# RM00298

## High Assurance Boot Version 4 API Reference Manual

Rev. 2.0 — 5 December 2024 | Reference manual

**Document information**

| Information | Content |
|---|---|
| Keywords | RM00298, HAB, HABv4, code signing tool, CST, i.MX |
| Abstract | This document provides details on the application programming interface (API) for the NXP High Assurance Boot (HAB) library. |

# 1 About this book

This document provides details on the application programming interface (API) for the NXP High Assurance Boot (HAB) library. The HAB library is included as a component of the boot read-only memory (ROM) on some NXP processors. The HAB API allows image code, external to the ROM, to make calls back to the HAB for authenticating other boot stages.

**Audience**:

This document describes the details of the HAB API for engineers architecting and implementing a secure boot.

**Scope**:

This document describes the API for HAB version 4 only. For information on the HAB version 3 API, refer to the "System Boot" section of the relevant NXP processor reference manual.

**Organization**:

This manual is divided into the following sections:

- Section 2 "Introduction": Provides an overview of the HAB API.
- Section 3 "Functions": Provides detailed description of the HAB API functions.
- Section 4 "Data structures": Describes the data structures used by HAB including device configuration and command sequence file commands.
- Section 5 "Security hardware": Describes the security hardware block used by HAB.
- Section 6 "Constants": Defines constant values that are used by the HAB API.
- Section 7 "Interpreting HAB event data from report_event() API": Illustrates how to interpret the event data returned from the `report_event()` API.

## 2 Introduction

The HAB library included in the on-chip ROMs of some NXP processors provides a means to authenticate and sometimes, even encrypt execution images. This secure boot process starts with the ROM authenticating the first image in the boot flow, which is typically a bootloader, such as U-Boot.

The HAB library provides several APIs for image authentication. These APIs can be called from boot images, such as U-Boot, to extend the secure boot chain further. This process is illustrated in Figure 1.

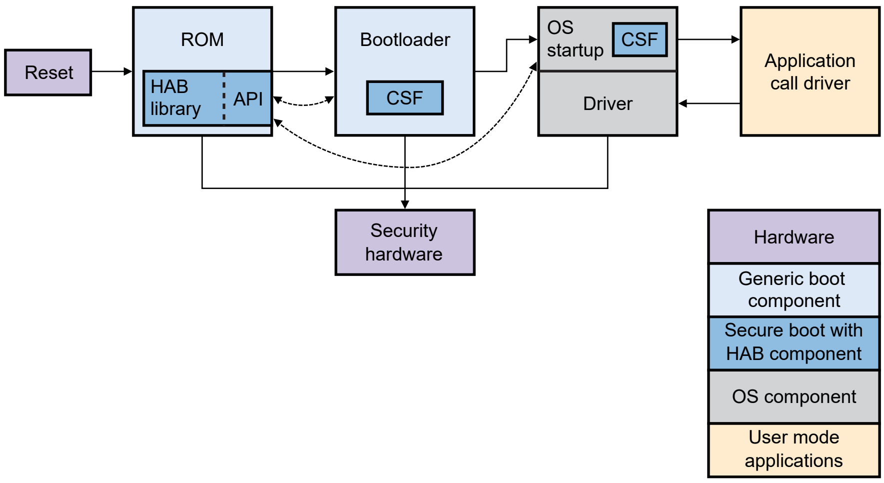


**Figure 1. Secure boot with HAB**

Boot components, such as U-Boot, can use these APIs by locating the ROM vector table (RVT), which is a table of the HAB API addresses. The RVT address located in the ROM is specific to each NXP processor. To determine the RVT address, refer to the "System Boot" section of the relevant NXP processor reference manual.

The remainder of this document provides the details for all HAB API functions, HAB commands, and engines used by HAB.

***Note:*** *Even though all APIs are documented in this document, NXP recommends using the hab_rvt.authenticate_image() API whenever possible, instead of calling other APIs separately. It ensures that all authentication steps are performed properly.*

For more details and examples illustrating the use of the HAB APIs, obtain *Secure Boot with i.MX28 HAB v4* (AN4555) and *i.MX Secure Boot on HABv4 Supported Devices* (AN4581) from nxp.com.

# 3 Functions

This section describes each of the HAB API functions. The addresses of the API functions are collected in a data structure called the ROM Vector Table (RVT). The RVT is placed at a fixed location in the ROM. For RVT address details, refer to the "System Boot" section of the relevant NXP processor reference manual.

## 3.1 Entry

**Usage**:

```
hab_status_t(*hab_rvt::entry)(void)
```

Enters and initializes the HAB library.

**Purpose**:

This function initializes the HAB library and security hardware plug-ins. Post-ROM boot stage components use this function via the ROM vector table, before calling a HAB function other than hab_rvt.report_event() and hab_rvt.report_status().

**Operations**:

This function performs the following operations every time it is called:

- Initialize the HAB library internal state.
- Initialize the internal secret key store (cleared at the next hab_rvt.exit()).
- Run the entry sequence of each available security hardware plug-in.

When called from boot ROM for the first time, this function performs the following operations before performing the above operations:

- Initialize the internal public key store (persists beyond hab_rvt.exit()).
- Run the self-test sequence of each available security hardware plug-in.
- If a state machine is present and enabled, change the security state as follows:
  - If the IC is configured as HAB_CFG_OPEN or HAB_CFG_RETURN, move to HAB_STATE_NONSECURE.
  - If the IC is configured as HAB_CFG_CLOSED, move to HAB_STATE_TRUSTED.
  - Otherwise, leave the security state unchanged.

If any failure occurs in the operations above:

- An audit event is logged.
- All remaining operations are abandoned (except that all security hardware self-test and entry sequences are still executed).
- If a state machine is present and enabled, the security state is set as follows:
  - Move to HAB_STATE_NONSECURE.
    ***Note:*** *If the hardware detects a security violation, the final state is HAB_STATE_FAIL_SOFT or HAB_STATE_FAIL_HARD, depending on the hardware configuration.*

***Warning:*** *A boot sequence may contain several images to be launched one after another. It may also contain some alternative images to be used when one boot device or boot image be unavailable or unusable. The authentication of each image in a boot sequence must be limited to its hab_rvt.entry() / hab_rvt.exit() pair. It ensures that the security state information gathered for one image is not applied to any other image.*

**Postconditions**:

- HAB library internal state is initialized.
- Available security hardware plug-ins are initialized.

- If a failure or warning occurs during security hardware plug-in initialization, an audit event is logged with the relevant engine tag. The status and reason logged are described in the relevant security hardware plug-in documentation.
- If a state machine is present and enabled, the security state is initialized.

**Return values**:

- HAB_SUCCESS, for an IC not configured as HAB_CFG_CLOSED; though unsuccessful operations still generate audit log events.
- HAB_SUCCESS, for other ICs if all commands were completed successfully without failure (even if warnings were generated).
- HAB_FAILURE otherwise.

## 3.2 Get version

**Usage**:

```
uint32_t(*hab_rvt::get_version)(void)
```

Gets the HAB version.

**Purpose**:

This function returns the version of the HAB library. The ROM boot stage uses this function via the ROM vector table.

**Operations**:

This function performs the following operations:

- Returns HAB library version.

**Return values**:

- HAB library version 32-bit unsigned integer

## 3.3 Check target

**Usage**:

```
hab_status_t(*hab_rvt::check_target)(hab_target_t type,
                                     const void *start,
                                     size_t bytes)
```

Checks the target address.

**Purpose**:

This function reports whether a given target region is allowed for either peripheral configuration or image loading in the memory. Post-ROM boot stage components use this function via the ROM vector table to avoid:

- Configuring security-sensitive peripherals.
- Loading images over sensitive memory regions or outside the recognized memory devices in the address map.

**Operations**:

The list of allowed target regions vary by IC and core. To get the details, refer to the relevant NXP processor reference manual.

**Parameters**:

| Parameter | Type | Description |
|---|---|---|
| *type* | `[in]` | Indicates the type of the target (memory or peripheral). |
| *start* | `[in]` | Indicates the address of the target region. |
| *bytes* | `[in]` | Indicates the size of the target region. |

**Postconditions**:

- If the given target region goes beyond the allowed regions, an audit event is logged with status HAB_FAILURE and reason HAB_INV_ADDRESS, together with the function call parameters. For more details, see the Events section.
- For successful commands, no audit event is logged.

**Return values**:

- HAB_SUCCESS, for an IC not configured as HAB_CFG_CLOSED; though unsuccessful operations still generate audit log events.
- HAB_SUCCESS, if the given target region falls completely within the allowed regions for the requested type of target.
- HAB_FAILURE otherwise.

**Definitions**:

```
enum hab_target {
  HAB_TGT_MEMORY = 0x0f, /* Check memory white list */
  HAB_TGT_PERIPHERAL = 0xf0, /* Check peripheral white list*/
  HAB_TGT_ANY = 0x55, /**< Check memory & peripheral white list */
} hab_target_t
```

## 3.4 Authenticate image

**Usage**:

```
hab_image_entry_f(*hab_rvt::authenticate_image)(
                            uint8_t cid,
                            ptrdiff_t ivt_offset,
                            void **start,
                            size_t *bytes,
                            hab_loader_callback_f loader)
```

Authenticates the image.

**Purpose**:

This function combines device configuration data (DCD), command sequence file (CSF), and assert functions in a standard sequence to authenticate a loaded image. Post-ROM boot stage components use this function via the ROM vector table. Support for images partially loaded to an initial location is provided via a callback function.

***Note:***

- *After the boot process exits the ROM, do not use the DCD based SoC initialization mechanism.*

- *A non-ROM user can use only the hab_rvt.authenticate_image_no_dcd() function (in place of the hab_rvt.authenticate_image() function) or must ensure that a null DCD pointer is passed as an argument.*
- *If called from outside the boot ROM, each of the following functions returns an error (applicable for HAB version 4.3.7 or higher):*
  - *The hab_rvt.run_dcd() function*
  - *The hab_rvt.authenticate_image() function called with a non-null DCD pointer*
- *Older versions of HAB run DCD commands if available. It may lead to an incorrect authentication boot flow.*

**Operations**:

This function performs the following sequence of operations:

- Verify that the initial image load addresses pass hab_rvt.check_target().
- Verify that the image vector table (IVT) offset falls within the initial image bounds.
- Verify that the *self* and *entry* pointers of the image vector table are not null.
- Verify the image vector table header for consistency and compatibility.
- If provided in the image vector table, calculate the device configuration data initial location, verify that it falls within the initial image bounds, and run the device configuration data commands.
- If provided in the image vector table, calculate the boot data initial location and verify that it falls within the initial image bounds.
- If provided in the parameters, invoke the callback function with the initial image bounds and initial location of the image vector table boot data.

From this point on, the full image is assumed to be in its final location. The following operations are performed on all IC configurations, but are enforced only on ICs that are configured as HAB_CFG_CLOSED:

- Verify that the final image load addresses pass hab_rvt.check_target().
- Verify that the CSF falls within the image bounds and runs the CSF commands.
- Verify that the following data has been authenticated (using their final locations):
  - IVT
  - DCD (if provided)
  - Boot data (initial byte if provided)
  - Entry point (initial word)

**Parameters**:

| Parameter | Type | Description |
|---|---|---|
| *cid* | `[in]` | Indicates the caller ID, which determines the software that issued this call. |
| *ivt_offset* | `[in]` | Indicates the address of the target region. |
| *start* | `[in,out]` | Indicates the initial (possibly partial) image load address on entry or the final image load address on exit. |
| *bytes* | `[in,out]` | Indicates the initial (possibly partial) image size on entry or the final image size on exit. |
| *loader* | `[in]` | Indicates the callback function to load the full image to its final load address. If not required, set this parameter to null. |

**Remarks**:

- Caller ID can be bound to signatures that are verified using keys installed with the HAB_CMD_INS_KEY_CID flag. For more details, see the Install key section.
- A loader callback function can be supplied even if the image is already loaded to its final location on entry.
- If the loader callback function pointer is set to null, boot data (boot_data in image vector table) is ignored.

***Warning:***

- *The loader callback function must be used within existing authenticated areas.*
- *Before loading the initial image and calling the hab_rvt.authenticate_image() function, the caller must verify the initial image load addresses using the hab_rvt.check_target() function.*
- *After completion of the hab_rvt.authenticate_image() function, the caller must verify if the boot data was authenticated, using the hab_rvt.assert() function.*

**Postconditions**:

- The post-conditions of the functions hab_rvt.check_target(), hab_rvt.run_dcd(), hab_rvt.run_csf(), and hab_rvt.assert() apply also to this function. In particular, any audit events logged within the given functions have the context field appropriate to that function rather than HAB_CTX_AUTHENTICATE. In addition, the side effects and post conditions of any callback function supplied apply.
- If a failure or warning occurs outside these contexts, an audit event is logged with status:
  - HAB_FAILURE, with further reasons:
    - HAB_INV_ADDRESS: The initial or final image address is outside the allowed region.
    - HAB_INV_ADDRESS: IVT, DCD, boot data, or CSF is outside the image bounds.
    - HAB_INV_ADDRESS: The IVT *self* or *entry* pointer is null.
    - HAB_INV_CALL: hab_rvt.entry() is not run successfully before the call.
    - HAB_INV_IVT: IVT is malformed.
    - HAB_INV_IVT: The IVT version number is less than the HAB library version.
    - HAB_INV_RETURN: The callback function failed.

**Return values**:

- *entry*, field from image vector table for an IC not configured as HAB_CFG_CLOSED if the following conditions are met (other unsuccessful operations generate audit log events):
  - The start pointer and the pointer it locates are not null.
  - The initial image vector table location is not null.
  - The image vector table header (given in the *hdr* field) is valid.
  - The final image vector table location (given by the *self* field) is not null.
  - Any loader callback completed successfully.
- *entry*, field from image vector table for other ICs if all operations were completed successfully without failure (even if warnings were generated).
- *null* otherwise.

**Definitions**:

```
/* This typedef serves as the return type for
 * hab_rvt.authenticate_image().  It specifies a void-void function
 * pointer, but can be cast to another function
 * pointer type if required.
 */
typedef void (*hab_image_entry_f)(void);
```

### 3.4.1 Authenticate image – loader callback

**Usage**:

```
hab_status_t(*)hab_loader_callback_f(void **start,
                                     size_t *bytes,
                                     const void *boot_data)
```

 

Loader callback.

**Purpose**:

The library caller supplies this function, if required. This function finalizes image loading in boot modes where only a portion of the image is loaded to a temporary initial location, before device configuration.

**Operations**:

This function is called during the hab_rvt.authenticate_image() function between running the device configuration data and command sequence file. The caller defines the operation of this function.

**Parameters**:

| Parameter | Type | Description |
|---|---|---|
| *start* | `[in,out]` | Indicates the initial (possibly partial) image load address on entry or the final image load address on exit. |
| *bytes* | `[in,out]` | Indicates the initial (possibly partial) image size on entry or the final image size on exit. |
| *boot_data* | `[in]` | Indicates the initial load address of the image vector table boot data. |

**Remarks**:

- The caller defines the interpretation of the boot data. Different boot components or modes can use different boot data, or even different loader callback functions.

***Warning:***

- *This function must ensure that the boot data is valid/authentic.*
- *Before copying any image data, the loader callback must verify the final image load addresses, using hab_rvt.check_target().*

**Preconditions**:

- The (possibly partial) image has been loaded in the initial load address, and the boot data is within the initial image.
- The device configuration data has been run, if provided.

**Return values**:

- HAB_SUCCESS, if all operations are completed successfully.
- HAB_FAILURE otherwise.

## 3.5 Authenticate image no DCD

**Usage**:

```
hab_image_entry_f(*hab_rvt::authenticate_image_no_dcd)(
                                  uint8_t cid,
                                  ptrdiff_t ivt_offset,
                                  void **start,
                                  size_t *bytes,
                                  hab_loader_callback_f
                                  loader)
```

Authenticates the image without executing the DCD commands.

**Purpose**:

 

This function is identical to the hab_rvt.authenticate_image() function except that it does not run the DCD. For more details, see the Section 3.4 section.

## 3.6 Authenticate container

**Usage**:

```
hab_status_t(*hab_rvt:: authenticate_container)(
                                   uint8_t cid,
                                   ptrdiff_t ivt_offset,
                                   void **start,
                                   size_t *bytes,
                                   hab_loader_callback_f
                                   loader,
                                   uint32_t srkmask,
                                   int skip_dcd)
```

Authenticates the container.

**Purpose**:

This function is identical to the hab_rvt.authenticate_image() function except that:

- It supports different image vector table versions.
- It authenticates the contents by using a specified super root key (SRK) set.
- It can skip the DCD commands.

For more details, see the Section 3.4 section.

**Modified and additional parameters**:

| | | |
|---|---|---|
| ... [in] | *bytes* | Initial (possibly partial) image size on entry. |
| ... [in] | *srkmask* | Mask of bits that indicate which SRK set is allowed to authenticate the contents of this container. |
| [in] | *skip_dcd* | Set to disable DCD processing. |

**Return values**:

- HAB_SUCCESS, for an IC not configured as HAB_CFG_CLOSED if the following conditions are met (other unsuccessful operations generate audit log events):
  – The start pointer and the pointer it locates are not null.
  – The initial image vector table location is not null.
  – The image vector table header (given in the *hdr* field) is valid.
  – The final image vector table location (given by the *self* field) is not null.
  – Any loader callback completed successfully.
- HAB_SUCCESS, for other ICs if all operations were completed successfully without failure (even if warnings were generated).
- HAB_FAILURE otherwise.

## 3.7 Run DCD

**Usage**:

```
hab_status_t(*hab_rvt::run_dcd)(const uint8_t *dcd)
```

Executes the boot configuration script.

**Purpose**:

This function configures the IC based on a device configuration data table. Post-ROM boot stage components use this function via the ROM vector table. For each boot stage, this function can be invoked as often as required.

The difference between the configuration functionality in this function and the hab_rvt.run_csf() function arises because the device configuration data table is not authenticated before running the commands. Therefore, there is a more limited range of commands allowed, and a limited range of parameters to allowed commands.

***Note:***

- *After the boot process exits the ROM, do not use the DCD based SoC initialization mechanism.*
- *A non-ROM user does not need to use the hab_rvt.run_dcd() function.*
- *Each of the following functions returns an error if called from outside the boot ROM (applicable for HAB version 4.3.7 or higher):*
  - *The hab_rvt.run_dcd() function*
  - *The hab_rvt.authenticate_image() function called with a non-null DCD pointer*
- *Older versions of HAB run DCD commands if available. It may lead to an incorrect authentication boot flow.*

**Operations**:

This function performs the following operations:

- Checks the header for compatibility and consistency
- Makes an internal copy of the device configuration data table
- Executes the commands in sequence from the internal copy of the device configuration data

If any failure occurs, an audit event is logged, and all remaining operations are abandoned.

**Parameters**:

| Parameter | Type | Description |
| --- | --- | --- |
| *dcd* | `[in]` | Indicates the address of the device configuration data. |

***Warning:***

- *It is the responsibility of the caller to ensure that the dcd parameter points to a valid memory location.*
- *To avoid unauthorized configurations subverting a secure operation, a subsequent command sequence file command must authenticate the device configuration data, before launching the next boot image. This constraint can be enforced using the hab_rvt.assert() function, even if the contents of the CSF of the next boot stage are out of scope for the hab_rvt.run_dcd() caller. It ensures that both the DCD and any pointers used to locate it have been authenticated.*
- *Each invocation of the hab_rvt.run_dcd() function must occur between a pair of the hab_rvt.entry() and hab_rvt.exit() calls, though multiple hab_rvt.run_dcd() calls (and other HAB calls) can be made in one bracket. This constraint applies whether the hab_rvt.run_dcd() call is successful or not. A subsequent call to the hab_rvt.exit() function is required before launching the authenticated image or switching to another boot target.*

**Postconditions**:

- Many commands may cause side effects. For more details, see the device configuration data documentation.
- If a failure or warning occurs within a command handler, an audit event is logged with the offending command, copied from the DCD. The status and reason logged are described in the relevant command documentation.
- For other failures or warning, the status logged is:
  - HAB_WARNING, with further reasons:
    - HAB_UNS_COMMAND: An unsupported command is encountered, where the DCD version differs from the HAB library version.
  - HAB_FAILURE, with further reasons:
    - HAB_INV_ADDRESS: The *dcd* parameter is null.
    - HAB_INV_CALL: Before the call, the hab_rvt.entry() function did not run successfully.
    - HAB_INV_COMMAND: The command is not allowed in DCD.
    - HAB_UNS_COMMAND: An unrecognized command is encountered, where the DCD version matches the HAB library version.
    - HAB_INV_DCD: DCD is malformed or too large.
    - HAB_INV_DCD: The DCD version number is less than the HAB library version.

**Return values**:

- HAB_SUCCESS, for an IC not configured as HAB_CFG_CLOSED; though unsuccessful operations still generate audit log events.
- HAB_SUCCESS, for other ICs if all commands were completed successfully without failure (even if warnings were generated).
- HAB_FAILURE otherwise.

## 3.8 Run CSF

**Usage**:

```
hab_status_t(*hab_rvt::run_csf)(const uint8_t *csf,
                                uint8_t cid,
                                uint32_t srkmask)
```

Executes an authentication script.

**Purpose**:

This function authenticates software images and configures the IC based on a command sequence file. Post-ROM boot stage components use this function via the ROM vector table. For each boot stage, this function can be invoked as often as required.

**Operations**:

This function performs the following operations:

- Checks the header for compatibility and consistency
- Makes an internal copy of the command sequence file
- Executes the commands in sequence from the internal copy of the command sequence file

An explicit command in the sequence authenticates the internal copy of the command sequence file. Before authentication, only a limited set of commands is available to:

- Install a super root key (unless previously installed).
- Install a CSF key (unless previously installed).
- Specify any variable configuration items.
- Authenticate the CSF.

 

After CSF authentication, the full set of commands is available.

If any failure occurs, an audit event is logged, and all remaining operations are abandoned.

**Parameters**:

| Parameter | Type | Description |
|---|---|---|
| *csf* | `[in]` | Indicates the address of the command sequence file. |
| *cid* | `[in]` | Indicates the caller ID, which determines the software that issued this call. |
| *srkmask* | `[in]` | Indicates the mask of bits (SRK set) that is allowed to authenticate the contents of this CSF (bit 0 = OEM / bit 1 = NXP). If *srkmask* is equal to 0, HAB uses OEM by default. This argument was added in HAB 4.3.0. |

**Remarks**:

- Caller ID can be bound to signatures that are verified using keys installed with the HAB_CMD_INS_KEY_CID flag. For more details, see the Install key section.

***Warning:***

- *It is the responsibility of the caller to ensure that the csf parameter points to a valid memory location.*
- *Each invocation of the hab_rvt.run_csf() function must occur between a pair of the hab_rvt.entry() and hab_rvt.exit() calls, though multiple hab_rvt.run_csf() calls (and other HAB calls) can be made in one bracket. This constraint applies whether the hab_rvt.run_csf() call is successful or not. A subsequent call to the hab_rvt.exit() function is required before launching the authenticated image or switching to another boot target.*

**Postconditions**:

- Many commands may cause side effects. For more details, see the command sequence file documentation. ***Note:*** *Keys installed by the command sequence file remain available for use in subsequent operations.*
- If a failure or warning occurs within a command handler, an audit event is logged with the offending command, copied from the CSF. The status and reason logged are described in the relevant command documentation.
- For other failures or warning, the status logged is:
  - HAB_WARNING, with further reasons:
    - HAB_UNS_COMMAND: An unsupported command is encountered, where the CSF version differs from the HAB library version.
  - HAB_FAILURE, with further reasons:
    - HAB_INV_ADDRESS: The *csf* parameter is null.
    - HAB_INV_CALL: Before the call, the hab_rvt.entry() function did not run successfully.
    - HAB_INV_COMMAND: The command is not allowed before CSF authentication.
    - HAB_UNS_COMMAND: An unrecognized command is encountered, where the CSF version matches the HAB library version.
    - HAB_INV_CSF: CSF is not authenticated.
    - HAB_INV_CSF: CSF is malformed or too large.
    - HAB_INV_CSF: The CSF version number is less than the HAB library version.

**Return values**:

- HAB_SUCCESS, for an IC not configured as HAB_CFG_CLOSED; though unsuccessful operations still generate audit log events.
- HAB_SUCCESS, for other ICs if all commands were completed successfully without failure (even if warnings were generated).

- HAB_FAILURE otherwise.

## 3.9 Assert

**Usage**:

```
hab_status_t(*hab_rvt::assert)(hab_assertion_t type,
                               const void *data,
                               uint32_t count)
```

Tests an assertion against the audit log.

**Purpose**:

This function allows the audit log to be interrogated. Post-ROM boot stage components use this function via the ROM vector table to determine the status of authentication operations. For each boot stage, this function can be invoked as often as required.

**Operations**:

This function checks the required assertion as detailed below.

**Parameters**:

| Parameter | Type | Description |
|---|---|---|
| *type* | `[in]` | Assertion type |
| *data* | `[in]` | Assertion data |
| *count* | `[in]` | Data size or count |

**Memory block authentication**:

For HAB_ASSERT_BLOCK assertion type (defined as 0x0), the hab_rvt.assert() function verifies that the given memory block has been authenticated after running a CSF. The parameters are interpreted as follows:

- data: Memory block starting address
- count: Memory block size (in bytes)

A simple interpretation of "memory block has been authenticated" is taken, such that the given block must fall completely within a single contiguous block authenticated while running a CSF. A given memory block covered by the union of several neighboring or overlapping authenticated blocks could fail the test with this interpretation. However, it is assumed that such cases do not arise in practice.

**Postconditions**:

- If the assertion fails, an audit event is logged with status HAB_FAILURE and reason HAB_INV_ASSERTION, together with the function call parameters. For more details, see the event record documentation.
- For successful commands, no audit event is logged.

**Return values**:

- HAB_SUCCESS, for an IC not configured as HAB_CFG_CLOSED; though unsuccessful operations still generate audit log events.
- HAB_SUCCESS, for other ICs if the assertion is confirmed.
- HAB_FAILURE otherwise.

## 3.10 Report event

**Usage**:

```
hab_status_t(*hab_rvt::report_event)(hab_status_t status,
                                     uint32_t index,
                                     uint8_t *event,
                                     size_t *bytes)
```

Reports an event from the audit log.

**Purpose**:

This function allows the audit log to be interrogated. Post-ROM boot stage components use this function via the ROM vector table to determine the status of authentication operations. This function can be called outside a hab_rvt.entry() / hab_rvt.exit() pair.

The boot ROM can also use this function to report boot failures as part of a tethered boot protocol.

**Operations**:

This function performs the following operations:

- Scans the audit log for a matching event
- Copies the required details to the output parameters (if found)

**Parameters**:

| Parameter | Type | Description |
|---|---|---|
| *status* | `[in]` | Indicates the status level of the required event. |
| *index* | `[in]` | Indicates the index of the required event, at a given status level. |
| *event* | `[in]` | Indicates the event record. |
| *bytes* | `[in, out]` | Indicates the size of the *event* buffer on entry or the size of the event record on exit. |

**Remarks**:

- Use *status* = HAB_STS_ANY to match any logged event, regardless of the status value logged.
- Use *index* = 0 to return the first matching event, *index* = 1 to return the second matching event, and so on.
- The data logged with each event is context-dependent. For more details, see the event record documentation.

***Warning:***

- *Parameter bytes cannot be null.*
- *If the event buffer is a null pointer or too small to fit the event record, the required size is written to bytes. However, no part of the event record is copied to the output buffer.*

**Return values**:

- HAB_SUCCESS, if the required event is found and the event record is copied to the output buffer.
- HAB_SUCCESS, if the required event is found and the event buffer passed is a null pointer.
- HAB_FAILURE otherwise.

## 3.11 Report status

**Usage**:

```
hab_status_t(*hab_rvt::report_status)(
                                       hab_config_t *config,
                                       hab_state_t *state)
```

Reports the security status.

**Purpose**:

This function reports the security configuration and state of the IC. It also searches the audit log to determine the status of the boot process. Post-ROM boot stage components use this function via the ROM vector table. This function can also be called outside a hab_rvt.entry() / hab_rvt.exit() pair.

**Operations**:

This function reads the fuses that indicate the security configuration. The fuse map varies by IC. To get the details, refer to the relevant NXP processor reference manual.

To report the security state, this function also uses the security hardware state machine, if present and enabled.

**Parameters**:

| Parameter | Type | Description |
| --- | --- | --- |
| *config* | `[out]` | Indicates the security configuration, null if not required. |
| *state* | `[out]` | Indicates the security state, null if not required. |

**Remarks**:

- If no security hardware state machine is present and enabled, the state HAB_STATE_NONE is output.

**Return values**:

- HAB_SUCCESS, if no warning or failure audit event is logged.
- HAB_WARNING, if only warning events are logged.
- HAB_FAILURE otherwise.

## 3.12 Fail-Safe mode

**Usage**:

```
void(*hab_rvt::failsafe)(void)
```

Enters the Fail-Safe mode.

**Purpose**:

This function provides a safe path when image authentication has failed and all possible boot paths have been exhausted. Post-ROM boot stage components use this function via the ROM vector table.

**Operations**:

The exact details of this function vary by IC and core. To get the details, refer to the relevant NXP processor security reference manual.

***Warning:*** *This function does not return.*

**Remarks**:

- Because this function does not return, it implicitly performs the functionality of hab_rvt.exit() to ensure an appropriate configuration of the security hardware plug-ins. Two typical implementations are:
  - A low-level provisioning protocol in which an image is downloaded to the RAM from an external host, is authenticated, and is launched. The downloaded image may report the reasons for boot failure to the tools on the external host, resulting in reprovisioning the end product with authentic boot images.
  - A Fail-Safe boot mode that does not allow the execution to leave the ROM until the IC is reset.

## 3.13 Exit

**Usage**:

```
hab_status_t(*hab_rvt::exit)(void)
```

Finalizes and exits the HAB library.

**Purpose**:

This function finalizes the HAB library and security hardware plug-ins. Post-ROM boot stage components use this function via the ROM vector table:

- After calling other HAB functions.
- Before launching the next boot stage.
- Before switching to another boot path.

**Operations**:

This function performs the following operations:

- Finalize the HAB library internal state
- Clear the internal secret key store
- Run the finalization sequence of each available security hardware plug-in

If any failure occurs, an audit event is logged, and all remaining operations are abandoned (except that all security hardware exit sequences are still executed).

***Warning:*** *See warnings for the hab_rvt.entry() function.*

**Postconditions**:

- Records are cleared from the audit log.
  ***Note:*** *Other event records are not cleared.*
- Any public keys installed by command sequence file commands remain active.
- Any secret keys installed by command sequence file commands are deleted.
- Available security hardware plug-ins are in their final state as described in the relevant sections.
- If a failure or warning occurs, an audit event is logged with the engine tag of the security hardware plug-in concerned. The status and reason logged are described in the relevant security hardware plug-in documentation.

**Return values**:

- HAB_SUCCESS, for an IC not configured as HAB_CFG_CLOSED; though unsuccessful operations still generate audit log events.
- HAB_WARNING, for other ICs if all commands were completed successfully without failure (even if warnings were generated).
- HAB_FAILURE otherwise.

# 4 Data structures

**Detailed description**:

External data structures required or provided by HAB.

**Purpose**:

This section defines the data structures that HAB uses to guide image authentication and device configuration operation. It also mentions the information provided by HAB that can be used in authenticating boot sequence components after the execution leaves the boot ROM.

**Format**:

All data structures other than C "structs" are interpreted as byte arrays. Data structure diagrams in this document show the first byte (lowest address) in the top-left corner, with subsequent bytes read across the rows from left to right, and then down the page to the final byte (highest address) in the lower-right corner. Multi-byte byte fields such as addresses, offsets and other integers are in big-endian format, regardless of the underlying processor architecture.

No constraint is imposed on alignment for the start or end of data structures, though word alignment may improve the processing speed.

Each HAB data structure starts with a header. In the header, the *par* bit field contains the HAB version for which the data structure was constructed. Some structures contain fields of variable length or a variable number of fields; other structures are of fixed format.

| tag | len | V | v |
|---|---|---|---|
| | .<br>.<br>. | | |

**Parameters**:

| Parameter | Description |
|---|---|
| *tag* | Indicates the constant identifying the data structure. Tags are unique across HAB and are separated by at least Hamming distance 2. |
| *len* | Indicates the structure length in bytes (including the header). The minimum length is 4 bytes. |
| *V* | Indicates HAB_MAJOR_VERSION for this data structure. |
| *v* | Indicates HAB_MINOR_VERSION for this data structure. |

**Note on the version field**:

Any data structure where the version field (the combination of V and v) is less than the base HAB library version results in the HAB API returning HAB_FAILURE. Also, a corresponding event is added to the audit log.

**Note on address fields**:

Several of the data structures here contain address, offset and size fields. On ICs with 32-bit address spaces, each such field is represented as four bytes in big-endian order. On ICs with different width address spaces, the minimum number of bytes to represent the address width is used, again in big-endian order.

## 4.1 Image vector table

Details on the image vector table can be found in the "System Boot" section of the relevant NXP processor reference manual.

## 4.2 Device configuration data

Details on the device configuration data can be found in the "System Boot" section of the relevant NXP processor reference manual. DCD supports the write data, check data, and no operation commands.

## 4.3 Command sequence file

**Detailed description**:

Authentication and configuration script.

**Purpose**:

A command sequence file (CSF) is a script of commands used to guide image authentication and device configuration operations. In a typical high-assurance boot, each image in the boot sequence has a CSF. This CSF is used by the preceding image to verify the authenticity before passing the control.

**Format**:

A CSF contains a header, followed by a sequence of one or more commands, as shown below.

| hdr |
|---|
| [cmd]<br>.<br>.<br>. |
| [cmd]<br>.<br>.<br>. |

**Parameters**:

A CSF contains a header, followed by a sequence of one or more commands, as shown below.

| Parameter | Description |
|---|---|
| *hdr* | Indicates a header with tag HAB_TAG_CSF, length, and HAB version fields. |
| *cmd* | Indicates the CSF command. |

***Warning:***

- *Every CSF must contain an authenticate data command to authenticate the CSF contents using the CSF key.*
- *The first CSF in the boot sequence must contain an install key command to install the CSF key, before CSF authentication.*
- *The first CSF in the boot sequence must contain an install key command to install the super root key (SRK) table, before CSF key installation.*
- *Any other commands encountered before the above commands behave as per their individual command descriptions.*

**Remarks**:

- After installation, the keys can be reused by subsequent CSFs run by boot sequence components.
- This section lists all hardware-independent commands supported by HAB. Further hardware-specific commands, where supported, are described in the subsections of the Security hardware section. The selection of commands available on a given IC is described in the corresponding NXP processor reference manual. At a minimum, the available commands always include:
  - Check data
  - Set
  - Install key
  - Authenticate data
- The maximum size of CSF supported is given in the "System Boot" section of the relevant NXP processor reference manual.

### 4.3.1 Write data

**Detailed description**:

Write data command (DCD).

Writes a list of given 1-, 2-, or 4-byte values or bitmasks to a corresponding list of target addresses. This command can be used in a device configuration data structure. However, in the former, the set of allowed target addresses is restricted, as described in the "System Boot" section of the relevant NXP processor reference manual.

The command format is shown below.

<table>
<tr><td>tag</td><td>len</td><td>par</td></tr>
<tr><td colspan="3">address</td></tr>
<tr><td colspan="3">val_msk</td></tr>
<tr><td colspan="3">[address]</td></tr>
<tr><td colspan="3">[val_msk]</td></tr>
<tr><td colspan="3">...</td></tr>
<tr><td colspan="3">...</td></tr>
<tr><td colspan="3">[address]</td></tr>
<tr><td colspan="3">[val_msk]</td></tr>
</table>

**Parameters**:

<table>
<tr><th>Parameter</th><th>Description</th></tr>
<tr><td>tag</td><td>Indicates the constant HAB_CMD_WRT_DAT.</td></tr>
<tr><td>len</td><td>Indicates a constant.</td></tr>
<tr><td>par</td><td>Indicates command parameters. The par parameter is divided into bit fields as shown below.
<table>
<tr><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="5">flags</td><td colspan="3">bytes</td></tr>
</table>
In the above bit field structure:
<ul>
<li>bytes: Indicates the width of the target locations: 1 (8-bit value), 2 (16-bit value), or 4 (32-bit value)</li>
<li>flags: Indicates the control flags for command behavior. It has one of the following values:
<ul>
<li>0x01 (HAB_CMD_WRT_DAT_MSK) – mask/value flag:
<ul>
<li>If this flag is set, only specific bits can be overwritten at the target address.</li>
</ul></li>
</ul></li>
</ul></td></tr>
</table>

 

| Parameter | Description |
|---|---|
| | – Otherwise, all bits can be overwritten.<br>– 0x02 (HAB_CMD_WRT_DAT_SET) – write data set – set/clear flag:<br>– If the HAB_CMD_WRT_DAT_MSK flag is set, bits at the target address are overwritten with this flag.<br>– Otherwise, this flag is ignored. |
| *address* | Indicates the target address for writing the data. |
| *val_msk* | Indicates the data value or bitmask to be written at *address*. |

**See also**:

Note on address fields.

**Remarks**:

- One or more target address and value/bitmask pairs can be specified. The same *bytes* and *flags* parameters apply to all locations in the command. When successful, this command writes to each target address according to the flags shown below.

| "MSK" | "SET" | Action | Interpretation |
|---|---|---|---|
| 0 | 0 | *address = val_msk | Write value |
| 0 | 1 | *address = val_msk | Write value |
| 1 | 0 | *address &= ~val_msk | Clear bitmask |
| 1 | 1 | *address \|= val_msk | Set bitmask |

***Warning:***

- *When used in a device configuration data structure, if any of the target addresses does not fall within an allowed region, none of the values is written. The allowed target regions are the union of the allowed regions for hab_rvt.check_target(). Details on the allowable regions are available in the "System Boot" section of the relevant NXP processor reference manual.*
- *If any of the target addresses does not have the same alignment as the data width indicated in the parameter field, none of the values is written.*
- *If any of the values is larger or any of the bitmasks is wider than permitted by the data width indicated in the parameter field, none of the values is written.*

**Postconditions**:

- On successful completion, values or bitmasks are written to target locations.
- On unsuccessful completion, an audit event is logged giving the status as follows:
  - HAB_FAILURE, with further reasons:
    - HAB_INV_COMMAND: The command is malformed.
    - HAB_INV_ADDRESS: Access is denied for the target address.
    - HAB_INV_ADDRESS: The target address is not aligned.
    - HAB_INV_SIZE: The value is larger than the data width.
    - HAB_INV_SIZE: The data width is not supported.
- For successful commands, no audit event is logged.

### 4.3.2 Check data

**Detailed description**:

Check data command (DCD and CSF).

Test for a given 1-, 2-, or 4-byte bitmask from a source address. This command can be used in either a device configuration data structure or a command sequence file.

The command format is shown below.

| tag | len | par |
|---|---|---|
| address | | |
| mask | | |
| [count] | | |

**Parameters**:

| Parameter | Description |
|---|---|
| *tag* | Indicates the constant HAB_CMD_CHK_DAT. |
| *len* | Indicates a constant. |
| *par* | Indicates command parameters. The *par* parameter is divided into bit fields as shown below.<br>Bits 7–3: flags; bits 2–0: bytes<br>In the above bit field structure:<br>• *bytes*: Indicates the width of the target locations: 1 (8-bit value), 2 (16-bit value), or 4 (32-bit value)<br>• *flags*: Indicates the control flags for command behavior. It can have one of the following values:<br>– 0x02 (HAB_CMD_CHK_DAT_SET) – set/clear flag: Bits set in the mask must match this flag.<br>– 0x04 (HAB_CMD_CHK_DAT_ANY) – any/all flag: If clear, all bits set in the mask must match; otherwise, any bit suffices. |
| *address* | Indicates the source address to test. |
| *mask* | Indicates the bitmask to test. |
| *count* [optional] | Indicates the poll count. |

**See also**:

Note on address fields.

**Remarks**:

- This command polls the source address until either the exit condition is satisfied, or the poll count is reached. The exit condition is determined by the flags, as shown below.

| "ANY" | "SET" | Exit condition | Interpretation |
|---|---|---|---|
| 0 | 0 | `(*address & mask) == 0` | All bits clear |
| 0 | 1 | `(*address & mask) == mask` | All bits set |
| 1 | 0 | `(*address & mask) != mask` | Any bit clear |
| 1 | 1 | `(*address & mask) != 0` | Any bit set |

- This command can be used in either a device configuration data structure or a command sequence file structure, before or after command sequence file authentication without any difference.
- If *count* is not specified, this command polls indefinitely until the exit condition is met. If *count* = 0, this command behaves as for NOP.

***Warning:***

 

- *If the source address does not have the same alignment as the data width indicated in the parameter field, the value is not read.*
- *If the bitmask is wider than permitted by the data width indicated in the parameter field, the value is not read.*

**Postconditions**:

- On unsuccessful completion, an audit event is logged giving the status as follows:
  - HAB_FAILURE, with further reasons:
    - HAB_INV_COMMAND: The command is malformed.
    - HAB_INV_ADDRESS: The source address is not aligned.
    - HAB_INV_SIZE: The bitmask is wider than the data width.
    - HAB_INV_SIZE: The data width is not supported.
    - HAB_OVR_COUNT: Before the exit condition met, the poll count reached.
- For successful commands, no audit event is logged.

### 4.3.3 NOP

**Detailed description**:

This command has no effect (DCD and CSF).

The command format is shown below.

| | | |
|---|---|---|
| tag | len | und |

**Parameters**:

| Parameter | Description |
|---|---|
| *tag* | Indicates the constant HAB_CMD_NOP. |
| *len* | Indicates a constant (4). |
| *und* | Undefined (ignored) |

**Remarks**:

- This command can be used in a command sequence file structure, before or after the command sequence file authentication without any difference.

### 4.3.4 Set

**Detailed description**:

Set command (CSF only).

Sets the value of variable configuration items.

The command format is shown below.

<table>
<tr><td>tag</td><td>len</td><td>itm</td></tr>
<tr><td colspan="3">value<br>.<br>.<br>.</td></tr>
</table>

**Parameters**:

| Parameter | Description |
|---|---|
| *tag* | Indicates the constant HAB_CMD_SET. |
| *len* | Indicates a constant. |
| *itm* | Indicates command parameters (see hab_var_cfg_itm_t). |
| *value* | Indicates the value to be used for *itm*. The format for the *value* field is shown below.<br>0x0 \| alg \| eng \| cfg<br>In the above format:<br>• *alg*: Indicates the algorithm identifier.<br>• *eng*: Indicates the engine identifier.<br>• *cfg*: Indicates the engine configuration flags (if applicable). |

**Remarks**:

- This command can be used in a command sequence file structure, before or after the command sequence file authentication without any difference.

***Warning:*** *Only one value is active at a time for each itm. Each set command replaces the active value for that itm, with subsequent operations using the new value. The active value persists across subsequent calls to HAB functions, including subsequent hab_rvt.exit() and hab_rvt.entry() calls.*

**Postconditions**:

- On successful completion, the active value for the variable configuration item is replaced by the new value. No audit event is logged.
- On unsuccessful completion, the active value is not changed, and an audit event is logged giving the status as follows:
  - HAB_FAILURE, with further reasons:
    - HAB_INV_COMMAND: The command is malformed.
    - HAB_UNS_ITM: The configuration item is not supported.
    - HAB_UNS_ALGORITHM: The specified algorithm is not supported.

**Set default engine**:

Whenever an algorithm computation is required, the algorithm tag and parameters are used to search for an engine capable of performing the computation. This default behavior can be overridden in the following two ways (in decreasing order of precedence):

- By specifying an engine other than HAB_ENG_ANY in a CSF command, such as authenticate data. It has the highest priority, but its execution depends on the individual CSF command. The default behavior resumes after the command is completed. The range of algorithms for which engines can be specified is also limited by the parameters available in the command.
- By specifying an engine other than HAB_ENG_ANY in a set command. It overrides the default behavior, and applies to all subsequent operations, including later boot phases, until modified by another set command.

A set command specifying HAB_ENG_ANY restores the default behavior.

**Definitions**:

```
/* Variable configuration items */
typedef enum hab_var_cfg_itm {
    /* Preferred engine for a given algorithm */
    HAB_VAR_CFG_ITM_ENG = 0x03
    } hab_var_cfg_itm_t;
```

 

### 4.3.5 Initialize

**Detailed description**:

Initialize the specified engine features when exiting the ROM (CSF only).

The command format is shown below.

| tag | len | eng |
|---|---|---|
| [val]<br>.<br>.<br>. | | |

**Parameters**:

| Parameter | Description |
|---|---|
| *tag* | Indicates the constant HAB_CMD_INIT. |
| *len* | Depends on *eng*. |
| *eng* | Indicates the engine to be initialized. |
| *val* [optional] | Indicates the initialization values required by *eng*. |

**Remarks**:

- Engine-specific values and initialization sequences are described in the relevant subsection of the Security hardware section.
- The initialize commands are cumulative. A feature is initialized if specified in one or more initialize commands.

***Warning:*** *If the IC is configured as HAB_CFG_CLOSED, this command cannot be used in a device configuration data structure.*

**Postconditions**:

- On successful completion, the features specified for the engine are initialized when the hab_rvt.exit() function is called.
- On unsuccessful completion, the initialization sequences are omitted unless specified in a separate, successful initialize command. An audit event is logged giving the status as follows:
  - HAB_FAILURE, with further reasons:
    - HAB_INV_COMMAND: When not allowed by the IC security configuration, the command is used outside the authenticated CSF.
- For successful commands, no audit event is logged.

### 4.3.6 Unlock

**Detailed description**:

Prevent specific engine features being locked when exiting the ROM (CSF only).

The command format is shown below.

| tag | len | eng |
|---|---|---|
| [val]<br>.<br>. | | |

.

**Parameters**:

| Parameter | Description |
|---|---|
| *tag* | Indicates the constant HAB_CMD_UNLK. |
| *len* | Depends on *eng*. |
| *eng* | Indicates the engine to be left unlocked. |
| *val* [optional] | Indicates the unlock values required by *eng*. |

**Remarks**:

- Engine-specific values and locks are described in the relevant subsection of the Security hardware section.
- The unlock command is cumulative. A feature is left unlocked if specified in one or more unlock commands.

***Warning:*** *If the IC is configured as HAB_CFG_CLOSED, this command cannot be used in a device configuration data structure.*

**Postconditions**:

- On successful completion, the features specified for engine are not locked when the hab_rvt.exit() function is called.
- On unsuccessful completion, the features specified are locked unless specified in a separate, successful unlock command. An audit event is logged giving the status as follows:
  - HAB_FAILURE, with further reasons:
    - HAB_INV_COMMAND: When not allowed by the IC security configuration, the command is used outside the authenticated CSF.
- For successful commands, no audit event is logged.

### 4.3.7 Install key

**Detailed description**:

Install key command (CSF only).

Authenticate and install a public key or secret key for use in the subsequent install key or authenticate data commands.

Public key authentication can be restricted to a specific key, by including a hash of the certificate of the key in the command parameters. Also, public key authentication can be extended to include any key certified by the verifying key.

Secret key installation involves unwrapping with a key encryption key (KEK) using a supported key wrap protocol, with authentication integral to that protocol.

Other key usages are set in this command and apply to subsequent operations using the installed key.

HAB uses three internal key stores for key data, each with its own zero-based array of key slots:

- The public key store for public keys installed by this command
- The secret key store for secret keys installed by this command
- The master KEK store for master KEKs preinstalled on the IC

The user is responsible for managing the key slots in the internal key stores to establish the desired public or secret key hierarchy and determine the keys used in authentication operations. Overwriting occupied key slots is not allowed, though a repeat command to reinstall the same public key occupying the target slot is skipped and it does not generate an error.

The command format is shown below.

<table>
<tr><td colspan="2">tag</td><td colspan="4">len</td><td colspan="2">flg</td></tr>
<tr><td colspan="2">pcl</td><td colspan="2">alg</td><td colspan="2">src</td><td colspan="2">tgt</td></tr>
<tr><td colspan="8">key_dat</td></tr>
<tr><td colspan="8">[crt_hsh]</td></tr>
<tr><td colspan="8">...</td></tr>
<tr><td colspan="8">[crt_hsh]</td></tr>
</table>

**Parameters**:

| Parameter | Description |
|---|---|
| *tag* | Indicates the constant HAB_CMD_INS_KEY. |
| *len* | Depends on the size of *crt_hsh* (if present). |
| *flg* | Indicates flags from hab_cmd_ins_key_flg. |
| *pcl* | Indicates the key authentication protocol. |
| *alg* | Indicates the hash algorithm. |
| *src* | Indicates the source key (verification key, KEK) index. |
| *tgt* | Indicates the target key index. |
| *key_dat* | Indicates the start address of the key data to be installed:<br>• If HAB_CMD_INS_KEY_ABS is set in the *flg* parameter, the start address is absolute.<br>• Otherwise, the start address is relative to the CSF start. |
| *crt_hsh* [optional] | Indicates the hash of the certificate structure indicated by *key_dat*. |

**See also**:

Note on address fields regarding the *key_dat* parameter.

**Remarks**:

- For super root key installation:
  - *pcl* is HAB_PCL_SRK.
  - *alg* is used in the SRK authentication protocol.
  - *src* is the source key index within the super root key table (with 0 denoting the first key in the table).
  - *tgt* is the public key store index for installation.
  - *key_dat* locates the super root key table data.
- For other public key installation:
  - *pcl* is the certificate format.
  - *alg* is the algorithm used to compute *crt_hsh*.
  - *src* is the public key store index of the verification key. It is used to determine the signature algorithm.
  - *tgt* is the public key store index for installation.
  - *key_dat* locates the certificate data.
- For secret key installation from a secret key blob:
  - *pcl* is HAB_PCL_BLOB.
  - *src* is the KEK index within the master KEK store.
  - *tgt* is the secret key store index for installation.

  - *key_dat* locates the secret key blob data.
  - The key wrap algorithm is determined by the on-chip security hardware.
  - The *tgt* index may match the *src* index without overwriting an existing key because they refer to different key stores.
- For other secret key installation:
  - *pcl* is the wrapped key format.
  - *src* is the KEK index within the secret key store.
  - *tgt* is the secret key store index for installation.
  - *key_dat* locates the wrapped key data.
  - The key wrap algorithm is determined from the KEK *src*.

***Warning:***

*The following constraints apply to the command parameters:*

- *For public key installation:*
  - *If tgt is HAB_IDX_SRK:*
    - *pcl must be HAB_PCL_SRK.*
    - *Only HAB_CMD_INS_KEY_ABS can be set in flg.*
    - *crt_hsh must be absent.*
  - *Otherwise, if tgt is HAB_IDX_CSFK:*
    - *pcl must not be HAB_PCL_SRK.*
    - *alg must be HAB_ALG_ANY.*
    - *src must be HAB_IDX_SRK.*
    - *HAB_CMD_INS_KEY_CSF must be set in flg.*
    - *HAB_CMD_INS_KEY_HSH must not be set in flg.*
    - *crt_hsh must be absent.*
  - *Otherwise:*
    - *pcl must not be HAB_PCL_SRK.*
    - *tgt must not be HAB_IDX_SRK or HAB_IDX_CSFK.*
  - *Finally, if HAB_CMD_INS_KEY_HSH is not set in flg:*
    - *alg must be HAB_ALG_ANY.*
    - *crt_hsh must be absent.*
- *For secret key installation:*
  - *alg must be HAB_ALG_ANY.*
  - *Only HAB_CMD_INS_KEY_ABS can be set in flg.*
  - *crt_hsh must be absent.*
- *The crt_hsh parameter (if used) is calculated across the whole of the HAB Certificate structure, including header. If there is a mismatch, the key installation is aborted.*
- *For SRK installation, the valid values of src are limited by the number of keys present in the super root key table. Further IC-specific constraints may apply when multiple cores on a single IC share a super root key table, or when an IC implements SRK revocation fuses. For more details, refer to the relevant NXP processor reference manual.*
- *For secret key installation with pcl set to HAB_PCL_BLOB, the valid values of src are limited to the master KEKs available on the IC. The relevant security hardware description describes the available master KEK indexes.*

**Postconditions**:

- On successful completion, the key indicated by the *key_dat* parameter is installed at the *tgt* index in the appropriate HAB internal key store, along with:
  - The usage data extracted from the *flg* parameter
  - The verification protocol or decryption protocol to be used with the installed key, extracted from the *key_dat* parameter
- On unsuccessful completion, an audit event is logged giving the status as follows:
  - HAB_WARNING: Key installed but the command did not complete as expected, with further reasons:
    - HAB_UNS_ENGINE: The default engine (from the set command) is either not recognized or does not support the specified algorithm or parameters. An alternative engine is used.
    - HAB_ENG_FAIL: An attempt to release the default engine (from the set command) failed.
  - HAB_FAILURE otherwise, with further reasons:
    - HAB_INV_COMMAND: The command is malformed.
    - HAB_INV_COMMAND: After CSF authentication, an attempt is made to install the SRK or CSF key.
    - HAB_INV_COMMAND: Before CSF authentication, an attempt is made to install keys other than the SRK or CSF key.
    - HAB_INV_KEY: The source key in the super root key table is of type HAB_KEY_HASH.
    - HAB_INV_INDEX: No verification key is available at the given index.
    - HAB_INV_INDEX: No KEK is available at the given index.
    - HAB_INV_INDEX: No source key is available at the given index in the super root key table.
    - HAB_INV_INDEX: The target index is not available for the installed key.
    - HAB_UNS_KEY: Public key type or domain parameters (for example, field size) are not supported.
    - HAB_UNS_KEY: Secret key type or domain parameters (for example, key size) are not supported.
    - HAB_UNS_PROTOCOL: The certificate protocol is not supported or not suitable.
    - HAB_UNS_PROTOCOL: The key wrap protocol is not supported or not suitable.
    - HAB_UNS_ALGORITHM: The hash algorithm is not supported or not suitable.
    - HAB_ENG_FAIL: An attempt to allocate the default engine (from the set command) failed.
    - HAB_INV_SIGNATURE: Certificate signature verification failed.
    - HAB_INV_SIGNATURE: Key unwrap authentication failed.
    - HAB_INV_CERTIFICATE: Other certificate or super root key table verification (including mismatch with *crt_hsh*) failed.
- For successful commands, no audit event is logged.

**Definitions**:

```
/* Public Key types */
#define HAB_KEY_PUBLIC 0xe1     /**< Public key type: data present */
#define HAB_KEY_HASH 0xee      /**< Any key type: hash only */

/* Public key store indices */
#define HAB_IDX_SRK0  0         /**< Super-Root Key, set 0 index */
     #define HAB_IDX_CSFK0 1         /**< CSF key 0 index */
#define HAB_IDX_SRK1  5         /**< Super-Root Key, set 1 index */
#define HAB_IDX_CSFK1 6         /**< CSF key 1 index */

/* Flags for Install Key commands. */
typedef enum hab_cmd_ins_key_flg
{
     HAB_CMD_INS_KEY_CLR = 0,  /**< No flags set */
     HAB_CMD_INS_KEY_ABS = 1,  /**< Absolute certificate address */
     HAB_CMD_INS_KEY_CSF = 2,  /**< Install CSF key */
     HAB_CMD_INS_KEY_DAT = 4,  /**< Key binds to Data Type */
     HAB_CMD_INS_KEY_CFG = 8,  /**< Key binds to Configuration */
     HAB_CMD_INS_KEY_FID = 16, /**< Key binds to Fabrication UID */
```

 

```
      HAB_CMD_INS_KEY_MID = 32, /**< Key binds to Manufacturing ID */
      HAB_CMD_INS_KEY_CID = 64, /**< Key binds to Caller ID */
      HAB_CMD_INS_KEY_HSH = 128 /**< Certificate hash present */
 } hab_cmd_ins_key_flg_t;
```

#### 4.3.7.1 Wrapped key

**Detailed description**:

Wrapped secret key data for installation.

Supported in HAB version 4.1 and later with the appropriate protocols.

**Purpose**:

A HAB wrapped key structure specifies a secret key to be installed, along with the data required to verify the authenticity of the key. Wrapped secret keys include the key value in encrypted form. Wrapped keys are attached to or referenced by a command sequence file and installed using the install key command.

**Format**:

A HAB wrapped key structure contains a generic header, followed by protocol-specific data containing the key and authentication data. A parameter within the install key command determines the wrapped key protocol.

***Note:*** *The protocol-specific data can have an arbitrary length in bytes.*

The storage format is shown below.

| |
|---|
| hdr |
| wrp_dat<br>.<br>.<br>. |

**Parameters**:

| Parameter | Description |
|---|---|
| *hdr* | Indicates a header with tag HAB_TAG_WRP, length, and HAB version fields. |
| *wrp_dat* | Indicates the protocol-specific wrapped key and authentication data. |

**Remarks**:

- This section lists all wrapped key formats or protocols supported by HAB. The selection of formats available on a given IC is described in the corresponding NXP processor reference manual.

#### 4.3.7.2 Secret key blob

**Purpose**:

HAB secret key blobs are used to install secret keys using the special HAB_PCL_BLOB protocol in the install key command. This protocol is specific to the available security hardware and it always uses a master KEK, which is usually unique to an IC.

**Format**:

HAB secret key blobs are stored using the HAB wrapped key data structure. The storage format for the wrp_dat section is shown below.

| mode | alg | size | flg |
|---|---|---|---|
| data<br>.<br>.<br>. | | | |

**Parameters**:

| Parameter | Description |
|---|---|
| *mode* | Indicates the key cipher mode. |
| *alg* | Indicates the key cipher algorithm. |
| *siz* | Indicates the size (in bytes) of the unwrapped key value. |
| *flg* | Indicates key flags from hab_key_secret_flg. |
| *data* | Indicates the encrypted key value. |

**Remarks**:

- In the wrp_dat section, the unencrypted parameters are authenticated as part of the wrapping protocol.
- Details of the wrapping protocol, including the authentication mechanism and storage format for the encrypted key value, are available in the relevant security hardware engine documentation.

**Definitions**:

```
/* Secret key flags. */
typedef enum hab_key_secret_flg
{
    /* Seven more flag values available */
    HAB_KEY_FLG_KEK = 128   /**< KEK */
} hab_key_secret_flg_t;
```

### 4.3.8 Authenticate data

**Detailed description**:

Authenticate data command (CSF only).

Verify the authenticity of preloaded data using a preinstalled key. The data may include executable software instructions and it may be spread across multiple noncontiguous blocks in the memory.

The authentication protocol can be based on either public keys using a digital signature or secret keys using a message authentication code. Secret key authentication protocols may include in-place decryption of the preloaded data.

The command format is shown below.

| tag | len | | flg |
|---|---|---|---|
| key | pcl | eng | cfg |
| aut_start | | | |
| [blk_start] | | | |
| [blk_bytes] | | | |
| ... | | | |

| |
|---|
| ... |
| [blk_start] |
| [blk_bytes] |

**Parameters**:

| Parameter | Description |
|---|---|
| *tag* | Indicates the constant HAB_CMD_AUT_DAT. |
| *len* | Indicates the number of blocks. |
| *flg* | Indicates a flag from the set hab_cmd_aut_dat_flg. |
| *key* | Indicates the verification key index. |
| *pcl* | Indicates the authentication protocol. |
| *eng* | Indicates the engine used to process data blocks. Use HAB_ENG_ANY to allow the protocol implementation to choose the first compatible engine. |
| *cfg* | Indicates the engine configuration flags (if applicable). |
| *aut_start* | Indicates the address of the authentication data:<br>• If HAB_CMD_AUT_DAT_ABS is set in the *flg* parameter, the authentication data address is absolute.<br>• Otherwise, the authentication data address is relative to the CSF start. |
| *blk_start* | Indicates the absolute address of a data block to be authenticated. |
| *blk_bytes* | Indicates the size (in bytes) of a data block to be authenticated. |

**See also**:

Note on address fields regarding the *key_dat* parameter.

**Remarks**:

- When multiple data blocks are indicated, authentication is done on the contents of the data blocks as if they were concatenated in the specified order.
- For public key authentication protocols:
  - *key* is an index within the public key store.
  - *pcl* is the signature format.
  - *eng* is a hash engine.
  - *aut_start* locates the signature data.
  - The signature algorithm is determined from *key*.
  - The hash algorithm is determined from the signature data.
  - If *key* is HAB_IDX_CSFK, the current CSF is authenticated. It is the only CSF authentication recognized by HAB. For example, including the CSF within a region authenticated by another key is not recognized as authenticating the CSF.
- For secret key authentication protocols:
  - *key* is an index within the secret key store.
  - *pcl* is the message authentication code format.
  - *eng* is a medium access control (MAC) engine.
  - *aut_start* locates the Message Authentication Code data.

  – The MAC algorithm is determined from *key*.
- In addition, for secret key authentication protocols with decryption:
  – *eng* is a cipher and MAC engine.
  – The cipher and MAC algorithms are determined from *key*.
  – Encrypted data is overwritten in-place with decrypted data as the decryption proceeds.

***Warning:***

- *If eng is HAB_ENG_ANY, cfg must be zero.*
- *For public key authentication protocols:*
  – *If key is HAB_IDX_CSFK, the blk_start and blk_bytes parameters must be absent.*

**Postconditions**:

- This command may alter the configuration of an engine used in the authentication. See the Security hardware section for the engine in question.
- On completion of a secret key authentication protocol using counter mode, the data encryption key at the index *key* is uninstalled from the secret key store. It is done to prevent the use of the same key and nonce combination again.
- On failure of a secret key authentication and decryption protocol, the decrypted data regions are overwritten with zero bytes.
- On completion (for any protocol), an audit event is logged, giving the status as follows:
  – HAB_SUCCESS: The data is authenticated as specified. The data blocks are logged (except for CSF authentication, where no audit event is logged).
  – HAB_WARNING: The data is authenticated but the command did not complete as specified, with further reasons:
    – HAB_UNS_ENGINE: The specified engine is either not recognized or does not support a specified algorithm or parameter. An alternative engine is used.
    – HAB_ENG_FAIL: An attempt to release the specified engine failed.
  – HAB_FAILURE otherwise, with further reasons:
    – HAB_INV_COMMAND: The command is malformed.
    – HAB_INV_COMMAND: An attempt is made to authenticate the image data before CSF authentication.
    – HAB_INV_COMMAND: An attempt is made to re-authenticate the CSF data after CSF authentication.
    – HAB_INV_COMMAND: An attempt is made to authenticate the image data with either HAB_IDX_SRK or HAB_IDX_CSFK.
    – HAB_INV_COMMAND: An attempt is made to authenticate the CSF data with a key other than HAB_IDX_CSFK.
    – HAB_INV_INDEX: No key is available at the given index or the index is out of range.
    – HAB_INV_KEY: The specified key is identified as a certificate authority (CA) key.
    – HAB_UNS_KEY: No engine is available for the specified key parameters.
    – HAB_UNS_PROTOCOL: The protocol is not supported.
    – HAB_UNS_ALGORITHM: The algorithm is not supported or not suitable.
    – HAB_ENG_FAIL: An attempt to allocate the specified engine failed.
    – HAB_INV_SIGNATURE: Data authentication failed. It covers both the signature and message authentication code failures.

**Definitions**:

```
/* Flags for Authenticate Data commands. */
typedef enum hab_cmd_aut_dat_flg
{
    HAB_CMD_AUT_DAT_CLR = 0,     /**< No flags set */
    HAB_CMD_AUT_DAT_ABS = 1      /**< Absolute signature address */
```

```
} hab_cmd_aut_dat_flg_t;
```

## 4.4 Events

**Detailed description**:

Audit log event record.

**Purpose**:

A HAB event record contains data from an event in the audit log. It is generated as an output from the hab_rvt.report event() API.

**Format**:

An event record contains a header, followed by a list of parameters, as described below.

| | | | |
|---|---|---|---|
| hdr | | | |
| sts | rsn | ctx | eng |
| [data] | | | |

**Parameters**:

| Parameter | Description |
|---|---|
| *hdr* | Indicates a header with tag HAB_TAG_EVT, length, and HAB version fields. |
| *sts* | Indicates the status level logged. |
| *rsn* | Indicates further reason logged. |
| *ctx* | Indicates the context from which the event is logged. |
| *eng* | Indicates the engine associated with the failure or HAB_ENG_ANY if none. |
| *data* | Indicates the context-dependent data. |

**Remarks**:

- The length of the *data* field can be calculated from the overall length of the record.

**Context-dependent data**:

In most contexts, the *data* field is absent. The exceptions are as follows:

- HAB_CTX_AUT_DAT: Authenticated data event is used internally by HAB. The *data* field specifies an authenticated data block in an internally defined format.
- HAB_CTX_ENTRY, HAB_CTX_EXIT: Unless mentioned in the relevant subsection of the Security hardware section, the *data* field is empty.
- HAB_CTX_TARGET: The *data* field contains the hab_rvt.check_target() call parameters in the order that they appear in the parameter list.
- HAB_CTX_COMMAND: The *data* field contains the entire command that failed, and then it was copied from the device configuration data or command sequence file.
- HAB_CTX_ASSERT: The *data* field contains the hab_rvt.assert() call parameters in the order that they appear in the parameter list.

## 4.5 ROM vector table

**Detailed description**:

HAB library hooks.

**Purpose**:

The ROM vector table (RVT) provides function pointers into the HAB library in the ROM for use by post-ROM boot sequence components.

**Format**:

The ROM vector table contains a header, followed by a list of addresses, as described below. For details on the address locations, refer to the "System Boot" chapter of the relevant NXP processor reference manual.

**Data fields**:

- ```
  hab_hdr_t hdr
  ```
  A header with tag HAB_TAG_RVT, length, and HAB version fields (see the Data Structures section).
- ```
  hab_status_t(* entry)(void)
  ```
  Enters and initializes the HAB library.
- ```
  hab_status_t(* exit)(void)
  ```
  Finalizes and exits the HAB library.
- ```
  hab_status_t(* check_target)(hab_target_t type, const void *start, size_t
   bytes)
  ```
  Checks the target address.
- ```
  hab_image_entry_f(* authenticate_image)(uint8_t cid, ptrdiff_t ivt_offset, void
   **start,
  size_t *bytes, hab_loader_callback_f loader)
  ```
  Authenticates the image.
- ```
  hab_status_t(* run_dcd )(const uint8_t *dcd)
  ```
  Executes a boot configuration script.
- ```
  hab_status_t(* run_csf )(const uint8_t *csf, uint8_t cid)
  ```
  Executes an authentication script.
- ```
  hab_status_t(* assert )(hab_assertion_t type, const void *data, uint32_t count)
  ```
  Tests an assertion against the audit log.
- ```
  hab_status_t(* report_event )(hab_status_t status, uint32_t index, uint8_t
   *event, size_t
  *bytes)
  ```
  Reports an event from the audit log.
- ```
  hab_status_t(* report_status )(hab_config_t *config, hab_state_t *state)
  ```
  Reports the security status.
- ```
  void(* failsafe )(void)
  ```
  ```

Enters the Fail-Safe boot mode. The ROM vector table contains a header, followed by a list of addresses, as described below.

- ```
  hab_image_entry_f(* authenticate_image_no_dcd)(uint8_t cid, ptrdiff_t
   ivt_offset, void **start,
  size_t *bytes, hab_loader_callback_f loader)
  ```
  Authenticates the image.
- ```
  uint32_t(* get_version)(void)
  ```
  Gets the HAB version.
- ```
  hab_status_f(* authenticate_container)(uint8_t cid, ptrdiff_t ivt_offset, void
   **start, size_t
  *bytes, hab_loader_callback_f loader, uint32_t srkmask, int skip_dcd)
  ```
  Authenticates the container.
  ```

# 5 Security hardware

This section describes all versions of all security hardware blocks supported by HAB. For details on the available security hardware, refer to the security reference manual for the relevant NXP processor.

## 5.1 Security Controller (SCC)

**Purpose**:

The SCC provides secure RAM storage and monitors the security state of the IC. HAB supports SCCv2.

**Entry sequence**:

During entry, the SCC status registers are examined for any errors.

**Self-tests**:

During the initial call to hab_rvt.entry() in the ROM, SCC performs a known-answer test. If the known-answer test fails, a failure event is logged to the audit log. Subsequent invocations of hab_rvt.entry() do not repeat the self-tests.

**Exit sequence**:

During exit, the SCC status registers are examined for any errors.

**Events**:

If an entry, exit or test operation fails, an audit event is logged with status field HAB_FAILURE, reason field HAB_ENG_FAIL, engine field HAB_ENG_SCC and data field containing the following registers (in the specified order):

- Command Status
- Error Status
- Security Monitor Status
- Security Violation Detector

***Note:*** *If a failure occurs when the SCC is not enabled, the audit event reason field is HAB_WARNING rather than HAB_FAILURE.*

**Security state mapping**:

SCCv2 does not support all the states in HAB version 4 (HABv4). The mapping between the HABv4 and SCCv2 states is shown below.

| HABv4 state | SCCv2 state |
|---|---|
| HAB_STATE_INITIAL | Initialize |
| HAB_STATE_CHECK | Health check |
| HAB_STATE_NONSECURE | Non-secure |
| HAB_STATE_TRUSTED | Secure |
| HAB_STATE_ SECURE | Not supported |
| HAB_STATE_FAIL_SOFT | Fail soft |
| HAB_STATE_FAIL_HAR | Fail hard |

If the *eng* parameter is HAB_ENG_ANY and the hash computation is compatible with the constraints of the software engine, HAB may select software automatically.

## 5.2 Data Co-Processor (DCP)

**Purpose**:

DCP is used by HAB to accelerate hash computations. HAB supports DCPv2, depending on the IC configuration.

**Entry sequence**:

Apart from the self-tests, no externally visible operation occurs for this engine.

**Self-tests**:

During the initial call to hab_rvt.entry() in the ROM, DCP performs several known-answer tests. If any known-answer test fails, DCP is marked as inoperative, and operations are directed to other engines where available. Subsequent invocations of hab_rvt.entry() do not repeat the self-tests.

**Exit sequence**:

No externally visible operation occurs for this engine.

**Events**:

If an entry, exit or test operation fails, an audit event is logged with status field HAB_WARNING, reason field HAB_ENG_FAIL, engine field HAB_ENG_DCP and data field containing the following registers (in the specified order):

- Status
- Channel Status

**Algorithms – hash**:

DCP supports SHA-1 and SHA-256 hash algorithms.

If the following constraints are met, DCP can be used for hash computation in commands, such as the authenticate data command, through the HAB_ENG_DCP *eng* parameter:

- DCP is enabled.
- The *alg* parameter is HAB_ALG_SHA1 or HAB_ALG_SHA256 and is supported on this IC.
- At most HAB_DCP_BLOCK_MAX data blocks are covered by the hash. For more details, see the Authenticate data section.
- Except the final one, all data blocks are multiples of 64 bytes in length; the final data block can be of an arbitrary length.
- The combined length of all data blocks is less than 512 MB.
- All data blocks reside in the memory accessible to the DMA engine of DCP.

Using HAB_ENG_DCP without meeting the constraints results in an unsuccessful operation, with a HAB_UNS_ALGORITHM audit event logged.

If the *eng* parameter is HAB_ENG_ANY and the hash computation is compatible with the constraints of DCP, HAB may select DCP automatically.

**Configuration**:

- Using appropriate write data commands in the device configuration data, DCP can be configured for optimal performance and various memory types.
- Using the set command, DCP can be selected as the default hash engine for a specific algorithm. A default configuration is established in the same command.

## 5.3 Run-Time Integrity Checker (RTIC)

**Purpose**:

RTIC is used to accelerate hash algorithm calculations and can be configured to retain computed hashes for later use in run-time monitoring. HAB supports RTICv3, depending on the IC configuration.

**Entry sequence**:

Apart from the self-tests, no externally visible operation occurs for this engine.

**Self-tests**:

During the initial call to hab_rvt.entry() in the ROM, RTIC performs a known-answer test. If the known-answer test fails, RTIC is marked as inoperative, and hash operations are directed to other engines where available. Subsequent invocations of hab_rvt.entry() do not repeat the self-tests.

**Exit sequence**:

No externally visible operation occurs for this engine.

**Events**:

If an entry, exit or test operation fails, an audit event is logged with *status* field HAB_WARNING, *reason* field HAB_ENG_FAIL, *engine* field HAB_ENG_RTIC and *data* field containing the following registers (in the specified order):

- Status
- Control
- Fault Address

Using HAB_ENG_RTIC without meeting the constraints results in an unsuccessful operation, with a HAB_UNS_ALGORITHM audit event logged.

If the *eng* parameter is HAB_ENG_ANY and the hash computation is compatible with the constraints of RTIC, HAB may select RTIC automatically.

**Configuration**:

- Using appropriate write data commands in the device configuration data, RTIC can be configured for optimal performance and various memory types.
- Using the set command, RTIC can be selected as the default hash engine for a specific algorithm. A default configuration is established in the same command.

**Retaining computed hash values**:

RTIC supports storing several independent reference hash values, which can be monitored at run time. HAB provides a means to compute and retain the reference hash values in preparation for run-time monitoring later.

If HAB_RTIC_KEEP is set when using HAB_ENG_RTIC, the computed hash value is retained in the reference hash register of RTIC, the corresponding run-time enable bit is set, and the corresponding run-time unlock bit is cleared. A subsequent hash calculation using RTIC uses the next available reference hash register.

If HAB_RTIC_KEEP is not set, a subsequent hash calculation using RTIC overwrites the current reference hash register.

Using HAB_ENG_RTIC (with or without HAB_RTIC_KEEP) after all the reference hash registers are exhausted results in an unsuccessful operation, with a HAB_UNS_ALGORITHM audit event logged. It is especially important for multicore ICs with a shared RTIC because the available reference hashes must be shared between the cores. HAB uses the run-time enable bits in the RTIC control register to ensure that reference hashes retained by another core are not overwritten.

## 5.4 Symmetric, Asymmetric, Hash, and Random Accelerator (SAHARA)

**Purpose**:

SAHARA is used by HAB to accelerate hash computations. HAB supports SAHARAv4LT, depending on the IC configuration.

**Entry sequence**:

Apart from the self-tests, no externally visible operation occurs for this engine.

**Self-tests**:

During the initial call to hab_rvt.entry() in the ROM, SAHARA performs several known-answer tests. If any known-answer test fails, SAHARA is marked as inoperative, and operations are directed to other engines where available. Subsequent invocations of hab_rvt.entry() do not repeat the self-tests.

**Exit sequence**:

No externally visible operation occurs for this engine.

**Events**:

If an entry, exit or test operation fails, an audit event is logged with *status* field HAB_WARNING, *reason* field HAB_ENG_FAIL, *engine* field HAB_ENG_SAHARA and *data* field containing the following registers (in the specified order):

- Control
- Status
- Error Status
- Fault Address
- Current Descriptor Address
- Initial Descriptor Address
- Operation Status
- Configuration
- Multiple Master Status

**Algorithms – hash**:

Although SAHARA supports MD5, SHA-1, and SHA-256 hash algorithms; MD5 and SHA-1 are deprecated in HAB. Therefore, SAHARA can be used only for SHA-256.

If the following constraints are met, SAHARA can be used for hash computation in commands, such as the authenticate data command, through the HAB_ENG_SAHARA *eng* parameter:

- SAHARA is enabled.
- The *alg* parameter is HAB_ALG_SHA256.
- At most HAB_SAHARA_BLOCK_MAX data blocks are covered by the hash. For more details, see the Authenticate data section.
- All data blocks reside in the memory that is accessible to the DMA engine of SAHARA.

Using HAB_ENG_SAHARA without meeting the constraints results in an unsuccessful operation, with a HAB_UNS_ALGORITHM audit event logged.

If the *eng* parameter is HAB_ENG_ANY and the hash computation is compatible with the constraints of SAHARA, HAB may select SAHARA automatically. DCP supports SHA-1 and SHA-256 hash algorithms.

**Configuration**:

- Using appropriate write data commands, SAHARA can be configured for optimal performance and various memory types.

- Using the set command, SAHARA can be selected as the default engine for a specific algorithm. A default configuration is established in the same command.

## 5.5 Secure Real Time Clock (SRTC)

**Purpose**:

During the boot flow, HAB controls the SRTC state. HAB supports SRTC version 1.

**Entry sequence**:

No externally visible operation occurs for this engine.

**Self-tests**:

No self-test occurs for this engine.

**Commands**:

When used with HAB_ENG_SRTC:

- The initialize command prepares to clear any failure status flags and zero the low-power counters and timers, if the SRTC is in the Init state when hab_rvt.exit() is first called on leaving the ROM. The optional *val* parameter is absent.
- The unlock command prepares to prevent the secure timer and monotonic counter being locked, if the SRTC is in the Valid state when hab_rvt.exit() is first called on leaving the ROM. The optional *val* parameter is absent.

**Exit sequence**:

During the initial call to hab_rvt.exit() in the ROM, the SRTC state is updated according to its configuration and state, the IC boot and security configurations, and SRTC-specific CSF commands, if any.

If the SRTC is not configured for low security, but the boot configuration is non-secure:

- It is an unsupported configuration.
- SRTC is forced into the Failure state.

Otherwise, if SRTC is configured for high security, the behavior depends on the SRTC state and whether SRTC-related CSF commands have been executed:

- If the SRTC is in the Init state:
  - The power glitch register is initialized.
  - The SRTC is moved out of the Init state:
    - If the initialize command has been executed since calling hab_rvt.entry(), the SRTC should move to the Non-Valid state with secure timer and monotonic counter cleared.
    - Otherwise, the SRTC could move to either the Non-Valid or Failure state, depending on the status register contents.
  - The secure timer and monotonic counters are not locked.
- If the SRTC is in the Valid state:
  - Unless the unlock command has been executed since calling hab_rvt.entry(), the secure timer and monotonic counters are locked.
- Otherwise, no changes are made to the SRTC settings.

During subsequent calls to hab_rvt.exit(), no externally visible operation occurs for this engine.

**Events**:

If an exit operation fails, an audit event is logged with engine field HAB_ENG_SRTC and data field containing the following registers (in the specified order):

- LP Control
- LP Status
- HP Control
- HP Interrupt Status

The event status is as follows:

- HAB_WARNING, with further reasons:
  - HAB_UNS_STATE: The initialize command used with SRTC is not in the Init state.
  - HAB_UNS_STATE: The unlock command used with SRTC is not in the Valid state.
- HAB_FAILURE, with further reasons:
  - HAB_ENG_FAIL: SRTC could not be allocated.

## 5.6 Cryptographic Accelerator and Assurance Module (CAAM)

**Purpose**:

CAAM is used by HAB to accelerate hash computations. HAB supports CAAMv1, depending on the IC configuration.

**Entry sequence**:

During calls to hab_rvt.entry(), the following operations are performed:

- Self-tests are run if it is the initial entry (see below).
- The status register is examined to verify that CAAM is idle and not busy.
- The secure memory status register is examined for errors.
- A secure memory partition is allocated with a single page for the secret key store.

**Self-tests**:

During the initial call to hab_rvt.entry() in the ROM, CAAM performs the following known-answer tests:

- Hash example
- AEAD example
- Key unwrap example with known test key
- SHA-256 hash DRBG example

If any known-answer test fails, the relevant functionality in CAAM is marked as inoperative, and operations are directed to other engines where available. Subsequent invocations of hab_rvt.entry() do not repeat the self-tests.

**Commands**:

When used with HAB_ENG_CAAM, the unlock command prevents specific locks being applied and the initialize command enforces specific initializations when hab_rvt.exit() is first called on leaving the ROM. The format of the *val* command parameter is shown below.

| | |
|---|---|
| 0x000000 | flg |

where *flg* specifies the features to leave unlocked or to initialize by using the values HAB_CAAM_UNLOCK_MID or HAB_CAAM_INIT_RNG.

**Exit sequence**:

During the initial call to hab_rvt.exit() in the ROM, CAAM is updated according to the IC boot and security configurations and CAAM unlock commands (if any), as explained below:

- If the IC is configured as HAB_CFG_CLOSED, unless the unlock command with HAB_CAAM_UNLOCK_MID flagged has been executed:
  - Job Ring and DECO Master ID registers are locked.
- If the IC is configured as HAB_CFG_CLOSED, if the initialize command with HAB_CAAM_INIT_RNG flagged has been executed:
  - True Random Number Generator (TRNG) status is checked for errors in entropy generation.
  - Deterministic Random Number Generator (DRNG) state handle 0 is instantiated (without prediction resistance) using entropy from TRNG.
  - Descriptor keys (JDKEK, TDKEK, and TDSK) are generated.
  - Advanced Encryption Standard (AES) differential power analysis (DPA) mask is generated.

During all calls to hab_rvt.exit(), the following operations are performed:

- The secure memory partition allocated for the secret key store is released.

**Events**:

If a CAAM operation fails, an audit event is logged with reason field HAB_ENG_FAIL and engine field HAB_ENG_CAAM. For known-answer test failures, the status field is HAB_WARNING; otherwise, it is HAB_ENG_FAIL. Where possible, the data field contains the following registers (in the specified order):

- Secure Memory Status
- Job Ring Output Status Register
- Secure Memory Partition Owners
- Fault Address
- Fault Address Master ID
- Fault Address Detail
- CAAM Status

**Algorithms – hash**:

Although CAAM supports MD5, SHA-1, and SHA-256 hash algorithms, MD5 and SHA-1 are deprecated in HAB. Therefore, CAAM can be used only for SHA-256.

If the following constraints are met, CAAM can be used for hash computation in commands, such as the authenticate data command, through the HAB_ENG_CAAM *eng* parameter:

- CAAM is enabled.
- The *alg* parameter is HAB_ALG_SHA256.
- At most HAB_CAAM_BLOCK_MAX data blocks are covered by the hash. For more details, see the Authenticate data section.
- All data blocks reside in the memory that is accessible to the DMA engine of CAAM.

Using HAB_ENG_CAAM without meeting the constraints results in an unsuccessful operation, with a HAB_UNS_ALGORITHM audit event logged.

If the *eng* parameter is HAB_ENG_ANY and the hash computation is compatible with the constraints of CAAM, HAB may select CAAM automatically.

**Algorithms – key wrap**:

If the following constraints are met, HAB uses CAAM for secret key installation from a secret key blob in the install key command:

- CAAM is enabled.
- The AES engine in CAAM is not disabled due to the export control configuration.
- The *pcl* parameter is HAB_PCL_BLOB.
- The *key_dat* parameter locates the memory accessible to the DMA engine of CAAM.

Using CAAM without meeting the constraints results in an unsuccessful operation, with a HAB_ENG_FAIL or HAB_UNS_PROTOCOL audit event logged.

The CAAM blob decapsulation protocol is used to unwrap the secret key blob. That protocol requires a 64-bit Key_Modifier input, which is used by HAB to authenticate the unencrypted data in the secret key blob data structure. The Key_Modifier is constructed by padding the unencrypted data on the right with zero bytes as shown below. The same Key_Modifier must be used in the CAAM blob encapsulation protocol when wrapping the key.

| | | | | |
|---|---|---|---|---|
| mode | alg | siz | flg | 0x00000000 |

**Algorithms – AEAD**:

CAAM supports the AES-CCM (CCM stands for "Counter with CBC-MAC") algorithm for Authenticated Encryption with Associated Data (AEAD). This mode can be selected for any supported key size.

If the following constraints are met, CAAM can be used for AEAD MAC computation in the authenticate data command through the HAB_ENG_CAAM *eng* parameter:

- CAAM is enabled.
- The AES engine in CAAM is not disabled due to export control configuration.
- The *alg* parameter in the selected key is HAB_ALG_AES.
- The *mode* parameter in the selected key is HAB_MODE_CCM.
- At most eight data blocks are covered by the hash. For more details, see the Authenticate data section.
- All data blocks reside in the memory accessible to the DMA engine of CAAM.

Using HAB_ENG_CAAM without meeting the constraints results in an unsuccessful operation, with a HAB_UNS_ALGORITHM audit event logged.

If the *eng* parameter is HAB_ENG_ANY and the AEAD MAC computation is compatible with the constraints of CAAM, HAB may select CAAM automatically.

**Configuration**:

- Using appropriate write data commands, CAAM can be configured for optimal performance and various memory types.
- Using the set command, CAAM can be selected as the default engine for a specific algorithm. A default configuration is established in the same command.

## 5.7 Secure Non-Volatile Storage (SNVS)

**Purpose**:

The SNVS provides secure non-volatile (battery-backed) storage, security state monitoring, and master key selection. Non-volatile features include a secure real time clock and a zeroizable master key. Master key selection determines the major input to the master KEK used when unwrapping a secret key blob. HAB supports SNVSv1.

**Entry sequence**:

During all calls to hab_rvt.entry(), HAB verifies that the SNVS SSM state is either Trusted or Secure, only when the IC is configured as HAB_CFG_CLOSED. If a fault is detected, a failure event is logged to the audit log.

**Commands**:

When used with HAB_ENG_SNVS, the unlock command prevents specific locks being applied when hab_rvt.exit() is first called on leaving the ROM. The format of the *val* command parameter is shown below.

| | |
|---|---|
| 0x000000 | flg |

where flg specifies the features to leave unlocked by using a bitwise OR of values from hab_snvs_unlock_flag_t.

**Exit sequence**:

During the initial call to hab_rvt.exit() in the ROM, the SNVS configuration is updated according to the IC boot and security configurations and SNVS unlock commands, if any.

If the IC is configured as HAB_CFG_CLOSED, but the boot configuration is non-secure:

- It is an unsupported configuration.
- SNVS is forced into the Soft Fail state.

Otherwise, the behavior depends on the IC security configuration and any SNVS unlock command:

- If the IC is configured as HAB_CFG_CLOSED (unless a matching unlock command has been executed):
  - The SNVS LP software reset is disabled.
  - The SNVS zeroizable master key is locked against write.
- If the IC is configured as HAB_CFG_OPEN or HAB_CFG_RETURN:
  - Non-privileged access to the SNVS registers is enabled.
- Otherwise, no changes are made to the SNVS settings.

During all calls to hab_rvt.exit(), HAB verifies that the SNVS SSM state is either Trusted or Secure, only when the IC is configured as HAB_CFG_CLOSED. If a fault is detected, a failure event is logged to the audit log.

**Master key selection**:

When present on an IC, SNVS provides the master key for use in the HAB_PCL_BLOB key wrap protocol. SNVS allows selection of a master key using a value from hab_snvs_keys_t as the KEK index src in an install key command. A HAB_INV_INDEX event can result from SNVS master key selection with HAB_PCL_BLOB in one of the following circumstances:

- Using a value not in hab_snvs_keys_t
- Using a value involving the zeroizable master key when it is not validly programmed
- Using a value when a different master key selection has been locked in the LP Master Key Control register

Following the install key command, the SNVS master key selection is restored to the value it had before running the command.

**Events**:

If an entry, exit or test operation fails, an audit event is logged with status field HAB_FAILURE, reason field HAB_ENG_FAIL, engine field HAB_ENG_SNVS, and data field containing the following registers (in the specified order):

- HP Security Violation Control
- HP Status
- HP Security Violation Status
- LP Control
- LP Master Key Control
- LP Security Violation Control
- LP Status
- LP Secure Real Time Counter MSB
- LP Secure Real Time Counter LSB

***Note:*** *If a failure occurs when the SNVS is not enabled, the audit event reason field is HAB_WARNING rather than HAB_FAILURE.*

**Security state mapping**:

SNVS supports all the states in HAB. The specific mapping is shown in the table below.

***Note:*** *HAB does not move SNVS into Secure or Hard Fail state automatically.*

| HAB state | SCCv2 state |
|---|---|
| HAB_STATE_INITIAL | Initialize |
| HAB_STATE_CHECK | Check |
| HAB_STATE_NONSECURE | Non-Secure |
| HAB_STATE_TRUSTED | Trusted |
| HAB_STATE_ SECURE | Secure |
| HAB_STATE_FAIL_SOFT | Soft Fail |
| HAB_STATE_FAIL_HAR | Hard Fail |

**Definitions**:

```
/* SNVS master keys
 * Note that the first two master key selections are completely
 */
typedef enum hab_snvs_keys {
    HAB_SNVS_OTPMK = 0,           /**< OTP master key */
    HAB_SNVS_OTPMK_ALIAS = 1,     /**< OTP master key (alias) */
    HAB_SNVS_ZMK = 2,             /**< Zeroizable master key */
    HAB_SNVS_CMK = 3              /**< Combined master key  */
    } hab_snvs_keys_t;

/* SNVS unlock flags */
typedef enum hab_snvs_unlock_flag {
    HAB_SNVS_UNLOCK_LP_SWR = 0x01,    /**< Leave LP SW reset unlocked */
    HAB_SNVS_UNLOCK_ZMK_WRITE = 0x02 /**< Leave Zeroisable Master Key
                                          * write unlocked */
} hab_snvs_unlock_flag_t;
```

## 5.8 Software

**Purpose**:

The software engine is used to implement cryptographic algorithms in contexts where a hardware accelerator is either unavailable or unusable.

HAB supports hash computations and public key algorithm calculations, depending on the IC configuration.

**Entry sequence**:

Apart from the self-tests, no externally visible operation occurs for this engine.

**Self-tests**:

During the initial call to hab_rvt.entry() in the ROM, the software engine performs several known-answer tests. If any known-answer test fails, the software engine is marked as inoperative, and operations are directed to other engines where available. Subsequent invocations of hab_rvt.entry() do not repeat the self-tests.

**Exit sequence**:

No externally visible operation occurs for this engine.

**Events**:

If an entry, an exit, or a test operation fails, an audit event is logged with status field HAB_FAILURE, reason field HAB_ENG_FAIL, engine field HAB_ENG_SW, and empty data field.

**Algorithms – hash**:

If the following constraint is met, software can be used for hash computation in commands, such as the authenticate data command, through the HAB_ENG_SW *eng* parameter:

- The required algorithm is either HAB_ALG_SHA1 or HAB_ALG_SHA256.

Using HAB_ENG_SW without meeting the constraints results in an unsuccessful operation, with a HAB_UNS_ALGORITHM audit event logged.

If the *eng* parameter is HAB_ENG_ANY and the hash computation is compatible with the constraints of the software engine, HAB may select software automatically.

**Algorithms – signature**:

If no suitable hardware accelerator is available and the following constraint is met, HAB may select software automatically for signature computation in commands, such as authenticate data:

- The required algorithm is of the PKCS#1 signature.

Using HAB_ENG_SW without meeting the constraints results in an unsuccessful operation, with a HAB_UNS_ALGORITHM audit event logged.

**Algorithms – prime field arithmetic**:

If the software engine supports public key operations and the following constraints are met, HAB may select software automatically for performing prime field arithmetic calculations in support of the relevant signature algorithms:

- The input integers are at most 256 bytes (2048 bits).
- The modulus length is a multiple of 32 bits.
- The most significant bit of the modulus is 1.
- The modulus is an odd integer.
- The signature value is less than the modulus value.
- The exponent length is at least 1 byte.
- The exponent length is at most 4 bytes.

If the constraints are not met and no suitable alternative engine is found, the current operation is unsuccessful and a HAB_UNS_ALGORITHM audit event is logged.

**State machine**:

If the IC has no hardware security state machine, a software engine is loaded to maintain the security state. It is a simple state machine that enforces no constraints on the transitions between the HAB states. It is initialized to HAB_STATE_CHECK on first entry to the HAB library.

The software engine state machine may be removed in future versions.

**Configuration**:

No further configuration is supported when selecting the software engine.

# 6 Constants

This section contains the constant definitions used by HAB.

## 6.1 Header

**Purpose**:

The header fields are used to mark the start of various HAB data structures, which can contain a variable number of fields or fields of variable size.

**Format**:

A header is a 4-byte array containing the following three components.

| tag | len | par |
|---|---|---|

**Parameters**:

| Parameter | Description |
|---|---|
| *tag* | Indicates the constant identifying the data structure. Tags are unique across HAB and are separated by at least Hamming distance 2. |
| *len* | Indicates the structure length in bytes (including the header). The minimum length is 4 bytes. |
| *V* | Indicates HAB_MAJOR_VERSION for this data structure. |
| *v* | Indicates HAB_MINOR_VERSION for this data structure. |

**Remarks**:

- Apart from the self-tests, no externally visible operation occurs for this engine.

## 6.2 Structure

**Description**:

Data structure constants.

**External data structure tags**:

| Definition | Value | Description |
|---|---|---|
| HAB_TAG_IVT | 0xd1 | Image vector table |
| HAB_TAG_DCD | 0xd2 | Device configuration data |
| HAB_TAG_CSF | 0xd4 | Command sequence file |
| HAB_TAG_CRT | 0xd7 | Certificate |
| HAB_TAG_SIG | 0xd8 | Signature |
| HAB_TAG_EVT | 0xdb | Event |
| HAB_TAG_RVT | 0xdd | ROM vector table |
| HAB_TAG_WRP | 0x81 | Wrapped key |
| HAB_TAG_MAC | 0xac | Message authentication code |

**HAB version**:

| Definition | Value | Description |
|---|---|---|
| HAB_MAJOR_VERSION | 0x04 | Major version of this HAB release |
| HAB_MINOR_VERSION | | Varies depending on NXP processor and HAB release |

## 6.3 Command

**Description**:

Command constants.

**Command tags**:

| Definition | Value | Description |
|---|---|---|
| HAB_CMD_SET | 0xb1 | Set |
| HAB_CMD_INS_KEY | 0xbe | Install key |
| HAB_CMD_AUT_DAT | 0xca | Authenticate data |
| HAB_CMD_WRT_DAT | 0xcc | Write data |
| HAB_CMD_CHK_DAT | 0xcf | Check data |
| HAB_CMD_NOP | 0xc0 | No operation |
| HAB_CMD_INIT | 0xb4 | Initialize |
| HAB_CMD_UNLK | 0xb2 | Unlock |

## 6.4 Protocol

**Description**:

Protocol constants.

**Protocol tags**:

| Definition | Value | Description |
|---|---|---|
| HAB_PCL_SRK | 0x03 | SRK certificate format |
| HAB_PCL_X509 | 0x09 | X.509v3 certificate format |
| HAB_PCL_CMS | 0xc5 | Cryptographic message syntax (CMS) / PKCS#7 signature format |
| HAB_PCL_BLOB | 0xbb | SHW-specific wrapped key format |
| HAB_PCL_AEAD | 0xa3 | Proprietary AEAD MAC format |

## 6.5 Algorithms

**Description**:

Algorithm constants.

**Algorithm types**:

The most-significant nibble of an algorithm ID denotes the algorithm type. Algorithms of the same type share the same interface.

| Definition | Value | Description |
|---|---|---|
| HAB_ALG_ANY | 0x00 | Algorithm type ANY |
| HAB_ALG_HASH | 0x01 | Hash algorithm type |
| HAB_ALG_SIG | 0x02 | Signature algorithm type |
| HAB_ALG_F | 0x03 | Finite field arithmetic |
| HAB_ALG_EC | 0x04 | Elliptic curve arithmetic |
| HAB_ALG_CIPHER | 0x05 | Cipher algorithm type |
| HAB_ALG_MODE | 0x06 | Cipher/hash modes |
| HAB_ALG_WRAP | 0x07 | Key wrap algorithm type |

**Hash algorithms**:

| Definition | Value | Description |
|---|---|---|
| HAB_ALG_SHA1 | 0x11 | SHA-1 algorithm ID |
| HAB_ALG_SHA256 | 0x17 | SHA-256 algorithm ID |
| HAB_ALG_SHA512 | 0x1b | SHA-512 algorithm ID |

**Signature algorithms**:

| Definition | Value | Description |
|---|---|---|
| HAB_ALG_PKCS1 | 0x21 | PKCS#1 RSA signature algorithm |

**Cipher algorithms**:

| Definition | Value | Description |
|---|---|---|
| HAB_ALG_AES | 0x55 | AES algorithm ID |

**Cipher or hash modes**:

| Definition | Value | Description |
|---|---|---|
| HAB_MODE_CCM | 0x66 | Counter with CBC-MAC |

**Key wrap algorithms**:

| Definition | Value | Description |
|---|---|---|
| HAB_ALG_BLOB | 0x71 | SHW-specific key wrap |

## 6.6 Engine

**Description**:

Security hardware (or software) constants. The term engine denotes a peripheral involved in one or more of the following functions:

- Cryptographic computation
- Security state management
- Security alarm handling

- Access control

By extension, software implementations of the above functionality are also termed engines.

**Engine plug-in tags**:

| Definition | Value | Description |
|---|---|---|
| HAB_ENG_ANY | 0x00 | The first compatible engine is selected automatically. No engine configuration parameter is allowed. |
| HAB_ENG_SCC | 0x03 | Security controller |
| HAB_ENG_RTIC | 0x05 | Run-time integrity checker |
| HAB_ENG_SAHARA | 0x06 | Crypto accelerator |
| HAB_ENG_CSU | 0x0a | Central Security Unit |
| HAB_ENG_SRTC | 0x0c | Secure clock |
| HAB_ENG_DCP | 0x1b | Data Co-Processor |
| HAB_ENG_CAAM | 0x1d | Cryptographic Acceleration and Assurance Module |
| HAB_ENG_SNVS | 0x1e | Secure Non-Volatile Storage |
| HAB_ENG_OCOTP | 0x21 | Fuse controller |
| HAB_ENG_DTCP | 0x22 | DTCP co-processor |
| HAB_ENG_ROM | 0x36 | Protected ROM area |
| HAB_ENG_HDCP | 0x24 | HDCP co-processor |
| HAB_ENG_SW | 0xff | Software engine |

**Miscellaneous engine definitions**:

| Definition | Value | Description |
|---|---|---|
| HAB_RTIC_KEEP | 0x80 | Retain the hash value as a reference for run-time checking later. |
| HAB_DCP_BLOCK_MAX | 6 | The maximum number of non-contiguous memory blocks supported for DCP operations |
| HAB_SAHARA BLOCK_MAX | 12 | The maximum number of non-contiguous memory blocks supported for SAHARA operations |
| HAB_CAAM_BLOCK_MAX | 8 | The maximum number of non-contiguous memory blocks supported for CAAM operations |
| HAB_CAAM_UNLOCK_MID | 0x1 | Leave the Job Ring and DECO Master ID registers unlocked. |
| HAB_CAAM_INIT_RNG | 0x2 | Instantiate Random Number Generator (RNG) state handle 0, generate descriptor keys, set AES DPA mask, and block state handle 0 test instantiation. |

## 6.7 Audit events

### 6.7.1 Reason

**Description**:

Event reason definitions.

 

**Reason definitions**:

| Definition | Value | Description |
|---|---|---|
| HAB_RSN_ANY | 0x00 | Match any reason in hab_rvt.report_event() |
| HAB_ENG_FAIL | 0x30 | Engine failure |
| HAB_INV_ADDRESS | 0x22 | Invalid address. Access is denied. |
| HAB_INV_ASSERTION | 0x0c | Invalid assertion |
| HAB_INV_CALL | 0x28 | Function called out of sequence |
| HAB_INV_CERTIFICATE | 0x21 | Invalid certificate |
| HAB_INV_COMMAND | 0x06 | Invalid command. The command is malformed. |
| HAB_INV_CSF | 0x11 | Invalid command sequence file |
| HAB_INV_DCD | 0x27 | Invalid device configuration data |
| HAB_INV_INDEX | 0x0f | Invalid index. Access is denied. |
| HAB_INV_IVT | 0x05 | Invalid image vector table |
| HAB_INV_KEY | 0x1d | Invalid key |
| HAB_INV_RETURN | 0x1e | Failed callback function |
| HAB_INV_SIGNATURE | 0x18 | Invalid signature |
| HAB_INV_SIZE | 0x17 | Invalid data size |
| HAB_MEM_FAIL | 0x2e | Memory failure |
| HAB_OVR_COUNT | 0x2b | Expired poll count |
| HAB_OVR_STORAGE | 0x2d | Exhausted storage region |
| HAB_UNS_ALGORITHM | 0x12 | Unsupported algorithm |
| HAB_UNS_COMMAND | 0x03 | Unsupported command |
| HAB_UNS_ENGINE | 0x0a | Unsupported engine |
| HAB_UNS_ITEM | 0x24 | Unsupported configuration item |
| HAB_UNS_KEY | 0x1b | Unsupported key type or parameters |
| HAB_UNS_PROTOCOL | 0x14 | Unsupported protocol |
| HAB_UNS_STATE | 0x09 | Unsuitable state |

### 6.7.2 Context

**Description**:

Event context definitions.

**Context definitions**:

| Definition | Value | Description |
|---|---|---|
| HAB_CTX_ANY | 0x00 | Matches any context in hab_rvt.report_event(). |
| HAB_CTX_ENTRY | 0xe1 | Indicates an event logged in hab_rvt.entry(). |
| HAB_CTX_TARGET | 0x33 | Indicates an event logged in hab_rvt.check_target(). |
| HAB_CTX_AUTHENTICATE | 0x0a | Indicates an event logged in hab_rvt.authenticate_image(). |

| Definition | Value | Description |
|---|---|---|
| HAB_CTX_DCD | 0xdd | Indicates an event logged in hab_rvt.run_dcd(). |
| HAB_CTX_CSF | 0xcf | Indicates an event logged in hab_rvt.run_csf(). |
| HAB_CTX_COMMAND | 0xc0 | Indicates an event logged while executing the command sequence file or device configuration data command. |
| HAB_CTX_AUT_DAT | 0xdb | Indicates the authenticated data block. |
| HAB_CTX_ASSERT | 0xa0 | Indicates an event logged in hab_rvt.assert(). |
| HAB_CTX_EXIT | 0xee | Indicates an event logged in hab_rvt.exit(). |

## 6.8 Configuration, status, and state

### 6.8.1 Configuration

**Description**:

HAB configuration definitions.

**Configuration definitions**:

| Definition | Value | Description |
|---|---|---|
| HAB_CFG_RETURN | 0x33 | Field return IC |
| HAB_CFG_OPEN | 0xf0 | Non-secure IC |
| HAB_CFG_CLOSED | 0xcc | Secure IC |

### 6.8.2 Status

**Description**:

HAB status definitions.

**Configuration definitions**:

| Definition | Value | Description |
|---|---|---|
| HAB_STS_ANY | 0x00 | Match any status in hab_rvt.report_event(). |
| HAB_FAILURE | 0x33 | Operation failed. |
| HAB_WARNING | 0x69 | Operation completed with warning. |
| HAB_SUCCESS | 0xf0 | Operation completed successfully. |

### 6.8.3 State

**Description**:

HAB state definitions.

**Configuration definitions**:

| Definition | Value | Description |
|---|---|---|
| HAB_STATE_INITIAL | 0x33 | Initializing state (transitory) |
| HAB_STATE_CHECK | 0x55 | Check state (non-secure) |

| Definition | Value | Description |
|---|---|---|
| HAB_STATE_NONSECURE | 0x66 | Non-Secure state |
| HAB_STATE_TRUSTED | 0x99 | Trusted state |
| HAB_STATE_SECURE | 0xaa | Secure state |
| HAB_STATE_FAIL_SOFT | 0xcc | Soft Fail state |
| HAB_STATE_FAIL_HARD | 0xff | Hard Fail state (terminal) |
| HAB_STATE_NONE | 0xf0 | No security state machine |

# 7 Interpreting HAB event data from report_event() API

This section provides two sets of event data returned from hab_rvt.report_event() and illustrates how to interpret the data.

**Example 1**:

```
0xdb 0x00 0x14 0x41
0x33 0x0c 0xa0 0x00
0x00 0x00 0x00 0x00
0x27 0x80 0x00 0x00
0x00 0x00 0x00 0x20
0x00 0x91 0x00 0x00
0x00 0x00 0x02 0xf0
```

- First confirm that the data is an event consisting of a header, an SRCE (status, reason, context, engine) word and context-dependent data. The first byte is the tag field, which indicates an event when set to HAB_TAG_EVENT. The next two bytes determine the length and the last byte is the HAB version.

```
Header Field: db 00 14 41
              |  |  |  |
              |  |  |  +-- HAB version
              |  +--+-- Event data length in bytes
              +-- Tag: 0xdb = Event
```

- The next word is the SRCE (status|reason|context|engine), which indicates the type of event that occurred. The following is an example:

```
SRCE Field: 33 0c a0 00
            |  |  |  |
            |  |  |  +-- ENG = HAB_ENG_ANY
            |  |  +-- CTX = HAB_CTX_ASSERT
            |  +-- RSN = HAB_INV_ASSERTION
            +-- STS = HAB_FAILURE
```

- In this case, the context is the hab_rvt.assert() API. An assertion event means that one of the following required areas is not signed as documented in the "Operations" subsection for authenticate_image() API:
  - IVT
  - DCD (if provided)
  - Boot Data (initial byte, if provided)
  - Entry point (initial word)

The post condition for hab_rvt.assert() indicates the data portion of the event are:

- A type
- Data pointer
- A count indicating the size of the block in bytes

Currently, the only type defined is `0x00000000` (assertion block). Therefore, for this event the remaining bytes define the data blocks that do not have a required valid signature:

- Address event 1: `27 80 00 00`
- Length event 1: `00 00 00 20`
- Address event 2: `00 91 00 00`
- Length event 2: `00 00 02 f0`

The key to interpreting events is to start always with the context of the SRCE field in the event. The format of the data included in an event depends on the context. If the context is HAB_CTX_CMD, the first byte of the data field matches one of the defined command tags. For example, 0xBE (HAB_CMD_INS_KEY) means that the remaining data matches the install key command format. It identifies the command causing the event that is useful for debugging CSFs.

There are also cases where the context is HAB_CMD_INS_KEY and the first byte does not match a command tag. In this case, check the engine field of (SRCE) to see if it is non-zero (that is, not HAB_ENG_ANY). If so, it means that the event was triggered by a hardware engine, and the remaining data contains registers selected from the hardware engine. The Section 5 section provides details of the registers included in engine-related events.

**Example 2**:

```
0xdb 0x00 0x1c 0x41
0x33 0x18 0x0c 0x00
0xca 0x00 0x14 0x00
0x02 0xc5 0x00 0x00
0x00 0x00 0x07 0x40
0x77 0x80 0x04 0x00
0x00 0x02 0x9c 0x00
```

- As in the previous example, the first word is the header with the first byte as the tag field (for example, `0xdb`).
- Now, for the SRCE field:

```
SRCE Field: 33 18 0c 00
            |  |  |  |
            |  |  |  +-- ENG = HAB_ENG_ANY (0x00)
            |  |  +-- CTX = HAB_CTX_COMMAND (0x0c)
            |  +-- RSN = HAB_INV_SIGNATURE (0x0C)
            +-- STS = HAB_FAILURE (0x33)
```

- Given the context is HAB_CTX_COMMAND, it means, the remaining bytes correspond to the CSF command that caused the event:

```
ca 00 1c 00
|  |  |  |
|  |  |  +-- Event  flags
|  +--+-- Length = 0x001c
+-- HAB_CMD_AUT_DAT = Authenticate data command

02 c5 00 00
|  |  |  |
|  |  |  +-- Configuration = default
|  |  +-- Engine = HAB_ENG_ANY
|  +-- Protocol = HAB_PCL_CMS
+-- Verification key index = 2
    Index 0 corresponds to the SRK
    Index 1 corresponds to the CSF key
    Index 2 or greater corresponds to an Image key

00 00 07 40 - Signature start address (relative offset
              from CSF address in IVT

77 80 04 00 - Data block to be verified starting address

00 02 9c 00 - Length of data block to verify in bytes
```

This event indicates that the digital signature authentication of the data block starting at 0x77800400 has failed.

# 8 References

The following are some additional documents that you can refer to for more information on the HAB APIs:

- Secure Boot with i.MX28 HAB v4 (AN4555)
- i.MX Secure Boot on HABv4 Supported Devices (AN4581)
- HABv4 RVT Guidelines and Recommendations (AN12263)

# 9 Acronyms

Table 1 lists the acronyms used in this document.

Table 1. Acronyms

| Acronym | Description |
|---|---|
| AEAD | Authenticated Encryption with Associated Data |
| AES | Advanced Encryption Standard |
| API | Application programming interface |
| CA | Certificate authority |
| CAAM | Cryptographic Acceleration and Assurance Module |
| CCM | Counter with CBC-MAC |
| CMS | Cryptographic message syntax |
| CSF | Command sequence file |
| DCD | Device configuration data |
| DCP | Data Co-Processor |
| DPA | Differential power analysis |
| DRNG | Deterministic Random Number Generator |
| EC | Elliptic curve |
| HAB | High Assurance Boot |
| HABv4 | HAB version 4 |
| IC | Integrated circuit |
| IVT | Image vector table |
| KEK | Key encryption key |
| MAC | Medium access control |
| OEM | Original equipment manufacturer |
| OS | Operating system |
| PKCS | Public-Key Cryptography Standards |
| RNG | Random Number Generator |
| ROM | Read-only memory |
| RSA | Public key encryption algorithm created by Rivest, Shamir, and Adleman |
| RTIC | Run-Time Integrity Checker |
| RVT | ROM vector table |
| SAHARA | Symmetric/Asymmetric Hash and Random Accelerator |
| SCC | Security Controller |
| SHA | Secure Hash Algorithm |
| SNVS | Secure Non-Volatile Storage |
| SoC | System-on-chip |
| SRK | Super root key |

**Table 1. Acronyms***...continued*

| Acronym | Description |
|---|---|
| SW | Software |
| TRNG | True Random Number Generator |
| UID | Unique ID, a field in the processor and CSF, identifying a device or group of devices |

## 10 Note about the source code in the document

Example code shown in this document has the following copyright and BSD-3-Clause license:

Copyright 2024 NXP Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

# 11 Revision history

Table 2 summarizes the revisions to this document.

**Table 2. Revision history**

| Document ID | Release date | Description |
|---|---|---|
| RM00298 v.2.0 | 5 December 2024 | Put the document into new NXP template and made several enhancements |
| RM00298 v.1.4 | 29 March 2019 | Updated parameter description for the run CSF API |
| RM00298 v.1.3 | 4 April 2018 | • Updated supported APIs:<br>– Authenticate data no DCD<br>– Get version<br>– Authenticate container<br>• Removed the set MID command<br>• Replaced the unlock RNG command with the instantiate RNG command |
| RM00298 v.1.2 | 10 October 2014 | Made minor updates in the hab_rvt.report_event() and hab_rvt.report_status() APIs |
| RM00298 v.1.1 | 27 November 2012 | • Updated parameter description for secret key blob data structure<br>• Made minor updates to the authenticate data command variables<br>• Updated example 2 in Section 7 |
| RM00298 v.1.0 | 5 November 2012 | Initial release. It covers HABv4.1. |

# Legal information

## Definitions

**Draft** — A draft status on a document indicates that the content is still under internal review and subject to formal approval, which may result in modifications or additions. NXP Semiconductors does not give any representations or warranties as to the accuracy or completeness of information included in a draft version of a document and shall have no liability for the consequences of use of such information.

## Disclaimers

**Limited warranty and liability** — Information in this document is believed to be accurate and reliable. However, NXP Semiconductors does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information. NXP Semiconductors takes no responsibility for the content in this document if provided by an information source outside of NXP Semiconductors.

In no event shall NXP Semiconductors be liable for any indirect, incidental, punitive, special or consequential damages (including - without limitation - lost profits, lost savings, business interruption, costs related to the removal or replacement of any products or rework charges) whether or not such damages are based on tort (including negligence), warranty, breach of contract or any other legal theory.

Notwithstanding any damages that customer might incur for any reason whatsoever, NXP Semiconductors' aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms and conditions of commercial sale of NXP Semiconductors.

**Right to make changes** — NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

**Suitability for use** — NXP Semiconductors products are not designed, authorized or warranted to be suitable for use in life support, life-critical or safety-critical systems or equipment, nor in applications where failure or malfunction of an NXP Semiconductors product can reasonably be expected to result in personal injury, death or severe property or environmental damage. NXP Semiconductors and its suppliers accept no liability for inclusion and/or use of NXP Semiconductors products in such equipment or applications and therefore such inclusion and/or use is at the customer's own risk.

**Applications** — Applications that are described herein for any of these products are for illustrative purposes only. NXP Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

Customers are responsible for the design and operation of their applications and products using NXP Semiconductors products, and NXP Semiconductors accepts no liability for any assistance with applications or customer product design. It is customer's sole responsibility to determine whether the NXP Semiconductors product is suitable and fit for the customer's applications and products planned, as well as for the planned application and use of customer's third party customer(s). Customers should provide appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP Semiconductors does not accept any liability related to any default, damage, costs or problem which is based on any weakness or default in the customer's applications or products, or the application or use by customer's third party customer(s). Customer is responsible for doing all necessary testing for the customer's applications and products using NXP Semiconductors products in order to avoid a default of the applications and the products or of the application or use by customer's third party customer(s). NXP does not accept any liability in this respect.

**Terms and conditions of commercial sale** — NXP Semiconductors products are sold subject to the general terms and conditions of commercial sale, as published at https://www.nxp.com/profile/terms, unless otherwise agreed in a valid written individual agreement. In case an individual agreement is concluded only the terms and conditions of the respective agreement shall apply. NXP Semiconductors hereby expressly objects to applying the customer's general terms and conditions with regard to the purchase of NXP Semiconductors products by customer.

**Export control** — This document as well as the item(s) described herein may be subject to export control regulations. Export might require a prior authorization from competent authorities.

**Suitability for use in non-automotive qualified products** — Unless this document expressly states that this specific NXP Semiconductors product is automotive qualified, the product is not suitable for automotive use. It is neither qualified nor tested in accordance with automotive testing or application requirements. NXP Semiconductors accepts no liability for inclusion and/or use of non-automotive qualified products in automotive equipment or applications.

In the event that customer uses the product for design-in and use in automotive applications to automotive specifications and standards, customer (a) shall use the product without NXP Semiconductors' warranty of the product for such automotive applications, use and specifications, and (b) whenever customer uses the product for automotive applications beyond NXP Semiconductors' specifications such use shall be solely at customer's own risk, and (c) customer fully indemnifies NXP Semiconductors for any liability, damages or failed product claims resulting from customer design and use of the product for automotive applications beyond NXP Semiconductors' standard warranty and NXP Semiconductors' product specifications.

**HTML publications** — An HTML version, if available, of this document is provided as a courtesy. Definitive information is contained in the applicable document in PDF format. If there is a discrepancy between the HTML document and the PDF document, the PDF document has priority.

**Translations** — A non-English (translated) version of a document, including the legal information in that document, is for reference only. The English version shall prevail in case of any discrepancy between the translated and English versions.

**Security** — Customer understands that all NXP products may be subject to unidentified vulnerabilities or may support established security standards or specifications with known limitations. Customer is responsible for the design and operation of its applications and products throughout their lifecycles to reduce the effect of these vulnerabilities on customer's applications and products. Customer's responsibility also extends to other open and/or proprietary technologies supported by NXP products for use in customer's applications. NXP accepts no liability for any vulnerability. Customer should regularly check security updates from NXP and follow up appropriately.

Customer shall select products with security features that best meet rules, regulations, and standards of the intended application and make the ultimate design decisions regarding its products and is solely responsible for compliance with all legal, regulatory, and security related requirements concerning its products, regardless of any information or support that may be provided by NXP.

NXP has a Product Security Incident Response Team (PSIRT) (reachable at PSIRT@nxp.com) that manages the investigation, reporting, and solution release to security vulnerabilities of NXP products.

**NXP B.V.** — NXP B.V. is not an operating company and it does not distribute or sell products.

## Trademarks

Notice: All referenced brands, product names, service names, and trademarks are the property of their respective owners.

**NXP** — wordmark and logo are trademarks of NXP B.V.

## Contents







**Date of release: 5 December 2024**
**Document identifier: RM00298**