High Assurance Boot Version 4 Application Programming Interface Reference Manual

Table of Contents

1 About This Book

This manual provides the 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 ROM on certain NXP processors. The HAB API allows image code, external to the ROM, to make calls back to the HAB for authenticating additional boot stages.

1.1 Audience

This document describes the details of the HAB API for engineers architecting and implementing a secure boot. Note that this document describes the API for HAB version 4 only. For information on the HAB version 3 API please refer to the system boot chapter of the relevant NXP processor reference manual.

1.2 Organization

This manual is divided into the following sections:

- Introduction – provides an overview of the HAB API
- Functions – provides detailed description of the HAB API functions
- Data Structures – describes the data structures used by HAB including Device Configuration and Command Sequence File commands.
- Security Hardware – describes the security hardware block used by HAB
- Constants – defines constant values that are used by the HAB API
- Appendix A: Interpreting HAB Event Data from Report_Event() API

1.3 Definitions, Acronyms and Abbreviations

The following list defines the acronyms and abbreviations used in this document.

API	Applications Programming Interface\
CA	Certificate Authority
CAAM	Cryptographic Accelerator and Assurance Module
CSF	Command Sequence File
DCD	Device Configuration Table
DCP	Data Co-Processor
HAB	High Assurance Boot version 4
IVT	Image Vector Table
RTIC	Run-Time Integrity Checker
RVT	ROM Vector Table

SCC	Security Controller
SAHARA	Symmetric, Asymmetric, Hash and Random Accelerator
SNVS	Secure Non-Volatile Storage
SRK	Super Root Key

1.4 Revision History

Date	Version	Description
Nov. 5, 2012	1.0	• Initial Version – covers HAB4.1
Nov. 27, 2012	1.1	• Fix parameter description for Secret Key Blob data structure • Minor update to Authenticate Data command variables • Fix example 2 in Appendix A
Oct. 10, 2014	1.2	• Minor updates for report_event and report_status APIs
Apr. 4, 2018	1.3	• Update supported APIs: Authenticate Data no DCD, Get Version, Authenticate Container • Remove Set MID command, replace Unlock RNG by Init RNG command
March 29, 2019	1.4	• Fix parameters description for Run CSF API

2 Introduction

The HAB library included in on-chip ROMs of certain NXP processors provides a means to authenticate and, in some cases, even encrypt execution images. This secure boot process starts with 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 which can be called from boot images such as U-boot to further extend the secure boot chain. This is illustrated in the figure below.

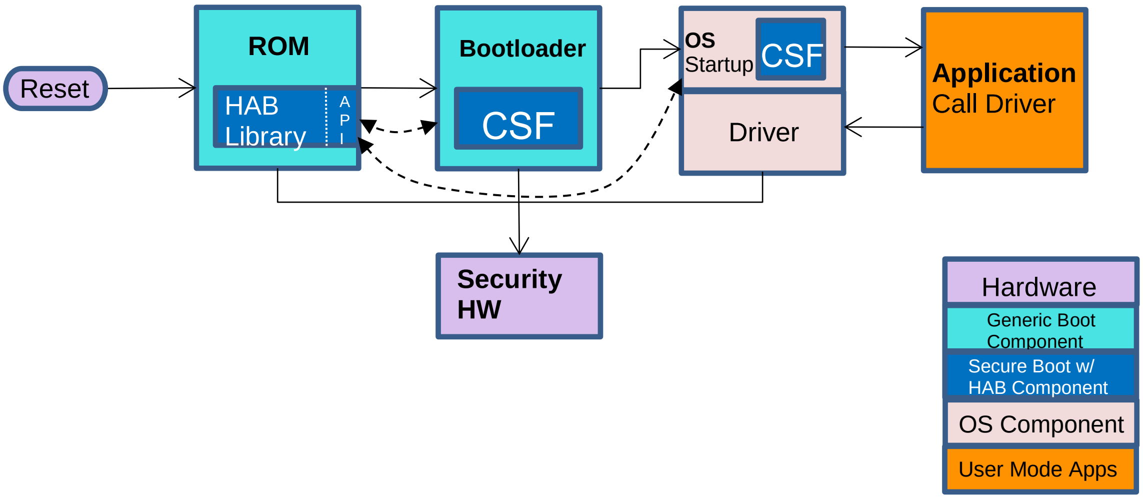


Figure 1 Secure Boot with HAB

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 ROM is specific to each NXP processor. To determine the address of the RVT consult the System Boot Chapter of the NXP processor Reference Manual you are using.

The remainder of this document provides the details for all HAB API functions, HAB commands and engines used by HAB. Note that even though all APIs are documented here NXP recommends using the hab_rvt.authenticate_image() whenever possible. This is instead of calling the other APIs separately ensuring all the proper authentication steps are performed.

For additional details and examples illustrating the use of the HAB API please see application notes AN4555 and AN4581 available for download from www.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 ROM. Consult the System Boot chapter of Reference Manual for the relevant NXP process for the RVT address.

3.1 Entry

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

Enter and initialize HAB library

Purpose:

This function initializes the HAB library and Security Hardware plugins. It is intended for use by post-ROM boot stage components, via the ROM Vector Table, prior to calling any other HAB functions other than hab_rvt.report_event() and hab_rvt.report_status().

Operation:

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 plugin

When first called from boot ROM, this function also performs the following operations prior to those given above:

- Initialize the internal public key store (persists beyond hab_rvt.exit())
- Run the self-test sequence of each available Security Hardware plugin
- 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 that if a security violation has been detected by the HW, the final state will be HAB_STATE_FAIL_SOFT or HAB_STATE_FAIL_HARD depending on the HW configuration.

Warning:

Boot sequences may comprise several images with each launching the next as well as alternative images should one boot device or boot image be unavailable or unusable. The authentication of each image in a boot sequence must be bracketed by its own hab_rvt.entry() ... hab_rvt.exit() pair in order to ensure that security state information gathered for one image cannot be misapplied to another image.

Post Condition:

- HAB library internal state is initialized.
- Available Security Hardware plugins are initialized.
- If a failure or warning occurs during Security Hardware plugin initialization, an audit event is logged with the relevant Engine tag. The status and reason logged are described in the relevant Security Hardware plugin documentation.
- Security state is initialized, if a state machine is present and enabled.

Return values:

- *HAB_SUCCESS* on an IC not configured as HAB_CFG_CLOSED, although unsuccessful operations will still generate audit log events,
- *HAB_SUCCESS* on other ICs if all commands completed without failure (even if warnings were generated),
- *HAB_SUCCESS* otherwise

3.2 Get Version

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

Get HAB version.

Purpose:

This function returns the version of the HAB library. It is intended for use by ROM boot stage, via the ROM Vector Table.

Operation:

This function performs the following operations:

- Return HAB library version.

Return values:

HAB library version 32-bit unsigned integer

3.3 Check Target

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

Check target address.

Purpose:

This function reports whether a given target region is allowed for either peripheral configuration or image loading in memory. It is intended for use by post-ROM boot stage components, via the ROM Vector Table, in order to avoid configuring security-sensitive peripherals, or loading images over sensitive memory regions or outside recognized memory devices in the address map.

Operation:

The lists of allowed target regions vary by IC and core and should be taken from the relevant NXP Processor Reference Manual.

Parameters:

[in]	*type*	Type of target (memory or peripheral)
[in]	*start*	Address of target region
[in]	*bytes*	Size of target region

Postcondition:

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 call parameters. See the Event record documentation for details.

For successful commands, no audit event is logged.

Return values:

- *HAB_SUCCESS* on an IC not configured as HAB_CFG_CLOSED, although unsuccessful operations will still generate audit log events,
- *HAB_SUCCESS* if the given target region lies wholly 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

```
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)
```

Authenticate image.

Purpose:

This function combines DCD, CSF and Assert functions in a standard sequence in order to authenticate a loaded image. It is intended for use by post-ROM boot stage components, via the ROM Vector Table. Support for images partially loaded to an initial location is provided via a callback function.

Important note:

The DCD based SoC initialization mechanism should not be used once the boot process exits the ROM.

The non-ROM user is required to only use the Authenticate Image no DCD function if available, or make sure a null DCD pointer is passed as argument.

Starting from HAB 4.3.7, the Run DCD function, as well as the Authenticate Image function called with a non-null DCD pointer, will return an error if called outside of the boot ROM.

Older versions of HAB will run DCD commands if available, this could lead to an incorrect authentication boot flow.

Operation:

This function performs the following sequence of operations:

- Check that the initial image load addresses pass hab_rvt.check_target().
- Check that the IVT offset lies within the initial image bounds.
- Check that the Image Vector Table *self* and *entry* pointers are not NULL
- Check the Image Vector Table header for consistency and compatibility.
- If provided in the Image Vector Table, calculate the Device Configuration Data initial location, check that it lies 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 check that it lies 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 will be performed on all IC configurations, but will be only enforced on an IC configured as HAB_CFG_CLOSED:

- Check that the final image load addresses pass hab_rvt.check_target().
- Check that the CSF lies within the image bounds, and run the CSF commands.
- Check that all of the following data have been authenticated (using their final locations):
  - IVT;
  - DCD (if provided);
  - Boot Data (initial byte if provided);
  - Entry point (initial word).

Parameters:

`[in]`	*cid*	Caller ID, used to identify which SW issued this call.
`[in]`	*ivt_offset*	Address of target region
`[in,out]`	*start*	Initial (possibly partial) image load address on entry. Final image load address on exit.
`[in,out]`	*bytes*	Initial (possibly partial) image size on entry. Final image size on exit.
`[in]`	*loader*	Callback function to load the full image to its final load address. Set to NULL if not required.

Remarks:

- Caller ID may be bound to signatures verified using keys installed with HAB_CMD_INS_KEY_CID flag. See Install Key.
- A loader callback function may be supplied even if the image is already loaded to its final location on entry.
- Boot Data (boot_data in Image Vector Table) will be ignored if the loader callback function point is set to Null.

Warnings:

- The loader callback function should lie within existing authenticated areas.
- It is the responsibility of the caller to check the initial image load addresses using hab_rvt.check_target() prior to loading the initial image and calling this function.
- After completion of hab_rvt.authenticate_image(), the caller should test using hab_rvt.assert() that the Boot Data was authenticated.
- After completion of hab_rvt.authenticate_image(), the caller should test using hab_rvt.assert() that the Boot Data was authenticated.

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: initial or final image addresses outside allowed regions
  - HAB_INV_ADDRESS: IVT, DCD, Boot Data or CSF outside image bounds
  - HAB_INV_ADDRESS: IVT *self* or *entry* pointer is NULL
  - HAB_INV_CALL: hab_rvt.entry() not run successfully prior to call
  - HAB_INV_IVT: IVT malformed
  - HAB_INV_IVT: IVT version number is less than HAB library version
  - HAB_INV_RETURN: Callback function failed

Return values:

entry field from Image Vector Table on an IC not configured as HAB_CFG_CLOSED provided that the following conditions are met (other unsuccessful operations will 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 on other ICs if all operations completed 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

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

Loader callback.

Purpose:

This function must be supplied by the library caller if required. It is intended to finalize image loading in those boot modes where only a portion of the image is loaded to a temporary initial location prior to device configuration.

Operation:

This function is called during hab_rvt.authenticate_image() between running the Device Configuration Data and Command Sequence File. The operation of this function is defined by the caller.

Parameters:

`[in,out]`	*start*	Initial (possibly partial) image load address on entry. Final image load address on exit.
`[in,out]`	*bytes*	Initial (possibly partial) image size on entry. Final image size on exit.
`[in]`	*boot_data*	Initial Image Vector Table Boot Data load address.

Remarks:

The interpretation of the Boot Data is defined by the caller. Different boot components or modes may use different boot data, or even different loader callback functions.

Warnings:

- It should not be assumed by this function that the Boot Data is valid or authentic.
- It is the responsibility of the loader callback to check the final image load addresses using hab_rvt.check_target() prior to copying any image data.

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 completed successfully,
- *HAB_FAILURE* otherwise

3.5 Authenticate Image no DCD

```
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)
```

Authenticate image without executing the DCD commands.

Purpose:

This function is identical to the Authenticate Image function, except it does not run the DCD. Please refer to the Authenticate Image chapter for more detailed information.

3.6 Authenticate Container

```
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)
```

Authenticate container.

Purpose:

This function is identical to the Authenticate Image function, except it supports different Image Vector Table versions, it authenticates the contents by using a specified SRK set, and it can skip the DCD commands. Please refer to the Authenticate Image chapter for more detailed information.

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 on an IC not configured as HAB_CFG_CLOSED provided that the following conditions are met (other unsuccessful operations will 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 on other ICs if all operations completed without failure (even if warnings were generated),

HAB_FAILURE otherwise

3.7 Run DCD

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

Execute boot configuration script.

Purpose:

This function configures the IC based upon a Device Configuration Data table. It is intended for use by post-ROM boot stage components, via the ROM Vector Table. This function may be invoked as often as required for each boot stage.

The difference between the configuration functionality in this function and hab_rvt.run_csf() arises because the Device Configuration Data table is not authenticated prior to running the commands. Hence, there is a more limited range of commands allowed, and a limited range of parameters to allowed commands.

Important note:

The DCD based SoC initialization mechanism should not be used once the boot process exits the ROM.

The non-ROM user is required to not use the Run DCD function.

Starting from HAB 4.3.7, the Run DCD function, as well as the Authenticate Image function called with a non-null DCD pointer, will return an error if called outside of the boot ROM.

Older versions of HAB will run DCD commands if available, this could lead to an incorrect authentication boot flow.

Operation:

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:

`[in]`	*dcd*	Address of the Device Configuration Data.

Warnings:

It is the responsibility of the caller to ensure that the dcd parameter points to a valid memory location.

The Device Configuration Data must be authenticated by a subsequent Command Sequence File command prior to launching the next boot image, in order to avoid unauthorized configurations which may subvert secure operation. Although the content of the next boot stage's CSF may be out of scope for the hab_rvt.run_dcd() caller, it is possible to enforce this constraint by using hab_rvt.assert() to ensure that both the DCD and any pointers used to locate it have been authenticated.

Each invocation of hab_rvt.run_dcd() must occur between a pair of hab_rvt.entry() and hab_rvt.exit() calls, although multiple hab_rvt.run_dcd() calls (and other HAB calls) may be made in one bracket. This constraint applies whether hab_rvt.run_dcd() is successful or not: a subsequent call to hab_rvt.exit() is required prior to launching the authenticated image or switching to another boot target.

Postconditions:

Many commands may cause side-effects. 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: unsupported command encountered, where DCD version and HAB library version differ
- HAB_FAILURE, with further reasons:
  - HAB_INV_ADDRESS: NULL dcd parameter
  - HAB_INV_CALL: hab_rvt.entry() not run successfully prior to call

  - HAB_INV_COMMAND: command not allowed in DCD
  - HAB_UNS_COMMAND: unrecognized command encountered, where DCD version and HAB library version match
  - HAB_INV_DCD: DCD malformed or too large
  - HAB_INV_DCD: DCD version number is less than HAB library version

Return values:

- *HAB_SUCCESS* on an IC not configured as HAB_CFG_CLOSED, although unsuccessful operations will still generate audit log events,
- *HAB_SUCCESS* on other ICs if all commands completed without failure (even if warnings were generated),
- *HAB_FAILURE* otherwise

3.8 Run CSF

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

Execute an authentication script.

Purpose:

This function authenticates SW images and configures the IC based upon a Command Sequence File. It is intended for use by post-ROM boot stage components, via the ROM Vector Table. This function may be invoked as often as required for each boot stage.

Operation:

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

The internal copy of the Command Sequence File is authenticated by an explicit command in the sequence. Prior to authentication, 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:

`[in]`	*csf*	Address of the Command Sequence File
`[in]`	*cid*	Carrer ID, used to identify which SW issued this call.
`[in]`	*srkmask*	Mask of bits that indicates which SRK set is allowed to authenticate the contents of this CSF (bit 0 = OEM / bit 1 = NXP). If *srkmask* is equal to 0, HAB uses OEM as default. This argument was added in HAB 4.3.0.

Remarks:

Caller ID may be bound to signatures verified using keys installed with HAB_CMD_INS_KEY_CID flag. See Install Key command for details.

Warnings:

It is the responsibility of the caller to ensure that the csf parameter points to a valid memory location.

Each invocation of hab_rvt.run_csf() must occur between a pair of hab_rvt.entry() and hab_rvt.exit() calls, although multiple hab_rvt.run_csf() calls (and other HAB calls) may be made in one bracket. This constraint applies whether hab_rvt.run_csf() is successful or not: a subsequent call to hab_rvt.exit() is required prior to launching the authenticated image or switching to another boot target.

Postconditions:

Many commands may cause side-effects. See the Command Sequence File documentation. In particular, note that 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: unsupported command encountered, where CSF version and HAB library version differ
- HAB_FAILURE, with further reasons:
  - HAB_INV_ADDRESS: NULL csf parameter
  - HAB_INV_CALL: hab_rvt.entry() not run successfully prior to call
  - HAB_INV_COMMAND: command not allowed prior to CSF authentication
  - HAB_UNS_COMMAND: unrecognized command encountered, where CSF version and HAB library version match
  - HAB_INV_CSF: CSF not authenticated

  - HAB_INV_CSF: CSF malformed or too large
  - HAB_INV_CSF: CSF version number is less than HAB library version

Return values:

- *HAB_SUCCESS* on an IC not configured as HAB_CFG_CLOSED, although unsuccessful operations will still generate audit log events,
- *HAB_SUCCESS* on other ICs if all commands completed without failure (even if warnings were generated),
- *HAB_FAILURE* otherwise

3.9 Assert

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

Test an assertion against the audit log.

Purpose:

This function allows the audit log to be interrogated. It is intended for use by post-ROM boot stage components, via the ROM Vector Table, to determine the state of authentication operations. This function may be invoked as often as required for each boot stage.

Operation:

This function checks the required assertion as detailed below.

Parameters:

[in]	*type*	Assertion type
[in]	*data*	Assertion data
[in]	*count*	Data size or count

Memory block authentication:

For HAB_ASSERT_BLOCK assertion type (defined as 0x0), hab_rvt.assert() checks 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 lie wholly 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, but it is assumed that such cases will 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 call parameters. See the Event record documentation for details.

For successful commands, no audit event is logged.

Return values:

- *HAB_SUCCESS* on an IC not configured as HAB_CFG_CLOSED, although unsuccessful operations will still generate audit log events,
- *HAB_SUCCESS* on other ICs if the assertion is confirmed
- *HAB_FAILURE* otherwise

3.10 Report event

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

Report an event from the audit log.

Purpose:

This function allows the audit log to be interrogated. It is intended for use by post-ROM boot stage components, via the ROM Vector Table , to determine the state of authentication operations. This function may be called outside an hab_rvt.entry() / hab_rvt.exit() pair.

It is also available for use by the boot ROM, where it may be used to report boot failures as part of a tethered boot protocol.

Operation:

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:

`[in]`	*status*	Status level of required event
`[in]`	*index*	Index of required event at given status level
`[in]`	*event*	Event record

`[in, out]`	*bytes*	Size of *event* buffer on entry, size of 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. Refer to Event record documentation.

Warning:

Parameter *bytes* may not 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*, but 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 *event* buffer passed is a NULL, pointer.
- *HAB_FAILURE* otherwise

3.11 Report status

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

Report security status.

Purpose:

This function reports the security configuration and state of the IC as well as searching the audit log to determine the status of the boot process. It is intended for use by post-ROM boot stage components, via the ROM Vector Table. This function may be called outside an hab_rvt.entry() / hab_rvt.exit() pair.

Operation:

This function reads the fuses which indicate the security configuration. The fuse map varies by IC and should be taken from the relevant NXP processor reference manual. It also uses the Security Hardware state machine, if present and enabled, to report on the security state.

Parameters:

`[out]`	*config*	Security configuration, NULL if not required
`[out]`	*state*	Security state, NULL if not required

Remarks:

If no Security Hardware state machine is present and enabled, the state HAB_STATE_NONE will be output.

Return values:

- *HAB_SUCCESS* if no warning or failure audit events have been logged,
- *HAB_WARNING* otherwise, if only warning events have been logged.
- *HAB_FAILURE otherwise*

3.12 Failsafe mode

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

Enter failsafe mode.

Purpose:

This function provides a safe path when image authentication has failed, and all possible boot paths have been exhausted. It is intended for use by post-ROM boot stage components, via the ROM Vector Table.

Operation:

The precise details of this function vary by IC and core and should be taken from the relevant NXP processor Security Reference Manual.

Warning:

This function does not return.

Remarks:

Since this function does not return, it implicitly performs the functionality of hab_rvt.exit() in order to ensure an appropriate configuration of the Security Hardware plugins. Two typical implementations are:

- a low-level provisioning protocol in which an image is downloaded to RAM from an external host, authenticated and launched. The downloaded image may communicate with tools on the external host to report the reasons for boot failure and may re-provision the end-product with authentic boot images.

- a failsafe boot mode which does not allow execution to leave the ROM until the IC is reset.

3.13 Exit

hab_status_t (* hab_rvt::exit)(void)

Finalize and exit HAB library.

Purpose:

This function finalizes the HAB library and Security Hardware plugins. It is intended for use by post-ROM boot stage components, via the ROM Vector Table, after calling other HAB functions and prior to launching the next boot stage or switching to another boot path.

Operation:

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 plugin

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 hab_rvt.entry().

Postcondition:

Records are cleared from the audit log. Note that 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 plugins 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 plugin concerned. The status and reason logged are described in the relevant Security Hardware plugin documentation.

Return values:

- *HAB_SUCCESS* on an IC not configured as HAB_CFG_CLOSED, although unsuccessful operations will still generate audit log events,

- *HAB_WARNING* on other ICs if all commands completed without failure (even if warnings were generated),
- *HAB_FAILURE otherwise*

4 Data Structures

Detailed Description:

External data structures required or provided by HAB.

Purpose:

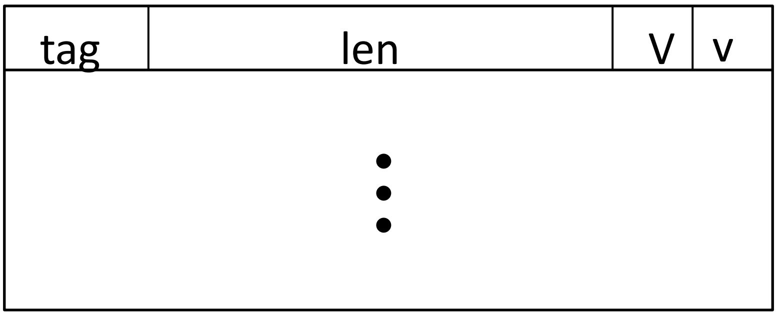
This section defines data structures used by HAB to guide image authentication and device configuration operation, as well as information provided by HAB to assist in authenticating boot sequence components after execution leaves the boot ROM.

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.

There is no constrain imposed on alignment for the start or end of data structures, although word alignment may improve processing speed.

Each HAB data structure starts with a Header in which 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, while others are fixed.



Parameters:

tag constant identifying data structure. Tags are unique across HAB and separated by at least Hamming distance two.

len structure length in 8-bit bytes, including the Header and must be at least four.

V HAB_MAJOR_VERSION for this data structure

v 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 and a corresponding event being 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 chapter 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 chapter of the relevant NXP processor reference manual. DCD supports the Write Data, Check Data and NOP commands.

4.3 Command Sequence File

Detailed Description:

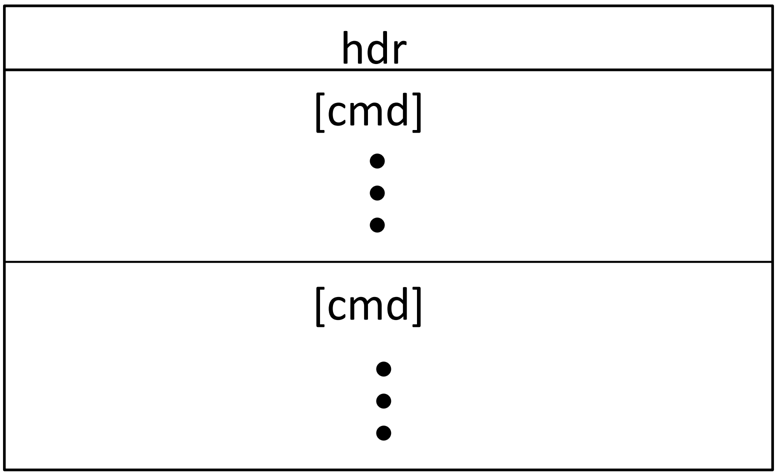
Authentication and configuration script.

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 is accompanied by a CSF which is used by the preceding image to verify authenticity before passing control.

Format:

A CSF consists of a Header followed by a sequence of one or more commands as shown below.



Parameters:

A CSF consists of a Header followed by a sequence of one or more commands as shown below.

hdr Header with tag HAB_TAG_CSF, length and HAB version fields.

cmd 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 prior to CSF authentication.

The first CSF in the boot sequence must contain an Install Key command to install the Super-Root Key Table prior to CSF key installation.

Any other commands encountered before those mentioned above will behave as mentioned in the individual command descriptions.

Remarks:

Once installed, keys may be re-used by subsequent CSFs run by later boot sequence components.

This section lists all HW-independent commands supported by HAB. Further HW-specific commands, where supported, are described in the Security Hardware sections. 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 System boot chapter of the relevant NXP processor reference manual.

4.3.1 Write Data

Detailed Description:

Write Data command (DCD).

Write a list of given 1, 2 or 4-byte values or bitmasks to a corresponding list of target addresses. This command may be used in a Device Configuration Data structure, but in the former, the set of allowed target addresses is restricted, as described in the System boot chapter of the relevant NXP processor reference manual.

The command format is:

<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:

tag	constant HAB_CMD_WRT_DAT
len	constant
par	command parameters - see below
address	target address to which data should be written
val_msk	data value or bitmask to be written to preceding address

See also:

Note on address fields.

The *par* parameter is divided into bit fields as follows:

<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>

Parameters:

bytes width of target locations, 1 (8 bit value), 2 (16 bit value), or 4 (32 bit value)

flags control flags for command behavior, a value from:

0x01 HAB_CMD_WRT_DAT_MSK: Mask/value flag – if set, only specific bits may be overwritten at target address (otherwise all bits may be overwritten).

0x02 HAB_CMD_WRT_DAT_SET: Write Data Set – Set/clear flag: if HAB_CMD_WRT_DAT_MSK is set, bits at the target address overwritten with this flag (otherwise it is ignored).

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 in accordance with the flags as follows:

"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 lie 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 chapter of the relevant NXP process or 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.

Postcondition:

On successful completion, values or bitmasks are written to target locations.

On unsuccessful completion, an audit event is logged giving the status as follows. For successful commands, no audit event is logged:

- HAB_FAILURE, with further reasons:
  - HAB_INV_COMMAND: command malformed.
  - HAB_INV_ADDRESS: access denied for target address.
  - HAB_INV_ADDRESS: misaligned target address.

- HAB_INV_SIZE: value larger than data width.
- HAB_INV_SIZE: unsupported data width.

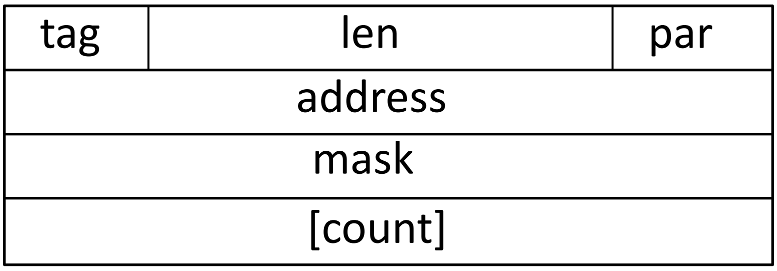
4.3.2 Check Data

Detailed Description:

Check Data command (DCD and CSF).

Test for a given 1, 2 or 4-byte bitmask from a source address. This command may be used in either a Device Configuration Data structure or a Command Sequence File.

The command format is:



tag	len	par
address		
mask		
[count]		

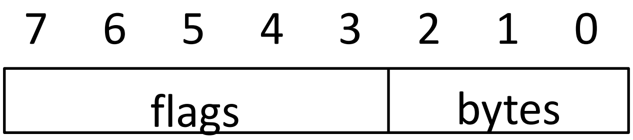
Parameters:

tag	constant HAB_CMD_CHK_DAT
len	constant
par	command parameters - see below
address	source address to test
mask	bitmask to test
count	optional poll count

See also:

Note on address fields.

The *par* parameter is divided into bit fields as follows:



7	6	5	4	3	2	1	0
flags					bytes		

Parameters:

bytes	width of target locations, 1 (8-bit value), 2 (16-bit value), or 4 (32-bit value)
flags	control flags for command behavior, a value from:

0x02 HAB_CMD_CHK_DAT_SET: Set/clear flag – bits set in mask must match this flag.

0x04 HAB_CMD_CHK_DAT_ANY: Any/all flag – if clear, all bits set in mask must match (otherwise any bit suffices).

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 follows:

“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 prior to or after Command Sequence File authentication without any difference.

If *count* is not specified this command will poll 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.

Postcondition:

On unsuccessful completion, an audit event is logged giving the status as follows. For successful commands, no audit event is logged:

- HAB_FAILURE, with further reasons:
  - HAB_INV_COMMAND: command malformed.
  - HAB_INV_ADDRESS: misaligned source address.
  - HAB_INV_SIZE: bitmask wider than data width.
  - HAB_INV_SIZE: unsupported data width.
  - HAB_OVR_COUNT: poll count reached before exit condition met.

4.3.3 NOP

Detailed Description:

This command has no effect (DCD and CSF).

The command format is:

tag	len	und

Parameters:

tag	constant HAB_CMD_NOP
len	constant (four)
und	undefined (ignored)

Remarks:

This command can be used in a Command Sequence File structure prior to or after the Command Sequence File authentication without any difference.

4.3.4 Set

Detailed Description:

Set command (CSF only).

Set the value of variable configuration items.

The command format is:

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

Parameters:

tag	constant HAB_CMD_SET
len	constant
itm	command parameters – see definitions for `hab_var_cfg_itm_t`
value	value to be used for *itm*

Remarks:

This command can be used in a Command Sequence File structure prior to 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.

Postcondition:

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: command malformed.
  - HAB_UNS_ITM: unsupported configuration item.
  - HAB_UNS_ALGORITHM: unsupported algorithm specified.

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 may be overridden in two ways, in decreasing order of precedence:

- specifying an engine other than HAB_ENG_ANY in a CSF command such as Authenticate Data. This has highest priority, but the choice applies only to the individual CSF command: the default behavior resumes once the command is completed. The range of algorithms for which engines may be specified is also limited by the parameters available in the command.
- specifying an engine other than HAB_ENG_ANY in a Set command. This 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.

The format for the value field in the Set command is:

0x0	alg	eng	cfg

Parameters:

alg Algorithm identifier
eng Engine identifier
cfg engine configuration flags (if applicable)

Definitions:

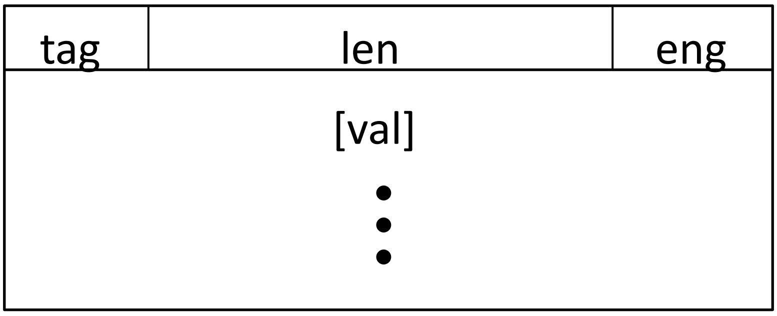
```
/* Variable configuration items */
typedef enum hab_var_cfg_itm {
    /* Preferred engine for a given algorithm */
    HAB_VAR_CFG_ITM_ENG = 0x03
} hab_var_cfg_itm_t;
```

4.3.5 Initialize

Detailed Description:

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

The command format is:



Parameters:

tag	constant HAB_CMD_INIT
len	variable, depending on *eng*
eng	Engine to be initialized
val	[optional] initialization values required by *eng*

Remarks:

Engine-specific values and initialization sequences are described in the relevant Security Hardware section.

Initialize commands are cumulative. A feature will be initialized if specified in one or more Initialize commands.

Warning:

This command may not be used in a Device Configuration Data structure if the IC is configured as HAB_CFG_CLOSED.

Postcondition:

On successful completion, the features specified for Engine will be initialized when hab_rvt.exit() is called.

On unsuccessful completion, the initialization sequences will be omitted unless specified in a separate, successful Initialize command. An audit event is logged giving the status as follows. For successful commands, no audit event is logged.

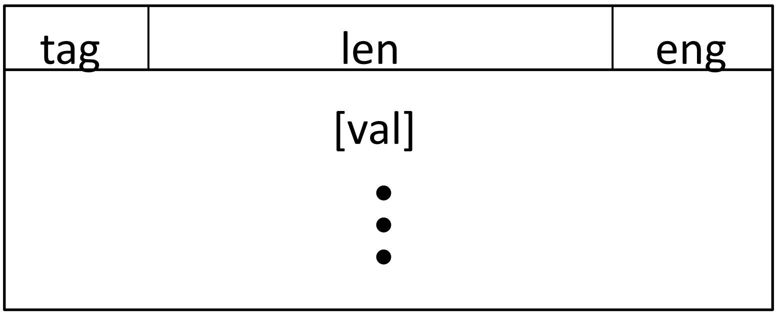
- HAB_FAILURE, with further reasons:
  - HAB_INV_COMMAND: command used outside of authenticated CSF when disallowed by IC security configuration.

4.3.6 Unlock

Detailed Description:

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

The command format is:



Parameters:

tag	constant HAB_CMD_UNLK
len	variable, depending on *eng*
eng	Engine to be left unlocked
val	[optional] unlocks values required by *eng*

Remarks:

Engine-specific values and locks are described in the relevant Security Hardware section.

Unlock commands are cumulative. A feature will be left unlocked if specified in one or more Unlock commands.

Warning:

> This command may not be used in a Device Configuration Data structure if the IC is configured as HAB_CFG_CLOSED.

Postcondition:

On successful completion, the features specified for Engine will not be locked when hab_rvt.exit() is called.

On unsuccessful completion, the features specified will be locked unless specified in a separate, successful Unlock command. An audit event is logged giving the status as follows. For successful commands, no audit event is logged.

- HAB_FAILURE, with further reasons:
  - HAB_INV_COMMAND: command used outside of authenticated CSF when disallowed by IC security configuration.

4.3.7 Install Key

Detailed Description:

Install Key command (CSF only).

Authenticate and install a public key or secret key for use in subsequent Install Key or Authenticate Data commands.

Public key authentication can be restricted to a specific key by including a hash of the key's certificate in the command parameters or 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 pre-installed 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, although a repeat command to re-install the same public key occupying the target slot will be skipped and not generate an error.

The command format is:

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

Parameters:

tag	constant HAB_CMD_INS_KEY
len	variable, depending on size of *crt_hsh* (if present)
flg	flags from hab_cmd_ins_key_flg
pcl	key authentication Protocol
alg	hash Algorithm.
src	source key (verification key, KEK) index.
tgt	target key index.
key_dat	start address of key data to install. Absolute if HAB_CMD_INS_KEY_ABS is set in *flg* parameter, relative to CSF start otherwise
crt_hsh	[optional] 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, and
- *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,
- *tgt* is the public key store index for installation,
- *key_dat* locates the Certificate data, and
- the signature algorithm is determined from the verification key *src*.

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, and
- *src* and *tgt* indices may match 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, and
- 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, then

- *pcl* must be HAB_PCL_SRK,
- only HAB_CMD_INS_KEY_ABS may be set in *flg*, and
- *crt_hsh* must be absent.

Otherwise, if *tgt* is HAB_IDX_CSFK, then

- *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*, and
- *crt_hsh* must be absent.

Otherwise,

- *pcl* must not be HAB_PCL_SRK, and
- *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, and
- *crt_hsh* must be absent.

For secret key installation,

- *alg* must be HAB_ALG_ANY,
- only HAB_CMD_INS_KEY_ABS may be set in *flg*, and
- *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: details should be taken from 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 indices.

Postcondition:

On successful completion, key indicated by *key_dat* parameter is installed at *tgt* index in the appropriate HAB internal key store, along with usage data extracted from *flg* parameter, and the verification protocol or decryption protocol for use with installed key, extracted from the *key_dat* parameter.

On unsuccessful completion, an audit event is logged giving the status as follows. For successful commands, no audit event is logged.

- HAB_WARNING: key installed, but command did not complete as expected, with further reasons:
  - HAB_UNS_ENGINE: default engine (from Set command) is either not recognized or does not support specified algorithm or parameters. Alternative engine used.
  - HAB_ENG_FAIL: failure to release default engine (from Set command).
- HAB_FAILURE otherwise, with further reasons:
  - HAB_INV_COMMAND: command malformed.
  - HAB_INV_COMMAND: Attempt to Install SRK or CSF Key after CSF authentication.
  - HAB_INV_COMMAND: Attempt to Install Keys other than SRK or CSF Key prior to CSF authentication.
  - HAB_INV_KEY: source key in Super-Root Key Table is of type HAB_KEY_HASH.
  - HAB_INV_INDEX: no verification key at given index.
  - HAB_INV_INDEX: no KEK at given index.
  - HAB_INV_INDEX: no source key at given index in Super-Root Key Table.
  - HAB_INV_INDEX: source key unavailable at given index in Super-Root Key Table.
  - HAB_INV_INDEX: target index for installed key unavailable.
  - HAB_UNS_KEY: unsupported public key type or domain parameters (e.g. field size).
  - HAB_UNS_KEY: unsupported secret key type or domain parameters (e.g. key size).
  - HAB_UNS_PROTOCOL: unsupported or unsuitable certificate protocol.
  - HAB_UNS_PROTOCOL: unsupported or unsuitable key wrap protocol.
  - HAB_UNS_ALGORITHM: unsupported or unsuitable hash algorithm.
  - HAB_ENG_FAIL: failure to allocate default engine (from Set command).
  - 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 failed (including mismatch with *crt_hsh*).

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;
```

4.3.7.1Wrapped Key

Detailed Description:

Wrapped secret key data for installation.

Supported in HAB version 4.1 and later with the appropriate protocol(s).

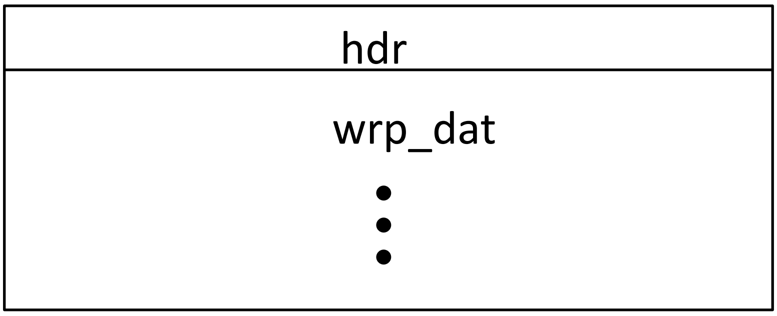
Purpose:

A HAB Wrapped Key structure specifies a secret key to be installed, along with the data required to verify the key's authenticity. 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 Install Key commands.

Format:

A HAB Wrapped Key structure consists of a generic Header followed by protocol-specific data containing the key and authentication data. The wrapped key protocol is determined by a parameter within the Install Key command. Note that the protocol-specific data may have an arbitrary length in bytes.

The storage format is:



Parameters:

hdr	Header with tag HAB_TAG_WRP, length and HAB version fields.
wrp_dat	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.

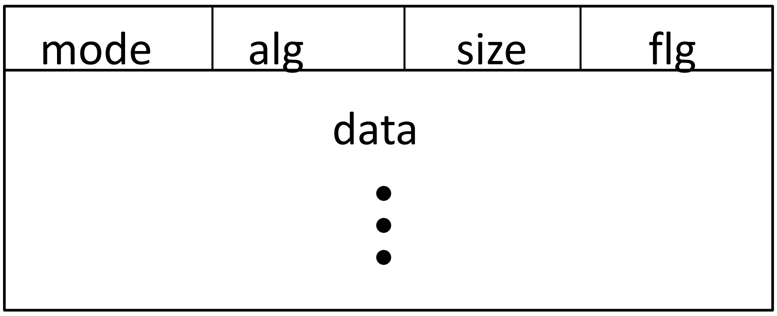
4.3.7.2Secret Key Blob

Purpose:

HAB secret key blobs are used to install secret keys using the special HAB_PCL_BLOB protocol in an Install Key command. This protocol is specific to the available Security Hardware and 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:



Parameters:

mode	key cipher mode
alg	key cipher algorithm
siz	unwrapped key value size in bytes
flg	key flags from the hab_key_secret_flg

data 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 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

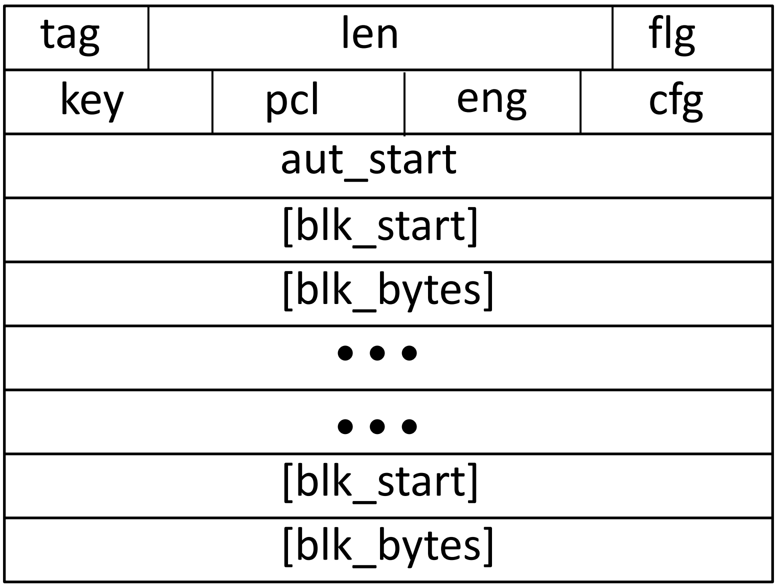
Detailed Description:

Authenticate Data command (CSF only).

Verify the authenticity of pre-loaded data using a pre-installed key. The data may include executable SW instructions and may be spread across multiple non-contiguous blocks in memory.

The authentication protocol may 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 pre-loaded data.

The command format is:



Parameters:

tag	constant HAB_CMD_AUT_DAT
len	variable, depending on number of blocks
flg	flags from the set hab_cmd_aut_dat_flg
key	verification key index
pcl	authentication protocol.
eng	Engine used to process data blocks. Use HAB_ENG_ANY to allow the protocol implementation to choose the first compatible engine.
cfg	Engine configuration flags (if applicable)
aut_start	address of authentication data. Absolute if HAB_CMD_AUT_DAT_ABS is set in *flg* parameter, relative to CSF start otherwise
blk_start	Absolute address of a data block to be authenticated.
blk_bytes	Size in bytes of a data block to be authenticated.

See also:

Note on address fields regarding the *key_dat* parameter.

Remarks:

When more than one data block is indicated, authentication is done on the contents of those data blocks as if they were concatenated in the order given.

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. This is the only CSF authentication recognized by HAB. For example, including the CSF within a region authenticated by another key will not be 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 MAC Engine
- *aut_start* locates the Message Authentication Code data
- the MAC algorithm is determined from the *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 the *key*
- encrypted data is over-written 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, all *blk_start* and *blk_bytes* parameters must be absent.

Postcondition:

This command may alter the configuration of an Engine used in 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 index *key* is uninstalled from the secret key store. This is a precaution against using the same key and nonce combination.

On failure of a secret key authentication and decryption protocol, the decrypted data regions are over-written with zero bytes.

On completion (for any protocol), an audit event is logged giving the status as follows.

- HAB_SUCCESS: data authenticated as specified. Data blocks are logged (except for CSF authentication, where no audit event is logged)
- HAB_WARNING: data authenticated, but command did not complete as specified, with further reasons:
  - HAB_UNS_ENGINE: specified engine is either not recognized or does not support specified algorithm or parameters. Alternative engine used.
  - HAB_ENG_FAIL: failure to release specified engine.
- HAB_FAILURE otherwise, with further reasons:
  - HAB_INV_COMMAND: command malformed.

  - HAB_INV_COMMAND: Attempt to authenticate Image Data prior to CSF Authentication.
  - HAB_INV_COMMAND: Attempt to re-authenticate CSF Data after CSF Authentication.
  - HAB_INV_COMMAND: Attempt to authenticate Image Data with either HAB_IDX_SRK, HAB_IDX_CSFK.
  - HAB_INV_COMMAND: Attempt to authenticate CSF Data with a key other than HAB_IDX_CSFK.
  - HAB_INV_INDEX: no key available at given index or index out of range.
  - HAB_INV_KEY: specified key is identified as a CA key.
  - HAB_UNS_KEY: no engine available for specified key parameters.
  - HAB_UNS_PROTOCOL: unsupported protocol.
  - HAB_UNS_ALGORITHM: unsupported or unsuitable algorithm.
  - HAB_ENG_FAIL: failure to allocate specified engine.
  - HAB_INV_SIGNATURE: data authentication failed. Covers both Signature and Message Authentication Code failure.

Postcondition:

```
/* 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 Report event API.

Format:

An Event record consists of a Header followed by a list of parameters as described below.

<table>
<tr><td colspan="4">hdr</td></tr>
<tr><td>sts</td><td>rsn</td><td>ctx</td><td>eng</td></tr>
<tr><td colspan="4">[data]</td></tr>
</table>

Parameters:

hdr	Header with tag HAB_TAG_EVT, length and HAB version fields
sts	Status level logged.
rsn	Further reason logged
ctx	Context from which the event is logged
eng	Engine associated with the failure, or HAB_ENG_ANY if none
data	Context-dependent data

Remarks:

The length of the *data* field may 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 used internally by HAB. The data field specifies an authenticated data block in an internally-defined format.
- HAB_CTX_ENTRY, HAB_CTX_EXIT: unless specifically mentioned in the relevant Security Hardware section, the data field is empty.
- HAB_CTX_TARGET: the data field consists of the hab_rvt.check_target() call parameters in the order they appear in the parameter list.
- HAB_CTX_COMMAND: the data field consists of the entire command which failed, copied from the Device Configuration Data or Command Sequence File.
- HAB_CTX_ASSERT: the data field consists of the hab_rvt.assert() call parameters in the order 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 ROM for use by post-ROM boot sequence components.

Format:

The ROM Vector Table consists of a Header followed by a list of addresses as described further below. For details on the location of the please refer to the System Boot chapter of the relevant NXP processor reference manual.

Data Fields:

hab_hdr_t hdr

Header with tag HAB_TAG_RVT, length and HAB version fields (see Data Structures)

hab_status_t(* entry)(void)

Enter and initialize HAB library.

hab_status_t(* exit)(void)

Finalize and exit HAB library.

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

Check 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)

Authenticate image.

hab_status_t(* run_dcd)(const uint8_t *dcd)

Execute a boot configuration script.

hab_status_t(* run_csf)(const uint8_t *csf, uint8_t cid)

Execute an authentication script.

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

Test an assertion against the audit log.

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

Report an event from the audit log.

hab_status_t(* report_status)(hab_config_t *config, hab_state_t *state)

Report security status.

void(* failsafe)(void)

Enter failsafe boot mode. The ROM Vector Table consists of a Header followed by a list of addresses as described further 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)

Authenticate image.

uint32_t(* get_version)(void)

Get 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)

Authenticate container.

5 Security Hardware

This section describes all versions of all Security Hardware blocks supported by HAB. For details on the security hardware available please refer to the Security Reference Manual for the relevant NXP processor.

5.1 Security Controller (SCC)

Purpose:

The SCC provides secure RAM storage as well as monitoring the security state of the IC. HAB supports SCCv2

Entry Sequence:

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

Self-test:

During the initial call to hab_rvt.entry() in 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 order):

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

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

Security state mapping:

SCCv2 does not support all of the states in HAB4. The specific mapping is shown in the table below:

HAB State	SCCv2 state

HAB_STATE_INITIAL	Initialize
HAB_STATE_CHECK	Health Check
HAB_STATE_NONSECURE	Non-Secure
HAB_STATE_TRUSTED	Secure
HAB_STATE_ SECURE	-UNSUPPORTED-
HAB_STATE_FAIL_SOFT	Fail Soft
HAB_STATE_FAIL_HAR	Fail Hard

SW may also be selected automatically by HAB if the eng parameter is HAB_ENG_ANY and the hash computation is compatible with the SW engine's constraints.

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-test, no externally-visible operations occur for this engine.

Self test:

During the initial call to hab_rvt.entry() in ROM, DCP performs a number of 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 operations occur 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 order):

- Status
- Channel Status

Algorithms – hash:

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

DCP may be used for hash computation in commands such as Authenticate Data by using a HAB_ENG_DCP eng parameter providing the following constraints are met:

- DCP is enabled
- the alg parameter is HAB_ALG_SHA1 or HAB_ALG_SHA256 and supported on this IC
- at most HAB_DCP_BLOCK_MAX data blocks are covered by the hash (see Authenticate Data)
- except the final one, all data blocks are multiples of 64 bytes in length (the final data block may be an arbitrary length)
- the combined length of all data blocks is less than 512 MB
- all data blocks reside in memory accessible to DCP's DMA engine

Use of HAB_ENG_DCP without meeting the constraints will result in an unsuccessful operation with an HAB_UNS_ALGORITHM audit event being logged.

DCP may also be selected automatically by HAB if the eng parameter is HAB_ENG_ANY and the hash computation is compatible with DCP's constraints.

Configuration:

DCP may be configured for optimal performance and various memory types by means of appropriate Write Data commands in the Device Configuration Data.

DCP may be selected as the default hash engine for a specific algorithm using the Set command. 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-test, no externally-visible operations occur for this engine.

Self-test:

During the initial call to hab_rvt.entry() in 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 operations occur 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 order):

- Status
- Control
- Fault Address

Use of HAB_ENG_RTIC without meeting the constraints will result in an unsuccessful operation with an HAB_UNS_ALGORITHM audit event being logged.

RTIC may also be selected automatically by HAB if the eng parameter is HAB_ENG_ANY and the hash computation is compatible with RTIC's constraints.

Configuration:

RTIC may be configured for optimal performance and various memory types by means of appropriate Write Data commands in the Device Configuration Data.

RTIC may be selected as the default hash engine for a specific algorithm using the Set command. A default configuration is established in the same command.

Retaining computed hash values:

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

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

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

Use of HAB_ENG_RTIC (with or without HAB_RTIC_KEEP) once all the reference hash registers are exhausted will result in an unsuccessful operation with an HAB_UNS_ALGORITHM audit event being logged. This is especially important to note for multiple core ICs with a shared RTIC since 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-test, no externally-visible operations occur for this engine.

Self-test:

During the initial call to hab_rvt.entry() in ROM, SAHARA performs a number of 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 operations occur 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 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, so SAHARA may be used only for SHA-256.

SAHARA may be used for hash computation in commands such as Authenticate Data by using a HAB_ENG_SAHARA eng parameter providing the following constraints are met:

- SAHARA is enabled
- the alg parameter is HAB_ALG_SHA256
- at most HAB_SAHARA_BLOCK_MAX data blocks are covered by the hash (see Authenticate Data)
- all data blocks reside in memory accessible to SAHARA's DMA engine

Use of HAB_ENG_SAHARA without meeting the constraints will result in an unsuccessful operation with an HAB_UNS_ALGORITHM audit event being logged.

SAHARA may also be selected automatically by HAB if the eng parameter is HAB_ENG_ANY and the hash computation is compatible with SAHARA's constraints. DCP supports SHA-1 and SHA-256 hash algorithms.

Configuration:

SAHARA may be configured for optimal performance and various memory types by means of appropriate Write Data commands.

SAHARA may be selected as the default engine for a specific algorithm using the Set command. A default configuration is established in the same command.

5.5 Secure Real Time Clock (SRTC)

Purpose:

SRTC state is controlled by HAB during the boot flow. HAB supports SRTC version 1.

Entry Sequence:

No externally-visible operations occur for this engine.

Self-test:

No self-tests occur 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 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 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 ROM, the SRTC state is updated in accordance with its configuration and state, the IC boot and security configurations, and any SRTC-specific CSF commands.

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

- this 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 Init state,
    - the power glitch register is initialized
    - the SRTC is moved out of Init state
        - if an 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 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 Valid state,
    - unless an 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 operations occur 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 order):

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

The event status is as follows.

- HAB_WARNING, with further reasons:
    - HAB_UNS_STATE: Initialize used with SRTC not in Init state
    - HAB_UNS_STATE: Unlock used with SRTC not in 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 this 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, and
- a secure memory partition is allocated with a single page for the secret key store

Self-test:

During the initial call to hab_rvt.entry() in ROM, CAAM performs a number of 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 Init command enforces specific initializations when hab_rvt.exit() is first called on leaving the ROM. The val command parameter is of the form

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 ROM, CAAM is updated in accordance with the IC boot and security configurations, and any CAAM Unlock commands as follows

- if the IC is configured as HAB_CFG_CLOSED, unless an 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 an Init command with HAB_CAAM_INIT_RNG flagged has been executed,
  - TRNG status is checked for errors in entropy generation
  - DRNG state handle 0 is instantiated (without prediction resistance) using entropy from TRNG
  - descriptor keys (JDKEK, TDKEK and TDSK) are generated
  - AES 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 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, so CAAM may be used only for SHA-256.

CAAM may be used for hash computation in commands such as Authenticate Data by using a HAB_ENG_CAAM eng parameter providing the following constraints are met:

- CAAM is enabled
- the alg parameter is HAB_ALG_SHA256
- at most HAB_CAAM_BLOCK_MAX data blocks are covered by the hash (see Authenticate Data)
- all data blocks reside in memory accessible to CAAM's DMA engine

Use of HAB_ENG_CAAM without meeting the constraints will result in an unsuccessful operation with an HAB_UNS_ALGORITHM audit event being logged.

CAAM may also be selected automatically by HAB if the eng parameter is HAB_ENG_ANY and the hash computation is compatible with CAAM's constraints.

Algorithms – key wrap:

CAAM will be used by HAB for secret key installation from a Secret Key Blob in the Install Key command providing the following constraints are met:

- CAAM is enabled
- the AES engine in CAAM is not disabled due to export control configuration
- the pcl parameter is HAB_PCL_BLOB
- the key_dat parameter locates memory accessible to CAAM's DMA engine

Use of CAAM without meeting the constraints will result in an unsuccessful operation with an HAB_ENG_FAIL or HAB_UNS_PROTOCOL audit event being logged.

The CAAM blob de-capsulation 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 algorithm in for authenticated encryption with associated data (AEAD). This mode may be selected for any supported key size.

CAAM may be used for AEAD MAC computation in Authenticate Data commands by using a HAB_ENG_CAAM eng parameter providing the following constraints are met:

- 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 8 data blocks are covered by the hash (see Authenticate Data)
- all data blocks reside in memory accessible to CAAM's DMA engine

Use of HAB_ENG_CAAM without meeting the constraints will result in an unsuccessful operation with an HAB_UNS_ALGORITHM audit event being logged.

CAAM may also be selected automatically by HAB if the eng parameter is HAB_ENG_ANY and the AEAD MAC computation is compatible with CAAM's constraints.

Configuration:

CAAM may be configured for optimal performance and various memory types by means of appropriate Write Data commands.

CAAM may be selected as the default engine for a specific algorithm using the Set command. 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 as well as 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 if, and only if, 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 val command parameter is of the form:

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 ROM, the SNVS configuration is updated in accordance with the IC boot and security configurations, and any SNVS Unlock commands.

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

- this 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 commands:

- if the IC is configured as HAB_CFG_CLOSED then, unless a matching Unlock command has been executed,
    - SNVS LP SW reset is disabled
    - SNVS zeroizable master key is locked against write
- if the IC is configured as HAB_CFG_OPEN or HAB_CFG_RETURN then,
    - non-privileged access to 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 if, and only if, 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 offers a choice of master keys which can be selected by 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 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 (see [SNVS]); or
- using a value when a different master key selection has been locked in the LP Master Key Control register (see [SNVS]).

Following an Install Key command, the SNVS master key selection is restored to the value it had prior to 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 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 then 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 that HAB itself does not automatically move SNVS into Secure or Hard Fail states.

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 SW engine is used to implement cryptographic algorithms in contexts where a HW 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-test, no externally-visible operations occur for this engine.

Self-test:

During the initial call to hab_rvt.entry() in ROM, the SW engine performs a number of known-answer tests. If any known-answer test fails, the SW 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 operations occur for this engine.

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_SW and empty data field.

Algorithms - hash:

SW may be used for hash computation in commands such as Authenticate Data by using a HAB_ENG_SW eng parameter providing all of the following constraints are met:

- the required algorithm is one of:
  - HAB_ALG_SHA1
  - HAB_ALG_SHA256

Use of HAB_ENG_SW without meeting the constraints will result in an unsuccessful operation with an HAB_UNS_ALGORITHM audit event being logged.

SW may also be selected automatically by HAB if the eng parameter is HAB_ENG_ANY and the hash computation is compatible with the SW engine's constraints.

Algorithms - signature:

Where there is no suitable HW accelerator, SW may be selected automatically by HAB for signature computation in commands such as Authenticate Data providing all the following constraints are met:

- the required algorithm is one of
  - PKCS#1 Signature

Use of HAB_ENG_SW without meeting the constraints will result in an unsuccessful operation with an HAB_UNS_ALGORITHM audit event being logged.

Algorithms – prime field arithmetic:

Where the SW engine supports public key operations, it may be selected automatically to perform prime field arithmetic calculations in support of relevant Signature algorithms, providing the following constraints are met:

- 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:

Where the IC has no HW security state machine, a SW engine is loaded to maintain the security state. This is a very simple state machine which enforces no constraints on the transitions between HAB states. It is initialized to HAB_STATE_CHECK on first entry to the HAB library.

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

Configuration:

No further configuration is supported when selecting the SW engine.

6 Constants

This section contains the constant definitions used by HAB.

6.1 Header

Purpose:

Header fields are used to mark the start of various HAB data structures which may contain a variable number of fields or fields of variable size.

Format:

A Header is a 4-byte array containing three components:

tag	len	par

Parameters:

Apart from the self-test, no externally-visible operations occur for this engine.

Parameters:

tag constant identifying data structure. Tags are unique across HAB and separated by at least Hamming distance two.

len structure length in 8-bit bytes, including the Header and must be at least four.

V HAB_MAJOR_VERSION for this data structure

v HAB_MINOR_VERSION for this data structure

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	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 SW) 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, SW implementations of the above functionality are also termed engines.

Engine plugin tags:

Definition	Value	Description
HAB_ENG_ANY	0x00	First compatible engine will be selected automatically (no engine configuration parameters are 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 reference hash value for later run time checking
HAB_DCP_BLOCK_MAX	6	Maximum on non-contiguous memory blocks supported for DCP operations
HAB_SAHARA BLOCK_MAX	12	Maximum on non-contiguous memory blocks supported for SAHARA operations
HAB_CAAM_BLOCK_MAX	8	Maximum on non-contiguous memory blocks supported for CAAM operations
HAB_CAAM_UNLOCK_MID	0x1	Leave Job Ring and DECO master ID registers unlocked
HAB_CAAM_INIT_RNG	0x2	Instantiate RNG state handle 0, generate descriptor keys, set AES

Definition	Value	Description
		DPA mask, 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 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: command malformed
HAB_INV_CSF	0x11	Invalid Command Sequence File
HAB_INV_DCD	0x27	Invalid Device Configuration Data.
HAB_INV_INDEX	0x0f	Invalid index: access 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	Match any context in hab_rvt.report_event()
HAB_CTX_ENTRY	0xe1	Event logged in hab_rvt.entry()
HAB_CTX_TARGET	0x33	Event logged in hab_rvt.check_target()
HAB_CTX_AUTHENTICATE	0x0a	Event logged in hab_rvt.authenticate_image()
HAB_CTX_DCD	0xdd	Event logged in hab_rvt.run_dcd()
HAB_CTX_CSF	0xcf	Event logged in hab_rvt.run_csf()
HAB_CTX_COMMAND	0xc0	Event logged executing Command Sequence File or Device Configuration Data command
HAB_CTX_AUT_DAT	0xdb	Authenticated data block
HAB_CTX_ASSERT	0xa0	Event logged in hab_rvt.assert()
HAB_CTX_EXIT	0xee	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)
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

Appendix A: Interpreting HAB Event Data from Report_Event() API

The below are two sets of event data returned from hab_rvt.report_event() and illustrate 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 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 Operation section 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:
  1. a type,
  2. data pointer and,
  3. a count indicating the size of the block in bytes.

Currently the only type defined is 0x00000000 (Assertion Block). So 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 Event2:	00 91 00 00
• Length Event 2:	00 00 02 f0

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

There are also cases where the context will be 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 (i.e. not HAB_ENG_ANY). If so, then this means the event was triggered by a HW engine and the remaining data contains select registers from the HW engine. Section 5 contains 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 being the tag field (e.g. 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 this 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.