HAB Code-Signing Tool

User's Guide

Rev. 2.3.1
7/2015

freescale
semiconductor

Contents

About This Book

This manual, the *HAB Code-Signing Tool User's Guide*, provides the details necessary to install, configure, and run the code-signing tool (CST).

Audience

This document provides installation instructions and describes the use of the HAB code signing tools for administrators and engineers performing codes signing for the Freescale High Assurance Boot (HAB) feature.

Scope

This document focuses on the use of the HAB CST to generate keys, certificates, HAB4 SRK tables, HAB3 SRK hash values and generating HAB data which include digital signatures. The use of the Freescale Manufacturing tool to load images and to burn e-fuses are beyond the scope of this document.

Organization

The remainder of this manual is divided into sections according to the main HAB Code Signing Tool user tasks:

- Section 1, "Introduction," describes the background of the code-signing tool and the goals of the procedures in later sections.
- Section 2, "Installation," describes the steps to install the Code-Signing Tool (CST) program files.
- Section 3, "HAB Key and Certificate Generation," details the steps to generate HAB signing keys and certificates for both HAB Version 3 and HAB Version 4.
- Section 4, "CST Usage," describes how to use the CST client command line interface.
- Section 5, "CSF Description Language," provides CST description language details required to create a CSF description file.

Two appendices also included:

- Appendix A, "HAB Library Version 3 Details," presents details about CST Certificates, Signature Verification, and a description of the HAB Version 3 constants.
- Appendix B, "Replacing the CST Backend Implementation," presents details about other possible solutions for the CST Backend such as a Hardware Security Module (HSM)

Revision History

Version	Date	Change Description
1.0	11/15/2011	Initial Version

2.0	11/09/2012	Bug fixes and other updates
2.1	4/15/2013	Add Support for HAB4 fast authentication
2.2	10/14//2014	Add note on Linux RNG dependency Add Appendix B containing details on replacing the CST Back End Corrected CA flag
2.3	3/30/2015	Bug fixes related to encrypted images
2.3.1	7/1/2015	Fix for 64 bit version of srktable

Conventions

Use this section to name, describe, and define any conventions used in the book. This document uses the following notational conventions:

- `Courier monospaced type` indicates commands, command parameters, code examples, expressions, datatypes, and directives.
- *Italic type* indicates replaceable command parameters.
- All source code examples are in C.

Definitions, Acronyms, and Abbreviations

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

AES   Advanced Encryption Standard
API   Application Programming Interface
ASN.1   Abstract Syntax Notation One
CA   Certificate Authority
CCM   Counter with CBC-MAC
CSF   Command Sequence File
CMS   Cryptographic Message Syntax
CST   Code-Signing Tool
DEK   Data Encryption Key
DER   ASN.1 Distinguished Encoding Rules
HAB   High Assurance Boot
HAB3   High Assurance Boot Version 3
HAB4   High Assurance Boot Version 4

HSM	Hardware Security Module
MMU	Memory Management Unit
OS	Operating System
PEM	Privacy Enhanced Mail
PKI	Public Key Infrastructure
PKCS	Public Key Cryptography Standards
RVT	ROM Vector Table
RSA	Public key encryption algorithm created by Rivest, Shamir and Adleman
SA	Signature Authority
SHA	Secure Hash Algorithm
SoC	System on Chip
SRK	Super Root Key
SW	Software
UID	Unique ID — a field in the processor and CSF identifying a device or group of devices
WTLS	Wireless Transport Layer Security

References
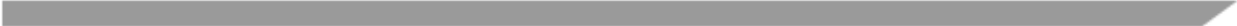
The following sources were referenced to produce this book:

1 *Open Secure Socket Layer (OpenSSL)*, http://www.openssl.org.

2 *RFC 3369: Cryptographic Message Syntax (CMS),* http://www.ietf.org/rfc/rfc3852.txt

3 *RFC 5280: Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*, http://www.ietf.org/rfc/rfc5280.txt

4 *RSA Private-Key Cryptography Standard #8 (PKCS #8) - Private-Key Information Syntax Standard*, version 1.2, RSA Laboratories, http://www.rsa.com/rsalabs.

5 *WAP Certificate and CRL Profiles (WAP-211-WAPCert)*, 22-May-2001, http://www.openmobilealliance.org

6 *RFC 3610: Counter with CBC-MAC (CCM)*, http://www.ietf.org/rfc/rfc3610.txt

7 *AES and Combined Encryption/Authentication Modes*, Brian Gladman, http://www.gladman.me.uk/

Additional Documents

The following documents provide additional information on secure boot with Freescale processors

8 *High Assurance Boot Version 4 Application Programming Interface Reference Manual*. Included as part of the Freescale Reference CST release.

9 *AN4547: Secure Boot on i.MX25, i.MX35, and i.MX51 using HABv3,* http://www.freescale.com

1 *AN4555: Secure Boot with i.MX28 HAB v4,* http://www.freescale.com

1 *AN4581: Secure Boot on i.MX50, i.MX53, and i.MX 6 Series using HABv4,* http://www.freescale.com

1 Introduction

This section provides an introduction to code signing for HAB using the HAB Code-Signing Tool (CST). The CST allows manufacturers to sign or encrypt the software for their products incorporating Freescale processors. Coupled with the High Assurance Boot (HAB) feature included in Freescale processors, the CST can be used to ensure that only genuine or authentic software is allowed to run on the end product.

1.1 HAB Code Signing Components

The secure boot feature using HAB included in many Freescale processors is based on Public Key Infrastructure. The HAB system consists of a two main components:

- The HAB library sub-component of Freescale Processor Boot ROMs.
- The HAB CST.

1.1.1 HAB Library

The HAB library is a sub-component of the boot ROM on select Freescale processors. The HAB library is responsible for verifying the digital signatures included as part of the product software and ensures that, when the processor is configured as a secure device, no unauthenticated code is allowed to run. On Freescale processors supporting the feature, encrypted boot may also be used to provide image cloning protection and, depending on the use case, image confidentiality. The HAB library can not only be used to authenticate the first stage of the boot chain, but the other components of the boot chain as well. The use of HAB is bootloader and OS agnostic. An example is shown in Figure 1 for a generic boot chain.
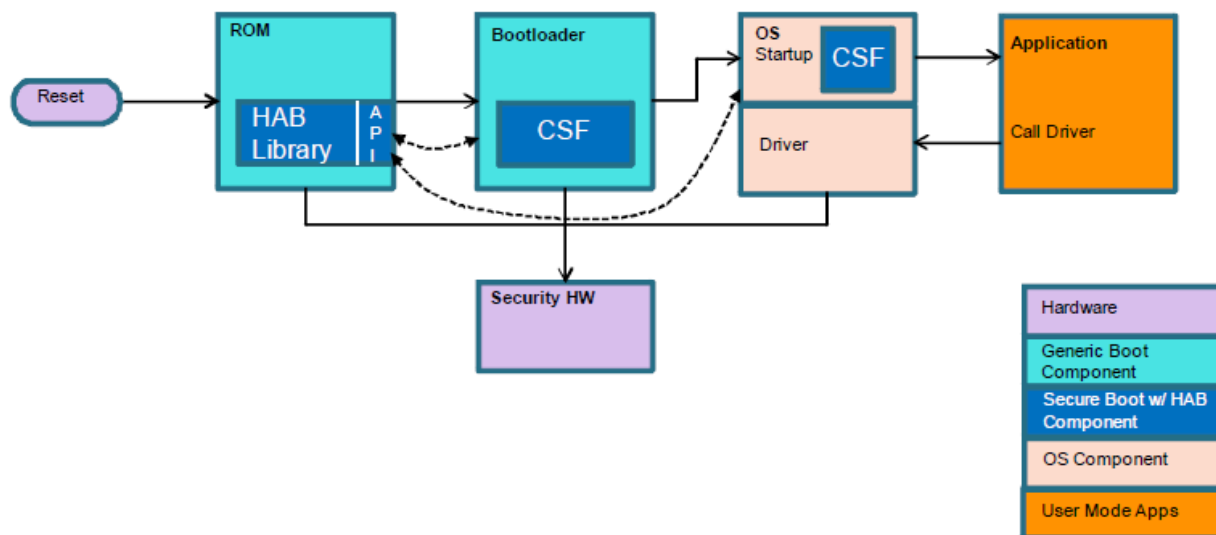
Figure 1. Generic Boot Flow using HAB

The secure boot process starts with the ROM reading efuses to determine the security configuration of the SoC and the type of the boot device. The ROM then loads the bootloader image to DDR memory. The bootloader image contains both the bootloader itself in addition to: commands that the HAB uses to verify the image, digital signature data and public key certificate data which are collectively called Command Sequence File (CSF) data. The CSF data is generated off-line using the HAB Code-Signing Tool (CST) which is introduced in the next section. Once ROM has completed loading the bootloader, execution is then passed to the HAB library which will verify the signatures of the bootloader stage. If signature verification fails, execution is not allowed to leave the ROM for securely configured SoCs. The exact behavior on signature verification failure at the ROM stage is SoC dependent. If all signatures, including image decryption, are successful then execution is passed to the bootloader stage which can perform similar steps to verify the next boot stage by calling back into the HAB API.

N

The ROM and HAB cannot be changed so they can be considered as trusted software components. This allows the use of ROM and HAB to establish a secure boot chain.

HAB requires the use of physical addresses, so if an MMU and a Level 2 cache are enabled within the bootloader stage then the address translation must be a one-to-one mapping. This ensures that all boot components provide HAB with physical addresses. Once all boot components have been verified, HAB is no longer needed and the MMU and Layer 2 cache may be re-configured as required by the Operating System (OS).

The ROM/HAB library integration also provides access to the HAB APIs that boot components outside the ROM may call for image verification. The HAB API are provided either as a jump table or a vector table at a fixed address in the ROM. The exact implementation of the jump or vector tables is SoC dependent so please refer to the Reference Manual for the Freescale processor you are using for specific details.

There are two major versions of the HAB library that exist on Freescale processors: HAB Version 3 (HAB3) and HAB Version 4 (HAB4). Both versions support the flow shown in Figure 1. HAB3 and HAB4 both use public key signature verification to ensure that product code is authentic. There some differences between these versions which are highlighted in Table 1 below. Please see the reference manual for the Freescale processor you are using to determine which version of HAB is supported.

T   **HAB3 - HAB4 Differences**

Feature	HAB3	HAB4
Image Authentication	Yes	Yes
Super Root Key	Single, fused hash	Multiple, revocable, fused hash
Public Key Type	RSA-2048 (Max)	RSA-4096 (Max)
Certificate Format	WTLS	X.509

Signature Format	Proprietary (PKCS#1)	CMS (PKCS#1)
Hash Algorithm	SHA-256	SHA-256
Image Decryption	No	Yes (HAB4.1 and later)
Image Decryption Algorithm	N/A	AES-CCM
Image Decryption Key Blob Algorithm	N/A	Freescale proprietary
Wrapped Key Format	None	CAAM Blob - Secret keys stored in CAAM secure RAM partiton
Secret Key Type	None	AES-128/192/256
Decryption Algorithm	None	AES-CCM - authenticated decryption
Device Configuration Commands	Write value	Write value Set/Clear bitmask Wait on bitmask
Unlock Commands	None	Field Return Fuse Revocation Fuses Secure JTAG etc.

1.1.1.1 HAB Library API

In order to allow boot components outside the ROM to continue the secure boot chain it must be possible for these components to call back into the HAB. There are two versions of the HAB API, one for HAB3 and one for HAB4.

HAB3 API information can be found in System Boot Chapter of either the Reference Manual or Security Reference Manual for the following Freescale processors: i.MX25, i.MX35 and i.MX51. This also includes the jump table and vector table location details for each of the API calls.

Information on the HAB4 API can be found in the HAB4 API Reference Manual.

1.1.2 HAB CST

There are a number of participants involved when performing cryptographic signatures as illustrated in Figure 2. These include:

- A Certificate Authority (CA). The CA is responsible for protecting the top level CA key and for certifying lower level code signing keys.
- A Signature Authority (SA). The SA is responsible for performing the act of code signing.
- A Manufacturer. The Manufacturer is responsible for requesting digital signatures across product software
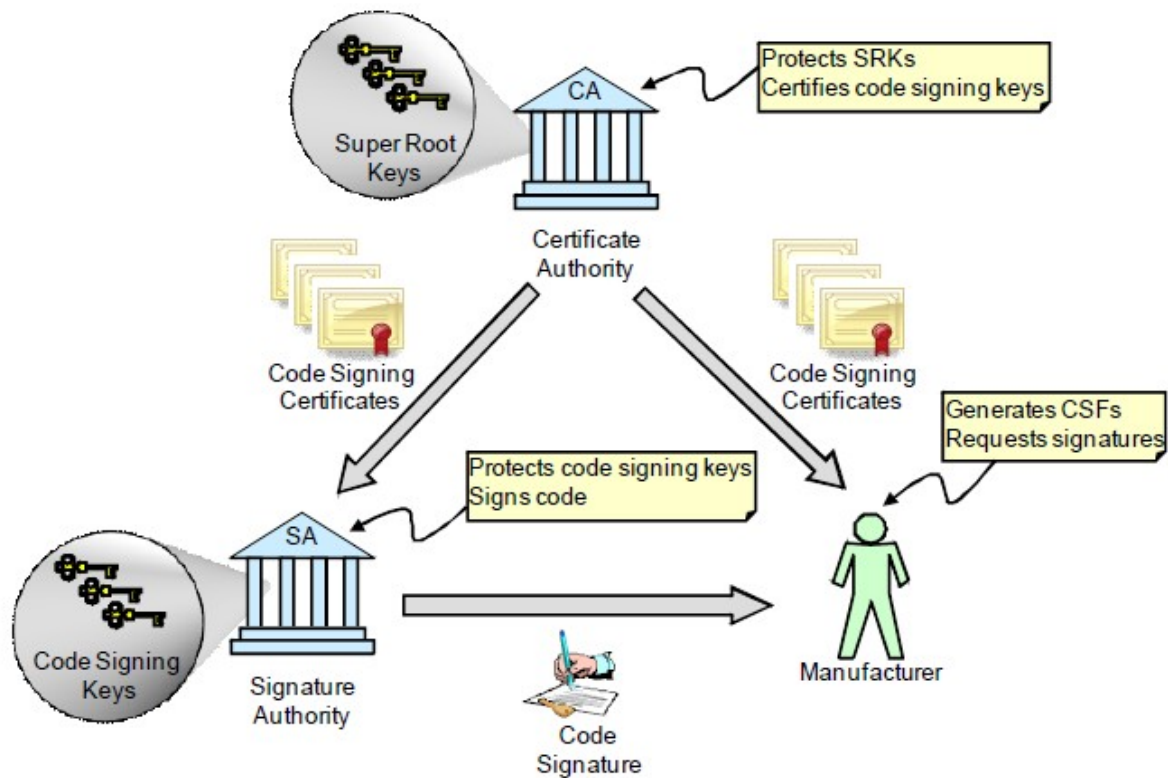
Generic Code Signing Participants

The CST is set of command line tools residing on a host computer which serves as both the Certificate Authority (CA) and Signature Authority (SA) allowing manufacturers to control all aspects of the HAB code signing process.

The CST can establish a PKI tree of keys and certificates (CA function) needed for code signing in addition to generating digital signatures across data provided by a user (SA Function). The signatures generated by the CST can then be included as part of the end product software image. The signatures are then verified by the HAB library on the Freescale processor at boot time.

Figure 3 shows how the CST is used to generate HAB data which includes signatures, certificates and CSF commands the HAB library in ROM will use to validate the product software. The CST takes two main inputs:

 A binary image or image(s) of the product software to be signed.

 A Command Sequence File (CSF). The CSF description file provides the instructions to the CST on what areas of the binary image need to be signed, which keys to sign the image with, etc.

The CST takes these inputs and generates binary HAB data, which includes signatures, certificates and CSF commands that can then be attached to the product software to create a signed image. This User Guide focuses on the details of how to generate the key, certificates, CSF description files and how to run the CST executable.

HAB Code-Signing Tool - Digital Signatures
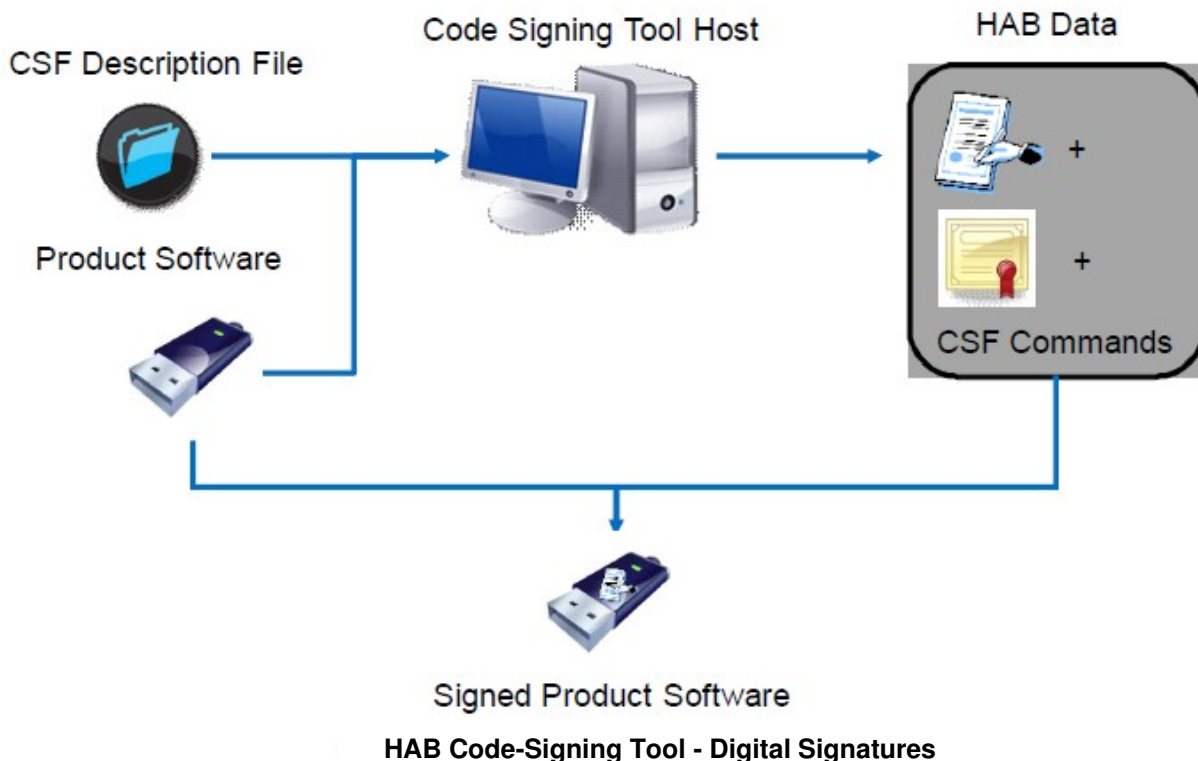
On certain Freescale processors supporting HAB4, encrypted boot may also be used. Figure 4 shows the encrypted boot process with the CST. The encrypted boot case is very similar to generating signed images, but there are two main differences. The first is that the binary image is both decrypted and authenticated using a symmetric key rather than signed using a private asymmetric key. The second is the CST generates a one-time AES Data Encryption Key (DEK) which is used to encrypt the image. Note that when performing an encrypted boot digital signatures are still required, see Section 5.3.4 for an example CSF description file. The DEK is independent of the public keys used for code signing. The DEK output from the CST is protected

but is not in the final form required for an encrypted boot on Freescale processors.
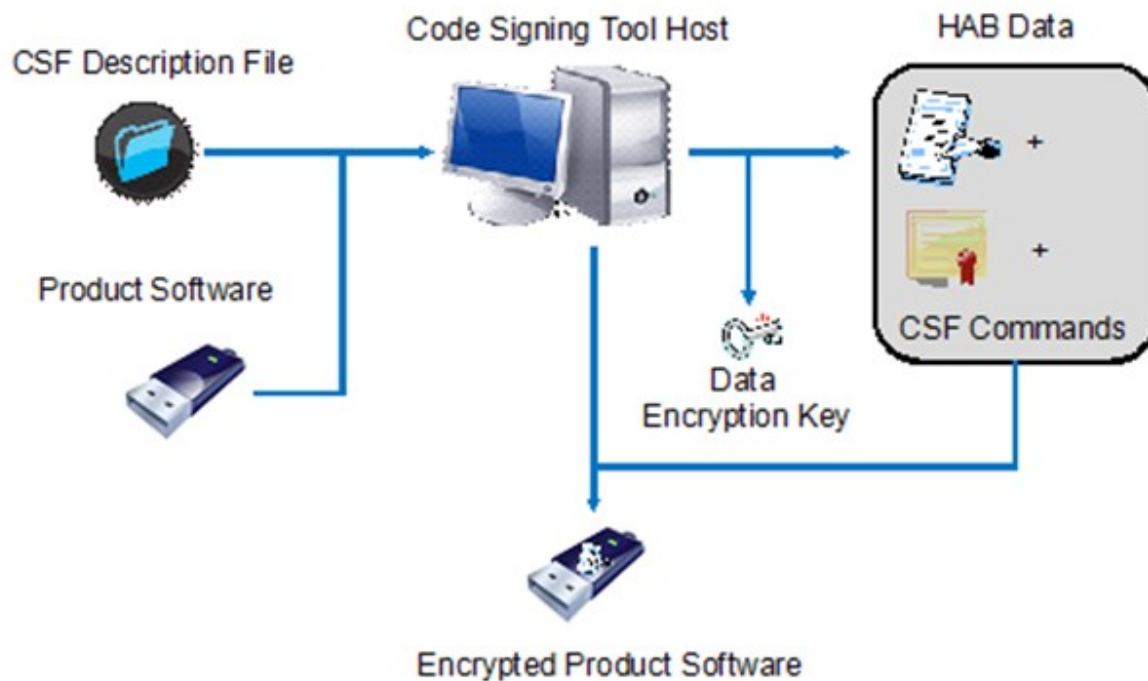
HAB Code Signing Tool - Encrypted Boot

A cryptographic blob of the DEK must be created during the OEM manufacturing stages on each processor and then attached to the image on the boot device. The reason for this is the DEK blob is created using the device unique key embedded into the Freescale processor which is only readable by the on-chip encryption engine. The DEK is common to all ICs using the same encrypted image but the DEK blob is unique per IC. Figure 5 provides an overview of DEK blob creation. The remaining details on DEK blob creation are beyond the scope of the CST and this document.

DEK Blob Creation
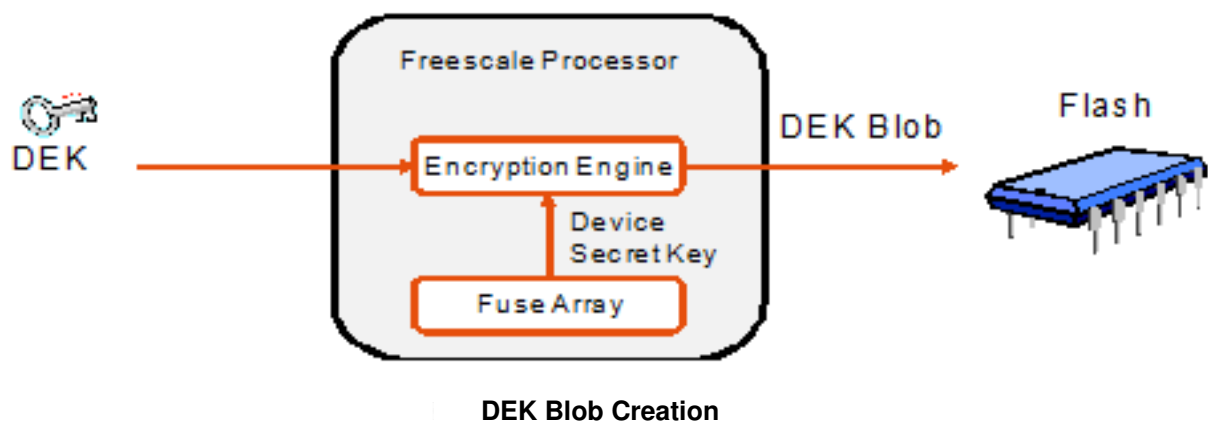
2 Installation

This section describes the installation of the CST code-signing client files.

2.1 CST Package Contents and Installation

The CST is delivered in an archive file, which contains a version for Linux. The archive contains a contents.txt file that lists the entire contents of the archive.

2.1.1 Linux System Requirements

The following checklist can be used to ensure appropriate software is available for the Linux CST. Check with your system administrator if any components are missing.

Table 2. Linux—CST System Requirements Checklist

	Required Component
⊟	A Linux distribution: RHEL4 and 5 are known to work although other distributions such as Ubuntu should also work but have not been formally tested. Check by viewing information shown on Linux login screen

⊟	GNU objcopy 2.15 or later
	    Check by running "`objcopy -V`"
	    Available at http://directory.fsf.org/wiki/Binutils

⊟	OpenSSL 1.0.0d for generating HAB code signing private keys and public key certificates
	    Check by running "`openssl version`"
	    Available at http://www.openssl.org/

NOTE

The Freescale Reference CST uses the Linux OS to generate random numbers for use as keys for HAB encrypted boot. Given this, the Linux host on which the reference CST is installed MUST have good sources of entropy. Generally, this requires multiple entropy sources such as keyboard input, mouse input, network packet arrival times etc. Running the CST without these sources of entropy will cause lengthy delays in seeding the Linux random number generator.

2.1.2 Unpacking the Files

Unpack the CST archive to the desired installation point. The following is an example for Linux and assumes that the client archive was saved in a directory named /home/*<username>*/cst:

```
%cd /home/<username>/cst/
%tar -zxvf <HAB release package name>.tgz
```

This creates the following directories:

ca/

Contains the OpenSSL configuration files. These configuration files are used when generating HAB code signing keys and certificates with the OpenSSL command line tool.

code/

Contains backend CST source and header files. The `/backend` directory contains the source and headers necessary for replacing the open source cryptographic libraries with a different implementation. The `/hab3` directory contains the header defining the data structure for the Super Root Key required when building signed images for Freescale processors containing HAB version 3.

crts/

Contains the public key certificates used for HAB code signing. Initially this directory is empty.

docs/

Contains the CST release notes and this user guide.

linux/

Contains the CST executables for the Linux OS:

`cst` — The CST executable used to sign code for HAB

`srktool` — Generate SRK table and e-fuse files for HAB4. Generates SRK e-fuse information for HAB3.

`x5092wtls` — Converts X.509 certificates to equivalent WTLS certificates required for HAB3.

`/lib` — Contains library files needed for replacing the CST backend implementation.

keys/

Contains the private key files used for HAB code signing. Initially this directory contains scripts to generate the PKI tree:

`hab3_pki_tree.sh` — Use to generate a series of keys and certificates on a Linux machine for use with a Freescale processor supporting HAB3.

`hab4_pki_tree.sh` — Use to generate a series of keys and certificates on a Linux machine for use with a Freescale processor supporting HAB4.

`add_key.sh` — Use to add new keys to an existing HAB3 or HAB4 PKI tree.

Once the archive is unpacked, there are no additional installation steps required in order to use the CST.

3 HAB Key and Certificate Generation

Once the CST installation is complete. The first step in signing code for HAB is generating private keys and certificates. The CST is not delivered with keys or certificates since these will be different for each manufacturer and perhaps even each product line.

The Freescale reference CST generates keys by making use of the OpenSSL command line tool and a set of shell scripts for Linux. This makes OpenSSL the CA component shown in Figure 2. The provided scripts illustrate how to generate a PKI tree of keys and certificates for HAB. There are two sets of scripts generating an initial PKI tree. One for HAB3 and one for HAB4. The reason for this is that the PKI tree structure is different for each version of HAB as well as the final public key certificate format. HAB3 requires public key certificates to be in WTLS [5] format where HAB4 requires X.509 [3] format certificates. The provided key and certificate generation scripts are for reference to illustrate how they should be generated with OpenSSL. Users may update these scripts or replace these scripts with something equivalent if required.

CAUTION
The Freescale reference CST requires a one-to-one correspondence between the key names in the /keys directory and the certificates in /crts directory.
The convention is <keyname>_key.<ext> for keys and <keyname>_crt.<ext> for certificates. For example, a key named keys/SRK1_sha256_2048_65537_v3_ca_key.der must have a corresponding certificate crts/SRK1_sha256_2048_65537_v3_ca_crt.der.

3.1 Generating HAB3 Keys and Certificates

This section covers key and certificate generation for HAB3.

3.1.1 HAB3 PKI Tree

The tree structure for HAB3 generated by the `hab3_pki_tree.sh` script for Linux. This script will generate a HAB3 PKI tree as shown in Figure 6 and is located in the /keys directory of

the Freescale Reference CST.
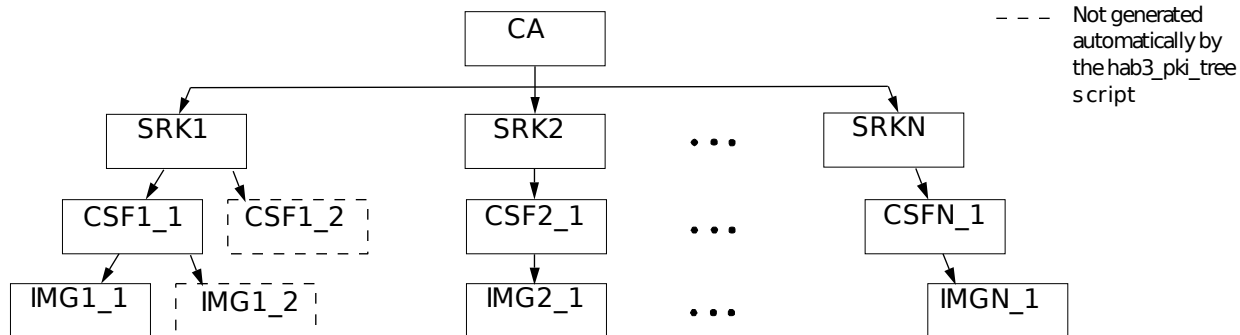
F   HAB3 PKI Tree

A HAB3 PKI tree consists of the following keys and certificates:

- CA key: is the top most key and is only used for signing SRK certificates.
- SRK: is the root key for HAB code signing keys. The cryptographic hash of this key is burned to one-time programmable efuses to establish a root of trust. The SRK must not be used for signing image code. Since the hash of the SRK is blown to efuses on the Freescale processor only one SRK may be used for a product or product line.
- CSF: is a subordinate key of the SRK and is used to verify the signature across CSF commands and to verify signatures of image key certificates.
- IMG: is a subordinate key of the CSF key. Image keys are used to verify signatures across product software.

The hab3_pki_tree script generates a basic tree in which up to a maximum of four SRKs may be generated. For each SRK a single CSF key and IMG key are also generated. Additional keys may be added to the tree later. It is also possible to replace the OpenSSL and the hab3_pki_tree script with an alternative key generation solution, but this is beyond the scope of this document. If the key generation scheme described here is replaced a new scheme must follow these constraints:

- Keys must be in PKCS#8 format
- Certificates must be in WTLS format
- Keys and Certificates must follow the file naming convention specified in the caution message found in Section 3, "HAB Key and Certificate Generation".

3.1.2   Running the hab3_pki_tree script Example

The following are the steps to generate a HAB3 PKI tree for Linux.

```
cd <CST Installation Path>/keys
```

Using your favorite text editor create a file called 'serial' in the /keys directory with contents 12345678. OpenSSL uses the contents of this file for the certificate serial numbers. You may choose to use another number for the initial certificate serial number.

Freescale Semiconductor

Using your favorite text editor create a file called 'key_pass.txt' in the /keys directory. This file contains your pass phrase that will protect the HAB code signing private keys. The format of this file is the pass phase repeated on the first and second lines of the file. For example:

my_pass_phrase

my_pass_phrase

N

Failure to generate the serial and key_pass.txt files prior to running the hab3_pki_tree script will result in OpenSSL errors and the script will fail to generate the requested tree.

C

It is up to the user how best to protect the pass phrase for the private keys. Loss of the pass phrase will result in not being able to sign code with the affected keys.

N

Note that OpenSSL enforces that the pass phrase must be at least four characters long.

Prior to running the `hab3_pki_tree.sh` ensure that OpenSSL is included in your search path by running:

```
> openssl version
OpenSSL 1.0.0d 8 Feb 2011
```

Run the `hab3_pki_tree.sh` script. The script will ask a series of questions:

Do you want to use an existing CA key (y/n)?

Choose no here unless you already have an existing CA key.

If you choose yes, the script will ask you to provide the filenames (including path information) to the location of the CA key and corresponding CA public key certificate.

Enter key length in bits for PKI tree:

This is the length in bit for the keys in the tree. For HAB3 1024 and 2048 bit RSA keys are supported. All keys in the tree are generated with the same length.

Enter PKI tree duration (years):

This defines the validity period of the corresponding certificates.

How many Super Root Keys should be generated?

Up to four SRKs may be generated by this script. This allows for different SRKs to be used for different product lines for example.

At this point the script will generate the SRK, CSF and IMG keys and certificates in the /keys and /crts directory. The generated keys will exist in PKCS#8 [4] format in both PEM and DER forms. The CST will accept key files in either form. Although the script

generates the public key certificates in the /crts in both X.509 [3]and WTLS [5] formats, the CST uses the WTLS certificates only for HAB3. The WTLS certificates exist only in binary format.

N

You may notice that there are a number of files .pem files such as 12345678.pem, serial.old, index.txt.attr and so on. It is normal to see these files and are left over from the OpenSSL key and certificate generation process.

Figure 7 below illustrates the use of the hab3_pki_tree script.

```
[94]> ./hab3_pki_tree.sh

    ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
    This script is a part of the Code signing tools for Freescale's
    High Assurance Boot.  It generates a basic HAB3 PKI tree.
    The PKI tree consists of one or more Super Root Keys (SRK),
    with each SRK having a one subordinate Command Sequence File
    (CSF) key.  Each CSF key Image key then has one subordinate.
    Image key.  Additional keys can be added to the PKI tree but a
    separate script is available for this.  This script
    assumes openssl is installed on your system and is included in
    your search path.  Note that this script automatically generates
    the WTLS certificates required for HAB3.
    Finally, the private keys generated are password
    protected with the password provided by the file key_pass.txt.
    The format of the file is the password repeated twice:
        my_password
        my_password
    All private keys in the PKI tree will be protected by the same
    password.

    ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
Do you want to use an existing CA key (y/n)?: n
Enter key length in bits for PKI tree: 2048
Enter PKI tree duration (years): 5
How many Super Root Keys should be generated? 1

++++++++++++++++++++++++++++++++++++++
+ Generating CA key and certificate +
++++++++++++++++++++++++++++++++++++++

Generating a 2048 bit RSA private key
.............................+++
...........................................................................
............+++
writing new private key to './temp_ca.pem'
-----

+++++++++++++++++++++++++++++++++++++++++
+ Generating SRK key and certificate 1 +
+++++++++++++++++++++++++++++++++++++++++

Generating RSA private key, 2048 bit long modulus
...........................................................................
...........................................................+++
.................+++
e is 65537 (0x10001)
Using configuration from ../ca/openssl.cnf
Check that the request matches the signature
```
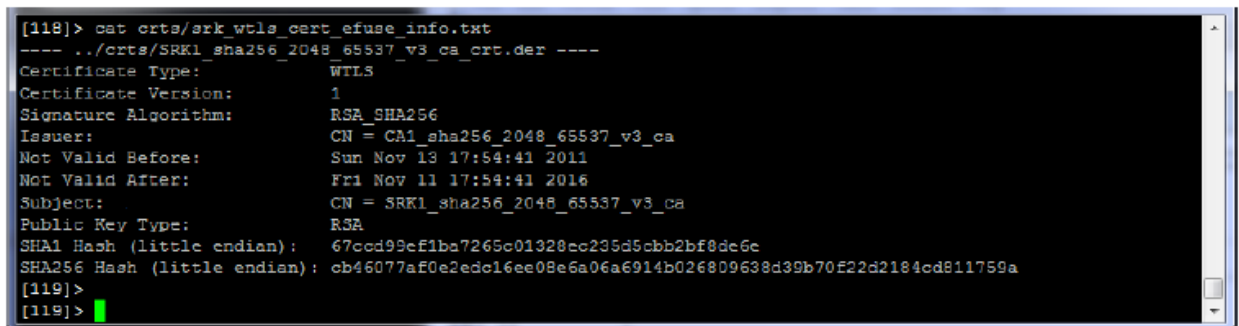
Example Usage of the hab3_pki_tree Script

Freescale Semiconductor

In addition to the keys and certificates there are additional files created which are required for generating signed images for HAB3. These include:

SRK C files: The script generates a C data structure containing the RSA public key exponent and modulus data. For the example above the file /crts/SRK1_sha256_2048_65537_v3_ca_crt.der.c is generated. This file in addition to the `/hdr/hab_super_root.h` file can be included as part of a signed image. This data structure is pointed to by the super root pointer of the ROM application header. Please see the System Boot chapter of the Freescale processor that you are using for further details.

srk_wtls_cert_efuse_info.txt: This file contains the certificate information in addition to a SHA-1 and SHA-256 hash value of the modulus and exponent for each SRK generated. This is the hash value that must be burned to the SRK_HASH fuse field. The results are displayed in little endian meaning the first byte of the hash listed is to be burned to the lowest SRK_HASH address which corresponds to SRK_HASH[255: 248] and the last byte of the hash corresponds to SRK_HASH[7:0]. See Section 3.1.3, "Programming the SRK Hash Value to Efuses" for further details.


```
[118]> cat crts/srk_wtls_cert_efuse_info.txt
---- ../crts/SRK1_sha256_2048_65537_v3_ca_crt.der ----
Certificate Type:        WTLS
Certificate Version:     1
Signature Algorithm:     RSA_SHA256
Issuer:                  CN = CA1_sha256_2048_65537_v3_ca
Not Valid Before:        Sun Nov 13 17:54:41 2011
Not Valid After:         Fri Nov 11 17:54:41 2016
Subject:                 CN = SRK1_sha256_2048_65537_v3_ca
Public Key Type:         RSA
SHA1 Hash (little endian):    67ccd99ef1ba7265c01328ec235d5cbb2bf8de6e
SHA256 Hash (little endian): cb46077af0e2edc16ee08e6a06a6914b026809638d39b70f22d2184cd811759a
[119]>
[119]>
```

Example Output for srk_wtls_cert_efuse_info.txt

At this point all key and certificate information required for signing an image for HAB3 is now available.

3.1.3   Programming the SRK Hash Value to Efuses

The previous section provided the details on how to generate keys and certificated for HAB3 code signing. Included as part of the certificate generation process is the computation of the hash value for each SRK. This hash value is intended to be burned to the SRK_HASH efuse field on the SoC supporting HAB3 and is computed automatically by the hab3_pki_tree script using the `x5092wtls` tool.

The hash value as shown in Figure 8 must be burned to the SoC efuses in the following order:

```
SRK_HASH[255:248] = 0xcb
SRK_HASH[247:240] = 0x46
SRK_HASH[239:232] = 0x07
...
SRK_HASH[15:8] = 0x75
SRK_HASH[7:0] = 0x9a
```

SRK Hash Value Assignment to SoC SRK_HASH Efuse Field for HAB3

Please refer to the fuse map for the Freescale processor you are using for location details of the SRK_HASH field.

3.1.4   Adding a Key to a HAB3 PKI Tree

Adding to an existing HAB3 PKI tree can be done using the add_key script. The following steps are used to add a new key:

1 Run the `add_key.sh` script for Linux. The script will prompt you with a number of questions:
   — Enter new key name (e.g. SRK5):
      – This the name of the new key, such as SRK2, CSF1_2, etc.
   — Enter new key length in bits:
      – This is the length of the new key in bits. This should match the key length of the signing key.
   — Enter certificate duration (years):
      – This defines the validity period for the corresponding certificate generated
   — Which version of HAB do you want to generate the key for (3/4)?:
      – Enter 3 here for HAB3
   — Is this an SRK key?:
      – If you are generating a new SRK enter 'y', otherwise enter 'n'
      – If you enter no you will be prompted with "Is this a CSF key?". Like the SRK enter 'y' if you are generating a CSF key and 'n' if you are generating an IMG key.
      – Note that if you are generating a new SRK the `./crts/srk_wtls_cert_efuse_info.txt` is automatically updated with the certificate information including the SRK hash value.
   — Enter <key type> signing key name:
      – If you are generating an new SRK <key type> is CA. Enter the path and filename of the CA key in the /keys directory.
      – If you are generating a new CSF key <key type> is SRK. Enter the path and filename of the SRK in the /keys directory you wish to use to generate the CSF key.

Freescale Semiconductor

- If you are generating a new IMG key <key type> is CSF. Enter the path and filename of the CSF key in the /keys directory you with to use to generate the IMG key.

— Enter <cert type> signing certificate name:

- If you are generating a new SRK certificate the <cert type> is CA. Enter the path and filename of the CA certificate in the /crts directory.

- If you are generating a new CSF certificate <cert type> is SRK. Enter the path and filename of the SRK certificate in the /certs directory you wish to use to generate the CSF certificate. Note this *must* be the filename of the SRK X.509 certificate not the SRK WTLS certificate.

- If you are generating a new IMG certificate <cert type> is CSF. Enter the path and filename of the CSF certificate in the /certs directory you wish to use to generate the IMG certificate. Note this *must* be the filename of the CSF X.509 certificate not the CSF WTLS certificate.

Using the example from the previous section Figure 10 below shows how to add a new SRK key to the PKI tree.

```
[125]> ./add_key.sh
Enter new key name (e.g. SRK5): SRK2
Enter new key length in bits: 2048
Enter certificate duration (years): 5
Which version of HAB do you want to generate the key for (3/4)?: 3
Is this an SRK key?: y
Enter CA signing key name: CA1_sha256_2048_65537_v3_ca_key.pem
Enter CA signing certificate name: ../crts/CA1_sha256_2048_65537_v3_ca_crt.pem
Generating RSA private key, 2048 bit long modulus
.........................+++
...............................................+++
e is 65537 (0x10001)
Using configuration from ../ca/openssl.cnf
Check that the request matches the signature
Signature ok
The Subject's Distinguished Name is as follows
commonName            :ASN.1 12:'SRK2_sha256_2048_65537_v3_ca'
Certificate is to be certified until Nov 12 01:54:22 2016 GMT (1825 days)

Write out database with 1 new entries
Data Base Updated
[126]>
```

F   Adding a New SRK to a HAB3 PKI Tree Example

C

Do not run this script without first generating a HAB3 PKI tree.
Failure to do so will result in errors.

3.2 Generating HAB4 Keys and Certificates

This section covers only key and certificate generation for HAB4. Note that when using making use of the encrypted boot feature digital signatures are still required. Data structures required by ROM and HAB cannot be encrypted but still must be covered by a valid digital signature. Also, a new symmetric key is dynamically generated by the CST for each Install Secret Key/Decypt Data command pair. These symmetric keys are an output of the CST and encrypted with a supplied public key. See Section 5.3.4 for an example encrypted boot CSF file.

3.2.1 HAB4 PKI Tree

The tree structure for HAB4 generated by the `hab4_pki_tree.sh` script for Linux. This script will generate a HAB4 PKI tree as shown in Figure 11 and is located in the /keys directory of the Freescale Reference CST.
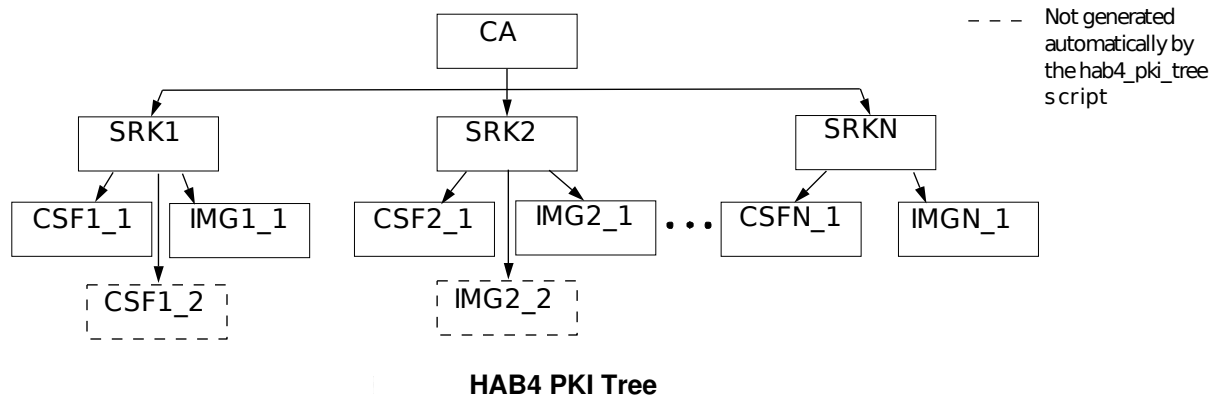
HAB4 PKI Tree

A HAB4 PKI tree consists of the following keys and certificates:

- CA key: is the top most key and is only used for signing SRK certificates.
- SRK: is the root key for HAB code signing keys. The cryptographic hash of a table of SRK is burned to one-time programmable efuses to establish a root of trust. Only one of the SRKs in the table may be selected for use on the Freescale processor per reset cycle. The selection of which SRK to use is a parameter within the Install Key CSF command (see Section 5.2.2, "Install SRK"). The SRK may only be used for signing certificate data of subordinate keys.
- CSF: is a subordinate key of the SRK and is used to verify the signature across CSF commands.
- IMG: is a subordinate key of the SRK key and is used to verify signatures across product software.
- NOTE: The CSF and IMG keys are not generated for a fast authentication PKI tree

The hab4_pki_tree script generates a basic tree in which up to a maximum of four SRKs may be generated. For each SRK a single CSF key and IMG key are also generated. Additional keys may be added to the tree later using a separate script. It is also possible to replace the OpenSSL and

the hab4_pki_tree script with an alternative key generation solution, but this is beyond the scope of this document. If the key generation scheme described here is replaced a new scheme must follow these constraints:

- Keys must be in PKCS#8 format
- Certificates must be in X.509 format following the certificate profile specified by HAB4. Keys and Certificates must follow the file naming convention specified in [Section 3, "HAB Key and Certificate Generation".](#)

3.2.2 Running the hab4_pki_tree script Example

The following are the steps to generate a HAB4 PKI tree for Linux.

```
cd <CST Installation Path>/keys
```

Using your favorite text editor create a file called 'serial' in the /keys directory with contents 12345678. OpenSSL uses the contents of this file for the certificate serial numbers. You may choose to use another number for the initial certificate serial number.

Using your favorite text editor create a file called 'key_pass.txt' in the /keys directory. This file contains your pass phrase that will protect the HAB code signing private keys. The format of this file is the pass phase repeated on the first and second lines of the file. For example:

    my_pass_phrase

    my_pass_phrase

<div align="center">N</div>

Failure to generate the serial and key_pass.txt files prior to running the hab4_pki_tree script will result in OpenSSL errors and the script will fail to generate the requested tree.

<div align="center">C</div>

It is up to the user how best to protect the pass phrase for the private keys. Loss of the pass phrase will result in not being able to sign code with the affected keys.

<div align="center">N</div>

Note that OpenSSL enforces that the pass phrase must be at least four characters long.

Prior to running the `hab4_pki_tree.sh` ensure that OpenSSL is included in your search path by running:

```
> openssl version
OpenSSL 1.0.0d 8 Feb 2011
```

Run the `hab4_pki_tree.sh` script. The script will ask a series of questions:

    Do you want to use an existing CA key (y/n)?

Choose no here unless you already have an existing CA key.

If you choose yes, the script will ask you to provide the filenames (including path information) to the location of the CA key and corresponding CA public key certificate.

Enter key length in bits for PKI tree:

This is the length in bit for the keys in the tree. For HAB4 1024, 2048, 3072 and 4096 bit RSA keys are supported. All keys in the tree are generated with the same length.

Enter PKI tree duration (years):

This defines the validity period of the corresponding certificates.

How many Super Root Keys should be generated?

Up to four SRKs may be generated by this script. This allows for up to four SRKs to be included in a HAB4 SRK table. See Section 4.2, "SRK Tool" for further details.

Do you want the SRK certificates to have the CA Flag set?

Answer 'y' for a standard tree, 'n' for fast authentication tree.

At this point the script will generate the SRK, CSF and IMG keys and certificates in the /keys and /crts directory. The generated keys will exist in PKCS#8 [4] format in both PEM and DER forms. Certificates are located in the /crts directory X.509 [3] format in both PEM and DER format. The `cst` will accept key and certificate files in either PEM or DER form.

N

You may notice that there are a number of files .pem files such as 12345678.pem, serial.old, index.txt.attr and so on. It is normal to see these files and are left over from the OpenSSL key and certificate generation process.

Figure 12 below illustrates the use of the hab4_pki_tree script.

```
[1]> ./hab4_pki_tree.sh

    ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
    This script is a part of the Code signing tools for Freescale's
    High Assurance Boot.  It generates a basic PKI tree.  The PKI
    tree consists of one or more Super Root Keys (SRK), with each
    SRK having two subordinate keys:
        + a Command Sequence File (CSF) key
        + Image key.
    Additional keys can be added to the PKI tree but a separate
    script is available for this.  This this script assumes openssl
    is installed on your system and is included in your search
    path.  Finally, the private keys generated are password
    protectedwith the password provided by the file key_pass.txt.
    The format of the file is the password repeated twice:
        my_password
        my_password
    All private keys in the PKI tree are in PKCS #8 format will be
    protected by the same password.

    ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
Do you want to use an existing CA key (y/n)?: n
Enter key length in bits for PKI tree: 2048
Enter PKI tree duration (years): 10
How many Super Root Keys should be generated? 4

+++++++++++++++++++++++++++++++++++++++++
+ Generating CA key and certificate +
+++++++++++++++++++++++++++++++++++++++++

Generating a 2048 bit RSA private key
.......................+++
..............................+++
writing new private key to 'temp_ca.pem'
-----

+++++++++++++++++++++++++++++++++++++++++++++
+ Generating SRK key and certificate 1 +
+++++++++++++++++++++++++++++++++++++++++++++

Generating RSA private key, 2048 bit long modulus
...........+++
.............................+++
e is 65537 (0x10001)
Using configuration from ../ca/openssl.cnf
Check that the request matches the signature
```

Example Usage of the hab4_pki_tree Script

At this point all key and certificate information required for signing an image for HAB4 is now available.

3.2.3 Generating HAB4 SRK tables and Efuse Hash

The previous section discussed the steps to generate the keys and certificates for a HAB4 PKI tree. Now that they have been generated, the next step is to generate a HAB4 SRK table and corresponding hash value for burning to efuses on the SoC. Unlike HAB3, in HAB4 it is possible to include up to four SRKs in a signed image, although only one may be used per reset cycle. By collecting SRKs in a table it is possible to select one of the SRKs at boot time. The Install SRK
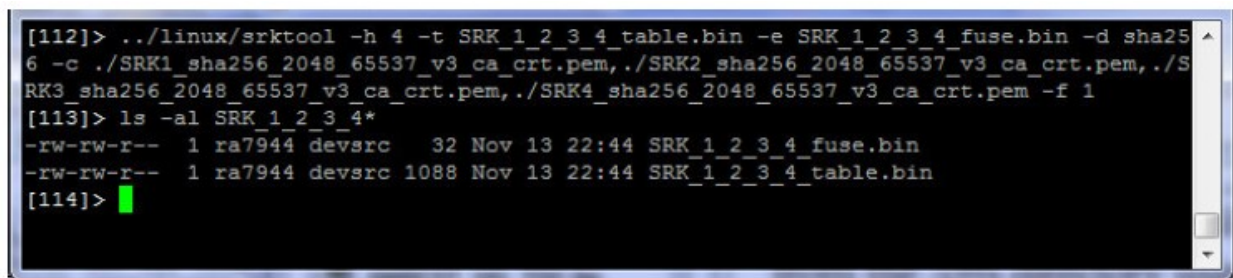
CSF command (see [Section 5.2.2, "Install SRK"](#)) selects which SRK to use from the table to establish the root of trust. Any of the SRKs in the table may be selected without having to change the SRK_HASH value burned to efuses on the SoC.

This is useful on Freescale processors where additional fuses are available for SRK revocation. That is, in the event one or more of the SRKs in the table are compromised, efuses corresponding to the compromised keys can be burned preventing those SRKs from ever being used again. This is enforced by the HAB library. The next SRK in the table can be used to sign new images. A minimum of one and maximum of four SRKs can be placed in an SRK table.

N

Only the first three SRKs in a table can be revoked, so it recommended to use an SRK table with four keys in order to have one SRK to fall back on which cannot be revoked.

SRK tables are generated using the `srktool`. The following illustrates the generation of an SRK table from the /crts directory using the four SRKs created in the previous section.

```
[112]> ../linux/srktool -h 4 -t SRK_1_2_3_4_table.bin -e SRK_1_2_3_4_fuse.bin -d sha25
6 -c ./SRK1_sha256_2048_65537_v3_ca_crt.pem,./SRK2_sha256_2048_65537_v3_ca_crt.pem,./S
RK3_sha256_2048_65537_v3_ca_crt.pem,./SRK4_sha256_2048_65537_v3_ca_crt.pem -f 1
[113]> ls -al SRK_1_2_3_4*
-rw-rw-r--  1 ra7944 devsrc   32 Nov 13 22:44 SRK_1_2_3_4_fuse.bin
-rw-rw-r--  1 ra7944 devsrc 1088 Nov 13 22:44 SRK_1_2_3_4_table.bin
[114]>
```

SRK Table and Efuse Generation Example

In this example:

All four SRKs are included in the table

The SHA-256 hash value is generated with 32 bit of fuse data per word. Some Freescale processors require the hash value to be generated with 8 bits of fuse data per word. In that case use the '-f 0' option.

The hash result in the resulting `SRK_1_2_3_4_fuse.bin` file is in little endian format. This means that the first byte in the file corresponds to SRK_HASH[255:248] and the last byte corresponds to SRK_HASH[7:0] in the efuse map. Similarly when using the '-f 0' option the first non-zero byte in the file corresponds to SRK_HASH[255:248] and the last non-zero byte corresponds to SRK_HASH[7:0].

C

Do not enter spaces between the ',' when specifying the SRKs in the -c or --certs option. Doing so will cause all certificates specified after the first space *not* to be included in the table and resulting efuse hash.

3.2.4   Programming the SRK Hash Value to Efuses

The previous section provided the details on how to SRK tables and the corresponding efuse data. In this section the hash value is of particular interest. The value located in the efuse file is intended to be burned to the SRK_HASH efuse field on the SoC supporting HAB4 and is computed automatically by the hab4_pki_tree script using the `srktool`. The `SRK1_2_3_4_fuse.bin` file from the example in the previous section has the following contents:

> 93ea61d0bd30ffb62aba0b9d5e144d082dd7faeb39223d9e3f9a22a06429895a

This hash value must be burned to the SoC efuses in the following order:

```
SRK_HASH[255:248] = 0x93
SRK_HASH[247:240] = 0xea
SRK_HASH[239:232] = 0x61
...
SRK_HASH[15:8] = 0x89
SRK_HASH[7:0] = 0x5a
```

SRK Hash Value Assignment to SoC SRK_HASH Efuse Field for HAB4

Please refer to the fuse map for the Freescale processor you are using for location details of the SRK_HASH field.

3.2.5   Adding a Key to a HAB4 PKI Tree

Adding to an existing HAB4 PKI tree can be done using the add_key script. The following steps are used to add a new key:

Run the `add_key.sh` script for Linux. The script will prompt you with a number of questions:

Enter new key name (e.g. SRK5):

This the name of the new key, such as SRK2, CSF1_2, etc.

Enter new key length in bits:

This is the length of the new key in bits. This should match the key length of the signing key.

Enter certificate duration (years):

This defines the validity period for the corresponding certificate generated

Which version of HAB do you want to generate the key for (3/4)?:

Enter 4 here for HAB4

Is this an SRK key?:

If you are generating a new SRK enter 'y', otherwise enter 'n'

If you enter no you will be prompted with "Is this a CSF key?". Like the SRK enter 'y' if you are generating a CSF key and 'n' if you are generating an IMG key.

Enter <key type> signing key name:

If you are generating an new SRK <key type> is CA. Enter the path and filename of the CA key in the /keys directory.

If you are generating a new CSF key <key type> is SRK. Enter the path and filename of the SRK in the /keys directory you wish to use to generate the CSF key.

If you are generating a new IMG key <key type> is CSF. Enter the path and filename of the CSF key in the /keys directory you with to use to generate the IMG key.
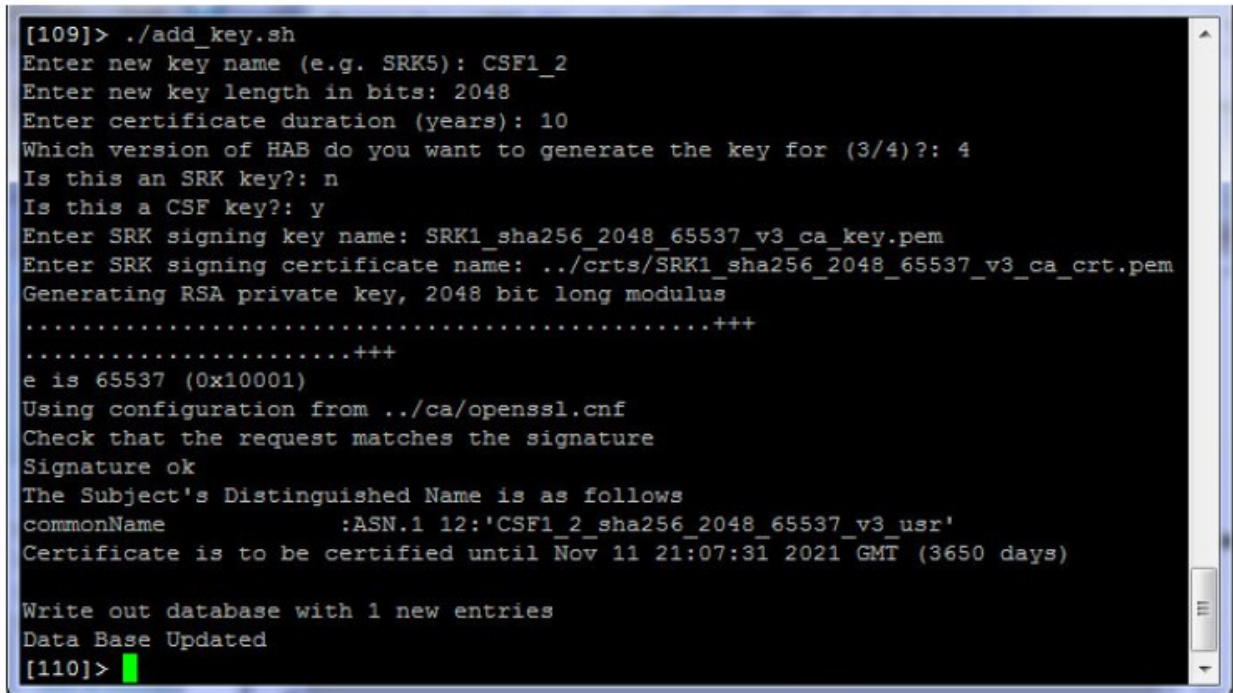
Enter <cert type> signing certificate name:

If you are generating a new SRK certificate the <cert type> is CA. Enter the path and filename of the CA certificate in the /crts directory.

If you are generating a new CSF certificate <cert type> is SRK. Enter the path and filename of the SRK certificate in the /certs directory you wish to use to generate the CSF certificate.

If you are generating a new IMG certificate <cert type> is CSF. Enter the path and filename of the CSF certificate in the /certs directory you wish to use to generate the IMG certificate.

Using the keys generated in Section 3.2.2, "Running the hab4_pki_ tree script Example", Figure 15 below shows how to add a new SRK key to the PKI tree.

```
[109]> ./add_key.sh
Enter new key name (e.g. SRK5): CSF1_2
Enter new key length in bits: 2048
Enter certificate duration (years): 10
Which version of HAB do you want to generate the key for (3/4)?: 4
Is this an SRK key?: n
Is this a CSF key?: y
Enter SRK signing key name: SRK1_sha256_2048_65537_v3_ca_key.pem
Enter SRK signing certificate name: ../crts/SRK1_sha256_2048_65537_v3_ca_crt.pem
Generating RSA private key, 2048 bit long modulus
..............................................+++
.....................+++
e is 65537 (0x10001)
Using configuration from ../ca/openssl.cnf
Check that the request matches the signature
Signature ok
The Subject's Distinguished Name is as follows
commonName            :ASN.1 12:'CSF1_2_sha256_2048_65537_v3_usr'
Certificate is to be certified until Nov 11 21:07:31 2021 GMT (3650 days)

Write out database with 1 new entries
Data Base Updated
[110]>
```

Adding a New SRK to a HAB4 PKI Tree Example

<center>C</center>

Do no run this script without first generating a HAB4 PKI tree.
Failure to do so will result in errors.

4   CST Usage

This section describes how to use the CST and other tools in the release package.

4.1   CST (Code Signing Tool)

The `cst` tool in the release package is the main application used to generate binary CSF data using input CSF description files passed as standard input. The CST can be executed from any location provided the correct absolute or relative path is provided. The paths to certificate and image files inside CSF can be either relative to the current working directory location or as absolute paths.

<center>C</center>

Due to limitation in current cst implementation the cst must be run
from a directory at the same level as <HAB Installation
path>/keys. For example <HAB Installation path>/product_code
where the product code to be signed is located.

Usage:

cst --output <*bin_csf*> [--cert <cert_file>] < <input_csf>

      [--license] [--help]

Description:

```
-o, --output <binary csf>:

    Output binary CSF filename. Required when generating binary CSF output. An
    input CSF passed as standard input is required when this option is given.

-l, --license:

    Optional, displays program license information. No additional arguments are
required

-h, --help:

    Optional, displays usage information. No additional arguments are required

<input_csf>:

    Text file with CSF language commands.

<bin_csf>:

    Binary CSF output file

--cert <cert_file>:
```

```
   Public key certificate filename.  Required when input CSF contains Install
Secret Key command(s).  Symmetric key(s) are encrypted using the public key and
saved to a filename specified in the CSF command
```

Command line arguments that specify a file or directory can contain spaces if they are quoted. File names with leading and trailing spaces are not supported.

If an error occurs during the operation of `cst`, an error message will be printed to the standard output stream and the executable will exit with a non-zero status.

Exit Status:

0   if the executable succeeded, or

>0   otherwise.

Cautions:

None.

Pre Conditions/Assumptions:

Input CSF must be present at specified path.

Certificates must be located in a directory called `crts`.

Keys must be located in a directory called `keys`. The `keys` directory must be located at the same level as the `crts` directory.

Filenames for the keys and certificates must use the following convention <filename>_<type>.pem or <filename>_<type>.der

where: <filename> is the root of the key/certificate filename

<type> is key for keys and crt for certificates.

Example: keys/SRK1_sha256_2048_65537_v3_ca_key.der must have a corresponding certificate crts/SRK1_sha256_2048_65537_v3_ca_crt.der

Post Conditions:

None.

Examples:

To generate out_csf.bin file from input hab4.csf, use

cst --o out_csf.bin < hab4.csf

To print program license information, use

cst --license

To print usage information, use

cst --help

Freescale Semiconductor

To generate out_csf.bin from input hab4.csf and public key certificate to encrypt symmetric key(s)

> cst --o out_csf.bin --cert `dek_protection_crt.pem` < hab4.csf

4.2　　　SRK Tool

For HAB4 the SRK tool is used to generate super root key table data and its hash (for efuses) and for HAB3 it is used to generate efuse information for given keys.

4.2.1　　SRK Tool Usage for HAB3

This section describes usage of SRK tool for HAB3.

Usage:

> srktool --hab_ver <version> --certs <srk>,<srk>,... [--output]

Description:

```
-h, --hab_ver <version>:

      HAB Version - set to 3 to generate HAB3 SRK data/files

-c, --certs <srk1>,<srk2>,...,<srk8>:

      WTLS certificate filenames.

        - WTLS certificates are always in binary format

        - Certificate filenames must be separated by a ',' with no spaces

        - A maximum of 8 certificate filenames may be provided. Additional
           certificate names are ignored

-o, --output:

      Optional, generate file containing the C structures of the
            public keys from the given WTLS certificates. C files use
            certificate filename appending a .c suffix

-l, --license:

      Optional, displays program license information. No additional
      arguments are required.
```

Command line arguments that specify a file or directory can contain spaces if they are quoted. File names with leading and trailing spaces are not supported.

If an error occurs during the operation of `srktool`, an error message will be printed to the standard error stream and the executable will exit with a non-zero status.

Exit Status:

0　　　if the executable succeeded, or

>0　　otherwise.

Cautions:

None.

Pre Conditions/Assumptions:

None.

Post Conditions:

None.

Examples:

To display information for two WTLS certificates

srktool --hab_ver 3 --certs SRK.CA1.FSL.wtls.crt,SRK.CA2.FSL.wtls.crt

To display information for two WTLS certificates and generate corresponding C data output files

srktool --hab_ver 3 --certs SRK.CA1.FSL.wtls.crt,SRK.CA2.FSL.wtls.crt –o

4.2.2 SRK Tool Usage for HAB4

This section describes usage of SRK tool for HAB4.

Usage:

srktool --hab_ver <version> --table <tablefile> --efuses <efusefile>

--digest <digestalg> --certs <srk>,%<srk>,...

[--fuse_format <format>] [--license]

Description:

```
-h, --hab_ver <version>:
     HAB Version - set to 4 for HAB4 SRK table generation
-t, --table <tablefile>:
     Filename for output SRK table binary file
-e, --efuses <efusefile>:
     Filename for the output SRK efuse binary file containing the SRK table
hash
-d, --digest <digestalg>:
     Message Digest algorithm. Either sha1 or sha256
-c, --certs <srk1>,<srk2>,...,<srk4>:
     X.509v3 certificate filenames.
```

```
          - Certificates may be either DER or PEM encoded format
          - Certificate filenames must be separated by a ',' with no spaces
          - A maximum of 4 certificate filenames may be provided. Additional
            certificate names are ignored
          - Placing a % in front of a filename replaces the public
            key data in the SRK table with a corresponding hash digest
  -f, --fuse_format <format>:
        Optional, Data format of the SRK efuse binary file. The
        format may be selected by setting <format> to either:
          - 0: 8 fuses per word, ex: 00 00 00 0a 00 00 00 01 ...
          - 1 (default): 32 fuses per word, ex: 0a 01 ff 8e
  -l, --license:
        Optional, displays program license information. No additional
        arguments are required.
```

Command line arguments that specify a file or directory can contain spaces if they are quoted. File names with leading and trailing spaces are not supported.

If an error occurs during the operation of `srktool`, an error message will be printed to the standard output stream and the executable will exit with a non-zero status.

Exit Status:

0   if the executable succeeded, or

>0   otherwise.

N

Using the % prefix in the -c option does not change the SRL fuse pattern generated, but does reduce the overall size of the SRK Table. However, an SRK prefixed with % cannot be selected in the Install SRK command using that SRK Table.

Cautions:

None.

Pre Conditions/Assumptions:

None.

Post Conditions:

None.

Examples:

To generate an SRK table and corresponding fuse pattern from 3 certificates
          using PEM encoded certificate files

Freescale Semiconductor

using full key for first two certificates and hash digest for the third

using the default 32 fuse bits per word for the efuse file

```
srktool --hab_ver 4 --table table.bin  --efuses fuses.bin \
            --digest sha256 \
            --certs srk1_crt.pem,srk2_crt.pem,%srk3_crt.pem
```

To generate an alternative SRK Table with the same fuse pattern as in example 1 and with SRK3 selectable:

```
srktool --hab_ver 4 --table table.bin  --efuses fuses.bin \
            --digest sha256 \
            --certs %srk1_crt.pem,%srk2_crt.pem,srk3_crt.pem
```

To generate an SRK table and corresponding fuse pattern from 2 certificates

using DER encoded certificate files

using the optional 8 fuse bits per word for the efuse file

```
srktool --hab_ver 4 --table table.bin  --efuses fuses.bin \
            --digest sha256 \
            --certs srk1_crt.der,srk2_crt.der\
              --fuse_format 1
```

4.3   X5092WTLS Tool

The x5092wtls tool is used to convert certificates in X509 format to WTLS required for HAB3.

Usage:

```
x5092wtls --cert <certfile> --key <keyfile>
          --wtls <wtlsfile> [--passin <password file>] [--license]
```

Description:

```
-c, --cert <certfile>:
    X509 certificate file. May be either a PEM or DER encoded file
-k, --key <keyfile>:
    PKCS #8 private key used to sign the input X.509 certificate. May be
    either a PEM or DER encoded file
```

```
-w, --wtls <wtlsfile>:

      Output WTLS file in binary format

-p, --passin <passfile>:

      Optional password file. Max. supported password length is 20 characters

-l, --license:

      Optional, displays program license information. No additional

      arguments are required.
```

Command line arguments that specify a file or directory can contain spaces if they are quoted. File names with leading and trailing spaces are not supported.

PEM files must have a .pem extension. All other extensions are considered to be binary DER encoded files.

If an error occurs during the operation of `x5092wtls`, an error message will be printed to the standard output stream and the executable will exit with a non-zero status.

Exit Status:

0   if the executable succeeded, or

>0   otherwise.

Cautions:

None.

Pre Conditions/Assumptions:

None.

Post Conditions:

None.

Examples:

To generate a WTLS certificate (wtls.der) from a given x509 certificate (x509_cert.pem) by the given key (key.pem):

          x5092wtls.exe --cert x509_cert.pem --key key.pem --wtls wtls.der

To generate a WTLS certificate (wtls.der) from a given x509 certificate (x509_cert.pem) by the given encrypted key (key.pem):

          x5092wtls.exe -c x509_cert.pem -k key.pem -w wtls.der -p key_pass.txt

          where key_pass.txt is a file containing the password for the encrypted key.

5 CSF Description Language

This section describes the CSF description language. A CSF description file is written in the CSF description language, which is parsed and processed by the CST application and generates a CSF binary file containing the CSF commands, HAB certificates, and signatures, which are interpreted by the HAB ROM component on the end product device.

5.1 Overview

The following are the general properties of CSF description files:

- The CSF description file is a text file containing statements, one per line.
- A backslash character '\' at the end of a line (ignoring white space or comments) continues the statement to the next line.
- Blank lines are ignored.
- Comments beginning with the # character on any line are ignored.
- Multiple white space characters are equivalent to a single space. Except where noted, keywords and parameters are separated by white space. White space at the beginning or end of a line is ignored.
- Except for file names, all keywords and parameters are case-insensitive.
- All certificate file parameters are relative to current folder from where CST application is being executed.
- All byte parameters are specified as integers in the range 0...255. They can be specified in hexadecimal or decimal.
- All parameters that specify a file name must be double quoted. A quoted file name can contain spaces. The following file names are not supported:
  - File name with leading or trailing spaces.
  - File name that contains a double quote (") as part of the file name.
- Ordering of commands within the CSF description is significant only to the following extent:
  - The Header command must precede any other command.
  - The Install SRK command must precede the Install CSFK command.
  - The Install CSFK must precede the Authenticate CSF command.
  - Install SRK, Install CSFK and Authenticate CSF commands must appear exactly once in a CSF description file.
  - A verification index in an Authenticate Data command must appear as the target index in a previous Install Key command.
  - Commands in the binary CSF follow the order in which they appear in the CSF description.

5.2 CSF Commands

This section describes each CSF command in detail.

5.2.1 Header

The Header command contains data used in the CSF header as well as default values used by the CST for other commands throughout the remaining CSF.

There must be exactly one Header command and it must appear first in the CSF. Table 3 below lists the Header command arguments.

Table 3. Header arguments

Argument name	Description	Valid values	HAB3	HAB4
Version	Version of HAB	3, 3.5, 4.x, where x=0,1,...	M	M
Security Configuration	Fused security configuration	Engineering, Production	M	X
UID	Value expected in UID fuses	Generic (matches any value) U0, U1,... Un where each Ui=0..255 and n<255	M	X
CODE	Value expected in "customer code" fuses	0..255	M	X
Hash Algorithm	Default hash algorithm	SHA1, SHA256, SHA512	O	O
Engine	Default engine.	ANY, SAHARA, RTIC, DCP, CAAM and SW	O	O
Engine Configuration	Default engine configuration	See	O	O
Certificate Format	Default certificate format)	WTLS, X509	O	O
Signature Format	Default signature format	PKCS1, CMS	O	O

M = mandatory, O = optional, D = use default from Header if absent and X = not present

Table 4 below lists valid engine configuration values for each engine type

Engine name	Valid engine configuration values
ANY	0
SAHARA	One or more of these, separated by '|': 0 IN SWAP8 IN SWAP16 DSC BE816

Freescale Semiconductor

	DSC BE832
DCP	One or more of these, separated by '|': 0 IN SWAP8 IN SWAP32 OUT SWAP8 OUT SWAP32
CAAM	One or more of these, separated by '|': 0 IN SWAP8 IN SWAP16 OUT SWAP8 OUT SWAP16 DSC SWAP8 DSC SWAP16
RTIC	One or more of these, separated by '|': 0 IN SWAP8 IN SWAP16 OUT SWAP8 KEEP
SW	0

5.2.1.1 Header Examples

```
[Header]
    Version = 3.5 # HAB3 example
    Security Configuration = Engineering
    UID = Generic
    Hash Algorithm = SHA256
    Certificate Format = WTLS
    Signature Format = PKCS1


[Header]
    Version = 4.1  # HAB4 example
    Hash Algorithm = SHA256
    Engine = Any
    Engine Configuration = 0
    Certificate Format = X509
    Signature Format = CMS
```

5.2.2 Install SRK

The Install SRK command authenticates and installs the root public key for use in subsequent Install CSFK or Install Key (HAB4 only) commands.

HAB authenticates the SRK using the SRK hash (SRK_HASH) fuses. HAB4 allows revocation of individual keys within the SRK table using the SRK revocation (SRK_REVOKE) fuses.

HAB installs the SRK in slot 0 of its internal public key store.

There must be exactly one Install SRK command in a CSF, and it must occur before the Install CSFK command. Table 5 lists the Install SRK command arguments.

Argument name	Description	Valid values	HAB3	HAB4
File	SRK certificate (HAB3) SRK table (HAB4)	Valid file path	M	M
Source Index	SRK index within SRK table. Installation fails if the SRK revocation fuse with this index is burned.	0..3	X	M
Hash Algorithm	SRK table hash algorithm	SHA1, SHA256 and SHA512	X	D

Table 5. Install SRK arguments

M = mandatory, O = optional, D = use default from Header if absent and X = not present

5.2.2.1 Install SRK Examples

```
[Install SRK] # HAB3 example
    File = "../crts/srk.der"



[Install SRK] # HAB4 example
    File = "../crts/srk_table.bin"
    Source Index = 0
    Hash Algorithm = sha256
```

5.2.3 Install CSFK

The Install CSFK command authenticates and installs a public key for use in subsequent Install Key (HAB3 only) or Authenticate CSF commands.

HAB authenticates the CSFK from the CSFK certificate using the SRK.

HAB installs the CSFK in slot 1 of its internal public key store.

There must be exactly one Install CSFK command in a CSF, and it must occur before the Authenticate CSF command. Table 6 lists the Install CSFK command arguments.

Table 6. Install CSFK arguments

Argument name	Description	Valid values	HAB3	HAB4
File	CSFK certificate	Valid file path	M	M
Certificate Format	CSFK certificate format	WTLS, X509	D	D

M = mandatory, O = optional, D = use default from Header if absent and X = not present

5.2.3.1 Install CSFK Examples

```
[Install CSFK] # HAB3 example
    File = "../crts/csf.der"
    Certificate Format = WTLS


[Install CSFK] # HAB4 example
    File = "../crts/csf.pem"
    Certificate Format = X509
```

5.2.4 Install NOCAK (HAB4 only)

The Install NOCAK command authenticates and installs a public key for use with the fast authentication mechanism (HAB 4.1.2 and later only). With this mechanism, one key is used for all signatures.

HAB installs the no-CA key in slot 1 of its internal public key store.

There must be exactly one Install NOCAK command in a CSF, and it must occur before the Authenticate CSF command and there must be no Install Key commands. Table 7 lists the install NOCAK command arguments.

Table 7. Install CSFK arguments

Argument name	Description	Valid values	HAB3	HAB4
File	CSFK certificate	Valid file path	M	M
Certificate Format	CSFK certificate format	WTLS, X509	D	D

M = mandatory, O = optional, D = use default from Header if absent and X = not present

5.2.4.1 Install NOCAK Examples

```
[Install NOCAK] # HAB4 example
    File = "../crts/csf.pem"
    Certificate Format = X509
```

5.2.5 Authenticate CSF

The Authenticate CSF command authenticates the CSF from which it is executed.

HAB authenticates the CSF using the CSFK public key, from a digital signature generated automatically by the CST.

There must be exactly one Authenticate CSF command in a CSF file, and it must occur after the Install CSFK command. Most other CSF commands are allowed only after the Authenticate CSF command. Table 8 lists the Authenticate CSF command arguments.

Authenticate CSF arguments

Argument name	Description	Valid values	HAB3	HAB4

Engine	CSF signature hash engine	ANY, SAHARA, RTIC, DCP, CAAM and SW	X	D
Engine Configuration	Configuration flags for the hash engine. Note that the hash is computed over an internal RAM copy of the CSF.	see	X	D
Signature Format	CSF signature format	PKCS1, CMS	D	D

M = mandatory, O = optional, D = use default from Header if absent and X = not present

5.2.5.1 Authenticate CSF Examples

```
[Authenticate CSF] # HAB3/HAB4 example using all default arguments


[Authenticate CSF] # HAB4 example
    Engine = DCP
    Engine Configuration = 0
    Signature Format = CMS
```

5.2.6 Install Key

The Install Key command authenticates and installs a public key for use in subsequent Install Key or Authenticate Data commands.

HAB authenticates a public key from a public key certificate using a previously installed verifying key and a hash of the public key certificate.

HAB installs the authenticated public key in an internal public key store with a zero-based array of key slots.

The CSF author is responsible for managing the key slots in the internal public key store to establish the desired public 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.

Multiple Install Key commands are allowed in a CSF. An Install Key command must precede any command which uses the installed key, and all Install Key commands must come after the Authenticate CSF command. Table 9 lists the Install Key command arguments.

Install Key arguments

Argument name	Description	Valid values	HAB3	HAB4
File	Public key certificate	Valid file path	M	M
Verification Index	Verification key index in key store.	1, ..., 4 (HAB3) 0, 2, ..., 5 (HAB4) SRK (HAB3), CSFK (HAB4) not supported	M	M

Freescale Semiconductor

Target Index	Target key index in key store.	2, ..., 4 (HAB3) 2, ..., 5 (HAB4) SRK, CSFK slots reserved.	M	M
Certificate Format	Public key certificate format.	WTLS, X509	D	D
Hash Algorithm	Hash algorithm for certificate binding. If present, a hash of the certificate specified in the File argument is included in the command to prevent installation from other sharing the same verification key.	SHA1, SHA256 and SHA512	D	O

M = mandatory, O = optional, D = use default from Header if absent and X = not present

5.2.6.1 Install Key Examples

```
[Install Key] # HAB3 example
    Key = "../crts/imgk.der"
    Verification Index = 1
    Target Index = 2
    Certificate Format = WTLS
    Hash Algorithm = SHA1


[Install Key] # HAB4 example
    Key = "../crts/imgk.pem"
    Verification Index = 0
    Target Index = 2
    Certificate Format = X509
```

5.2.7 Authenticate Data

The Authenticate Data command verifies the authenticity of pre-loaded data in memory. The data may include executable SW instructions, and may be spread across multiple non-contiguous address ranges drawn from multiple object files.

HAB authenticates the pre-loaded data using a previously installed public key from a digital signature generated automatically by the CST. For HAB3, authentication may be restricted to a single chip and security configuration.

The security configuration is taken from the Header command. Table 10 lists the Authenticate Data command arguments.

Authenticate Data arguments

Argument name	Description	Valid values	HAB3	HAB4

Blocks	List of one or more data blocks. Each block is specified by four parameters: source file (must be **binary**), starting load address in memory starting offset within the source file length (in bytes)	*file address offset length* with *file*: valid pathname *address*: 32-bit unsigned integer *offset*: 0, ..., size of *file* *length*: 0, ..., size of *file - offset* Block parameters separated by spaces. Multiple blocks separated by commas.	M	M
Verification Index	Verification key index in key store.	2, ..., 4 (HAB3) 2, ..., 5 (HAB4) SRK, CSFK not supported NOTE: For HAB4 Fast Authentication, this must be 0	M	M
Engine	Data signature hash engine.	ANY, SAHARA, RTIC, DCP, CAAM and SW	D	D
Engine Configuration	Configuration flags for the engine.	See	D	D
Signature Format	Data signature format	PKCS1, CMS	D	D
Binding	64-bit unique ID (UID) for binding. If present, authentication succeeds only if the UID fuse value matches this argument, and the TYPE fuse value matches the Security Configuration argument from the Header command.	*U0, U1, ... U7* with *Ui*: 0, ..., 255. UID bytes separated by commas.	O	X

M = mandatory, O = optional, D = use default from Header if absent and X = not present

5.2.7.1   Authenticate Data Examples

```
[Authenticate Data] # HAB3 example
    Blocks = 0xf8000000 0x0 0x10000 "flash.bin", \
             0xf8010000 0x0 0x1000 "xyz.bin"
    Verification Index = 2


[Authenticate Data] # HAB3 example
    Blocks = 0xf8000000 0x0 0x10000 "flash.bin", \
             0xf8010000 0x0 0x1000 "xyz.bin", \
             0xf8012000 0x2000 0x4000 "xyz.bin", \
             0xf8018000 0x8000 0x1000 "xyz.bin"
    Verification Index = 3
    Engine = SAHARA
    Binding = 0x01, 0x23, 0x45, 0x67, 0x89, 0xab, 0xcd, 0xef


[Authenticate Data] # HAB4 example
    Blocks = 0xf8000000 0x0 0x10000 "flash.bin" , \
             0xf801000 0x0 0x1000 "xyz.bin"
    Verification Index = 2
```

```
Engine = DCP
Engine Configuration = 0
Signature Format = CMS
```

5.2.8　Install Secret Key

This command is applicable from HAB 4.1 onwards and only on processors which include
CAAM and SNVS. Each instance of this command generates a CSF command to install a secret
key in CAAM's secret key store. A key blob as described in Section 1.1.2 is unwrapped using a
master key encryption key (KEK) supplied by SNVS. A random key is generated and protected
by the CST back end and encrypted using a public key passed with --cert command line option to
CST and saved in a file under the name passed in the Key argument. This file is intended for
later use by provisioning software to create the blob. Table 11 lists the Install Secret Key
command arguments. Each execution of the CST will generate a different secret key, overwriting
any previous secret key in the given file.

Install Secret Key arguments

Argument name	Description	Valid values	HAB (> 4.0)
Key	Output filename for CST to create the cms encrypted data encryption key	Valid pathname	M
Key length	Key length in bits	128, 192 and 256	M
Verification Index	Master KEK index	0 or 1 : OTPMK from fuses 2 : ZMK from SNVS 3 : CMK from SNVS	D
Target Index	Target secret key store index	0, 1, 2 or 3 of secret key store	M
Blob Address	Absolute memory address where blob will be loaded	Internal or external DDR address	M

M = mandatory, O = optional, D = use default from Header if absent and X = not present

5.2.8.1　Install Secret Key Examples
```
[Install Secret Key] # Example using OPTMK (Default)
    Key = "data_encryption.key"
    Target Index = 0 /* Secret key store index */
    Blob Address = 0x0090a000 /* internal ram address */


[Install Secret Key] # Example using ZMK
    Key = "data_encryption.key"
    Verification Index = 2 /* ZMK */
    Target Index = 0 /* Secret key store index */
    Blob Address = 0x0090a000 /* internal ram address */
```

5.2.9   Decrypt Data

This command is applicable from HAB4.1 onwards. Each instance generates a CSF command to decrypt and authenticate a list of code/data blocks using secret key stored in the secret key store. CST will generate a corresponding AUT_DAT command. CST will encrypt the data blocks in-place in the given files using a secret key and generate MAC data which is appended to the CSF. Table 12 lists the Decrypt Data command arguments. The secret key index must have been the target key index in a preceding Install Secret Key command. ▦

Decrypt Data arguments

Argument name	Description	Valid values	HAB (> 4.0)
Blocks	List of one or more data blocks. Each block is specified by four parameters: source file (must be **binary**), starting load address in memory starting offset within the source file length (in bytes)	*file address offset length* with *file*: valid pathname *address*: 32-bit unsigned integer *offset*: 0, ..., size of *file* *length*: 0, ..., size of *file - offset* Block parameters separated by spaces. Multiple blocks separated by commas.	M
Verification Index	Secret key index in Secret key store	0, 1, 2 or 3 from secret key store	M
Engine	MAC engine	CAAM (Default)	D
Engine Configuration	Configuration flags for the engine.	See Default from header command	D
MAC Bytes	Size of MAC in bytes.	Even value between 4 and 16 (Default 16)	D

M = mandatory, O = optional, D = use default from Header if absent and X = not present

5.2.9.1   Decrypt Data Examples

```
[Decrypt Data]
    Blocks = 0xf8000000 0x0 0x10000 "flash.bin", \
    0xf8010000 0x0 0x1000 "xyz.bin"
    Verification Index = 0


[Decrypt Data]
Blocks = 0xf8000000 0x0 0x10000 "flash.bin" , \
            0xf8010000 0x0 0x1000 "xyz.bin", \
            0xf8012000 0x2000 0x4000 "xyz.bin", \
            0xf8018000 0x8000 0x1000 "xyz.bin"
    Verification Index = 3
    Engine = CAAM
    Engine Configuration = 0
```

5.2.10 Write Data

The Write Data command writes a list of given data values to a corresponding list of target addresses. The data values may be 1, 2 or 4 bytes wide. HAB3 supports only a single write operations per Write Data command.

Multiple Write Data commands may appear in a CSF after the Authenticate CSF command. For HAB4, Write Data commands may also appear between the Header and Authenticate CSF commands, but the allowed target locations are limited to the same address ranges as allowed for DCD commands. Table 13 lists the Write Data command arguments.

Write Data arguments

Argument name	Description	Valid values	HAB3	HAB4
Width	Width of the target locations in bytes. All target locations in a single command must have the same width.	1, 2 or 4	M	M
Address Data	List of one or more pairs giving target addresses and corresponding data values.	*address data* with *address* : 32-bit unsigned integer *data*: 8, 16 or 32-bit unsigned integer according to Width argument. Address and data are separated by spaces. Multiple pairs are separated by commas.	M	M

M = mandatory, O = optional, D = use default from Header if absent and X = not present

5.2.10.1 Write Data Examples

```
[Write Data] # HAB3 example
    Width = 2
    Address Data = 0xf8006000 0xf000


[Write Data] # HAB4 example
    Width = 4
    Address Data = 0xf8006000 0x0000f000, \
                   0xf8007000 0x0f0f0f0f


[Write Data] # HAB4 example
    Width = 1
    Address Data = 0xf8006000 0x00, \
                   0xf8007000 0x0f, \
                   0xf8008000 0xff
```

5.2.11　Clear Mask

The Clear Mask command clears a list of given bit masks at a corresponding list of target addresses. The bit masks may be 1, 2 or 4 bytes wide. HAB3 does not support the Clear Mask command.

Multiple Clear Mask commands may appear anywhere in a CSF after the Header command. Clear Mask commands may appear between the Header and Authenticate CSF commands, but the allowed target locations are limited to the same address ranges as allowed for DCD commands. Table 14 lists the Clear Mask command arguments.

Clear Mask arguments

Argument name	Description	Valid values	HAB3	HAB4
Width	Width of the target locations in bytes. All target locations in a single command must have the same width.	1, 2 or 4	X	M
Address Mask	List of one or more pairs giving target addresses and corresponding bit masks.	*address mask* with *address* : 32-bit unsigned integer *mask*: 8, 16 or 32-bit unsigned integer according to Width argument. Address and mask are separated by spaces. Multiple pairs are separated by commas.	X	M

M = mandatory, O = optional, D = use default from Header if absent and X = not present

5.2.11.1　Clear Mask Examples

```
[Clear Mask]
    Width = 2
    Address Mask = 0xf8006000 0xf000


[Clear Mask]
    Width = 4
    Address Mask = 0xf8006000 0x0000f000, \
                   0xf8007000 0x0f0f0f0f


[Clear Mask]
    Width = 1
    Address Mask = 0xf8006000 0x00, \
                   0xf8007000 0x0f, \
                   0xf8008000 0xff
```

5.2.12 Set Mask

The Set Mask command sets a list of given bit masks at a corresponding list of target addresses. The bit masks may be 1, 2 or 4 bytes wide. HAB3 does not support the Set Mask command.

Multiple Set Mask commands may appear anywhere in a CSF after the Header command. Set Mask commands may appear between the Header and Authenticate CSF commands, but the allowed target locations are limited to the same address ranges as allowed for DCD commands. Table 15 lists the Set Mask command arguments.

Set Mask arguments

Argument name	Description	Valid values	HAB3	HAB4
Width	Width of the target locations in bytes. All target locations in a single command must have the same width.	1, 2 or 4	X	M
Address Mask	List of one or more pairs giving target addresses and corresponding bit masks.	*address mask* with *address* : 32-bit unsigned integer *mask*: 8, 16 or 32-bit unsigned integer according to Width argument. Address and mask are separated by spaces. Multiple pairs are separated by commas.	X	M

M = mandatory, O = optional, D = use default from Header if absent and X = not present

5.2.12.1 Set Mask Examples

```
[Set Mask]
    Width = 2
    Address Mask = 0xf8006000 0xf000


[Set Mask]
    Width = 4
    Address Mask = 0xf8006000 0x0000f000, \
                   0xf8007000 0x0f0f0f0f


[Set Mask]
    Width = 1
    Address Mask = 0xf8006000 0x00, \
                   0xf8007000 0x0f, \
                   0xf8008000 0xff
```

5.2.13 Check All Clear

The Check All Clear command polls a given source address until all bits in a given bit mask are clear or a maximum poll count is reached. The bit mask may be 1, 2 or 4 bytes wide. HAB3 does not support the Check All Clear command.

Multiple Check All Clear commands may appear anywhere in a CSF after the Header command. Table 16 lists the Check All Clear command arguments.

Check All Clear arguments

Argument name	Description	Valid values	HAB3	HAB4
Width	Width of the source location in bytes.	1, 2 or 4	X	M
Count	Maximum poll count before returning error. If zero, behaves as NOP. If absent, the command polls indefinitely.	32-bit unsigned integer	X	O
Address Mask	Single pair giving source address and corresponding bit mask.	*address mask* with *address* : 32-bit unsigned integer *mask*: 8, 16 or 32-bit unsigned integer according to Width argument. Address and mask are separated by spaces.	X	M

M = mandatory, O = optional, D = use default from Header if absent and X = not present

5.2.13.1 Check All Clear Examples

```
[Check All Clear]
    Width = 2
    Address Mask = 0xf8006000 0xf000


[Check All Clear]
    Width = 4
    Count = 100
    Address Mask = 0xf8006000 0x0000f000


[Check All Clear]
    Width = 1
    Count = 100
    Address Mask = 0xf8006000 0x01
```

5.2.14   Check All Set

The Check All Set command polls a given source address until all bits in a given bit mask are set or a maximum poll count is reached. The bit mask may be 1, 2 or 4 bytes wide. HAB3 does not support the Check All Set command.

Multiple Check All Set commands may appear anywhere in a CSF after the Header command. Table 17 lists the Check All Set command arguments.

Check All Set arguments

Argument name	Description	Valid values	HAB3	HAB4
Width	Width of the source location in bytes.	1, 2 or 4	X	M
Count	Maximum poll count before returning error. If zero, behaves as NOP. If absent, the command polls indefinitely.	32-bit unsigned integer	X	O
Address Mask	Single pair giving source address and corresponding bit mask.	*address mask* with *address* : 32-bit unsigned integer *mask*: 8, 16 or 32-bit unsigned integer according to Width argument. Address and mask are separated by spaces.	X	M

M = mandatory, O = optional, D = use default from Header if absent and X = not present

5.2.14.1   Check All Set Examples

```
[Check All Set]
    Width = 2
    Address Mask = 0xf8006000 0xf000

[Check All Set]
    Width = 4
    Count = 100
    Address Mask = 0xf8006000 0x0f0f0f0f


[Check All Set]
    Width = 1
    Count = 100
    Address Mask = 0xf8006000 0xff
```

5.2.15   Check Any Clear

The Check Any Clear command polls a given source address until any bits in a given bit mask are clear or a maximum poll count is reached. The bit mask may be 1, 2 or 4 bytes wide. HAB3 does not support the Check Any Clear command.

Multiple Check Any Clear commands may appear anywhere in a CSF after the Header command. Table 18 lists the Check Any Clear command arguments.

Check Any Clear arguments

name	description	Valid values	HAB3	HAB4
Width	Width of the source location in bytes.	1, 2 or 4	X	M
Count	Maximum poll count before returning error. If zero, behaves as NOP. If absent, the command polls indefinitely.	32-bit unsigned integer	X	O
Address Mask	Single pair giving source address and corresponding bit mask.	*address mask* with *address* : 32-bit unsigned integer *mask*: 8, 16 or 32-bit unsigned integer according to Width argument. Address and mask are separated by spaces.	X	M

M = mandatory, O = optional, D = use default from Header if absent and X = not present

5.2.15.1   Check Any Clear Examples

```
[Check Any Clear]
    Width = 2
    Address Mask = 0xf8006000 0xf000

[Check Any Clear]
    Width = 4
    Count = 100
    Address Mask = 0xf8007000 0x0f0f0f0f
```

5.2.16   Check Any Set

The Check Any Set command polls a given source address until any bits in a given bit mask are set or a maximum poll count is reached. The bit mask may be 1, 2 or 4 bytes wide. HAB3 does not support the Check Any Set command.

Multiple Check Any Set commands may appear anywhere in a CSF after the Header command. Table 19 lists the Check Any Set command arguments.

Check Any Set arguments

Argument name	Description	Valid values	HAB3	HAB4
Width	Width of the source location in bytes.	1, 2 or 4	X	M

Count	Maximum poll count before returning error. If zero, behaves as NOP. If absent, the command polls indefinitely.	32-bit unsigned integer	X	O
Address Mask	Single pair giving source address and corresponding bit mask.	*address mask* with *address* : 32-bit unsigned integer *mask*: 8, 16 or 32-bit unsigned integer according to Width argument. Address and mask are separated by spaces.	X	M

M = mandatory, O = optional, D = use default from Header if absent and X = not present

5.2.16.1 Check Any Set Examples

```
[Check Any Set]
    Width = 2
    Address Mask = 0xf8006000 0xf000

[Check Any Set]
    Width = 4
    Count = 100
    Address Mask = 0xf8007000 0x0f0f0f0f
```

5.2.17 NOP

The NOP command has no effect.

Multiple NOP commands may appear in a CSF after the Authenticate CSF command. For HAB4, NOP commands may also appear between the Header and Authenticate CSF commands.

The NOP command has no arguments.

5.2.17.1 NOP Example

```
[NOP]
```

5.2.18 Set MID

The Set MID command selects a range of fuse locations to use as a manufacturing identifier (MID). MID values are bound with Authenticate Data signatures when verified using keys with the MID binding flag in the Install Key command. HAB3 does not support the Set MID command.

Multiple Set MID commands may appear anywhere in a CSF after the Header command. Authenticate Data commands use the MID value selected by the most recent Set MID command. Table 20 lists the Set MID command arguments.

Set MID arguments

Argument name	Description	Valid values	HAB3	HAB4
Bank	Fuse bank containing MID	0, ..., 255	X	M
Row	Starting row number of MID within bank	0, ..., 255	X	M
Fuse	Starting fuse of MID within row	0, ..., 255	X	M
Bits	Number of bits for MID	0, ..., 255	X	M

M = mandatory, O = optional, D = use default from Header if absent and X = not present

5.2.18.1 Set MID Example

```
[Set MID]
    Bank = 4
    Row = 0
    Fuse = 7
    Bits = 64
```

5.2.19 Set Engine

The Set Engine command selects the default engine and engine configuration for a given algorithm. HAB3 does not support the Set Engine command.

Some CSF commands allow the CSF author to select the engine used for an algorithm by specifying an argument other than ANY. However, if the engine argument is ANY, then HAB selects the engine to use based on internal criteria. The Set Engine command overrides the HAB internal criteria and selects the engine and configuration to use when ANY is specified.

Some algorithm types do not have an associated engine argument in the CSF commands (e.g. the signature algorithm in Authenticate Data commands). By default, HAB selects the engine to use for such algorithms based on internal criteria. The Set Engine command overrides the HAB internal criteria in such cases as well.

Multiple Set Engine commands may appear anywhere in a CSF after the Header command. Subsequent commands use the engine selected by the most recent Set Engine command. Table 21 lists the Set Engine command arguments.

Set Engine arguments

Argument name	Description	Valid values	HAB3	HAB4
Hash Algorithm	Hash algorithm	SHA1, SHA256 and SHA512	X	M
Engine	Engine Use ANY to restore the HAB internal criteria.	ANY, SAHARA, RTIC, DCP, CAAM and SW	X	M
Engine Configuration	Configuration flags for the engine.	See Table 4	X	O

M = mandatory, O = optional, D = use default from Header if absent and X = not present

5.2.19.1   Set Engine Example

```
[Set Engine]
    Hash Algorithm = SHA256
    Engine = DCP
    Engine Configuration = 0
```

5.2.20   Init

The Init command initializes specified engine features when exiting the internal boot ROM. HAB3 does not support the Init command.

Multiple Init commands may appear after the Authenticate CSF command. A feature will be initialized if specified in one or more Init commands. Table 22 lists the Init command arguments.

Init arguments

Argument name	Description	Valid values	HAB3	HAB4
Engine	Engine to initialize	SRTC	X	M
Features	Comma-separated list of features to initialize	See	X	O

M = mandatory, O = optional, D = use default from Header if absent and X = not present

5.2.20.1   Init Example

```
[Init]
    Engine = SRTC
```

5.2.21   Unlock

The Unlock command prevents specified engine features from being locked when exiting the internal boot ROM. HAB3 does not support the Unlock command.

Multiple Unlock commands may appear after the Authenticate CSF command. A feature will be unlocked if specified in one or more Unlock commands. Table 23 lists the Unlock command arguments.

T   Unlock arguments

Argument name	Description	Valid values	HAB3	HAB4
Engine	Engine to unlock	SRTC, CAAM, SNVS and OCOTP	X	M
Features	Comma-separated list of features to unlock	See Table 24	X	O

UID	Device specific 64-bit UID Required to unlock certain features, must be absent for others (see Table 24).	*U0,U1,... U7* with *Ui*=0..255 UID bytes separated by commas	X	M/X

M = mandatory, O = optional, D = use default from Header if absent and X = not present

Table 24 shows valid Features values available in Init/Unlock commands for each Engine argument.

T Valid feature values

Engine	Features	UID	Init/Unlock command effect
SRTC		X	The Init command clears any failure status flags and clears the low-power counters and timers if the SRTC is in Init state. The Unlock command prevents the secure timer and monotonic counter being locked if the SRTC is in Valid state
CAAM	MID	X	Leaves Job Ring and DECO master ID registers unlocked.
	RNG	X	Leaves RNG state handle 0 uninstantiated, does not generate descriptor keys, does not set the AES DPA mask, and does not block state handle 0 test instantiation.
SNVS	LP SWR	X	Leaves LP SW reset unlocked.
	ZMK WRITE	X	Leaves Zeroisable Master Key write unlocked.
OCOTP	FIELD RETURN	M	Leave Field Return activation unlocked.
	SRK REVOKE	X	Leave SRK revocation unlocked.
	SCS	M	Leave SCS register unlocked.
	JTAG	M	Unlock JTAG using SCS HAB_JDE bit.

M = mandatory, O = optional, D = use default from Header if absent and X = not present

5.2.21.1   Unlock Examples

```
[Unlock]
     Engine = SRTC

[Unlock]
     Engine = CAAM
     Features = RNG

[Unlock]
     Engine = OCOTP
     Features = JTAG, SRK REVOKE
     UID = 0x01, 0x23, 0x45, 0x67, 0x89, 0xab, 0xcd, 0xef
```

5.3 CSF Examples

This section provides some examples for HAB3 and HAB4 CSF.

5.3.1 HAB3 CSF Example

is an example of HAB3 CSF description. This example CSF description:
- Defines a version 3 CSF description.
- Uses a generic UID for a processor with the HAB_TYPE fuses burned to Engineering
- Covers three blocks of memory loaded from two different files with a single signature verification.

```
# Illustrative Command Sequence File Description (generic UID)

[Header]
    Version = 3.0
    Security Configuration = Engineering
    Hash Algorithm = sha256
    Engine = ANY
    Engine Configuration = 0
    Certificate Format = WTLS
    Signature Format = PKCS1
    UID = Generic
    Code = 0x0F

[Install SRK]
    File = "../crts/SRK1_sha256_2048_65537_v3_ca_crt.der   "

[Install CSFK]
    File = "../crts/CSF1_1_sha256_2048_65537_v3_ca_crt.der   "

[Authenticate CSF]

[Install Key]
    Verification index = 1
    Target index = 2
    File = "../crts/IMG1_1_sha256_2048_65537_v3_usr_crt.der   "

# whole line comment

[Authenticate Data]   # part line comment
    Verification index = 2
    Blocks = 0x80000000 0 0x00001000 "image.bin" \
             0x80004000 0x4000 0x00001000 "image.bin" \
              0x8000a000 0 0x00002000 "image2.bin"
```

Freescale Semiconductor

5.3.2   HAB3 Binding CSF Example

Figure 17 is an example of HAB3 CSF description. This example CSF description:

- Defines a version 3.5 CSF description.
- Uses a generic UID for the CSF of a processor with HAB_TYPE fuses set for Production.
- Uses a bound signature for the software load. The bound signature is specified by using the Binding argument in Authenticate Data command. This binds the signature to the device by including the UID in the signature process.
- Uses the RTIC hash engine and keeps the calculated hash value to allow activating RTIC run-time monitoring later in the boot flow.

```
# Illustrative Command Sequence File Description (generic UID)

[Header]
    Version = 3.5
    Security Configuration = Production
    Hash Algorithm = sha256
    Certificate Format = WTLS
    Signature Format = PKCS1
    UID = Generic
    Code = 0x0F

[Install SRK]
    File = "../crts/ SRK1_sha256_2048_65537_v3_ca_crt.der"

[Install CSFK]
    File = "../crts/ CSF1_1_sha256_2048_65537_v3_ca_crt.der"

[Authenticate CSF]

[Install Key]
    Verification index = 1
    Engine = RTIC
    Engine Configuration = KEEP
    Target index = 2
    File = "../crts/ IMG1_1_sha256_2048_65537_v3_usr_crt.der"

# whole line comment

[Authenticate Data]   # part line comment
    Verification index = 2
    Binding = 0x01 , 0x23, 0x45, 0x67, 0x89, 0xab, 0xcd, 0xef
    Blocks = 0x80000000 0 0x00001000 "image.bin" \
             0x80004000 0x4000 0x00001000 "image.bin" \
              0x8000a000 0 0x00002000 "image2.bin"
```

F   Example Production CSF Description File

5.3.3   HAB4 CSF Example

Figure 18 is an example of a HAB4 CSF description. This example CSF description:

- Defines a version 4 CSF description.
- Overrides default engine ANY with DCP in Authenticate Data command
- Lists three blocks from image for signing.

```
# Illustrative Command Sequence File Description

[Header]
    Version = 4.0
    Hash Algorithm = sha256
    Engine = ANY
    Engine Configuration = 0
    Certificate Format = X509
    Signature Format = CMS

[Install SRK]
    File = "../crts/TBL_1_sha256_tbl.bin"
    Source index = 0

[Install CSFK]
    File = "../crts/ CSF1_1_sha256_2048_65537_v3_usr_crt.pem"

[Authenticate CSF]

[Install Key]
    Verification index = 0
    Target index = 2
    File = "../crts/ IMG1_1_sha256_2048_65537_v3_usr_crt.pem"

# whole line comment

[Authenticate Data]   # part line comment
    Verification index = 2
    Engine = DCP
    Blocks = 0xf8009400 0x400 0x40 "MCUROM-OCRAM-ENG_img.bin", \
             0xf8009440 0x440 0x40 "MCUROM-OCRAM-ENG_img.bin", \
             0xf800a000 0x1000 0x8000 "MCUROM-OCRAM-ENG_img.bin"
```

F   **Example HAB4 CSF Description File**

5.3.4   HAB4 CSF Fast Authentication Example

Figure 19 is an example of a HAB4 CSF description for fast authentication. This example CSF description:

- Defines a version 4 CSF description.
- Tells HAB to use fast authentication mechanism
- Lists single block from image for signing

```
#Illustrative Command Sequence File Description
[Header]
    Version = 4.1
    Hash Algorithm = sha256
    Engine = ANY
    Engine Configuration = 0
    Certificate Format = X509
    Signature Format = CMS

[Install SRK]
    File = "../crts/TBL_1_sha256+tbl.bin"
    Source index = 0

[Install NOCAK]
    File = "../crts/SRK1_sha256_2048_65537_v3_usr_crt.pem"

[Authenticate CSF]
#whole line comment

[Authenticate Data]    # part line comment
    Verification index = 0
    Blocks = 0x877fb000 0x000 0x48000 "signed-uboot.bin"
```

Figure 19. Example HAB4 CSF Description File

5.3.5   HAB4 CSF Example for Encrypted Boot

Figure 20 is an example of a HAB version 4.1 CSF description demonstrating on how to use Install Secret Key and Decrypt Data commands. This example CSF description:

- Defines a version 4.1 CSF description.
- Necessary blocks from image for signing.
- Install Secret Key command
- Blocks for encryption by CST and decryption by ROM/HAB

```
# Illustrative Command Sequence File Description

[Header]
    Version = 4.1
     Hash Algorithm = SHA256
    Engine Configuration = 0
    Certificate Format = X509
    Signature Format = CMS
    Engine = CAAM
    Engine Configuration = 0

[Install SRK]
    File = "../crts/SRK_1_2_3_4_table.bin"
    Source index = 0

[Install CSFK]
    File = "../crts/CSF1_1_sha256_4096_65537_v3_usr_crt.der"

[Authenticate CSF]

[Install Key]
    Verification index = 0
    Target index = 2
    File = "../crts/IMG1_1_sha256_4096_65537_v3_usr_crt.der"

[Authenticate data]
    Verification index = 2
    Blocks = 0x27800400 0x400 800 "u-boot-mx6q-arm2_padded.bin"

[Install Secret Key]
    Verification index = 0
    Target index = 0
    Key = "dek.bin"
    Key Length = 128
    Blob address = 0x27831000

[Decrypt Data]
    Verification index = 0
    Mac Bytes = 16
    Blocks = 0x27800720 0x720 0x2E8E0 "u-boot-mx6q-arm2_padded.bin"
```

Figure 20. Example HAB4 CSF Description File with Decrypt Data Command

Appendix A Hab Library Version 3 Details

This section provides additional details on the Certificate and Signature formats supported by SoCs including the HAB library Version 3.

A.1   HAB CST Certificate Details

The HAB CST supports public key certificates in WTLS version 1 format using the RSA/SHA-256 signature format. This format is a proprietary extension to the RSA/SHA-1 algorithm specified in [5].

The RSA/SHA-1 signature format in [5] and the RSA/SHA-256 signature format both use PKCS#1 v1.5 Block Type 1 padding, but it is unclear whether the Signature scheme or Encryption scheme (which differs in the use of ASN.1 encoding) is intended. The HAB supports either choice (see Section A.2, "HAB Signature Verification Details"). Note that the Block Type 1 Encryption scheme is not supported in PKCS#1 v2.0.

A.2   HAB Signature Verification Details

HAB Version 3 signature verification is the authentication of digital signature(s) on a chain of data blocks, which can be optionally bound to other data. Signature Verification supports RSA PKCS#1 v1.5 Block Type 1 encoding with SHA-1, SHA-256, or MD5 as the hash algorithm. Either the Signature scheme or Encryption scheme is accepted as valid. MD5 support is disabled by default in HAB ROM code. For SoCs using SHA-256 as the default hash algorithm, both MD-5 and SHA-1 are disabled in the HAB ROM.

Signature lengths are equal to the modulus length of the verifying RSA key.

Signature verification in the HAB ROM component performs the following operations given a signature algorithm, chain of data blocks, an optional rehash chain, a signature, and a public key:

- Verifies that the signature algorithm is supported.
- Computes a hash on the data block chain with the hash algorithm appropriate to the signature algorithm.
- Computes an optional hash on the rehash chain if required as part of a bound signature.
- Recovers the padded hash digest from the signature using RSA computation and the public key.
- Verifies that the recovered padded hash conforms to the PKCS #1 v1.5 Block Type 1 padding format for either the Signature scheme or the Encryption scheme.
- Verifies that the hash within the recovered padded hash matches the computed hash (if the optional rehash chain is present, the hash on the rehash chain is used).

The hash algorithm to use is embedded in the padding recovered in the PKCS#1 v1.5 Signature scheme.

A.3   HAB3 SRK Structure Information

On SoCs supporting HAB3 the ROM expects an Application Header (also known as the Flash Header) to be included as part of the product code which is usually a bootloader. The SoC boot ROM expects the Application Header to be located at a fixed offset from the beginning of the product code image. Note that one of the fields in the Application Header is the pointer to the SRK data structure which is defined as:

```
typedef struct
{
 /* RSA public exponent */
 unsigned char rsa_exponent[HAB_MAX_EXPONENT_BYTES];
 unsigned char *rsa_modulus;            /* RSA modulus pointer */
 unsigned short int exponent_bytes;     /* Exponent size in bytes */
 unsigned short int modulus_bytes;      /* Modulus size in bytes */
 unsigned char init_flag;               /* Indicates if key initialized */
} hab_rsa_public_key;
```

This data structure is defined in the file hab_super_root.h included in the `/code/hab3/hdr` directory of the CST release. Further details on the Application Header are available in the System Boot chapter of the Reference Manual for the Freescale Processor that you are using.

A C file containing the static data for each SRK public key is generated by the hab3_pki_tree script (See Section 3.1.2, "Running the hab3_pki_tree script Example"). The data generated in the C files is the public key exponent and modulus data the HAB library on the SoC requires and follows the structure given above. To include the SRK public key data in your product code add the selected SRK C file in your product code build process, being sure to include hab_super_root.h to the include search path. Then be sure that the SRK field of the Application Header points to this data.

# Appendix B Replacing the CST Backend Implementation

The Freescale CST is a reference implementation and is sufficient for most use cases. However, there may be instances where a higher level of security is required. In such cases the level of protection required for HAB code signing keys needs to be much higher than what the CST reference implementation provides. To accommodate this Freescale has architected the CST in two parts a Front End and a Back End. The Front End contains all the Freescale proprietary operations of the CST with the Back End containing all standard cryptographic operations.

The CST Back End make use of standard cryptographic protocols allowing the use of common cryptographic library implementations. The CST makes use of the OpenSSL library [1] for performing basic cryptographic operations related to digital signature generation. For encrypted boot operations using AES in CCM mode the reference CST makes use of the AES library from Brian Gladman [7].

The remainder of this appendix describes how the reference Back End can be replaced with a different solution such as a Hardware Security Module (HSM). The replacement of the Back End with another solution is completely optional.

B.1 CST Architecture

Figure B-1 provides an overview of the CST components. To replace the Back End of the CST not only do you need to replace the OpenSSL commands, OpenSSL library and AES library but you must develop new Adaptation Layer software. The Adaptation Layer must be written to provide the equivalent functionality to the Front End as the reference Back End.
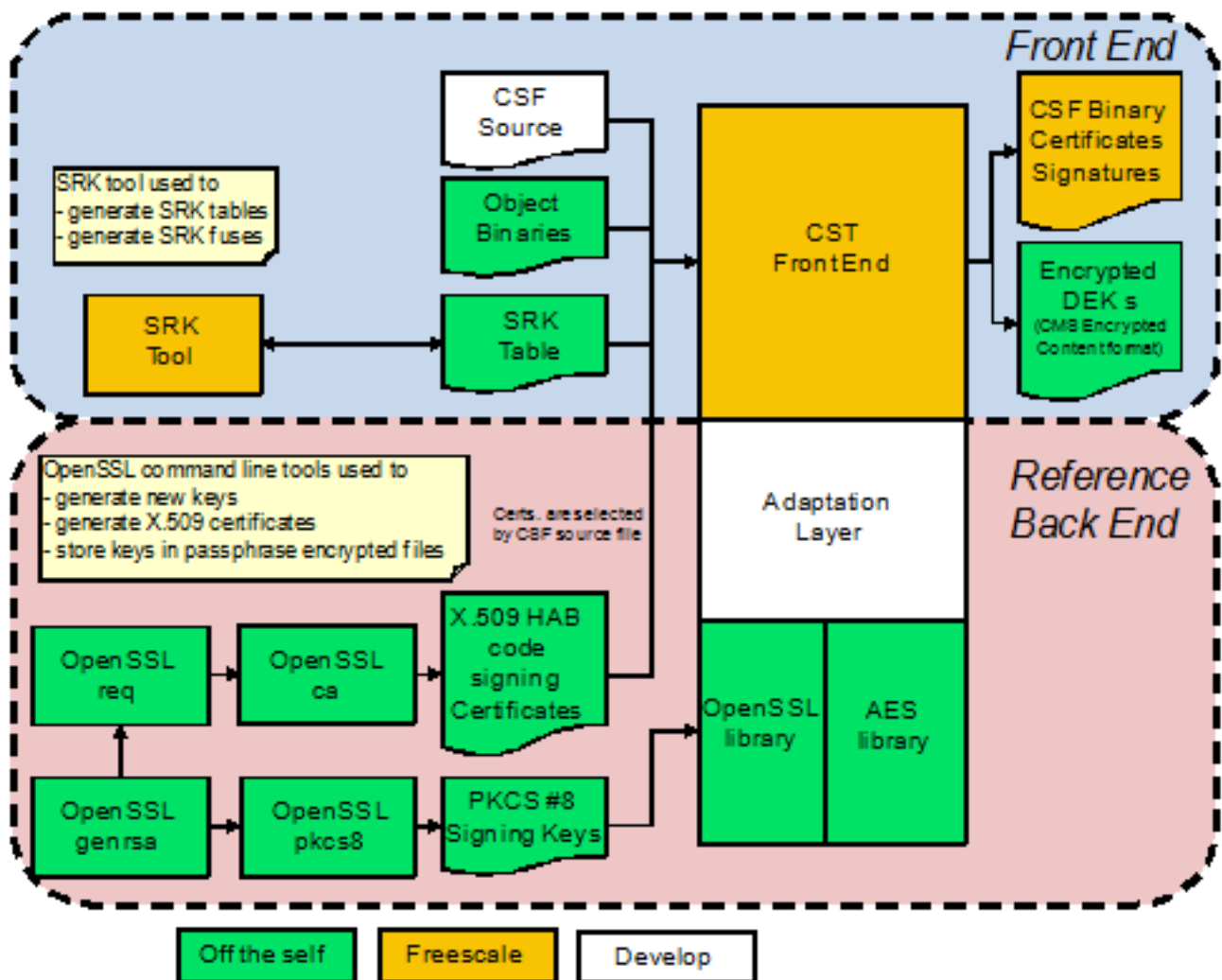
Overview of HAB Reference CST components

B.2 Back End Components

The section provides a brief overview of each of the components that make up the CST Back End. Note that the CST depends on OpenSSL in two ways. The first is by using the command line interface for generating code signing keys. The second is programmatically via the cryptographic library for generating signatures and making use of cryptographic algorithms. The reference implementation of the CST acts as both an CA through the use of OpenSSL command and an SA through the use of the OpenSSL library:

OpenSSL genrsa: OpenSSL command line option the CST key generation scripts use to create HAB code signing RSA key pairs.

OpenSSL pkcs8: OpenSSL command line option the CST key generation scripts use to protect the confidentiality of the private keys. Keys stored in PKCS#8 format encrypted with a pass phrase.

OpenSSL req: OpenSSL command line option used to generate a PKCS#10 certificate requests.

OpenSSL ca: OpenSSL command line option used to used to generate X.509 certificates for use with HAB code signing. The High Assurance Boot Version 4 Application Programming Interface Reference Manual [8] provides details on the X.509 certificate profile supported by HAB.

OpenSSL Library: Provides support for the PKCS standards and the underlying cryptographic algorithms. The exception is AES CCM mode which at the time of CST development OpenSSL did not include support.

AES Library: This library includes Brian Gladman's reference AES implementation [7] together with CCM mode [6] . This is to encrypt images for HAB encrypted boot.

CST Adaptation Layer: The adaptation layer provides the interface linking the CST Front End to the Back End services provided by the OpenSSL and AES libraries. To replace the Back End services provided open source libraries requires a new implementation of the adaptation layer based on the new Back End services. The implementation will be specific to the Back End Service implementation you have chosen. For example, if OpenSSL and the AES libraries are replaced with and HSM, then the adaptation layer must be re-written to interface with the HSM APIs.

B.3 Back End Replacement for Linux

In addition to the reference CST executables the CST is delivered with the following additional components:

- *./linux/lib/libfrontend.a*: This is the Front End library containing the Freescale proprietary features of the CST as shown in [Figure B-1]. This is a 32 bit library built with GCC 3.4.3 using the -m32 compiler option.

- *./code/back_end/hdr/adapt_layer.h*: Is the main header file for the adaptation layer and includes the documentation for the APIs used by the Front End. There are two APIs used by the Front End:

  — *gen_sig_data*: The CST Front End uses this API to generate HAB signatures.

  — *gen_auth_encrypted_data*: The CST Front End uses this API to generate encrypted data using AES-CCM.

- Any new Back End implementation must follow implement these two APIs in an equivalent adaptation layer corresponding to the new cryptographic services replacing OpenSSL and the AES libraries.

- For reference the source code and header files for the Freescale reference implementation are included in *./code/back_end.*

- To use a new method for public key generation, replace the key generation scripts with the new implementation.

These components will assist you in developing a new Adaptation Layer component.

N

Although the Back End may replace OpenSSL for code signing, SA and CA support, the CST Front End still makes use of OpenSSL for some non-code signing operations. This means that when linking library components together to generate a CST executable an OpenSSL library must also be included. Freescale recommends using OpenSSL 1.0.0d or later which is available at [1].

B.4 Front End References to Code Signing Keys

When replacing the CST Back End it is important to keep in mind that he CST Front End refers to code signing keys and certificates using file names. These are the key filenames that correspond to the RSA public key certificate and private key files generated by the CST. However, filenames may not be the native method for referencing keys in a new replacement Back End service. If this is the case, then the new Adaptation Layer is responsible for converting to and from file name references.

This is also true for Data Encryption Keys that the CST generates for encrypting images.