPACITY ((size_t)(-1))

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_set_generator_capacity (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_handle_t handle, psa_key_type_t type, size_t bits, psa↩ _crypto_generator_t ∗generator)
- psa_status_t psa_generator_abort (psa_crypto_generator_t ∗generator)

4.11.1 Detailed Description

4.11.2 Macro Definition Documentation

4.11.2.1 #define PSA_CRYPTO_GENERATOR_INIT {0}

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

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

Use the maximum possible capacity for a generator.

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

4.11.3 Typedef Documentation

4.11.3.1 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.

4.11.4 Function Documentation

4.11.4.1 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	

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

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

	handle	Handle to the slot where the key will be stored. It must have been obtained by calling psa_allocate_key() or psa_create_key() and must not contain key material yet.
	type	Key type (a `PSA_KEY_TYPE_XXX` value). This must be a symmetric key type.
	bits	Key size in bits.
`in,out`	*generator*	The generator object to read from.

Return values

PSA_SUCCESS	Success. If the key is persistent, the key material and the key's metadata have been saved to persistent storage.
PSA_ERROR_INSUFFICIENT_CAPACITY	There were fewer than `output_length` bytes in the generator. Note that in this case, no output is written to the output buffer. The generator's capacity is set to 0, thus subsequent calls to this function will not succeed, even with a smaller output buffer.
PSA_ERROR_NOT_SUPPORTED	The key type or key size is not supported, either by the implementation in general or in this particular slot.
PSA_ERROR_BAD_STATE	
PSA_ERROR_INVALID_HANDLE	
PSA_ERROR_OCCUPIED_SLOT	There is already a key in the specified slot.
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_INSUFFICIENT_STORAGE	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_TAMPERING_DETECTED	
PSA_ERROR_BAD_STATE	The library has not been previously initialized by psa_crypto_init(). It is implementation-dependent whether a failure to initialize results in this error code.

4.11.4.3 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	
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	

4.11.4.4 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	

4.11.4.5 psa_status_t psa_set_generator_capacity (psa_crypto_generator_t * *generator,* size_t *capacity*)

Set the maximum capacity of a generator.

Parameters

`in,out`	*generator*	The generator object to modify.
	capacity	The new capacity of the generator. It must be less or equal to the generator's current capacity.

Return values

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

4.12 Key derivation

Macros

- #define PSA_KDF_STEP_SECRET ((psa_key_derivation_step_t)0x0101)
- #define PSA_KDF_STEP_LABEL ((psa_key_derivation_step_t)0x0201)
- #define PSA_KDF_STEP_SALT ((psa_key_derivation_step_t)0x0202)
- #define PSA_KDF_STEP_INFO ((psa_key_derivation_step_t)0x0203)

Typedefs

- typedef uint16_t psa_key_derivation_step_t

  Encoding of the step of a key derivation.

Functions

- psa_status_t psa_key_derivation_setup (psa_crypto_generator_t ∗generator, psa_algorithm_t alg)
- psa_status_t psa_key_derivation_input_bytes (psa_crypto_generator_t ∗generator, psa_key_derivation_↩ step_t step, const uint8_t ∗data, size_t data_length)
- psa_status_t psa_key_derivation_input_key (psa_crypto_generator_t ∗generator, psa_key_derivation_↩ step_t step, psa_key_handle_t handle)
- psa_status_t psa_key_agreement (psa_crypto_generator_t ∗generator, psa_key_derivation_step_t step, psa_key_handle_t private_key, const uint8_t ∗peer_key, size_t peer_key_length)
- psa_status_t psa_key_agreement_raw_shared_secret (psa_algorithm_t alg, psa_key_handle_t private_key, const uint8_t ∗peer_key, size_t peer_key_length, uint8_t ∗output, size_t output_size, size_t ∗output_length)

4.12.1 Detailed Description

4.12.2 Macro Definition Documentation

4.12.2.1 #define PSA_KDF_STEP_INFO ((psa_key_derivation_step_t)0x0203)

An information string for key derivation.

This must be a direct input.

4.12.2.2 #define PSA_KDF_STEP_LABEL ((psa_key_derivation_step_t)0x0201)

A label for key derivation.

This must be a direct input.

4.12.2.3 #define PSA_KDF_STEP_SALT ((psa_key_derivation_step_t)0x0202)

A salt for key derivation.

This must be a direct input.

4.12.2.4 #define PSA_KDF_STEP_SECRET ((psa_key_derivation_step_t)0x0101)

A secret input for key derivation.

This must be a key of type PSA_KEY_TYPE_DERIVE.

4.12.3 Function Documentation

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

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

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

Parameters

`in,out`	*generator*	The generator object to use. It must have been set up with psa_key_derivation_setup() with a key agreement and derivation algorithm `alg` (`PSA_ALG_XXX` value such that PSA_ALG_IS_KEY_AGREEMENT(`alg`) is true and #PSA_ALG_IS_RAW_KEY_AGREEMENT(`alg`) is false). The generator must be ready for an input of the type given by `step`.
	step	Which step the input data is for.
	private_key	Handle to the private key to use.
`in`	*peer_key*	Public key of the peer. The peer key must be in the same format that psa_import_key() accepts for the public key type corresponding to the type of private_key. That is, this function performs the equivalent of ‘psa_import_key(internal_public_key_handle, PSA_KEY_TYPE_PUBLIC_KEY_OF_KEYPAIR(private_key_type), peer_key, peer_key_length)`where` private_key_type`is the type of`private_key‘. For example, for EC keys, this means that peer_key is interpreted as a point on the curve that the private key is on. The standard formats for public keys are documented in the documentation of psa_export_public_key().
	peer_key_length	Size of `peer_key` in bytes.

Return values

PSA_SUCCESS	Success.
PSA_ERROR_INVALID_HANDLE	
PSA_ERROR_EMPTY_SLOT	
PSA_ERROR_NOT_PERMITTED	
PSA_ERROR_INVALID_ARGUMENT	`private_key` is not compatible with `alg`, or `peer_key` is not valid for `alg` or not compatible with `private_key`.
PSA_ERROR_NOT_SUPPORTED	`alg` is not supported or is not a key derivation algorithm.
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_TAMPERING_DETECTED	

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

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

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

Warning

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

Parameters

	private_key	Handle to the private key to use.
`in`	*peer_key*	Public key of the peer. It must be in the same format that psa_import_key() accepts. The standard formats for public keys are documented in the documentation of psa_export_public_key().
	peer_key_length	Size of `peer_key` in bytes.
`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	Success.
PSA_ERROR_INVALID_HANDLE	
PSA_ERROR_EMPTY_SLOT	
PSA_ERROR_NOT_PERMITTED	
PSA_ERROR_INVALID_ARGUMENT	`alg` is not a key agreement algorithm
PSA_ERROR_INVALID_ARGUMENT	`private_key` is not compatible with `alg`, or `peer_key` is not valid for `alg` or not compatible with `private_key`.
PSA_ERROR_NOT_SUPPORTED	`alg` is not a supported key agreement algorithm.
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_TAMPERING_DETECTED	

4.12.3.3 psa_status_t psa_key_derivation_input_bytes (psa_crypto_generator_t ∗ *generator*, psa_key_derivation_step_t *step*, const uint8_t ∗ *data*, size_t *data_length*)

Provide an input for key derivation or key agreement.

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

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

Parameters

`in,out`	*generator*	The generator object to use. It must have been set up with psa_key_derivation_setup() and must not have produced any output yet.
	step	Which step the input data is for.
`in`	*data*	Input data to use.
	data_length	Size of the `data` buffer in bytes.

Return values

PSA_SUCCESS	Success.
PSA_ERROR_INVALID_ARGUMENT	`step` is not compatible with the generator's algorithm.
PSA_ERROR_INVALID_ARGUMENT	`step` does not allow direct inputs.
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_TAMPERING_DETECTED	
PSA_ERROR_BAD_STATE	The value of `step` is not valid given the state of `generator`.
PSA_ERROR_BAD_STATE	The library has not been previously initialized by psa_crypto_init(). It is implementation-dependent whether a failure to initialize results in this error code.

4.12.3.4 psa_status_t psa_key_derivation_input_key (psa_crypto_generator_t ∗ *generator*, psa_key_derivation_step_t *step*, psa_key_handle_t *handle*)

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

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

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

Parameters

`in,out`	*generator*	The generator object to use. It must have been set up with psa_key_derivation_setup() and must not have produced any output yet.
	step	Which step the input data is for.
	handle	Handle to the key. It must have an appropriate type for `step` and must allow the usage PSA_KEY_USAGE_DERIVE.

Return values

PSA_SUCCESS	Success.
PSA_ERROR_INVALID_HANDLE	
PSA_ERROR_EMPTY_SLOT	
PSA_ERROR_NOT_PERMITTED	
PSA_ERROR_INVALID_ARGUMENT	`step` is not compatible with the generator's algorithm.
PSA_ERROR_INVALID_ARGUMENT	`step` does not allow key inputs.

Return values

PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_TAMPERING_DETECTED	
PSA_ERROR_BAD_STATE	The value of `step` is not valid given the state of `generator`.
PSA_ERROR_BAD_STATE	The library has not been previously initialized by psa_crypto_init(). It is implementation-dependent whether a failure to initialize results in this error code.

4.12.3.5 psa_status_t psa_key_derivation_setup (psa_crypto_generator_t ∗ *generator,* psa_algorithm_t *alg*)

Set up a key derivation operation.

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

To use a generator for key derivation:

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

Parameters

`in,out`	*generator*	The generator object to set up. It must have been initialized but not set up yet.
	alg	The key derivation algorithm to compute (`PSA_ALG_XXX` value such that PSA_ALG_IS_KEY_DERIVATION(`alg`) is true).

Return values

PSA_SUCCESS	Success.
PSA_ERROR_INVALID_ARGUMENT	`alg` is not a key derivation algorithm.
PSA_ERROR_NOT_SUPPORTED	`alg` is not supported or is not a key derivation algorithm.
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_TAMPERING_DETECTED	
PSA_ERROR_BAD_STATE	

4.13 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_handle_t handle, psa_key_type_t type, size_t bits, const void ∗extra, size_t extra_size)

  Generate a key or key pair.

4.13.1 Detailed Description

4.13.2 Function Documentation

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

Generate a key or key pair.

Parameters

	handle	Handle to the slot where the key will be stored. It must have been obtained by calling psa_allocate_key() or psa_create_key() and must not contain key material yet.
	type	Key type (a `PSA_KEY_TYPE_XXX` value).
	bits	Key size in bits.
`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: • For a symmetric key type (a type such that PSA_KEY_TYPE_IS_ASYMMETRIC(`type`) is false), `extra` must be `NULL`. • For an elliptic curve key type (a type such that PSA_KEY_TYPE_IS_ECC(`type`) is false), `extra` must be `NULL`. • For an RSA key (`type` is PSA_KEY_TYPE_RSA_KEYPAIR), `extra` is an optional psa_generate_key_extra_rsa structure specifying the public exponent. The default public exponent used when `extra` is `NULL` is 65537. • For an DSA key (`type` is PSA_KEY_TYPE_DSA_KEYPAIR), `extra` is an optional structure specifying the key domain parameters. The key domain parameters can also be provided by psa_set_key_domain_parameters(), which documents the format of the structure. • For a DH key (`type` is PSA_KEY_TYPE_DH_KEYPAIR), the `extra` is an optional structure specifying the key domain parameters. The key domain parameters can also be provided by psa_set_key_domain_parameters(), which documents the format of the structure.
	extra_size	Size of the buffer that `extra` points to, in bytes. Note that if `extra` is `NULL` then `extra_size` must be zero.

Return values

PSA_SUCCESS	Success. If the key is persistent, the key material and the key's metadata have been saved to persistent storage.
PSA_ERROR_INVALID_HANDLE	
PSA_ERROR_OCCUPIED_SLOT	There is already a key in the specified slot.
PSA_ERROR_NOT_SUPPORTED	
PSA_ERROR_INVALID_ARGUMENT	
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_INSUFFICIENT_ENTROPY	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_TAMPERING_DETECTED	
PSA_ERROR_BAD_STATE	The library has not been previously initialized by psa_crypto_init(). It is implementation-dependent whether a failure to initialize results in this error code.

4.13.2.2 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	
PSA_ERROR_BAD_STATE	The library has not been previously initialized by psa_crypto_init(). It is implementation-dependent whether a failure to initialize results in this error code.

4.14 Error codes

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_ERROR_INVALID_HANDLE ((psa_status_t)19)

Typedefs

- typedef int32_t psa_status_t

  Function return status.

4.14.1 Detailed Description

4.14.2 Macro Definition Documentation

4.14.2.1 #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.

4.14.2.2 #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.

4.14.2.3 #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.

4.14.2.4 #define PSA_ERROR_EMPTY_SLOT ((psa_status_t)6)

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

If a handle is invalid, it does not designate an empty slot. The error for an invalid handle is PSA_ERROR_INVAL↩ID_HANDLE.

4.14.2.5 #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.

4.14.2.6 #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.

4.14.2.7 #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.

4.14.2.8 #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.

4.14.2.9 #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.

4.14.2.10 #define PSA_ERROR_INVALID_ARGUMENT ((psa_status_t)8)

The parameters passed to the function are invalid.

Implementations may return this error any time a parameter or combination of parameters are recognized as invalid.

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

Implementation shall not return this error code to indicate that a key handle is invalid, but shall return PSA_ERR↩ OR_INVALID_HANDLE instead.

4.14.2.11 #define PSA_ERROR_INVALID_HANDLE ((psa_status_t)19)

The key handle is not valid.

4.14.2.12 #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.

4.14.2.13 #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.

4.14.2.14 #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.

4.14.2.15 #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.

4.14.2.16 #define PSA_ERROR_OCCUPIED_SLOT ((psa_status_t)5)

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

If a handle is invalid, it does not designate an occupied slot. The error for an invalid handle is PSA_ERROR_INV↩ALID_HANDLE.

4.14.2.17 #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.

4.14.2.18 #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.

4.14.2.19 #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.

4.14.2.20 #define PSA_SUCCESS ((psa_status_t)0)

The action was completed successfully.

4.14.3 Typedef Documentation

4.14.3.1 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.

4.15 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)0x70000000)
- #define **PSA_KEY_TYPE_CATEGORY_SYMMETRIC** ((psa_key_type_t)0x40000000)
- #define **PSA_KEY_TYPE_CATEGORY_RAW** ((psa_key_type_t)0x50000000)
- #define **PSA_KEY_TYPE_CATEGORY_PUBLIC_KEY** ((psa_key_type_t)0x60000000)
- #define **PSA_KEY_TYPE_CATEGORY_KEY_PAIR** ((psa_key_type_t)0x70000000)
- #define **PSA_KEY_TYPE_CATEGORY_FLAG_PAIR** ((psa_key_type_t)0x10000000)
- #define PSA_KEY_TYPE_IS_VENDOR_DEFINED(type) (((type) & PSA_KEY_TYPE_VENDOR_FLAG) != 0)
- #define PSA_KEY_TYPE_IS_UNSTRUCTURED(type)
- #define PSA_KEY_TYPE_IS_ASYMMETRIC(type)
- #define PSA_KEY_TYPE_IS_PUBLIC_KEY(type) (((type) & PSA_KEY_TYPE_CATEGORY_MASK) == P↩ SA_KEY_TYPE_CATEGORY_PUBLIC_KEY)
- #define PSA_KEY_TYPE_IS_KEYPAIR(type) (((type) & PSA_KEY_TYPE_CATEGORY_MASK) == PSA_↩ KEY_TYPE_CATEGORY_KEY_PAIR)
- #define PSA_KEY_TYPE_KEYPAIR_OF_PUBLIC_KEY(type) ((type) | PSA_KEY_TYPE_CATEGORY_F↩ LAG_PAIR)
- #define PSA_KEY_TYPE_PUBLIC_KEY_OF_KEYPAIR(type) ((type) & ∼PSA_KEY_TYPE_CATEGORY↩ _FLAG_PAIR)
- #define PSA_KEY_TYPE_RAW_DATA ((psa_key_type_t)0x50000001)
- #define PSA_KEY_TYPE_HMAC ((psa_key_type_t)0x51000000)
- #define PSA_KEY_TYPE_DERIVE ((psa_key_type_t)0x52000000)
- #define PSA_KEY_TYPE_AES ((psa_key_type_t)0x40000001)
- #define PSA_KEY_TYPE_DES ((psa_key_type_t)0x40000002)
- #define PSA_KEY_TYPE_CAMELLIA ((psa_key_type_t)0x40000003)
- #define PSA_KEY_TYPE_ARC4 ((psa_key_type_t)0x40000004)
- #define PSA_KEY_TYPE_RSA_PUBLIC_KEY ((psa_key_type_t)0x60010000)
- #define PSA_KEY_TYPE_RSA_KEYPAIR ((psa_key_type_t)0x70010000)
- #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_DSA_PUBLIC_KEY ((psa_key_type_t)0x60020000)
- #define PSA_KEY_TYPE_DSA_KEYPAIR ((psa_key_type_t)0x70020000)
- #define PSA_KEY_TYPE_IS_DSA(type) (PSA_KEY_TYPE_PUBLIC_KEY_OF_KEYPAIR(type) == PSA_↩ KEY_TYPE_DSA_PUBLIC_KEY)
- #define **PSA_KEY_TYPE_ECC_PUBLIC_KEY_BASE** ((psa_key_type_t)0x60030000)
- #define **PSA_KEY_TYPE_ECC_KEYPAIR_BASE** ((psa_key_type_t)0x70030000)
- #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_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_KEY_TYPE_DH_PUBLIC_KEY ((psa_key_type_t)0x60040000)
- #define PSA_KEY_TYPE_DH_KEYPAIR ((psa_key_type_t)0x70040000)
- #define PSA_KEY_TYPE_IS_DH(type) (PSA_KEY_TYPE_PUBLIC_KEY_OF_KEYPAIR(type) == PSA_K←
  EY_TYPE_DH_PUBLIC_KEY)
- #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_DERIVATION** ((psa_algorithm_t)0x20000000)
- #define **PSA_ALG_CATEGORY_KEY_AGREEMENT** ((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_KEY_SELECTION_FLAG** ((psa_algorithm_t)0x01000000)
- #define PSA_ALG_IS_KEY_AGREEMENT(alg)

- #define PSA_ALG_IS_KEY_DERIVATION(alg) (((alg) & PSA_ALG_CATEGORY_MASK) == PSA_ALG_←
  CATEGORY_KEY_DERIVATION)
- #define PSA_ALG_IS_KEY_SELECTION(alg) (((alg) & PSA_ALG_CATEGORY_MASK) == PSA_ALG_C←
  ATEGORY_KEY_SELECTION)
- #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_ANY_HASH ((psa_algorithm_t)0x010000ff)
- #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_GET_HASH**(hmac_alg) (PSA_ALG_CATEGORY_HASH | ((hmac_alg) & PSA←
  _ALG_HASH_MASK))
- #define PSA_ALG_IS_HMAC(alg)
- #define **PSA_ALG_MAC_TRUNCATION_MASK** ((psa_algorithm_t)0x00003f00)
- #define **PSA_MAC_TRUNCATION_OFFSET** 8
- #define PSA_ALG_TRUNCATED_MAC(alg, mac_length)
- #define PSA_ALG_FULL_LENGTH_MAC(alg) ((alg) & ~PSA_ALG_MAC_TRUNCATION_MASK)
- #define PSA_MAC_TRUNCATED_LENGTH(alg) (((alg) & PSA_ALG_MAC_TRUNCATION_MASK) >> P←
  SA_MAC_TRUNCATION_OFFSET)
- #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_BLOCK_CIPHER_MAC(alg)
- #define **PSA_ALG_CIPHER_STREAM_FLAG** ((psa_algorithm_t)0x00800000)
- #define **PSA_ALG_CIPHER_FROM_BLOCK_FLAG** ((psa_algorithm_t)0x00400000)
- #define PSA_ALG_IS_STREAM_CIPHER(alg)
- #define PSA_ALG_ARC4 ((psa_algorithm_t)0x04800001)
- #define PSA_ALG_CTR ((psa_algorithm_t)0x04c00001)
- #define **PSA_ALG_CFB** ((psa_algorithm_t)0x04c00002)
- #define **PSA_ALG_OFB** ((psa_algorithm_t)0x04c00003)
- #define PSA_ALG_XTS ((psa_algorithm_t)0x044000ff)
- #define PSA_ALG_CBC_NO_PADDING ((psa_algorithm_t)0x04600100)
- #define PSA_ALG_CBC_PKCS7 ((psa_algorithm_t)0x04600101)
- #define **PSA_ALG_CCM** ((psa_algorithm_t)0x06001001)
- #define **PSA_ALG_GCM** ((psa_algorithm_t)0x06001002)
- #define **PSA_ALG_AEAD_TAG_LENGTH_MASK** ((psa_algorithm_t)0x00003f00)
- #define **PSA_AEAD_TAG_LENGTH_OFFSET** 8
- #define PSA_ALG_AEAD_WITH_TAG_LENGTH(alg, tag_length)
- #define PSA_ALG_AEAD_WITH_DEFAULT_TAG_LENGTH(alg)
- #define **PSA__ALG_AEAD_WITH_DEFAULT_TAG_LENGTH__CASE**(alg, ref)

- #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_IS_HASH_AND_SIGN(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)0x20000100)
- #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_ALG_TLS12_PRF_BASE** ((psa_algorithm_t)0x20000200)
- #define PSA_ALG_TLS12_PRF(hash_alg) (PSA_ALG_TLS12_PRF_BASE | ((hash_alg) & PSA_ALG_HA↩ SH_MASK))

- #define PSA_ALG_IS_TLS12_PRF(alg) (((alg) & ∼PSA_ALG_HASH_MASK) == PSA_ALG_TLS12_PRF↩_BASE)
- #define **PSA_ALG_TLS12_PRF_GET_HASH**(hkdf_alg) (PSA_ALG_CATEGORY_HASH | ((hkdf_alg) & P↩SA_ALG_HASH_MASK))
- #define **PSA_ALG_TLS12_PSK_TO_MS_BASE** ((psa_algorithm_t)0x20000300)
- #define PSA_ALG_TLS12_PSK_TO_MS(hash_alg) (PSA_ALG_TLS12_PSK_TO_MS_BASE | ((hash_alg) & PSA_ALG_HASH_MASK))
- #define PSA_ALG_IS_TLS12_PSK_TO_MS(alg) (((alg) & ∼PSA_ALG_HASH_MASK) == PSA_ALG_TL↩S12_PSK_TO_MS_BASE)
- #define **PSA_ALG_TLS12_PSK_TO_MS_GET_HASH**(hkdf_alg) (PSA_ALG_CATEGORY_HASH | ((hkdf↩_alg) & PSA_ALG_HASH_MASK))
- #define **PSA_ALG_KEY_DERIVATION_MASK** ((psa_algorithm_t)0x080fffff)
- #define **PSA_ALG_KEY_AGREEMENT_MASK** ((psa_algorithm_t)0x10f00000)
- #define PSA_ALG_KEY_AGREEMENT(ka_alg, kdf_alg) ((ka_alg) | (kdf_alg))
- #define **PSA_ALG_KEY_AGREEMENT_GET_KDF**(alg) (((alg) & PSA_ALG_KEY_DERIVATION_MASK) | PSA_ALG_CATEGORY_KEY_DERIVATION)
- #define **PSA_ALG_KEY_AGREEMENT_GET_BASE**(alg) (((alg) & PSA_ALG_KEY_AGREEMENT_MASK) | PSA_ALG_CATEGORY_KEY_AGREEMENT)
- #define **PSA_ALG_IS_RAW_KEY_AGREEMENT**(alg) (PSA_ALG_KEY_AGREEMENT_GET_KDF(alg) == PSA_ALG_CATEGORY_KEY_DERIVATION)
- #define **PSA_ALG_IS_KEY_DERIVATION_OR_AGREEMENT**(alg) ((PSA_ALG_IS_KEY_DERIVATI↩ON(alg) || PSA_ALG_IS_KEY_AGREEMENT(alg)))
- #define PSA_ALG_FFDH ((psa_algorithm_t)0x30100000)
- #define PSA_ALG_IS_FFDH(alg) (PSA_ALG_KEY_AGREEMENT_GET_BASE(alg) == PSA_ALG_FFDH)
- #define PSA_ALG_ECDH ((psa_algorithm_t)0x30200000)
- #define PSA_ALG_IS_ECDH(alg) (PSA_ALG_KEY_AGREEMENT_GET_BASE(alg) == PSA_ALG_ECDH)
- #define PSA_ALG_IS_WILDCARD(alg)
- #define PSA_ALG_IS_WILDCARD(alg)

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.

4.15.1 Detailed Description

4.15.2 Macro Definition Documentation

4.15.2.1 #define PSA__ALG_AEAD_WITH_DEFAULT_TAG_LENGTH__CASE(*alg, ref*)

Value:

```
PSA_ALG_AEAD_WITH_TAG_LENGTH(alg, 0) == \
    PSA_ALG_AEAD_WITH_TAG_LENGTH(ref, 0) ?  \
    ref :
```

4.15.2.2 #define PSA_ALG_AEAD_WITH_DEFAULT_TAG_LENGTH(*alg*)

Value:

```
(                                                                      \
        PSA__ALG_AEAD_WITH_DEFAULT_TAG_LENGTH__CASE(alg, PSA_ALG_CCM)    \
        PSA__ALG_AEAD_WITH_DEFAULT_TAG_LENGTH__CASE(alg, PSA_ALG_GCM)    \
        0)
```

Calculate the corresponding AEAD algorithm with the default tag length.

Parameters

alg	An AEAD algorithm (`PSA_ALG_XXX` value such that PSA_ALG_IS_AEAD(`alg`) is true).

Returns

The corresponding AEAD algorithm with the default tag length for that algorithm.

4.15.2.3 #define PSA_ALG_AEAD_WITH_TAG_LENGTH(*alg,* *tag_length*)

Value:

```
(((alg) & ~PSA_ALG_AEAD_TAG_LENGTH_MASK) |                              \
     ((tag_length) << PSA_AEAD_TAG_LENGTH_OFFSET &                        \
      PSA_ALG_AEAD_TAG_LENGTH_MASK))
```

Macro to build a shortened AEAD algorithm.

A shortened AEAD algorithm is similar to the corresponding AEAD algorithm, but has an authentication tag that consists of fewer bytes. Depending on the algorithm, the tag length may affect the calculation of the ciphertext.

Parameters

alg	A AEAD algorithm identifier (value of type psa_algorithm_t such that PSA_ALG_IS_AEAD(`alg`) is true).
tag_length	Desired length of the authentication tag in bytes.

Returns

The corresponding AEAD algorithm with the specified length.
Unspecified if `alg` is not a supported AEAD algorithm or if `tag_length` is not valid for the specified AEAD algorithm.

4.15.2.4 #define PSA_ALG_ANY_HASH ((psa_algorithm_t)0x010000ff)

Allow any hash algorithm.

This value may only be used to form the algorithm usage field of a policy for a signature algorithm that is parametrized by a hash. That is, suppose that `PSA_xxx_SIGNATURE` is one of the following macros:

- PSA_ALG_RSA_PKCS1V15_SIGN, PSA_ALG_RSA_PSS,
- PSA_ALG_DSA, #PSA_ALG_DETERMINISTIC_DSA,
- PSA_ALG_ECDSA, PSA_ALG_DETERMINISTIC_ECDSA. Then you may create a key as follows:
- Set the key usage field using PSA_ALG_ANY_HASH, for example:

```
psa_key_policy_set_usage(&policy,
                         PSA_KEY_USAGE_SIGN, //or PSA_KEY_USAGE_VERIFY
                         PSA_xxx_SIGNATURE(PSA_ALG_ANY_HASH));
psa_set_key_policy(handle, &policy);
```

- Import or generate key material.
- Call psa_asymmetric_sign() or psa_asymmetric_verify(), passing an algorithm built from `PSA_xxx_SIG↩ NATURE` and a specific hash. Each call to sign or verify a message may use a different hash.

```
psa_asymmetric_sign(handle, PSA_xxx_SIGNATURE(PSA_ALG_SHA_256), ...);
psa_asymmetric_sign(handle, PSA_xxx_SIGNATURE(PSA_ALG_SHA_512), ...);
psa_asymmetric_sign(handle, PSA_xxx_SIGNATURE(PSA_ALG_SHA3_256), ...);
```

This value may not be used to build other algorithms that are parametrized over a hash. For any valid use of this macro to build an algorithm `\p alg`, PSA_ALG_IS_HASH_AND_SIGN(`alg`) is true.

This value may not be used to build an algorithm specification to perform an operation. It is only valid to build policies.

4.15.2.5 #define PSA_ALG_ARC4 ((psa_algorithm_t)0x04800001)

The ARC4 stream cipher algorithm.

4.15.2.6 #define PSA_ALG_CBC_NO_PADDING ((psa_algorithm_t)0x04600100)

The CBC block cipher chaining mode, with no padding.

The underlying block cipher is determined by the key type.

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

4.15.2.7 #define PSA_ALG_CBC_PKCS7 ((psa_algorithm_t)0x04600101)

The CBC block cipher chaining mode with PKCS#7 padding.

The underlying block cipher is determined by the key type.

This is the padding method defined by PKCS#7 (RFC 2315) §10.3.

4.15.2.8 #define PSA_ALG_CTR ((psa_algorithm_t)0x04c00001)

The CTR stream cipher mode.

CTR is a stream cipher which is built from a block cipher. The underlying block cipher is determined by the key type. For example, to use AES-128-CTR, use this algorithm with a key of type PSA_KEY_TYPE_AES and a length of 128 bits (16 bytes).

4.15.2.9 #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). This includes PSA_ALG_ANY_HASH when specifying the algorithm in a usage policy.

Returns

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

4.15.2.10 #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). This includes PSA_ALG_ANY_HASH when specifying the algorithm in a usage policy.

Returns

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

4.15.2.11 #define PSA_ALG_ECDH ((psa_algorithm_t)0x30200000)

The elliptic curve Diffie-Hellman (ECDH) key agreement algorithm.

The shared secret produced by key agreement is the x-coordinate of the shared secret point. It is always `ceiling(m / 8)` bytes long where `m` is the bit size associated with the curve, i.e. the bit size of the order of the curve's coordinate field. When `m` is not a multiple of 8, the byte containing the most significant bit of the shared secret is padded with zero bits. The byte order is either little-endian or big-endian depending on the curve type.

- For Montgomery curves (curve types `PSA_ECC_CURVE_CURVEXXX`), the shared secret is the x-coordinate of `d_A Q_B = d_B Q_A` in little-endian byte order. The bit size is 448 for Curve448 and 255 for Curve25519.
- For Weierstrass curves over prime fields (curve types `PSA_ECC_CURVE_SECPXXX` and `PSA_ECC_CU↩RVE_BRAINPOOL_PXXX`), the shared secret is the x-coordinate of `d_A Q_B = d_B Q_A` in big-endian byte order. The bit size is `m = ceiling(log_2(p))` for the field `F_p`.
- For Weierstrass curves over binary fields (curve types `PSA_ECC_CURVE_SECTXXX`), the shared secret is the x-coordinate of `d_A Q_B = d_B Q_A` in big-endian byte order. The bit size is `m` for the field `F_{2^m}`.

4.15.2.12 #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). This includes PSA_ALG_ANY_HASH when specifying the algorithm in a usage policy.

Returns

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

4.15.2.13 #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.

4.15.2.14 #define PSA_ALG_FFDH ((psa_algorithm_t)0x30100000)

The finite-field Diffie-Hellman (DH) key agreement algorithm.

The shared secret produced by key agreement and passed as input to the derivation or selection algorithm `kdf↩_alg` is the shared secret `g^{ab}` in big-endian format. It is `ceiling(m / 8)` bytes long where `m` is the size of the prime `p` in bits.

4.15.2.15 #define PSA_ALG_FULL_LENGTH_MAC(*alg*) ((alg) & ∼PSA_ALG_MAC_TRUNCATION_MASK)

Macro to build the base MAC algorithm corresponding to a truncated MAC algorithm.

Parameters

alg	A MAC algorithm identifier (value of type psa_algorithm_t such that PSA_ALG_IS_MAC(`alg`) is true). This may be a truncated or untruncated MAC algorithm.

Returns

The corresponding base MAC algorithm.
Unspecified if `alg` is not a supported MAC algorithm.

4.15.2.16 #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.

This key derivation algorithm uses the following inputs:

- PSA_KDF_STEP_SALT is the salt used in the "extract" step. It is optional; if omitted, the derivation uses an empty salt.
- PSA_KDF_STEP_SECRET is the secret key used in the "extract" step.
- PSA_KDF_STEP_INFO is the info string used in the "expand" step. You must pass PSA_KDF_STEP_SALT before PSA_KDF_STEP_SECRET. You may pass PSA_KDF_STEP_INFO at any time after steup and before starting to generate output.

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.

4.15.2.17 #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.

4.15.2.18 #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.

4.15.2.19 #define PSA_ALG_IS_ASYMMETRIC_ENCRYPTION(*alg*) (((alg) & PSA_ALG_CATEGORY_MASK) == PSA_ALG_CATEGORY_ASYMMETRIC_ENCRYPTION)

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.

4.15.2.20 #define PSA_ALG_IS_BLOCK_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.

4.15.2.21 #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.

4.15.2.22 #define PSA_ALG_IS_DSA(*alg*)

Value:

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

4.15.2.23 #define PSA_ALG_IS_ECDH(*alg*) (PSA_ALG_KEY_AGREEMENT_GET_BASE(alg) == PSA_ALG_ECDH)

Whether the specified algorithm is an elliptic curve Diffie-Hellman algorithm.

This includes every supported key selection or key agreement algorithm for the output of the Diffie-Hellman calculation.

Parameters

alg	An algorithm identifier (value of type psa_algorithm_t).

Returns

1 if `alg` is an elliptic curve Diffie-Hellman algorithm, 0 otherwise. This macro may return either 0 or 1 if `alg` is not a supported key agreement algorithm identifier.

4.15.2.24 #define PSA_ALG_IS_ECDSA(*alg*)

Value:

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

4.15.2.25 #define PSA_ALG_IS_FFDH(*alg*) (PSA_ALG_KEY_AGREEMENT_GET_BASE(alg) == PSA_ALG_FFDH)

Whether the specified algorithm is a finite field Diffie-Hellman algorithm.

This includes every supported key selection or key agreement algorithm for the output of the Diffie-Hellman calculation.

Parameters

alg	An algorithm identifier (value of type psa_algorithm_t).

Returns

1 if `alg` is a finite field Diffie-Hellman algorithm, 0 otherwise. This macro may return either 0 or 1 if `alg` is not a supported key agreement algorithm identifier.

4.15.2.26 #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.

4.15.2.27 #define PSA_ALG_IS_HASH_AND_SIGN(*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))
```

Whether the specified algorithm is a hash-and-sign algorithm.

Hash-and-sign algorithms are public-key signature algorithms structured in two parts: first the calculation of a hash in a way that does not depend on the key, then the calculation of a signature from the hash value and the key.

Parameters

alg	An algorithm identifier (value of type psa_algorithm_t).

Returns

1 if `alg` is a hash-and-sign algorithm, 0 otherwise. This macro may return either 0 or 1 if `alg` is not a supported algorithm identifier.

4.15.2.28 #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.

4.15.2.29 #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.

4.15.2.30 #define PSA_ALG_IS_KEY_AGREEMENT(*alg*)

Value:

```
(((alg) & PSA_ALG_CATEGORY_MASK & ~PSA_ALG_KEY_SELECTION_FLAG) ==    \
     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.

4.15.2.31 #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.

4.15.2.32 #define PSA_ALG_IS_KEY_SELECTION(*alg*) (((alg) & PSA_ALG_CATEGORY_MASK) == PSA_ALG_CATEGORY_KEY_SELECTION)

Whether the specified algorithm is a key selection algorithm.

Parameters

alg	An algorithm identifier (value of type psa_algorithm_t).

Returns

1 if `alg` is a key selection algorithm, 0 otherwise. This macro may return either 0 or 1 if `alg` is not a supported algorithm identifier.

4.15.2.33 #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.

4.15.2.34 #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.

4.15.2.35 #define PSA_ALG_IS_STREAM_CIPHER(*alg*)

Value:

```
(((alg) & (PSA_ALG_CATEGORY_MASK | PSA_ALG_CIPHER_STREAM_FLAG)) == \
        (PSA_ALG_CATEGORY_CIPHER | PSA_ALG_CIPHER_STREAM_FLAG))
```

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.

4.15.2.36 #define PSA_ALG_IS_TLS12_PRF(*alg*) (((alg) & ∼PSA_ALG_HASH_MASK) == PSA_ALG_TLS12_PRF_BASE)

Whether the specified algorithm is a TLS-1.2 PRF algorithm.

Parameters

alg	An algorithm identifier (value of type psa_algorithm_t).

Returns

1 if `alg` is a TLS-1.2 PRF algorithm, 0 otherwise. This macro may return either 0 or 1 if `alg` is not a supported key derivation algorithm identifier.

4.15.2.37 #define PSA_ALG_IS_TLS12_PSK_TO_MS(*alg*) (((alg) & ∼PSA_ALG_HASH_MASK) == PSA_ALG_TLS12_PSK_TO_MS_BASE)

Whether the specified algorithm is a TLS-1.2 PSK to MS algorithm.

Parameters

alg	An algorithm identifier (value of type psa_algorithm_t).

Returns

1 if `alg` is a TLS-1.2 PSK to MS algorithm, 0 otherwise. This macro may return either 0 or 1 if `alg` is not a supported key derivation algorithm identifier.

4.15.2.38 #define PSA_ALG_IS_WILDCARD(*alg*)

Value:

```
(PSA_ALG_IS_HASH_AND_SIGN(alg) ?                    \
     PSA_ALG_SIGN_GET_HASH(alg) == PSA_ALG_ANY_HASH :    \
     (alg) == PSA_ALG_ANY_HASH)
```

Whether the specified algorithm encoding is a wildcard.

Wildcard values may only be used to set the usage algorithm field in a policy, not to perform an operation.

Parameters

alg	An algorithm identifier (value of type psa_algorithm_t).

Returns

1 if `alg` is a wildcard algorithm encoding.
0 if `alg` is a non-wildcard algorithm encoding (suitable for an operation).
This macro may return either 0 or 1 if `alg` is not a supported algorithm identifier.

4.15.2.39 #define PSA_ALG_IS_WILDCARD(*alg*)

Value:

```
(PSA_ALG_IS_HASH_AND_SIGN(alg) ?                    \
     PSA_ALG_SIGN_GET_HASH(alg) == PSA_ALG_ANY_HASH :    \
     (alg) == PSA_ALG_ANY_HASH)
```

Whether the specified algorithm encoding is a wildcard.

Wildcard values may only be used to set the usage algorithm field in a policy, not to perform an operation.

Parameters

alg	An algorithm identifier (value of type psa_algorithm_t).

Returns

1 if `alg` is a wildcard algorithm encoding.
0 if `alg` is a non-wildcard algorithm encoding (suitable for an operation).
This macro may return either 0 or 1 if `alg` is not a supported algorithm identifier.

4.15.2.40 #define PSA_ALG_KEY_AGREEMENT(*ka_alg, kdf_alg*) ((ka_alg) | (kdf_alg))

Macro to build a combined algorithm that chains a key agreement with a key derivation.

Parameters

ka_alg	A key agreement algorithm (`PSA_ALG_XXX` value such that PSA_ALG_IS_KEY_AGREEMENT(`ka_alg`) is true).
kdf_alg	A key derivation algorithm (`PSA_ALG_XXX` value such that PSA_ALG_IS_KEY_DERIVATION(`kdf_alg`) is true).

Returns

The corresponding key agreement and derivation algorithm.
Unspecified if `ka_alg` is not a supported key agreement algorithm or `kdf_alg` is not a supported key derivation algorithm.

4.15.2.41 #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.

4.15.2.42 #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)
```

4.15.2.43 #define PSA_ALG_RSA_PKCS1V15_CRYPT ((psa_algorithm_t)0x12020000)

RSA PKCS#1 v1.5 encryption.

4.15.2.44 #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). This includes PSA_ALG_ANY_HASH when specifying the algorithm in a usage policy.

Returns

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

4.15.2.45 #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.

4.15.2.46 #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). This includes PSA_ALG_ANY_HASH when specifying the algorithm in a usage policy.

Returns

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

4.15.2.47 #define PSA_ALG_SHA3_224 ((psa_algorithm_t)0x01000010)

SHA3-224

4.15.2.48 #define PSA_ALG_SHA3_256 ((psa_algorithm_t)0x01000011)

SHA3-256

4.15.2.49 #define PSA_ALG_SHA3_384 ((psa_algorithm_t)0x01000012)

SHA3-384

4.15.2.50 #define PSA_ALG_SHA3_512 ((psa_algorithm_t)0x01000013)

SHA3-512

4.15.2.51 #define PSA_ALG_SHA_224 ((psa_algorithm_t)0x01000008)

SHA2-224

4.15.2.52 #define PSA_ALG_SHA_256 ((psa_algorithm_t)0x01000009)

SHA2-256

4.15.2.53 #define PSA_ALG_SHA_384 ((psa_algorithm_t)0x0100000a)

SHA2-384

4.15.2.54 #define PSA_ALG_SHA_512 ((psa_algorithm_t)0x0100000b)

SHA2-512

4.15.2.55 #define PSA_ALG_SHA_512_224 ((psa_algorithm_t)0x0100000c)

SHA2-512/224

4.15.2.56 #define PSA_ALG_SHA_512_256 ((psa_algorithm_t)0x0100000d)

SHA2-512/256

4.15.2.57 #define PSA_ALG_SIGN_GET_HASH(*alg*)

Value:

```
(PSA_ALG_IS_HASH_AND_SIGN(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.

4.15.2.58 #define PSA_ALG_TLS12_PRF(*hash_alg*) (PSA_ALG_TLS12_PRF_BASE | ((hash_alg) & PSA_ALG_HASH_MASK))

Macro to build a TLS-1.2 PRF algorithm.

TLS 1.2 uses a custom pseudorandom function (PRF) for key schedule, specified in Section 5 of RFC 5246. It is based on HMAC and can be used with either SHA-256 or SHA-384.

For the application to TLS-1.2, the salt and label arguments passed to psa_key_derivation() are what's called 'seed' and 'label' in RFC 5246, respectively. For example, for TLS key expansion, the salt is the concatenation of Server←↩Hello.Random + ClientHello.Random, while the label is "key expansion".

For example, `PSA_ALG_TLS12_PRF(PSA_ALG_SHA256)` represents the TLS 1.2 PRF using HMAC-SHA-256.

Parameters

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

Returns

The corresponding TLS-1.2 PRF algorithm.
Unspecified if `alg` is not a supported hash algorithm.

4.15.2.59 #define PSA_ALG_TLS12_PSK_TO_MS(*hash_alg*) (PSA_ALG_TLS12_PSK_TO_MS_BASE | ((hash_alg) & PSA_ALG_HASH_MASK))

Macro to build a TLS-1.2 PSK-to-MasterSecret algorithm.

In a pure-PSK handshake in TLS 1.2, the master secret is derived from the PreSharedKey (PSK) through the application of padding (RFC 4279, Section 2) and the TLS-1.2 PRF (RFC 5246, Section 5). The latter is based on HMAC and can be used with either SHA-256 or SHA-384.

For the application to TLS-1.2, the salt passed to psa_key_derivation() (and forwarded to the TLS-1.2 PRF) is the concatenation of the ClientHello.Random + ServerHello.Random, while the label is "master secret" or "extended master secret".

For example, `PSA_ALG_TLS12_PSK_TO_MS(PSA_ALG_SHA256)` represents the TLS-1.2 PSK to Master←↩Secret derivation PRF 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 TLS-1.2 PSK to MS algorithm.
Unspecified if `alg` is not a supported hash algorithm.

4.15.2.60 #define PSA_ALG_TRUNCATED_MAC(*alg, mac_length*)

Value:

```
(((alg) & ~PSA_ALG_MAC_TRUNCATION_MASK) |                              \
     ((mac_length) << PSA_MAC_TRUNCATION_OFFSET & PSA_ALG_MAC_TRUNCATION_MASK))
```

Macro to build a truncated MAC algorithm.

A truncated MAC algorithm is identical to the corresponding MAC algorithm except that the MAC value for the truncated algorithm consists of only the first `mac_length` bytes of the MAC value for the untruncated algorithm.

Note

This macro may allow constructing algorithm identifiers that are not valid, either because the specified length is larger than the untruncated MAC or because the specified length is smaller than permitted by the implementation.
It is implementation-defined whether a truncated MAC that is truncated to the same length as the MAC of the untruncated algorithm is considered identical to the untruncated algorithm for policy comparison purposes.

Parameters

alg	A MAC algorithm identifier (value of type psa_algorithm_t such that PSA_ALG_IS_MAC(`alg`) is true). This may be a truncated or untruncated MAC algorithm.
mac_length	Desired length of the truncated MAC in bytes. This must be at most the full length of the MAC and must be at least an implementation-specified minimum. The implementation-specified minimum shall not be zero.

Returns

The corresponding MAC algorithm with the specified length.
Unspecified if `alg` is not a supported MAC algorithm or if `mac_length` is too small or too large for the specified MAC algorithm.

4.15.2.61 #define PSA_ALG_XTS ((psa_algorithm_t)0x044000ff)

The XTS cipher mode.

XTS is a cipher mode which is built from a block cipher. It requires at least one full block of input, but beyond this minimum the input does not need to be a whole number of blocks.

4.15.2.62 #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.

4.15.2.63 #define PSA_KEY_TYPE_AES ((psa_key_type_t)0x40000001)

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

The size of the key can be 16 bytes (AES-128), 24 bytes (AES-192) or 32 bytes (AES-256).

4.15.2.64 #define PSA_KEY_TYPE_ARC4 ((psa_key_type_t)0x40000004)

Key for the RC4 stream cipher.

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

4.15.2.65 #define PSA_KEY_TYPE_CAMELLIA ((psa_key_type_t)0x40000003)

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

4.15.2.66 #define PSA_KEY_TYPE_DERIVE ((psa_key_type_t)0x52000000)

A secret for key derivation.

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

4.15.2.67 #define PSA_KEY_TYPE_DES ((psa_key_type_t)0x40000002)

Key for a cipher or MAC algorithm based on DES or 3DES (Triple-DES).

The size of the key can be 8 bytes (single DES), 16 bytes (2-key 3DES) or 24 bytes (3-key 3DES).

Note that single DES and 2-key 3DES are weak and strongly deprecated and should only be used to decrypt legacy data. 3-key 3DES is weak and deprecated and should only be used in legacy protocols.

4.15.2.68 #define PSA_KEY_TYPE_DH_KEYPAIR ((psa_key_type_t)0x70040000)

Diffie-Hellman key exchange key pair (private and public key).

4.15.2.69 #define PSA_KEY_TYPE_DH_PUBLIC_KEY ((psa_key_type_t)0x60040000)

Diffie-Hellman key exchange public key.

4.15.2.70 #define PSA_KEY_TYPE_DSA_KEYPAIR ((psa_key_type_t)0x70020000)

DSA key pair (private and public key).

4.15.2.71 #define PSA_KEY_TYPE_DSA_PUBLIC_KEY ((psa_key_type_t)0x60020000)

DSA public key.

4.15.2.72 #define PSA_KEY_TYPE_ECC_KEYPAIR(*curve*) (PSA_KEY_TYPE_ECC_KEYPAIR_BASE | (curve))

Elliptic curve key pair.

4.15.2.73 #define PSA_KEY_TYPE_ECC_PUBLIC_KEY(*curve*) (PSA_KEY_TYPE_ECC_PUBLIC_KEY_BASE | (curve))

Elliptic curve public key.

4.15.2.74 #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.

4.15.2.75 #define PSA_KEY_TYPE_HMAC ((psa_key_type_t)0x51000000)

HMAC key.

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

HMAC keys should generally have the same size as the underlying hash. This size can be calculated with PSA_↩ HASH_SIZE(`alg`) where `alg` is the HMAC algorithm or the underlying hash algorithm.

4.15.2.76 #define PSA_KEY_TYPE_IS_ASYMMETRIC(*type*)

Value:

```
(((type) & PSA_KEY_TYPE_CATEGORY_MASK                            \
      & ~PSA_KEY_TYPE_CATEGORY_FLAG_PAIR) ==                        \
     PSA_KEY_TYPE_CATEGORY_PUBLIC_KEY)
```

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

4.15.2.77 #define PSA_KEY_TYPE_IS_DH(*type*) (PSA_KEY_TYPE_PUBLIC_KEY_OF_KEYPAIR(type) == PSA_KEY_TYPE_DH_PUBLIC_KEY)

Whether a key type is a Diffie-Hellman key exchange key (pair or public-only).

4.15.2.78 #define PSA_KEY_TYPE_IS_DSA(*type*) (PSA_KEY_TYPE_PUBLIC_KEY_OF_KEYPAIR(type) == PSA_KEY_TYPE_DSA_PUBLIC_KEY)

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

4.15.2.79 #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).

4.15.2.80 #define PSA_KEY_TYPE_IS_ECC_KEYPAIR(*type*)

Value:

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

Whether a key type is an elliptic curve key pair.

4.15.2.81 #define PSA_KEY_TYPE_IS_ECC_PUBLIC_KEY(*type*)

Value:

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

Whether a key type is an elliptic curve public key.

4.15.2.82 #define PSA_KEY_TYPE_IS_KEYPAIR(*type*) (((type) & PSA_KEY_TYPE_CATEGORY_MASK) == PSA_KEY_TYPE_CATEGORY_KEY_PAIR)

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

4.15.2.83 #define PSA_KEY_TYPE_IS_PUBLIC_KEY(*type*) (((type) & PSA_KEY_TYPE_CATEGORY_MASK) == PSA_KEY_TYPE_CATEGORY_PUBLIC_KEY)

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

4.15.2.84 #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).

4.15.2.85 #define PSA_KEY_TYPE_IS_UNSTRUCTURED(*type*)

Value:

```
(((type) & PSA_KEY_TYPE_CATEGORY_MASK & ~(psa_key_type_t)0x10000000) == \
     PSA_KEY_TYPE_CATEGORY_SYMMETRIC)
```

Whether a key type is an unstructured array of bytes.

This encompasses both symmetric keys and non-key data.

4.15.2.86 #define PSA_KEY_TYPE_IS_VENDOR_DEFINED(*type*) (((type) & PSA_KEY_TYPE_VENDOR_FLAG) != 0)

Whether a key type is vendor-defined.

4.15.2.87 #define PSA_KEY_TYPE_KEYPAIR_OF_PUBLIC_KEY(*type*) ((type) | PSA_KEY_TYPE_CATEGORY_FLAG_PAIR)

The key pair type corresponding to a public key type.

You may also pass a key pair type as `type`, it will be left unchanged.

Parameters

type	A public key type or key pair type.

Returns

The corresponding key pair type. If `type` is not a public key or a key pair, the return value is undefined.

4.15.2.88 #define PSA_KEY_TYPE_NONE ((psa_key_type_t)0x00000000)

An invalid key type value.

Zero is not the encoding of any key type.

4.15.2.89 #define PSA_KEY_TYPE_PUBLIC_KEY_OF_KEYPAIR(*type*) ((type) & ∼PSA_KEY_TYPE_CATEGORY_FLAG_PAIR)

The public key type corresponding to a key pair type.

You may also pass a key pair type as `type`, it will be left unchanged.

Parameters

type	A public key type or key pair type.

Returns

The corresponding public key type. If `type` is not a public key or a key pair, the return value is undefined.

4.15.2.90 #define PSA_KEY_TYPE_RAW_DATA ((psa_key_type_t)0x50000001)

Raw data.

A "key" of this type cannot be used for any cryptographic operation. Applications may use this type to store arbitrary data in the keystore.

4.15.2.91 #define PSA_KEY_TYPE_RSA_KEYPAIR ((psa_key_type_t)0x70010000)

RSA key pair (private and public key).

4.15.2.92 #define PSA_KEY_TYPE_RSA_PUBLIC_KEY ((psa_key_type_t)0x60010000)

RSA public key.

4.15.2.93 #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.

4.15.2.94 #define PSA_MAC_TRUNCATED_LENGTH(*alg*) (((alg) & PSA_ALG_MAC_TRUNCATION_MASK) >> PSA_MAC_TRUNCATION_OFFSET)

Length to which a MAC algorithm is truncated.

Parameters

alg	A MAC algorithm identifier (value of type psa_algorithm_t such that PSA_ALG_IS_MAC(`alg`) is true).

Returns

Length of the truncated MAC in bytes.
0 if `alg` is a non-truncated MAC algorithm.
Unspecified if `alg` is not a supported MAC algorithm.

4.15.3 Typedef Documentation

4.15.3.1 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.

4.15.3.2 typedef uint16_t psa_ecc_curve_t

The type of PSA elliptic curve identifiers.

4.16 Key lifetimes

Macros

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

Typedefs

- typedef uint32_t psa_key_lifetime_t
- typedef uint32_t psa_key_id_t

4.16.1 Detailed Description

4.16.2 Macro Definition Documentation

4.16.2.1 #define PSA_KEY_LIFETIME_PERSISTENT ((psa_key_lifetime_t)0x00000001)

The default storage area for persistent keys.

A persistent key remains in storage until it is explicitly destroyed or until the corresponding storage area is wiped. This specification does not define any mechanism to wipe a storage area, but implementations may provide their own mechanism (for example to perform a factory reset, to prepare for device refurbishment, or to uninstall an application).

This lifetime value is the default storage area for the calling application. Implementations may offer other storage areas designated by other lifetime values as implementation-specific extensions.

4.16.2.2 #define PSA_KEY_LIFETIME_VOLATILE ((psa_key_lifetime_t)0x00000000)

A volatile key only exists as long as the handle to it is not closed. The key material is guaranteed to be erased on a power reset.

4.16.3 Typedef Documentation

4.16.3.1 typedef uint32_t psa_key_id_t

Encoding of identifiers of persistent keys.

4.16.3.2 typedef uint32_t psa_key_lifetime_t

Encoding of key lifetimes.

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 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

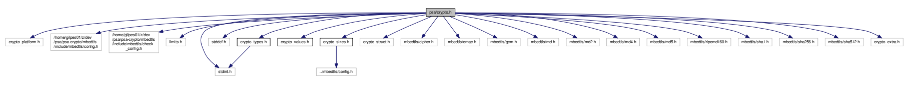
File Documentation

6.1 psa/crypto.h File Reference

Platform Security Architecture cryptography module.

```
#include "crypto_platform.h"
#include <stddef.h>
#include "crypto_types.h"
#include "crypto_values.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_KEY_POLICY_INIT {0}
- #define PSA_HASH_OPERATION_INIT {0}
- #define PSA_MAC_OPERATION_INIT {0}
- #define PSA_CIPHER_OPERATION_INIT {0}
- #define PSA_AEAD_OPERATION_INIT {0}
- #define PSA_CRYPTO_GENERATOR_INIT {0}
- #define PSA_GENERATOR_UNBRIDLED_CAPACITY ((size_t)(-1))

Typedefs

- typedef _unsigned_integral_type_ psa_key_handle_t

    Key handle.
- typedef struct psa_key_policy_s psa_key_policy_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_aead_operation_s psa_aead_operation_t
- typedef struct psa_crypto_generator_s psa_crypto_generator_t

Functions

- psa_status_t psa_crypto_init (void)

    Library initialization.
- 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_handle_t handle, const psa_key_policy_t ∗policy)

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

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

    Retrieve the lifetime of an open key.
- psa_status_t psa_allocate_key (psa_key_handle_t ∗handle)
- psa_status_t psa_open_key (psa_key_lifetime_t lifetime, psa_key_id_t id, psa_key_handle_t ∗handle)
- psa_status_t psa_create_key (psa_key_lifetime_t lifetime, psa_key_id_t id, psa_key_handle_t ∗handle)
- psa_status_t psa_close_key (psa_key_handle_t handle)
- psa_status_t psa_import_key (psa_key_handle_t handle, 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_handle_t handle)

    Destroy a key.
- psa_status_t psa_get_key_information (psa_key_handle_t handle, psa_key_type_t ∗type, size_t ∗bits)

    Get basic metadata about a key.
- psa_status_t psa_set_key_domain_parameters (psa_key_handle_t handle, psa_key_type_t type, const uint8_t ∗data, size_t data_length)

    Set domain parameters for a key.
- psa_status_t psa_get_key_domain_parameters (psa_key_handle_t handle, uint8_t ∗data, size_t data_size, size_t ∗data_length)

    Get domain parameters for a key.
- psa_status_t psa_export_key (psa_key_handle_t handle, 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_handle_t handle, 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.
- psa_status_t psa_copy_key (psa_key_handle_t source_handle, psa_key_handle_t target_handle, const psa_key_policy_t ∗constraint)

- psa_status_t psa_hash_compute (psa_algorithm_t alg, const uint8_t ∗input, size_t input_length, uint8_↩ t ∗hash, size_t hash_size, size_t ∗hash_length)
- psa_status_t psa_hash_compare (psa_algorithm_t alg, const uint8_t ∗input, size_t input_length, const uint8_t ∗hash, const size_t hash_length)
- 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_hash_clone (const psa_hash_operation_t ∗source_operation, psa_hash_operation_↩ t ∗target_operation)
- psa_status_t psa_mac_compute (psa_key_handle_t handle, psa_algorithm_t alg, const uint8_t ∗input, size↩ _t input_length, uint8_t ∗mac, size_t mac_size, size_t ∗mac_length)
- psa_status_t psa_mac_verify (psa_key_handle_t handle, psa_algorithm_t alg, const uint8_t ∗input, size_t input_length, const uint8_t ∗mac, const size_t mac_length)
- psa_status_t psa_mac_sign_setup (psa_mac_operation_t ∗operation, psa_key_handle_t handle, psa_↩ algorithm_t alg)
- psa_status_t psa_mac_verify_setup (psa_mac_operation_t ∗operation, psa_key_handle_t handle, 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 (psa_key_handle_t handle, psa_algorithm_t alg, const uint8_t ∗input, size_t input_length, uint8_t ∗output, size_t output_size, size_t ∗output_length)
- psa_status_t psa_cipher_decrypt (psa_key_handle_t handle, psa_algorithm_t alg, const uint8_t ∗input, size_t input_length, uint8_t ∗output, size_t output_size, size_t ∗output_length)
- psa_status_t psa_cipher_encrypt_setup (psa_cipher_operation_t ∗operation, psa_key_handle_t handle, psa_algorithm_t alg)
- psa_status_t psa_cipher_decrypt_setup (psa_cipher_operation_t ∗operation, psa_key_handle_t handle, 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_handle_t handle, psa_algorithm_t alg, const uint8_t ∗nonce, size_t nonce_length, const uint8_t ∗additional_data, size_t additional_data_length, const uint8_t ∗plaintext, size_t plaintext_length, uint8_t ∗ciphertext, size_t ciphertext_size, size_t ∗ciphertext_length)
- psa_status_t psa_aead_decrypt (psa_key_handle_t handle, psa_algorithm_t alg, const uint8_t ∗nonce, size_t nonce_length, const uint8_t ∗additional_data, size_t additional_data_length, const uint8_t ∗ciphertext, size_t ciphertext_length, uint8_t ∗plaintext, size_t plaintext_size, size_t ∗plaintext_length)
- psa_status_t psa_aead_encrypt_setup (psa_aead_operation_t ∗operation, psa_key_handle_t handle, psa↩ _algorithm_t alg)
- psa_status_t psa_aead_decrypt_setup (psa_aead_operation_t ∗operation, psa_key_handle_t handle, psa↩ _algorithm_t alg)
- psa_status_t psa_aead_generate_nonce (psa_aead_operation_t ∗operation, unsigned char ∗nonce, size_t nonce_size, size_t ∗nonce_length)

- psa_status_t psa_aead_set_nonce (psa_aead_operation_t ∗operation, const unsigned char ∗nonce, size_t nonce_length)
- psa_status_t psa_aead_set_lengths (psa_aead_operation_t ∗operation, size_t ad_length, size_t plaintext↩_length)
- psa_status_t psa_aead_update_ad (psa_aead_operation_t ∗operation, const uint8_t ∗input, size_t input_↩length)
- psa_status_t psa_aead_update (psa_aead_operation_t ∗operation, const uint8_t ∗input, size_t input_length, unsigned char ∗output, size_t output_size, size_t ∗output_length)
- psa_status_t psa_aead_finish (psa_aead_operation_t ∗operation, uint8_t ∗ciphertext, size_t ciphertext_size, size_t ∗ciphertext_length, uint8_t ∗tag, size_t tag_size, size_t ∗tag_length)
- psa_status_t psa_aead_verify (psa_aead_operation_t ∗operation, const uint8_t ∗tag, size_t tag_length)
- psa_status_t psa_aead_abort (psa_aead_operation_t ∗operation)
- psa_status_t psa_asymmetric_sign (psa_key_handle_t handle, psa_algorithm_t alg, const uint8_t ∗hash, size_t hash_length, uint8_t ∗signature, size_t signature_size, size_t ∗signature_length)

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

  Verify the signature a hash or short message using a public key.
- psa_status_t psa_asymmetric_encrypt (psa_key_handle_t handle, psa_algorithm_t alg, const uint8_t ∗input, size_t input_length, const uint8_t ∗salt, size_t salt_length, uint8_t ∗output, size_t output_size, size_↩t ∗output_length)

  Encrypt a short message with a public key.
- psa_status_t psa_asymmetric_decrypt (psa_key_handle_t handle, psa_algorithm_t alg, const uint8_t ∗input, size_t input_length, const uint8_t ∗salt, size_t salt_length, uint8_t ∗output, size_t output_size, size_↩t ∗output_length)

  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_set_generator_capacity (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_handle_t handle, 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_setup (psa_crypto_generator_t ∗generator, psa_algorithm_t alg)
- psa_status_t psa_key_derivation_input_bytes (psa_crypto_generator_t ∗generator, psa_key_derivation_↩step_t step, const uint8_t ∗data, size_t data_length)
- psa_status_t psa_key_derivation_input_key (psa_crypto_generator_t ∗generator, psa_key_derivation_↩step_t step, psa_key_handle_t handle)
- psa_status_t psa_key_agreement (psa_crypto_generator_t ∗generator, psa_key_derivation_step_t step, psa_key_handle_t private_key, const uint8_t ∗peer_key, size_t peer_key_length)
- psa_status_t psa_key_agreement_raw_shared_secret (psa_algorithm_t alg, psa_key_handle_t private_key, const uint8_t ∗peer_key, size_t peer_key_length, uint8_t ∗output, size_t output_size, size_t ∗output_length)
- psa_status_t psa_generate_random (uint8_t ∗output, size_t output_size)

  Generate random bytes.
- psa_status_t psa_generate_key (psa_key_handle_t handle, psa_key_type_t type, size_t bits, const void ∗extra, size_t extra_size)

  Generate a key or key pair.

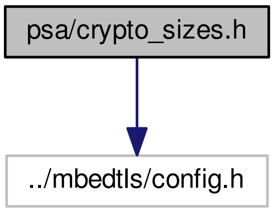
6.1.1 Detailed Description

Platform Security Architecture cryptography module.

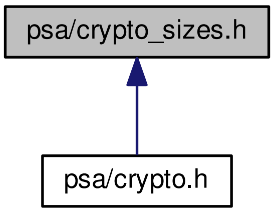
6.2 psa/crypto_sizes.h File Reference

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_BITS_TO_BYTES**(bits) (((bits) + 7) / 8)
- #define **PSA_BYTES_TO_BITS**(bytes) ((bytes) ∗ 8)
- #define PSA_HASH_SIZE(alg)
- #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_AEAD_TAG_LENGTH(alg)
- #define **PSA_VENDOR_RSA_MAX_KEY_BITS** 4096
- #define **PSA_VENDOR_ECC_MAX_CURVE_BITS** 521
- #define PSA_ALG_TLS12_PSK_TO_MS_MAX_PSK_LEN 128
- #define PSA_ASYMMETRIC_SIGNATURE_MAX_SIZE
- #define PSA_MAX_BLOCK_CIPHER_BLOCK_SIZE 16
- #define PSA_MAC_FINAL_SIZE(key_type, key_bits, alg)

- #define PSA_AEAD_ENCRYPT_OUTPUT_SIZE(alg, plaintext_length)
- #define PSA_AEAD_FINISH_OUTPUT_SIZE(alg, plaintext_length) ((size_t)0)
- #define PSA_AEAD_DECRYPT_OUTPUT_SIZE(alg, ciphertext_length)
- #define **PSA_RSA_MINIMUM_PADDING_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_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)
- #define **PSA_KEY_EXPORT_ASN1_INTEGER_MAX_SIZE**(bits) ((bits) / 8 + 5)
- #define **PSA_KEY_EXPORT_RSA_PUBLIC_KEY_MAX_SIZE**(key_bits) (PSA_KEY_EXPORT_ASN1_IN↩ TEGER_MAX_SIZE(key_bits) + 36)
- #define **PSA_KEY_EXPORT_RSA_KEYPAIR_MAX_SIZE**(key_bits) (9 ∗ PSA_KEY_EXPORT_ASN1_IN↩ TEGER_MAX_SIZE((key_bits) / 2 + 1) + 14)
- #define **PSA_KEY_EXPORT_DSA_PUBLIC_KEY_MAX_SIZE**(key_bits) (PSA_KEY_EXPORT_ASN1_IN↩ TEGER_MAX_SIZE(key_bits) ∗ 3 + 59)
- #define **PSA_KEY_EXPORT_DSA_KEYPAIR_MAX_SIZE**(key_bits) (PSA_KEY_EXPORT_ASN1_INTE↩ GER_MAX_SIZE(key_bits) ∗ 3 + 75)
- #define **PSA_KEY_EXPORT_ECC_PUBLIC_KEY_MAX_SIZE**(key_bits) (2 ∗ PSA_BITS_TO_BYTES(key↩ _bits) + 36)
- #define **PSA_KEY_EXPORT_ECC_KEYPAIR_MAX_SIZE**(key_bits) (PSA_BITS_TO_BYTES(key_bits))
- #define PSA_KEY_EXPORT_MAX_SIZE(key_type, key_bits)

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

Value:

```
(PSA_AEAD_TAG_LENGTH(alg) != 0 ?                              \
     (plaintext_length) - PSA_AEAD_TAG_LENGTH(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 #define PSA_AEAD_ENCRYPT_OUTPUT_SIZE(*alg,* *plaintext_length*)

Value:

```
(PSA_AEAD_TAG_LENGTH(alg) != 0 ?                              \
     (plaintext_length) + PSA_AEAD_TAG_LENGTH(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 #define PSA_AEAD_FINISH_OUTPUT_SIZE(*alg,* *plaintext_length*) ((size_t)0)

The maximum size of the output of psa_aead_finish(), in bytes.

If the size of the ciphertext buffer is at least this large, it is guaranteed that psa_aead_finish() will not fail due to an insufficient buffer size. Depending on the algorithm, the actual size of the ciphertext may be smaller.

Parameters

alg	An AEAD algorithm (`PSA_ALG_XXX` value such that PSA_ALG_IS_AEAD(alg) is true).

Returns

The maximum trailing ciphertext size for the specified algorithm. If the AEAD algorithm is not recognized, return 0. An implementation may return either 0 or a correct size for an AEAD algorithm that it recognizes, but does not support.

6.2.2.4 #define PSA_AEAD_TAG_LENGTH(*alg*)

Value:

```
(PSA_ALG_IS_AEAD(alg) ?                                    \
     (((alg) & PSA_ALG_AEAD_TAG_LENGTH_MASK) >> PSA_AEAD_TAG_LENGTH_OFFSET) : \
     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.

6.2.2.5 #define PSA_ALG_TLS12_PSK_TO_MS_MAX_PSK_LEN 128

This macro returns the maximum length of the PSK supported by the TLS-1.2 PSK-to-MS key derivation.

Quoting RFC 4279, Sect 5.3: TLS implementations supporting these ciphersuites MUST support arbitrary PSK identities up to 128 octets in length, and arbitrary PSKs up to 64 octets in length. Supporting longer identities and keys is RECOMMENDED.

Therefore, no implementation should define a value smaller than 64 for PSA_ALG_TLS12_PSK_TO_MS_MAX_↩ PSK_LEN.

6.2.2.6 #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.

6.2.2.7 #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.

6.2.2.8 #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.

6.2.2.9 #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.

6.2.2.10 #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.

6.2.2.11 #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.

6.2.2.12 #define PSA_HASH_SIZE(*alg*)

Value:

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

6.2.2.13 #define PSA_KEY_EXPORT_MAX_SIZE(*key_type*, *key_bits*)

Value:

```
(PSA_KEY_TYPE_IS_UNSTRUCTURED(key_type) ? PSA_BITS_TO_BYTES(key_bits) : \
     (key_type) == PSA_KEY_TYPE_RSA_KEYPAIR ? PSA_KEY_EXPORT_RSA_KEYPAIR_MAX_SIZE(
      key_bits) : \
     (key_type) == PSA_KEY_TYPE_RSA_PUBLIC_KEY ?
      PSA_KEY_EXPORT_RSA_PUBLIC_KEY_MAX_SIZE(key_bits) : \
     (key_type) == PSA_KEY_TYPE_DSA_KEYPAIR ? PSA_KEY_EXPORT_DSA_KEYPAIR_MAX_SIZE(
      key_bits) : \
     (key_type) == PSA_KEY_TYPE_DSA_PUBLIC_KEY ?
      PSA_KEY_EXPORT_DSA_PUBLIC_KEY_MAX_SIZE(key_bits) : \
     PSA_KEY_TYPE_IS_ECC_KEYPAIR(key_type) ? PSA_KEY_EXPORT_ECC_KEYPAIR_MAX_SIZE
      (key_bits) : \
     PSA_KEY_TYPE_IS_ECC_PUBLIC_KEY(key_type) ?
      PSA_KEY_EXPORT_ECC_PUBLIC_KEY_MAX_SIZE(key_bits) : \
     0)
```

Safe output buffer size for psa_export_key() or psa_export_public_key().

This macro returns a compile-time constant if its arguments are compile-time constants.

Warning

This function may call its arguments multiple times or zero times, so you should not pass arguments that contain side effects.

The following code illustrates how to allocate enough memory to export a key by querying the key type and size at runtime.

```
psa_key_type_t key_type;
size_t key_bits;
psa_status_t status;
status = psa_get_key_information(key, &key_type, &key_bits);
if (status != PSA_SUCCESS) handle_error(...);
size_t buffer_size = PSA_KEY_EXPORT_MAX_SIZE(key_type, key_bits);
unsigned char *buffer = malloc(buffer_size);
if (buffer != NULL) handle_error(...);
size_t buffer_length;
status = psa_export_key(key, buffer, buffer_size, &buffer_length);
if (status != PSA_SUCCESS) handle_error(...);
```

For psa_export_public_key(), calculate the buffer size from the public key type. You can use the macro PSA_KE↩Y_TYPE_PUBLIC_KEY_OF_KEYPAIR to convert a key pair type to the corresponding public key type.

```
psa_key_type_t key_type;
size_t key_bits;
psa_status_t status;
status = psa_get_key_information(key, &key_type, &key_bits);
if (status != PSA_SUCCESS) handle_error(...);
psa_key_type_t public_key_type =
      PSA_KEY_TYPE_PUBLIC_KEY_OF_KEYPAIR(key_type);
size_t buffer_size = PSA_KEY_EXPORT_MAX_SIZE(public_key_type, key_bits);
unsigned char *buffer = malloc(buffer_size);
if (buffer != NULL) handle_error(...);
size_t buffer_length;
status = psa_export_public_key(key, buffer, buffer_size, &buffer_length);
if (status != PSA_SUCCESS) handle_error(...);
```

Parameters

key_type	A supported key type.
key_bits	The size of the key in bits.

Returns

If the parameters are valid and supported, return a buffer size in bytes that guarantees that psa_asymmetric↩_sign() will not fail with PSA_ERROR_BUFFER_TOO_SMALL. If the parameters are a valid combination that is not supported by the implementation, this macro either shall return either a sensible size or 0. If the parameters are not valid, the return value is unspecified.

6.2.2.14 #define PSA_MAC_FINAL_SIZE(*key_type, key_bits, alg*)

Value:

```
((alg) & PSA_ALG_MAC_TRUNCATION_MASK ? PSA_MAC_TRUNCATED_LENGTH(alg) :
        \
     PSA_ALG_IS_HMAC(alg) ? PSA_HASH_SIZE(PSA_ALG_HMAC_GET_HASH(alg)) : \
     PSA_ALG_IS_BLOCK_CIPHER_MAC(alg) ?
      PSA_BLOCK_CIPHER_BLOCK_SIZE(key_type) : \
     ((void)(key_type), (void)(key_bits), 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.

6.2.2.15 #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.

6.2.2.16 #define PSA_MAX_BLOCK_CIPHER_BLOCK_SIZE 16

The maximum size of a block cipher supported by the implementation.

6.2.2.17 #define PSA_RSA_MINIMUM_PADDING_SIZE(*alg*)

Value:

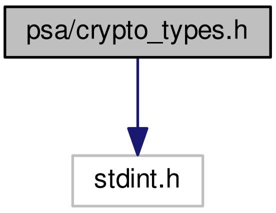
```
(PSA_ALG_IS_RSA_OAEP(alg) ?                                  \
     2 * PSA_HASH_FINAL_SIZE(PSA_ALG_RSA_OAEP_GET_HASH(alg)) + 1 :      \
     11 /*PKCS#1v1.5*/)
```

6.3 psa/crypto_types.h File Reference

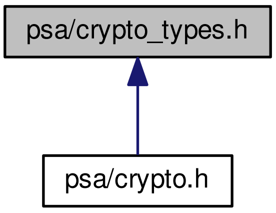
PSA cryptography module: type aliases.

```
#include <stdint.h>
```

Include dependency graph for crypto_types.h:



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



Typedefs

- 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_lifetime_t
- typedef uint32_t psa_key_id_t
- typedef uint32_t psa_key_usage_t

  Encoding of permitted usage on a key.
- typedef uint16_t psa_key_derivation_step_t

  Encoding of the step of a key derivation.

6.3.1 Detailed Description

PSA cryptography module: type aliases.

Note

This file may not be included directly. Applications must include psa/crypto.h. Drivers must include the appropriate driver header file.

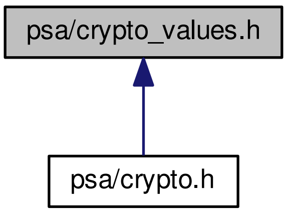
This file contains portable definitions of integral types for properties of cryptographic keys, designations of cryptographic algorithms, and error codes returned by the library.

This header file does not declare any function.

6.4 psa/crypto_values.h File Reference

PSA cryptography module: macros to build and analyze integer values.

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



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_ERROR_INVALID_HANDLE ((psa_status_t)19)
- #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)0x70000000)
- #define **PSA_KEY_TYPE_CATEGORY_SYMMETRIC** ((psa_key_type_t)0x40000000)
- #define **PSA_KEY_TYPE_CATEGORY_RAW** ((psa_key_type_t)0x50000000)
- #define **PSA_KEY_TYPE_CATEGORY_PUBLIC_KEY** ((psa_key_type_t)0x60000000)
- #define **PSA_KEY_TYPE_CATEGORY_KEY_PAIR** ((psa_key_type_t)0x70000000)
- #define **PSA_KEY_TYPE_CATEGORY_FLAG_PAIR** ((psa_key_type_t)0x10000000)
- #define PSA_KEY_TYPE_IS_VENDOR_DEFINED(type) (((type) & PSA_KEY_TYPE_VENDOR_FLAG) != 0)
- #define PSA_KEY_TYPE_IS_UNSTRUCTURED(type)
- #define PSA_KEY_TYPE_IS_ASYMMETRIC(type)
- #define PSA_KEY_TYPE_IS_PUBLIC_KEY(type) (((type) & PSA_KEY_TYPE_CATEGORY_MASK) == P↩
  SA_KEY_TYPE_CATEGORY_PUBLIC_KEY)
- #define PSA_KEY_TYPE_IS_KEYPAIR(type) (((type) & PSA_KEY_TYPE_CATEGORY_MASK) == PSA_↩
  KEY_TYPE_CATEGORY_KEY_PAIR)
- #define PSA_KEY_TYPE_KEYPAIR_OF_PUBLIC_KEY(type) ((type) | PSA_KEY_TYPE_CATEGORY_F↩
  LAG_PAIR)
- #define PSA_KEY_TYPE_PUBLIC_KEY_OF_KEYPAIR(type) ((type) & ~PSA_KEY_TYPE_CATEGORY↩
  _FLAG_PAIR)
- #define PSA_KEY_TYPE_RAW_DATA ((psa_key_type_t)0x50000001)
- #define PSA_KEY_TYPE_HMAC ((psa_key_type_t)0x51000000)
- #define PSA_KEY_TYPE_DERIVE ((psa_key_type_t)0x52000000)
- #define PSA_KEY_TYPE_AES ((psa_key_type_t)0x40000001)
- #define PSA_KEY_TYPE_DES ((psa_key_type_t)0x40000002)
- #define PSA_KEY_TYPE_CAMELLIA ((psa_key_type_t)0x40000003)
- #define PSA_KEY_TYPE_ARC4 ((psa_key_type_t)0x40000004)
- #define PSA_KEY_TYPE_RSA_PUBLIC_KEY ((psa_key_type_t)0x60010000)
- #define PSA_KEY_TYPE_RSA_KEYPAIR ((psa_key_type_t)0x70010000)
- #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_DSA_PUBLIC_KEY ((psa_key_type_t)0x60020000)
- #define PSA_KEY_TYPE_DSA_KEYPAIR ((psa_key_type_t)0x70020000)
- #define PSA_KEY_TYPE_IS_DSA(type) (PSA_KEY_TYPE_PUBLIC_KEY_OF_KEYPAIR(type) == PSA_↩
  KEY_TYPE_DSA_PUBLIC_KEY)
- #define **PSA_KEY_TYPE_ECC_PUBLIC_KEY_BASE** ((psa_key_type_t)0x60030000)
- #define **PSA_KEY_TYPE_ECC_KEYPAIR_BASE** ((psa_key_type_t)0x70030000)
- #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_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_KEY_TYPE_DH_PUBLIC_KEY ((psa_key_type_t)0x60040000)
- #define PSA_KEY_TYPE_DH_KEYPAIR ((psa_key_type_t)0x70040000)
- #define PSA_KEY_TYPE_IS_DH(type) (PSA_KEY_TYPE_PUBLIC_KEY_OF_KEYPAIR(type) == PSA_K↩ EY_TYPE_DH_PUBLIC_KEY)
- #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_DERIVATION** ((psa_algorithm_t)0x20000000)
- #define **PSA_ALG_CATEGORY_KEY_AGREEMENT** ((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_KEY_SELECTION_FLAG** ((psa_algorithm_t)0x01000000)
- #define PSA_ALG_IS_KEY_AGREEMENT(alg)

- #define PSA_ALG_IS_KEY_DERIVATION(alg) (((alg) & PSA_ALG_CATEGORY_MASK) == PSA_ALG_↩ CATEGORY_KEY_DERIVATION)
- #define PSA_ALG_IS_KEY_SELECTION(alg) (((alg) & PSA_ALG_CATEGORY_MASK) == PSA_ALG_C↩ ATEGORY_KEY_SELECTION)
- #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_ANY_HASH ((psa_algorithm_t)0x010000ff)
- #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_GET_HASH**(hmac_alg) (PSA_ALG_CATEGORY_HASH | ((hmac_alg) & PSA↩ _ALG_HASH_MASK))
- #define PSA_ALG_IS_HMAC(alg)
- #define **PSA_ALG_MAC_TRUNCATION_MASK** ((psa_algorithm_t)0x00003f00)
- #define **PSA_MAC_TRUNCATION_OFFSET** 8
- #define PSA_ALG_TRUNCATED_MAC(alg, mac_length)
- #define PSA_ALG_FULL_LENGTH_MAC(alg) ((alg) & ∼PSA_ALG_MAC_TRUNCATION_MASK)
- #define PSA_MAC_TRUNCATED_LENGTH(alg) (((alg) & PSA_ALG_MAC_TRUNCATION_MASK) >> P↩ SA_MAC_TRUNCATION_OFFSET)
- #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_BLOCK_CIPHER_MAC(alg)
- #define **PSA_ALG_CIPHER_STREAM_FLAG** ((psa_algorithm_t)0x00800000)
- #define **PSA_ALG_CIPHER_FROM_BLOCK_FLAG** ((psa_algorithm_t)0x00400000)
- #define PSA_ALG_IS_STREAM_CIPHER(alg)
- #define PSA_ALG_ARC4 ((psa_algorithm_t)0x04800001)
- #define PSA_ALG_CTR ((psa_algorithm_t)0x04c00001)
- #define **PSA_ALG_CFB** ((psa_algorithm_t)0x04c00002)
- #define **PSA_ALG_OFB** ((psa_algorithm_t)0x04c00003)
- #define PSA_ALG_XTS ((psa_algorithm_t)0x044000ff)
- #define PSA_ALG_CBC_NO_PADDING ((psa_algorithm_t)0x04600100)
- #define PSA_ALG_CBC_PKCS7 ((psa_algorithm_t)0x04600101)
- #define **PSA_ALG_CCM** ((psa_algorithm_t)0x06001001)
- #define **PSA_ALG_GCM** ((psa_algorithm_t)0x06001002)
- #define **PSA_ALG_AEAD_TAG_LENGTH_MASK** ((psa_algorithm_t)0x00003f00)
- #define **PSA_AEAD_TAG_LENGTH_OFFSET** 8
- #define PSA_ALG_AEAD_WITH_TAG_LENGTH(alg, tag_length)
- #define PSA_ALG_AEAD_WITH_DEFAULT_TAG_LENGTH(alg)
- #define **PSA__ALG_AEAD_WITH_DEFAULT_TAG_LENGTH__CASE**(alg, ref)

- #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_IS_HASH_AND_SIGN(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)0x20000100)
- #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_ALG_TLS12_PRF_BASE** ((psa_algorithm_t)0x20000200)
- #define PSA_ALG_TLS12_PRF(hash_alg) (PSA_ALG_TLS12_PRF_BASE | ((hash_alg) & PSA_ALG_HA↩ SH_MASK))

- #define PSA_ALG_IS_TLS12_PRF(alg) (((alg) & ~PSA_ALG_HASH_MASK) == PSA_ALG_TLS12_PRF←_BASE)
- #define **PSA_ALG_TLS12_PRF_GET_HASH**(hkdf_alg) (PSA_ALG_CATEGORY_HASH | ((hkdf_alg) & P←SA_ALG_HASH_MASK))
- #define **PSA_ALG_TLS12_PSK_TO_MS_BASE** ((psa_algorithm_t)0x20000300)
- #define PSA_ALG_TLS12_PSK_TO_MS(hash_alg) (PSA_ALG_TLS12_PSK_TO_MS_BASE | ((hash_alg) & PSA_ALG_HASH_MASK))
- #define PSA_ALG_IS_TLS12_PSK_TO_MS(alg) (((alg) & ~PSA_ALG_HASH_MASK) == PSA_ALG_TL←S12_PSK_TO_MS_BASE)
- #define **PSA_ALG_TLS12_PSK_TO_MS_GET_HASH**(hkdf_alg) (PSA_ALG_CATEGORY_HASH | ((hkdf←_alg) & PSA_ALG_HASH_MASK))
- #define **PSA_ALG_KEY_DERIVATION_MASK** ((psa_algorithm_t)0x080fffff)
- #define **PSA_ALG_KEY_AGREEMENT_MASK** ((psa_algorithm_t)0x10f00000)
- #define PSA_ALG_KEY_AGREEMENT(ka_alg, kdf_alg) ((ka_alg) | (kdf_alg))
- #define **PSA_ALG_KEY_AGREEMENT_GET_KDF**(alg) (((alg) & PSA_ALG_KEY_DERIVATION_MASK) | PSA_ALG_CATEGORY_KEY_DERIVATION)
- #define **PSA_ALG_KEY_AGREEMENT_GET_BASE**(alg) (((alg) & PSA_ALG_KEY_AGREEMENT_MASK) | PSA_ALG_CATEGORY_KEY_AGREEMENT)
- #define **PSA_ALG_IS_RAW_KEY_AGREEMENT**(alg) (PSA_ALG_KEY_AGREEMENT_GET_KDF(alg) == PSA_ALG_CATEGORY_KEY_DERIVATION)
- #define **PSA_ALG_IS_KEY_DERIVATION_OR_AGREEMENT**(alg) ((PSA_ALG_IS_KEY_DERIVATI←ON(alg) || PSA_ALG_IS_KEY_AGREEMENT(alg)))
- #define PSA_ALG_FFDH ((psa_algorithm_t)0x30100000)
- #define PSA_ALG_IS_FFDH(alg) (PSA_ALG_KEY_AGREEMENT_GET_BASE(alg) == PSA_ALG_FFDH)
- #define PSA_ALG_ECDH ((psa_algorithm_t)0x30200000)
- #define PSA_ALG_IS_ECDH(alg) (PSA_ALG_KEY_AGREEMENT_GET_BASE(alg) == PSA_ALG_ECDH)
- #define PSA_ALG_IS_WILDCARD(alg)
- #define PSA_ALG_IS_WILDCARD(alg)
- #define PSA_KEY_LIFETIME_VOLATILE ((psa_key_lifetime_t)0x00000000)
- #define PSA_KEY_LIFETIME_PERSISTENT ((psa_key_lifetime_t)0x00000001)
- #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_KDF_STEP_SECRET ((psa_key_derivation_step_t)0x0101)
- #define PSA_KDF_STEP_LABEL ((psa_key_derivation_step_t)0x0201)
- #define PSA_KDF_STEP_SALT ((psa_key_derivation_step_t)0x0202)
- #define PSA_KDF_STEP_INFO ((psa_key_derivation_step_t)0x0203)

6.4.1 Detailed Description

PSA cryptography module: macros to build and analyze integer values.

Note

This file may not be included directly. Applications must include psa/crypto.h. Drivers must include the appropriate driver header file.

This file contains portable definitions of macros to build and analyze values of integral types that encode properties of cryptographic keys, designations of cryptographic algorithms, and error codes returned by the library.

This header file only defines preprocessor macros.

Index