Freescale Semiconductor
Application Note

Document Number: AN4581
Rev. 0, 09/2012

Secure Boot on i.MX50, i.MX53, and i.MX 6 Series using HABv4

by Freescale Semiconductor, Inc.

This application note explains how to perform a secure boot on i.MX applications processors that support High Assurance Boot version 4 (HABv4). It provides steps to generate signed images. It also includes steps to configure the IC to run securely using tools provided freely by Freescale.

Contents

1 About this Document

1.1 Purpose

Executing trusted and authentic code on an applications processor starts with secure boot provided by the on chip boot ROM. The i.MX family of applications processors provides this capability with the HAB component of the on-chip ROM. The HAB enables the ROM to authenticate software, which executes immediately after ROM, by using digital signatures. This software is usually a bootloader.

The HAB provides a mechanism to establish a root of trust for the remaining software components and establishes a secure state on i.MX ICs that have secure state machine support in hardware.

The purpose of this application note is to explain how to perform a secure boot on i.MX applications processors that include HABv4. It tells the user how to generate signed images and configure the IC to run securely using tools provided freely by Freescale.

1.2 Scope

In this document, a practical example based on u-boot is used to illustrate the construction of a secure image, in addition to configuring the device to run securely. Extending the secure boot chain past the initial stage is also possible with HAB, but that is beyond the scope of this document.

This document answers all the following questions:

- What components are required?
  - A boot image
  - A set of signing keys and certificates
  - Resulting digital signatures
- How is each of these different components generated?
- How are all these components assembled to create a signed image?

NOTE

This document covers secure boot on the following i.MX processors using HABv4: i.MX50, i.MX53, and i.MX 6 Series. Secure boot features for other processors, such as i.MX25, i.MX35, and i.MX51, which use HABv3, are documented in *Secure Boot on i.MX25, i.MX35, and i.MX51 using HAB3* application note (AN4547).

NOTE

Secure boot features for i.MX28 are documented in *Secure Boot with i.MX28 HAB v4* application note (AN4555). i.MX28 supports HABv4 but its boot architecture is significantly different from other processors in the i.MX family.

NOTE

Secure boot using HAB is no longer supported on i.MX27 and i.MX31.

1.3 Audience

This document is intended for those who:

- Need an architectural-level technical understanding of how HAB works in the boot sequence
- Need to design signed software images to be used with a HAB-enabled processor

It is assumed that the reader is familiar with the basics of digital signatures and public key certificates.

1.4 Definitions, Acronyms, and Abbreviations

Definitions of the terms and acronyms used in this document are as follows:

- CA: Certificate Authority, the holder of a private key used to certify public keys.
- CAAM: Cryptographic Acceleration and Assurance Module, an accelerator for encryption, stream cipher, and hashing algorithms, with a random number generator and run time integrity checker.
- CMS: Cryptographic Message Syntax, a general format for data that may have cryptography applied to it, such as digital signatures and digital envelopes. HAB uses the CMS as a container holding PKCS#1 signatures.
- CSF: Command Sequence File, a binary data structure interpreted by the HAB to guide authentication operations.
- CST: Code Signing Tool, an application running on a build host to generate a CSF and associated digital signatures.
- DCD: Device Configuration Data, a binary table used by the ROM code to configure the device at early boot stage.
- DCP: Data co-processor, an accelerator for AES encryption and SHA hashing algorithms.
- HAB: High Assurance Boot, a software library executed in internal ROM on the Freescale processor at boot time which, among other things, authenticates software in external memory by verifying digital signatures in accordance with a CSF. This document is strictly limited to processors running HABv4.
- IVT: Image Vector Table.
- OS: Operating System.
- OTP: One-Time Programmable. OTP hardware includes masked ROM, and electrically programmable fuses (eFuses).
- PKCS#1: Standard specifying the use of the RSA algorithm.
- PKI: Public Key Infrastructure, a hierarchy of public key certificates in which each certificate (except the root certificate) can be verified using the public key above it.
- RTIC: Run-Time Integrity Checker, a SHA256 hash accelerator that ensures of software integrity during run time.
- RSA: Public key cryptography algorithm developed by Rivest, Shamir, and Adleman.
- SA: Signature Authority, the holder of a private key used to sign software components.
- SAHARA: Symmetric / Asymmetric Hashing and Random Accelerator, a cryptographic accelerator (including hash acceleration) found on some processors.

- SDP: Serial Download Protocol, also called UART/USB Serial Download Mode. This allows code provisioning through UART or USB during production and development phases.
- SRK: Super Root Key, an RSA key pair which forms the start of the boot-time authentication chain. The hash of the SRK public key is embedded in the processor using OTP hardware. The SRK private key is held by the CA. Unless explicitly noted, SRK in this document refers to the public key only.
- UID: Unique Identifier, a unique value (such as, a serial number) assigned to each processor during fabrication.
- XIP: Execute-In-Place, refers to a software image that is executed directly from its non-volatile storage location rather than first being copied to volatile memory.

1.5 References

- *i.MX50 Reference Manual, i.MX53 Reference Manual,* and *i.MX 6Dual/6Quad Reference Manual*.
- *i.MX53 Security Reference Manual, i.MX 6Dual/6Quad Security Reference Manual*, and *i.MX 6Solo/6DualLite Security Reference Manual.*
- *HAB CST User Guide* available in the *Code Signing Tool* package downloadable on freescale.com. Search for IMX_CST_TOOL.

2 Overview

This section gives a technical overview of HAB, and provides the background needed for understanding the use cases and processes described in later sections.

2.1 Boot ROM Code and HAB Library

In order to design a correctly signed boot image, it is necessary to understand both the components that make up the HAB and the basic boot-time authentication process. This section gives an architectural-level overview of these elements.

The boot ROM is the first software executed after reset, and it controls the initial phase of the boot process. The boot ROM uses the HAB library to authenticate the boot image in external memory, prior to its execution. Based on pin or fuse settings, the boot ROM executes different boot modes to locate, load, and execute the boot image from various boot peripherals (for example, NAND flash, SD/MMC card, Hard drive, serial EEPROM/flash, USB recovery mode, and so on). To ensure a secure boot, correct execution of the boot ROM must be guaranteed. To achieve this, the integrity of the boot ROM is protected by placing it inside a masked processor internal ROM that cannot be modified. Execution of the boot ROM is also protected by disabling external boot modes, unauthorized JTAG control, and interrupts.

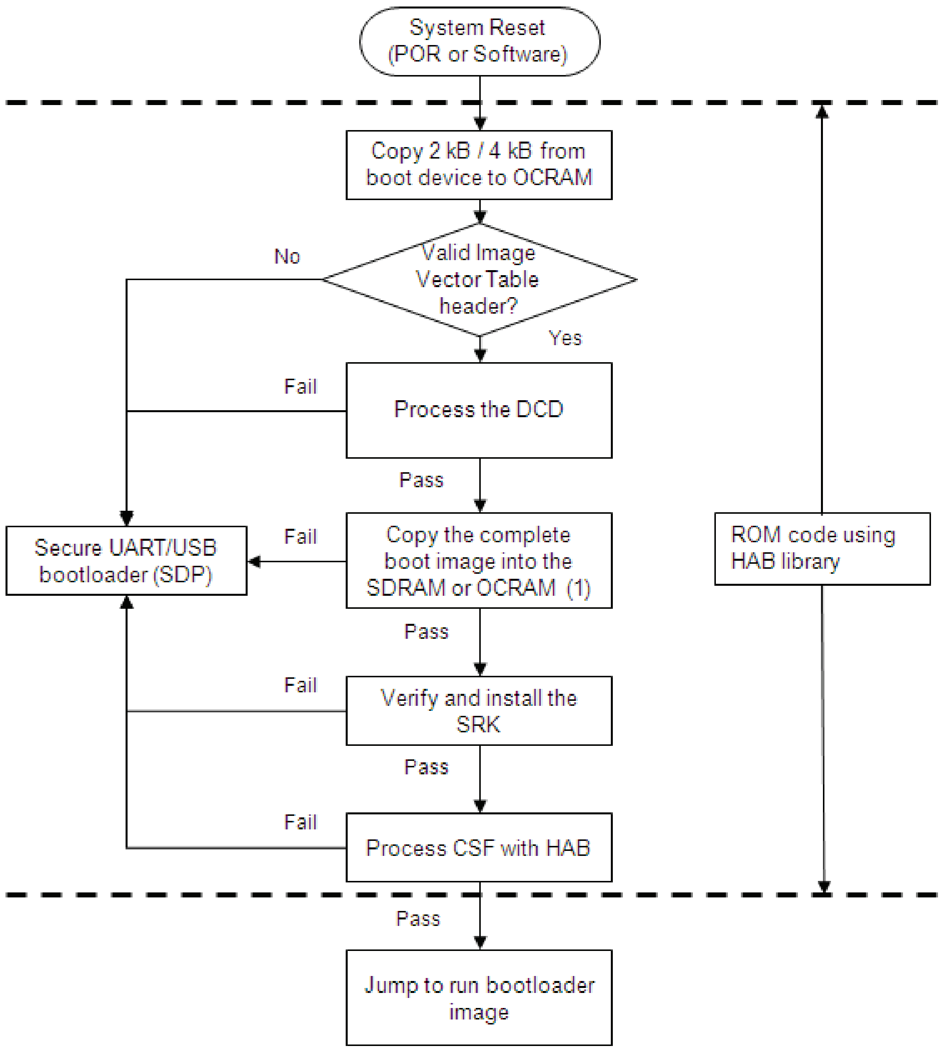
The HAB library, embedded in the processor ROM, contains functions to authenticate an image as well as initialize and test security hardware. The same library functions can be called from later boot stages to extend the secure boot chain past the stage immediately after the boot ROM.

The areas of an image that HAB verifies are completely customizable through a series of commands that are interpreted by HAB. These are known as Command Sequence File (CSF) commands that define the memory locations a digital signature covers, which keys to verify the signature with, and so on. All CSF processing, including PKI operations and cryptographic hash and digital signature verification, is performed within the HAB library. Where available, the HAB library makes use of on-board hardware accelerators (such as, RTIC, CAAM, or DCP) to improve boot performance. HABv4 makes exclusive use of the RSA algorithm, with all signatures following the CMS format, and all certificates following the X509v3 format.

For more details, see the reference manual of the processor being used, in addition to the *HAB CST User Guide* listed in Section 1.5, “References.”

2.2 Boot Flow

The boot ROM execution flow for i.MX applications processors that includes HABv4 is shown in Figure 1. “Process CSF with HAB” is the point in the flow where digital signatures across an image are verified. When configured for secure operation, the boot ROM on i.MX devices will not allow unauthenticated code to execute. Any signature failures or security violations forces the boot ROM to enter the Serial Download Protocol (SDP), which can be used to provide a new signed image to the boot device. Note that when configured for secure operation, even images downloaded through SDP must be properly signed before being executed.


System Reset (POR or Software)
Copy 2 kB / 4 kB from boot device to OCRAM
Valid Image Vector Table header?
No
Yes
Fail
Process the DCD
Pass
Secure UART/USB bootloader (SDP)
Fail
Copy the complete boot image into the SDRAM or OCRAM (1)
Pass
Fail
Verify and install the SRK
Pass
Fail
Process CSF with HAB
Pass
Jump to run bootloader image
ROM code using HAB library

(1) OCRAM is the preferred option for Trustzone.

Figure 1. Secure Boot Flow from Device

2.3 HAB Public Key Infrastructure

HAB authentication is based on public key cryptography using the RSA algorithm in which image data is signed offline using a series of private keys. The resulting signed image data is then verified on i.MX processor using the corresponding public keys. This key structure is known as a PKI tree.

Figure 2 gives an example of a typical PKI tree that is generated by the Freescale Code Signing Tools.

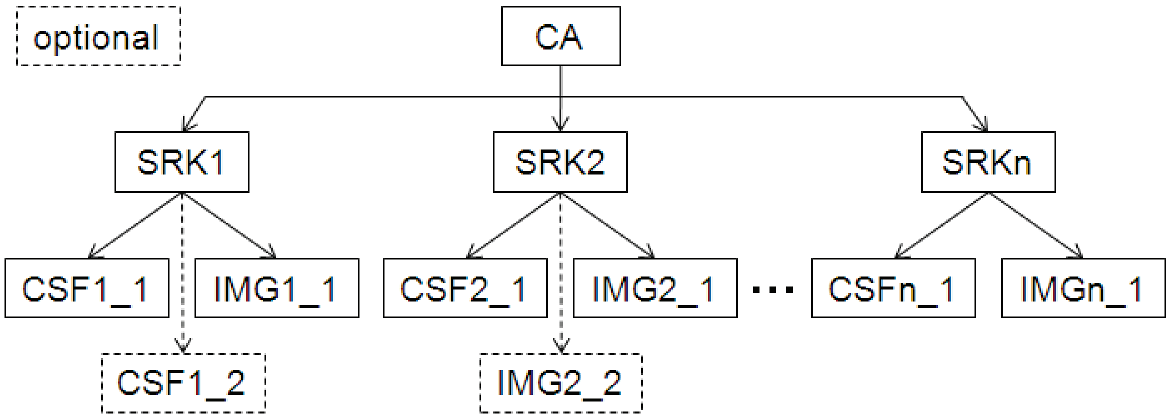


Figure 2. HABv4 Enabled Devices Typical PKI Tree

For further information on the PKI tree used for HABv4, see the *HAB CST User Guide* mentioned in Section 1.5, "References." The details of digital signature authentication with the RSA algorithm are beyond the scope of this document.

The authentication steps performed by HAB occur in stages that are shown in Figure 3.

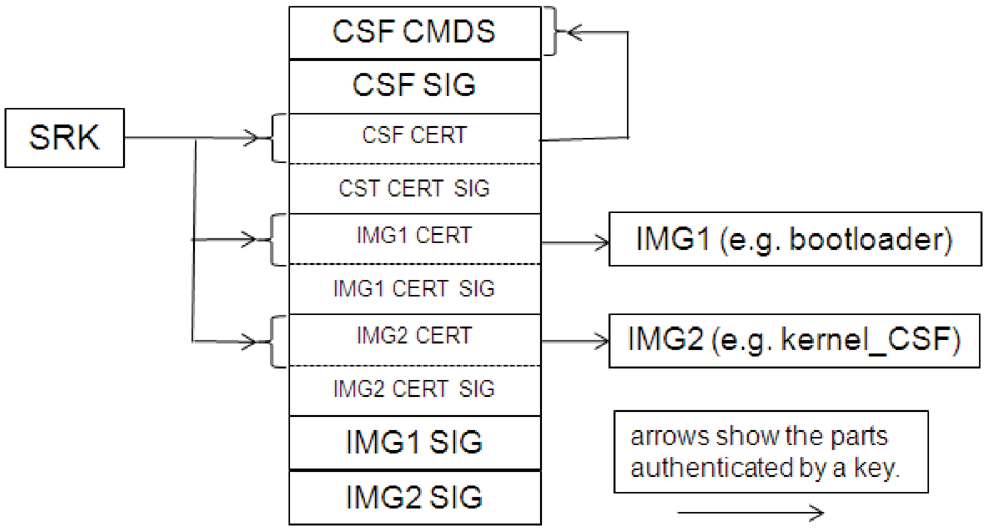


Figure 3. HABv4 Enabled Devices Detailed PKI Tree

In i.MX processors, the authentication begins with establishing a root of trust with the SRK. HAB does this by computing a cryptographic hash of the SRK table and comparing the result with a pre-computed hash that is provisioned in OTP fuses. This ensures that the integrity of the SRK table included in the image is intact. This is the beginning of the authentication chain in which the SRK is used to authenticate other keys which exist in the form of X509 certificates. For details on what is SRK table and how to generate it, see the *HAB CST User Guide*.

The arrows in Figure 3 show the authentication flow. For example, the SRK key is used to authenticate both the CSF and the image keys.

There are several important features to note. First, each key can certify multiple instances of the objects beneath it. For example, a single SRK can certify multiple CSF keys, and a single image key can certify multiple images. Nevertheless, the CSF key can only authenticate the CSF. The objects closer to the root of the PKI have greater impact if compromised, and require more protection. To some extent, this is offered by the CST tool, but it is important that the CST user adopt appropriate policies and processes. For example, a poorly planned CSF can open the way for malicious applications to be loaded in place of the authentic one. It is, therefore, prudent to plan the CSF so it does not need to change with every new version of the application, and restrict the group authorized to sign CSFs to a relatively small number of people.

Secondly, the authentication of an image key through the CSF may be a little different to the other links shown in Figure 3. In order to provide better key separation, an image key may be bound to the CSF, using a hash fingerprint embedded in the CSF. This means that it would not be possible to use the CSF with other image keys, even if they have been certified by the same SRK. Addition of the hash is optional and is dependent on whether the Hash Algorithm argument is present in the Install Key command. See the Install Key command documentation in the *HAB CST User Guide* for further details.

3 Code Signing

3.1 Identifying the Required Components

A secure boot requires a number of data components to be added to an image. This includes keys, certificates, signatures, IC configuration data, and CSF data.

When performing a secure boot on i.MX boot ROM and HAB, the following data components are required to be defined in the image:

- Image Vector Table: A table of pointers used by the boot ROM to locate other required data components.
- Boot data structure: A simple structure indicating where to load the boot image and specifying the size of the boot image.
- Device Configuration Data (DCD): A list of registers that the ROM will program with the provided data to perform an early initialization of the system. DCD is typically used to initialize the SDRAM.
- Command Sequence File (CSF) and associated data: A block of data containing the commands that the HAB will execute during boot, as well as the associated certificates and signatures HAB uses to verify an image.

3.1.1 Image Vector Table and Boot Data

The Image Vector Table (IVT) is a mandatory part of the boot image, and its structure could be defined as follows:

```
typedef struct
{
        uint32_t         header;
        uint32_t         *entry;
        uint32_t         reserved1;
        uint32_t         *dcd;
        boot_data_t      *boot_data;
        uint32_t         *self;
        uint32_t         *csf;
        uint32_t         reserved2;
} image_vector_table_t;
```

Where:

- `uint32_t`: A type representing a 32-bit unsigned integer.
- `header`: Header identifying the type of data structure (0xD1), its size (0x0020), and HAB version (such as, 0x40). For example, i.MX53 uses D100 2040h.
- `*entry`: Absolute address of the first instruction to be executed from the image.
- `reserved1`: Reserved and should be zero.
- `*dcd`: Absolute address of the image Device Configuration Table (DCD). The DCD is optional, so this field may be set to NULL, if no DCD is required.

- `*boot_data`: Absolute address of the Boot Data structure.
- `*self`: Absolute address of the IVT. Used internally by the ROM.
- `*csf`: Absolute address of the Command Sequence File (CSF) used by the HAB library. This field must be set to NULL, if not performing a secure boot.
- `reserved2`: Reserved and should be zero.

The Boot Data is an associated structure that indicates where to load the boot image and that specifies the size of the boot image. It is defined as follows:

```
typedef struct
{
        uint32_t         *start;
        uint32_t         length;
        uint32_t         plugin_flag;
} boot_data_t;
```

Where:

- `*start`: Absolute address of the boot image. Typically, somewhere in the SDRAM.
- `length`: Size of the boot image to copy from the boot device to address `*start`.
- `plugin_flag`: Reserved and must be zero.

The IVT is a block of data that must reside at a specific address that depends on the boot device, and is specified in the System Boot chapter of the reference manual.

3.1.2 Device Configuration Data

The main purpose of the DCD is to allow peripherals to be configured for optimal performance during application authentication. A second purpose is to allow memory controllers to be configured before loading the application from non-volatile storage to its run-time location in external RAM. Since DCD processing occurs prior to authentication, the scope of valid DCD operations is strictly limited to certain controllers (for example, clock, memories, and so on).

For more information, see the System Boot chapter of the reference manual.

NOTE

Some bootloaders use plug-in code to perform the device configuration; however, this method must not be used for a secure boot. The DCD method must be used instead, which provides the highest level of security and performance at boot.

3.1.3 Command Sequence File

The CSF is a binary data structure interpreted by the HAB library to guide the authentication process. It contains records that determine:

- The PKI tree to be used in authentication operations.
- The physical memory regions to be authenticated, along with the authentication method and referenced data.

- Device configuration operations.

Device configuration operations in the CSF are separate from those in the DCD. The important difference between the two is that DCD may configure only a limited range of peripherals (since DCD processing is performed prior to authentication) whereas device configuration operations within the CSF are unconstrained. This is because CSF commands are authenticated before they are executed.

With HAB, multiple non-contiguous regions of physical memory can be covered with a single digital signature. The maximum number of regions, which is limited by the hash computation engine used or its driver in ROM, can be defined as follows:

- When using RTIC for the hash computation of digital signature verification, a maximum of two (2) non-contiguous blocks are supported.
- When using SAHARA for the hash computation of digital signature verification, a maximum of twelve (12) non-contiguous blocks are supported.
- When using CAAM for the hash computation of digital signature verification, a maximum of eight (8) non-contiguous blocks are supported.
- When using DCP for the hash computation of digital signature verification, a maximum of six (6) non-contiguous blocks are supported. Note that all blocks except the last one must be a multiple of 64 bytes in length. The last data block may be of an arbitrary length.

If software implementation is used for hash computations included in HAB, then, there is no maximum limit.

3.2 Laying Out a Boot Image

3.2.1 Common Image Layout

When performing a secure boot on an i.MX processor based on HABv4, the image must contain a correctly formatted image vector table with a valid header and pointers, and a DCD table.

The first step of the boot process is to copy 1 kB, 2 kB, or 4 kB from the boot device into the OCRAM. The size copied depends on the boot device. This does not apply to a parallel NOR flash that uses XIP. See the System Boot chapter of the reference manual for more details.

At power up, the boot ROM expects a valid IVT that is placed at a known offset. The ROM code starts by extracting the pointer to the DCD table `*dcd`, and the DCD commands are processed. Note that when the processor is in Closed configuration, only certain memory regions are programmable by the DCD. The complete list is provided in the System Boot chapter of the reference manual.

If there are no errors to this point, then, the ROM reads the boot data information, and copies the specified length to the address `*start`.

Note that all the pointers included in the flash header are with respect to the final destination in memory `*start`.

At this stage, the rest of the boot process diverges depending on the security configuration: Open or Closed configuration, which is determined by the SEC_CONFIG fuse field.

3.2.2 Non-Secure Boot Image Layout

When performing a non-secure boot with the SEC_CONFIG fuse field set to Open, providing the CSF data as part of the image is optional. If CSF data is not provided, the `*csf` field of the IVT must be set to NULL. Regardless of whether a valid CSF present or not, HAB will attempt to authenticate the image performing the same steps as it would do for a secure boot in Closed configuration. If authentication fails, then, HAB will log events that can later be used for debugging purpose, and will continue execution of the normal boot flow. Eventually, ROM code will jump to the image pointed by `*entry`.

Note that when SEC_CONFIG fuse field is set to Open, all HAB failures are considered to be non-fatal and the boot process is allowed to continue. The Open configuration should also be used for development purposes of secure products, where CSFs and other data components for secure boot can be debugged. The Open configuration is the end configuration for non-secure products.

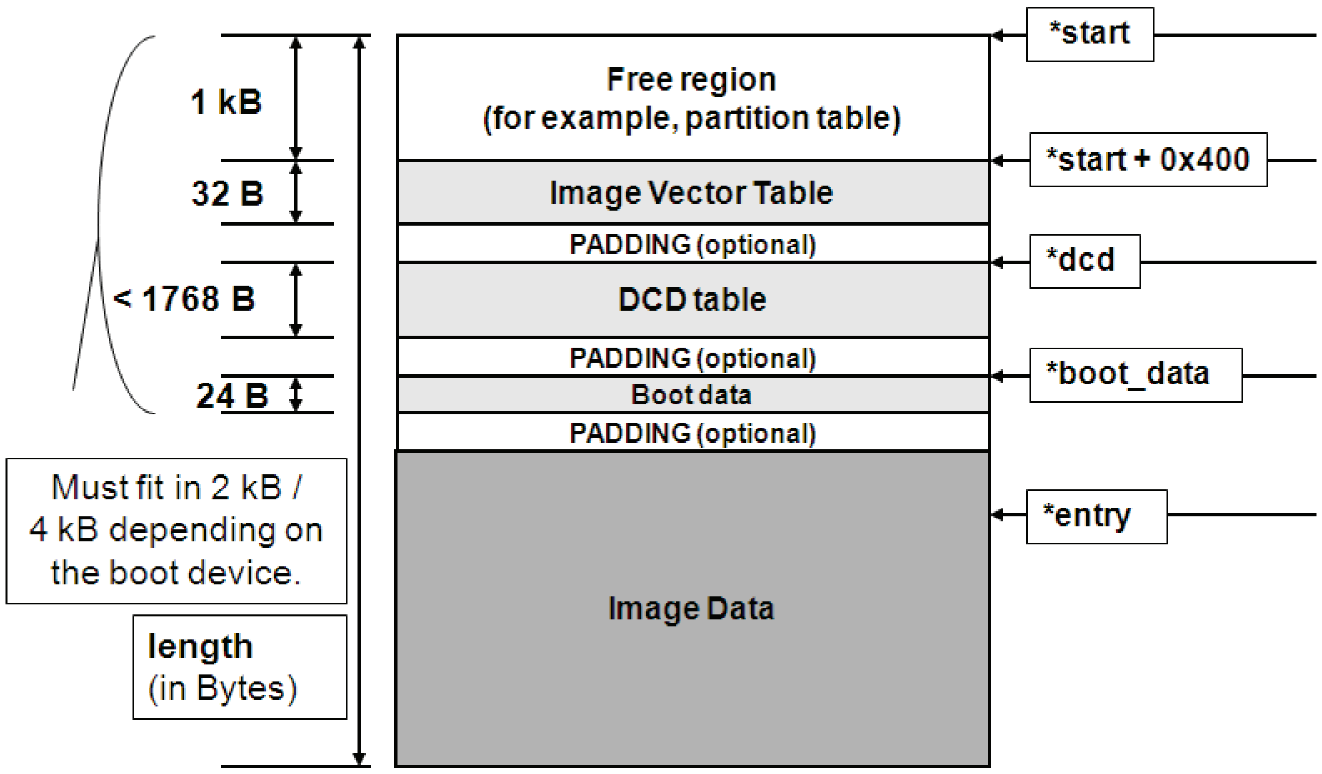


Figure 4. Typical Memory Layout of an Unsigned Image

Initially a small part (1 kB / 2 kB / 4 kB) of the boot code is read, so the IVT, DCD, and boot data must be located in that area.

The size of the free region and the IVT offset vary with the boot device. See the System Boot chapter of the reference manual for more details.

3.2.3 Secure Boot Image Layout

The SEC_CONFIG fuse field tells HAB whether or not to boot the device securely. If this field is set to Closed, the i.MX processor will only allow a properly signed image to execute. In Closed configuration, the CSF data component is mandatory and must be included in the image, along with valid pointers in the IVT structure. This is true regardless of which boot device is chosen, including USB recovery mode.

The first step performed by HAB when performing a secure boot is to install the SRK. It is important to have the SRK tied to the processor to avoid it being replaced by another non-trusted key. Therefore, during the installation of the SRK, the ROM computes a SHA-256 hash of the SRK table attached to the binary CSF data. The result is compared to the reference value provisioned into the OTP fuses during product manufacturing.

Following are the principal steps (not necessarily in order) involved in processing the CSF:

- Verify the CSF key certificate using the SRK.
- Verify the CSF signature using the CSF key.
- Verify image key certificates using the SRK.
- Verify image signature(s) using the image keys.
- Perform any device configuration operations specified in the CSF.

Note that not all steps apply to every CSF.

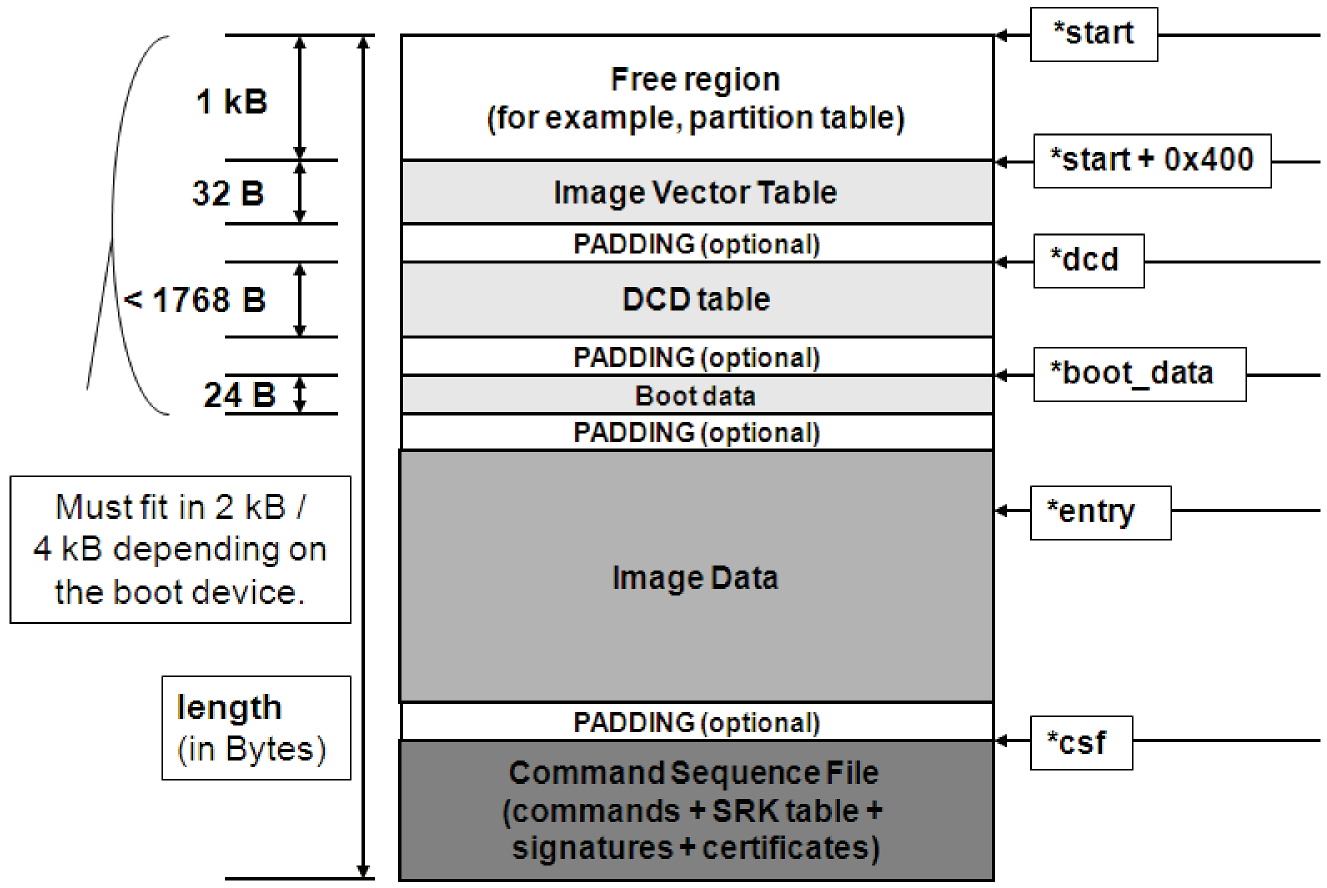


Figure 5. Typical Memory Layout of a Signed Image

Initially a small part (1 kB / 2 kB / 4 kB) of the boot code is read, so the IVT, DCD, and boot data must be located in that area.

The size of the free region and the IVT offset vary with the boot device. See the System Boot chapter of the reference manual for more details.

NOTE

The HAB requires that the IVT, initial byte of boot data, DCD table, and the first word of the image all must be covered by a digital signature.

3.3 Generating CSF Data and SRK Table

3.3.1 Generating Command Sequence File (CSF) Data

The CSF contains all the commands that the ROM executes during the secure boot. These commands instruct HAB on which memory areas of the image to authenticate, which keys to install and use, what data to write to a particular register, and so on. In addition, the necessary certificates and signatures involved in the verification of the image, as well as the SRK table, are attached to the CST.

The first CSF in the boot sequence must contain an Install Key command to install the SRK table prior to CSF key installation. It should also contain an Install Key command to install the CSF key prior to CSF authentication. Every CSF must contain an Authenticate Data command to authenticate the CSF contents using the CSF key.

To facilitate well-formed CSF generation, Freescale provides Code Signing Tool (IMX_CST_TOOL) as a reference for creating the CSF data. The package with the software executable and the associated documentation is available on the Freescale website as mentioned in Section 1.5, "References."

The binary output from the CST consists of the following components:

- CSF: Commands interpreted by HAB
- SRK table and corresponding fuse pattern
- Public key certificates
- CSF signature
- One or more image signatures

NOTE

Prior to continuing with examples described in this application note, see the *HAB CST User Guide*, available in the above mentioned package, to get a better understanding of the code signing process and how to use the CST.

3.3.2 Generating Keys and the Super Root Key (SRK) Table

To begin the code signing process, HAB code signing keys are required. The CST provides scripts to generate the required private keys and public key certificates. In addition to the keys, an SRK table must also be generated. The steps to generate the keys and the SRK table are as follows:

1. Generating HAB code signing keys: To generate the standard code signing keys for HAB, run the following command:

   ```
   ./hab4_pki_tree.sh
   ```

   The resulting private keys will be placed in the keys directory of the CST and the corresponding X.509 certificates will be placed in the crts directory. The private keys are stored in password protected files in PKCS#8 format, but care must be taken to ensure that the confidentiality of these keys is maintained.

   For details on key generation with the CST, see the *HAB CST User Guide*.

2. Generating an SRK table: HABv4 uses a concept of an SRK table to select the code signing root key. The SRK table allows installation of one of four (maximum) public keys. This key is used as the root of the HAB public key infrastructure. The SRK table is constructed from up to four public SRKs. A cryptographic hash of this table is generated by the CST; the generated cryptographic hash is then provisioned to the SRK_HASH field in OTP fuses during manufacturing. At boot time, an Install SRK CSF command specifies the location of the SRK table in memory, as well as the index of the SRK key, to be used for authenticating the remaining keys.

   To generate an SRK table, the CST provides the `srktool`, which requires X.509v3 public key certificates as inputs for the SRKs. The following example shows how to generate an SRK table with four keys:

```
../linux/srktool -h 4 -t SRK_1_2_3_4_table.bin -e SRK_1_2_3_4_fuse.bin -d sha256
-c
./SRK1_sha256_2048_65537_v3_ca_crt.pem,./SRK2_sha256_2048_65537_v3_ca_crt.pem,
./SRK3_sha256_2048_65537_v3_ca_crt.pem,./ SRK4_sha256_2048_65537_v3_ca_crt.pem
-f 1
```

NOTE

Section 5, "Managing Electrical Fuses," provides guidance on how to blow fuses, and which fuses are required to be blown for a secure product.

3.4 Assembling the CSF with the Boot Image

In Section 3.3.1, "Generating Command Sequence File (CSF) Data, we created a binary representing the CSF data. This section explains how to assemble that binary with the boot image to create a signed boot image. The purpose is to create a signed boot image organized as shown in the Figure 5 and Figure 6.

The following definitions will be referred to in the steps below:

- `boot_img.bin`: This is the boot image binary with the IVT configured for secure boot.
- `boot_img-signed.bin`: This is the signed boot image binary with the CSF included.
- `initial_length`: This is the image size of the unsigned boot image. The value for the initial length can be found in the boot data of the unsigned image.
- `csf_data.bin`: This is the CSF binary.

1. First step is to pad the boot image from `initial_length` to `*csf` address. This address points to a known and chosen free memory area. For instance, if `*csf` = 7782_C000h and `*start` = 7780_0000h, the binary has to be padded up to offset `*csf` - `*start` = 0002_C000h.

```
objcopy -I binary -O binary --pad-to 0x2C000 --gap-fill=0xff boot_img.bin
boot_img-pad.bin
```

2. Now, the CSF binary output from the CST can be merged with the previously padded boot image. With the padding, the CSF is localized as defined by the `*csf` pointer of the IVT:

```
cat boot_img-pad.bin csf_data.bin > boot_img-signed.bin
```

This signed boot image can now be used to boot an i.MX processor in Open configuration, without warnings or failures, or in Closed configuration.

4 Signed U-Boot Example

U-boot is a bootloader commonly used to boot a Linux device and is provided in the Freescale Linux BSP. There are multiple i.MX processors and boards to choose from to illustrate a secure u-boot, but this example will focus on the i.MX53 Quick Start board. The principle applied is exactly same when using other i.MX ICs or boards. An example for i.MX 6 Series is also included in the Linux BSP. The addresses in that case may be different but the concepts and steps performed here for i.MX53 will be identical for i.MX 6 Series. The signed u-boot will have the following memory map.

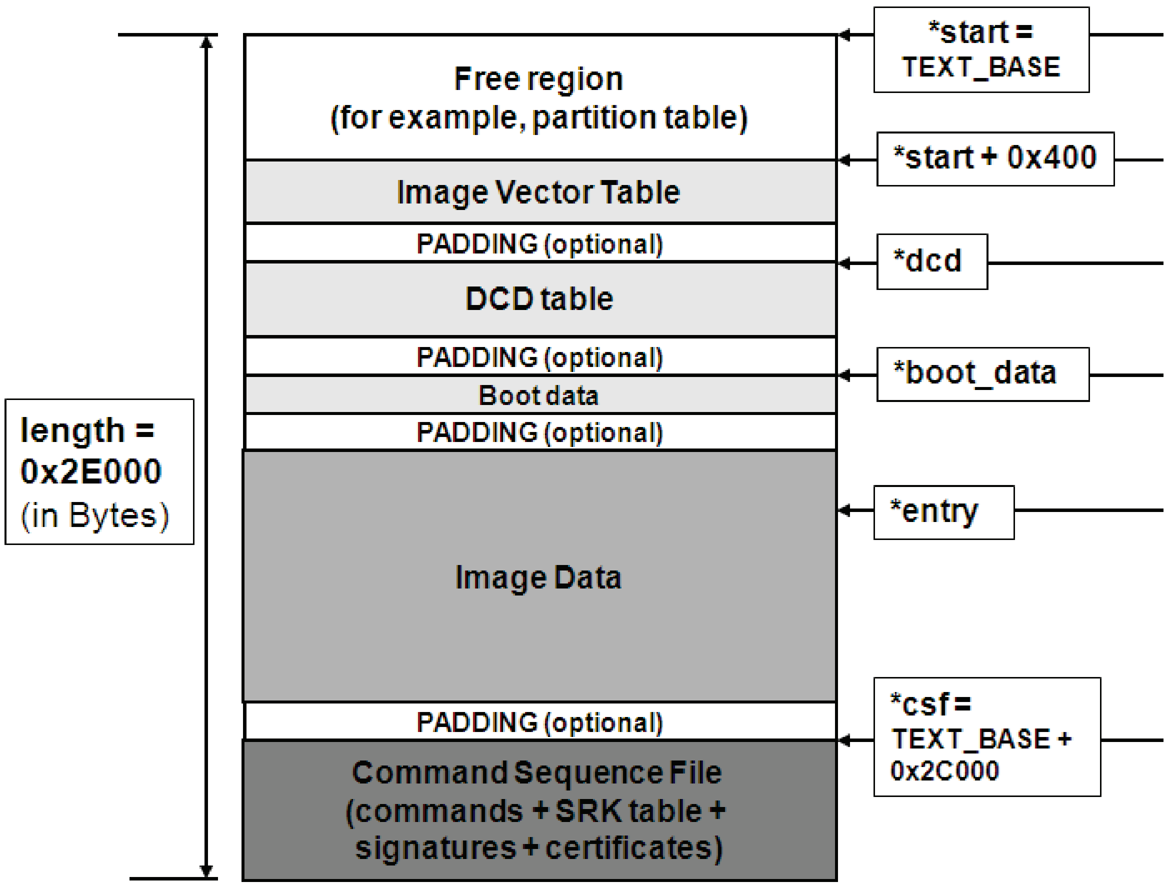


Figure 6. Chosen Memory Layout of a Signed U-Boot for i.MX53

TEXT_BASE is actually the pointer `*start` where the code is copied from the boot device. This variable is defined in the file:

```
./u-boot/board/freescale/mx53_loco/config.mk
```

If the boot is performed from an SD card, the offset of the flash header must equal 0000_0400h, so IVT is placed at TEXT_BASE + 0000_0400h. This layout is chosen based on the unsigned u-boot binary size. By default, the size is around 170 kB, so the offset for the CSF data must be located beyond that limit.

Using an offset of 0002_C000h, which corresponds to 176 kB, leaves some space for u-boot to grow, if necessary, without changing the addresses specified in the CSF description file, inputted to the CST. This empty area between the real end of u-boot code and the `*csf` is padded as explained previously. This places the CSF at the offset `*csf` = TEXT_BASE + 0002_C000h.

It is necessary to leave some space for the CSF and in this example a region of 8 kB is reserved. The end of that region determines the size of the final signed bootloader, which is the length value. The CSF will not consume all 8 kB of this region and it may or may not be padded. In any case, the ROM will copy length bytes of data.

The following are the steps to generate the layout defined above for the signed u-boot example:

1. Modify the linker script to reserve 8 kB space for the CSF data between u-boot data and the zero initialization section ".bss". Some labels, such as `__hab_data_start`, are used to locate the various created addresses, so that the IVT can use them as pointers. The file is located at:

   ```
   ./u-boot/board/freescale/mx53_loco/u-boot.lds
   ```

   See Section 8.1, "./u-boot/board/freescale/mx53_loco/u-boot.lds," for modified linker script.
2. Modify the IVT with the updated image length and the CSF pointer. The file is located at:

   ```
   ./u-boot/board/freescale/mx53_locoflash_header.S
   ```

   See Section 8.2, "./u-boot/board/freescale/mx53_loco/flash_header.S," for modified assembly file.
3. U-boot must be built to generate `u-boot.bin`. See u-boot BSP documentation for the detailed procedure, and to configure the environment, before executing these build commands:

   ```
   make mx53_loco_config
   make
   ```

   If u-boot is built through LTIB, follow the instructions provided in the BSP documentation, and use a command such as:

   ```
   ./ltib -m scbuild -p u-boot
   ```
4. The obtained binary has to be padded up to offset 0000_2C00h, as explained in Section 3.4, "Assembling the CSF with the Boot Image,":

   ```
   objcopy -I binary -O binary --pad-to 0x2C000 --gap-fill=0xff u-boot.bin
   u-boot-pad.bin
   ```
5. The padded u-boot is ready to be signed, and according to the *HAB CST User Guide*, the PKI tree is generated with the following command:

   ```
   /install_path/keys/hab4_pki_tree.sh
   ```

   See the *HAB CST User Guide* for all the prerequisites and for a step by step procedure to generate code signing keys and certificates.
6. The code can be signed, and the CSF instruction file is available in Section 8.3, "U-Boot CSF Example for i.MX53."

   ```
   /install_path/linux/cst --output csf_u-boot.bin < csf_u-boot.txt
   ```
7. Finally, by following the instructions provided in Section 3.4, "Assembling the CSF with the Boot Image," the final signed binary image can be merged, by using the below command:

   ```
   cat u-boot-pad.bin csf_ u-boot.bin > u-boot-signed.bin
   ```

5 Managing Electrical Fuses

5.1 Fuse Programming Solutions

5.1.1 Solution 1

Use the Freescale Manufacturing Tool, available on freescale.com. Search for IMX_MFG_TOOL.

Section 5.3, “Provisioning the SRK Hash eFuse Field for i.MX53,” and Section 5.4, “Provisioning the SRK Hash eFuse Field for i.MX50 and i.MX 6 Series provide examples to create the appropriate fuse programming profile in the `ucl.xml` tool file.

5.1.2 Solution 2

For i.MX53, which uses the IIM controller to manage the fuses, the following u-boot commands can be used with `<bank>` and `<row>` in hexadecimal format:

```
iim read <bank> <row>
iim blow <bank> <row> <value>
```

For i.MX 6 Series, which uses the OCOTP controller to manage the fuses, the following u-boot commands can be used with `<addr>` and `<value>` in hexadecimal format:

```
imxotp read <addr>
imxotp blow [--force] <addr> <value>
```

5.2 Fuse Configuration Recommendations

During development and production, it is suggested to change the SEC_CONFIG of the chip to Closed configuration only after the programming, provisioning, validation, and all other custom operations are complete and verified to be working in Open configuration. When only secure boot is allowed, every boot image, regardless of the boot device, must be signed and authenticated correctly prior to its execution. It is easier to fix any encountered issue with a chip still in Open configuration.

Many important boot parameters are set through the fuses. The fact that a fuse is only One Time Programmable is sufficient to prevent reverting a fuse that has been burned. This provides assurance for configuration parameters consisting of a single bit when it is set. However it does not provide protection for single bit fuse fields that remain intact or for fuse fields consisting of a number of fuses, such as the SRK digest.

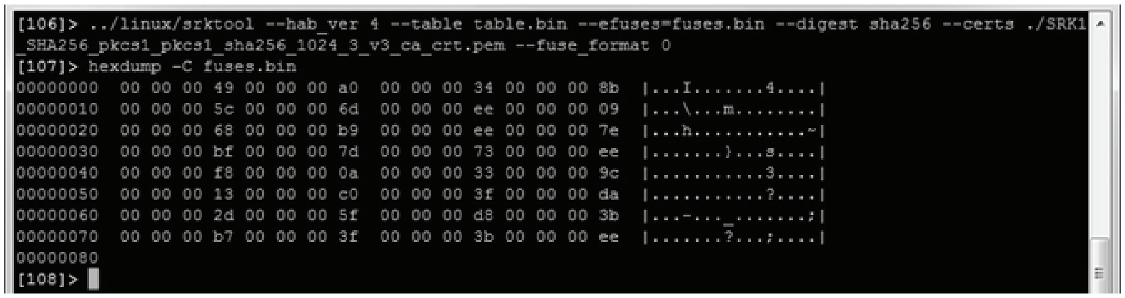
Therefore, some additional lock fuses are dedicated to protect such fields. It is recommended to blow these lock fuses once the protected value is programmed and verified to be functional

For example, with the i.MX53, once the SRK_HASH[255:0] is programmed, the SRK_LOCK88 and SRK_LOCK160 fuses must be programmed to disable any modification to the reference digest for the SRK table.

See the i.MX fuse map of the chip for the list of available protections with lock fuses.

5.3 Provisioning the SRK Hash eFuse Field for i.MX53

When generating the SRK table for i.MX53 with the CST `srktool`, the --fuse_format option is generally set to zero (0). This tells the `srktool` to generate one hash byte per 32-bit word. This is similar to how values are mapped in the IIM module of i.MX53.



```
[106]> ../linux/srktool --hab_ver 4 --table table.bin --efuses=fuses.bin --digest sha256 --certs ./SRK1
_SHA256_pkcs1_pkcs1_sha256_1024_3_v3_ca_crt.pem --fuse_format 0
[107]> hexdump -C fuses.bin
00000000  00 00 00 49 00 00 00 a0  00 00 00 34 00 00 00 8b  |...I.......4....|
00000010  00 00 00 5c 00 00 00 6d  00 00 00 ee 00 00 00 09  |...\...m........|
00000020  00 00 00 68 00 00 00 b9  00 00 00 ee 00 00 00 7e  |...h...........~|
00000030  00 00 00 bf 00 00 00 7d  00 00 00 73 00 00 00 ee  |.......}...s....|
00000040  00 00 00 f8 00 00 00 0a  00 00 00 33 00 00 00 9c  |...........3....|
00000050  00 00 00 13 00 00 00 c0  00 00 00 3f 00 00 00 da  |...........?....|
00000060  00 00 00 2d 00 00 00 5f  00 00 00 d8 00 00 00 3b  |...-..._.......;|
00000070  00 00 00 b7 00 00 00 3f  00 00 00 3b 00 00 00 ee  |.......?...;....|
00000080
[108]> 
```

Figure 7. SRK Fuse Values for i.MX53

Figure 7 illustrates the SRK hash eFuse result of the `srktool` for an SRK table consisting of a single key. The result is then displayed using the `hexdump` command available on Linux. These values can then be input to the `ucl.xml` script for use with the Freescale Manufacturing Tool. The XML example below illustrates how the results of the `srktool` are mapped to the bytes of the SRK hash eFuse field on i.MX53. The hash result for different keys will be different from what is shown here, but the mapping or ordering of the bytes will be the same. Failure to follow this byte ordering will cause a failure HAB when processing the CSF Installation SRK command.

```
<LIST name="MX53LOCO SRK Hash" desc="SRK hash fuse programming">
  <!--
    The method to enter serial download mode on MX53 LOCO boards:
    Don't insert Micro SD card. Press Power key to power on the board.
    The MFG tool should detect an USB-device.
  -->
<CMD type="boot" body="BootStrap" file ="u-boot-mx53-loco.bin" >Read from DDR script
from U-Boot to init DDR Memory.</CMD>
<CMD type="load" file="uImage" address="0x70800000"
    loadSection="OTH" setSection="OTH" HasFlashHeader="FALSE" >Doing Kernel.</CMD>
<CMD type="load" file="initramfs.cpio.gz.uboot" address="0x70B00000"
    loadSection="OTH" setSection="OTH" HasFlashHeader="FALSE" >Doing Init-
ramfs.</CMD>
<CMD type="jump" > Jumping to OS image. </CMD>
<CMD type="find" body="Updater" timeout="180"/>

  <!--
  The following blows the SRK hash fuses for the SRK table: TBL_1_sha256
  - The first byte in TBL_1_sha256 is SRK_HASH[255:248]
       + located at offet 0x0C04 in the i.MX53 iim
  - The second byte in TBL_1_sha256 is SRK_HASH[247:240]
       + located at offet 0x1404 in the i.MX53 iim
                              .
                              .
                              .
  - The last byte in TBL_1_sha256 is SRK_HASH[7:0]
```

```
       + located at offset 0x147c in the i.MX53 iim
-->

<CMD type="push" body="$ cat /sys/class/misc/mxc_iim/dev"/>
<CMD type="push" body="$ mknod /dev/mxc_iim c 10 63"/>
<CMD type="push" body="$ echo \"0xc04 0x49\" > /dev/mxc_iim "/>
<CMD type="push" body="$ dd if=/dev/mxc_iim of=/dev/temp.txt bs=1 skip=3076
count=1"/>
<CMD type="push" body="$ echo \"0x1404 0xA0\" > /dev/mxc_iim "/>
<CMD type="push" body="$ dd if=/dev/mxc_iim of=/dev/temp.txt bs=1 skip=5124
count=1"/>
<CMD type="push" body="$ echo \"0x1408 0x34\" > /dev/mxc_iim "/>
<CMD type="push" body="$ dd if=/dev/mxc_iim of=/dev/temp.txt bs=1 skip=5128
count=1"/>
<CMD type="push" body="$ echo \"0x140C 0x8B\" > /dev/mxc_iim "/>
<CMD type="push" body="$ dd if=/dev/mxc_iim of=/dev/temp.txt bs=1 skip=5132
count=1"/>
<CMD type="push" body="$ echo \"0x1410 0x5C\" > /dev/mxc_iim "/>
<CMD type="push" body="$ dd if=/dev/mxc_iim of=/dev/temp.txt bs=1 skip=5136
count=1"/>
<CMD type="push" body="$ echo \"0x1414 0x6D\" > /dev/mxc_iim "/>
<CMD type="push" body="$ dd if=/dev/mxc_iim of=/dev/temp.txt bs=1 skip=5140
count=1"/>
<CMD type="push" body="$ echo \"0x1418 0xEE\" > /dev/mxc_iim "/>
<CMD type="push" body="$ dd if=/dev/mxc_iim of=/dev/temp.txt bs=1 skip=5144
count=1"/>
<CMD type="push" body="$ echo \"0x141C 0x09\" > /dev/mxc_iim "/>
<CMD type="push" body="$ dd if=/dev/mxc_iim of=/dev/temp.txt bs=1 skip=5148
count=1"/>
<CMD type="push" body="$ echo \"0x1420 0x68\" > /dev/mxc_iim "/>
<CMD type="push" body="$ dd if=/dev/mxc_iim of=/dev/temp.txt bs=1 skip=5152
count=1"/>
<CMD type="push" body="$ echo \"0x1424 0xB9\" > /dev/mxc_iim "/>
<CMD type="push" body="$ dd if=/dev/mxc_iim of=/dev/temp.txt bs=1 skip=5156
count=1"/>
<CMD type="push" body="$ echo \"0x1428 0xEE\" > /dev/mxc_iim "/>
<CMD type="push" body="$ dd if=/dev/mxc_iim of=/dev/temp.txt bs=1 skip=5160
count=1"/>
<CMD type="push" body="$ echo \"0x142C 0x7E\" > /dev/mxc_iim "/>
<CMD type="push" body="$ dd if=/dev/mxc_iim of=/dev/temp.txt bs=1 skip=5164
count=1"/>
<CMD type="push" body="$ echo \"0x1430 0xBF\" > /dev/mxc_iim "/>
<CMD type="push" body="$ dd if=/dev/mxc_iim of=/dev/temp.txt bs=1 skip=5168
count=1"/>
<CMD type="push" body="$ echo \"0x1434 0x7D\" > /dev/mxc_iim "/>
<CMD type="push" body="$ dd if=/dev/mxc_iim of=/dev/temp.txt bs=1 skip=5172
count=1"/>
<CMD type="push" body="$ echo \"0x1438 0x73\" > /dev/mxc_iim "/>
<CMD type="push" body="$ dd if=/dev/mxc_iim of=/dev/temp.txt bs=1 skip=5176
count=1"/>
<CMD type="push" body="$ echo \"0x143C 0xEE\" > /dev/mxc_iim "/>
<CMD type="push" body="$ dd if=/dev/mxc_iim of=/dev/temp.txt bs=1 skip=5180
count=1"/>
```

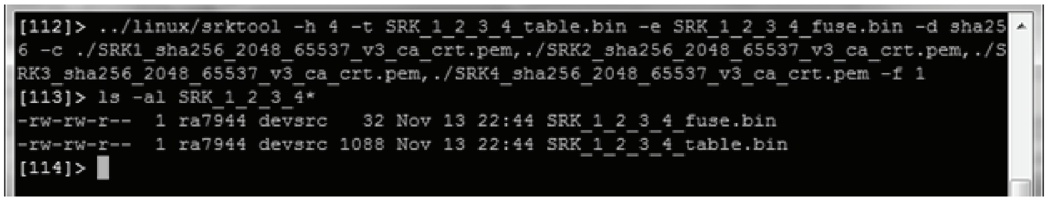
```
<CMD type="push" body="$ echo \"0x1440 0xF8\" > /dev/mxc_iim "/>
<CMD type="push" body="$ dd if=/dev/mxc_iim of=/dev/temp.txt bs=1 skip=5184
count=1"/>
<CMD type="push" body="$ echo \"0x1444 0x0A\" > /dev/mxc_iim "/>
<CMD type="push" body="$ dd if=/dev/mxc_iim of=/dev/temp.txt bs=1 skip=5188
count=1"/>
<CMD type="push" body="$ echo \"0x1448 0x33\" > /dev/mxc_iim "/>
<CMD type="push" body="$ dd if=/dev/mxc_iim of=/dev/temp.txt bs=1 skip=5192
count=1"/>
<CMD type="push" body="$ echo \"0x144C 0x9C\" > /dev/mxc_iim "/>
<CMD type="push" body="$ dd if=/dev/mxc_iim of=/dev/temp.txt bs=1 skip=5196
count=1"/>
<CMD type="push" body="$ echo \"0x1450 0x13\" > /dev/mxc_iim "/>
<CMD type="push" body="$ dd if=/dev/mxc_iim of=/dev/temp.txt bs=1 skip=5200
count=1"/>
<CMD type="push" body="$ echo \"0x1454 0xC0\" > /dev/mxc_iim "/>
<CMD type="push" body="$ dd if=/dev/mxc_iim of=/dev/temp.txt bs=1 skip=5204
count=1"/>
<CMD type="push" body="$ echo \"0x1458 0x3F\" > /dev/mxc_iim "/>
<CMD type="push" body="$ dd if=/dev/mxc_iim of=/dev/temp.txt bs=1 skip=5208
count=1"/>
<CMD type="push" body="$ echo \"0x145C 0xDA\" > /dev/mxc_iim "/>
<CMD type="push" body="$ dd if=/dev/mxc_iim of=/dev/temp.txt bs=1 skip=5212
count=1"/>
<CMD type="push" body="$ echo \"0x1460 0x2D\" > /dev/mxc_iim "/>
<CMD type="push" body="$ dd if=/dev/mxc_iim of=/dev/temp.txt bs=1 skip=5216
count=1"/>
<CMD type="push" body="$ echo \"0x1464 0x5F\" > /dev/mxc_iim "/>
<CMD type="push" body="$ dd if=/dev/mxc_iim of=/dev/temp.txt bs=1 skip=5220
count=1"/>
<CMD type="push" body="$ echo \"0x1468 0xD8\" > /dev/mxc_iim "/>
<CMD type="push" body="$ dd if=/dev/mxc_iim of=/dev/temp.txt bs=1 skip=5224
count=1"/>
<CMD type="push" body="$ echo \"0x146C 0x3B\" > /dev/mxc_iim "/>
<CMD type="push" body="$ dd if=/dev/mxc_iim of=/dev/temp.txt bs=1 skip=5228
count=1"/>
<CMD type="push" body="$ echo \"0x1470 0xB7\" > /dev/mxc_iim "/>
<CMD type="push" body="$ dd if=/dev/mxc_iim of=/dev/temp.txt bs=1 skip=5232
count=1"/>
<CMD type="push" body="$ echo \"0x1474 0x3F\" > /dev/mxc_iim "/>
<CMD type="push" body="$ dd if=/dev/mxc_iim of=/dev/temp.txt bs=1 skip=5236
count=1"/>
<CMD type="push" body="$ echo \"0x1478 0x3B\" > /dev/mxc_iim "/>
<CMD type="push" body="$ dd if=/dev/mxc_iim of=/dev/temp.txt bs=1 skip=5240
count=1"/>
<CMD type="push" body="$ echo \"0x147C 0xEE\" > /dev/mxc_iim "/>
<CMD type="push" body="$ dd if=/dev/mxc_iim of=/dev/temp.txt bs=1 skip=5244
count=1"/>

<CMD type="push" body="$ echo Update Complete!">Done</CMD>
</LIST>
```

This script can also be extended to burn the SRK Hash lock fuses.

5.4 Provisioning the SRK Hash eFuse Field for i.MX50 and i.MX 6 Series

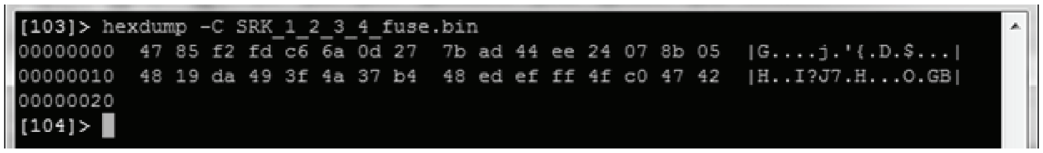
When generating the SRK table for i.MX50 or i.MX 6 Series with the CST `srktool`, the --fuse_format option is generally set to one (1), which is the default value. This tells the `srktool` to generate 4 hash bytes per 32-bit word. This is similar to how values are mapped in the OCOTP module of i.MX50 and i.MX 6 Series.



```
[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]>
```

Figure 8. Generating SRK for i.MX50 and i.MX 6 Series

Figure 8 illustrates the SRK hash eFuse result of the `srktool` for an SRK table consisting of four keys. The result is then displayed using the `hexdump` command available on Linux, as shown in Figure 9.



```
[103]> hexdump -C SRK_1_2_3_4_fuse.bin
00000000  47 85 f2 fd c6 6a 0d 27  7b ad 44 ee 24 07 8b 05  |G....j.'{.D.$...|
00000010  48 19 da 49 3f 4a 37 b4  48 ed ef ff 4f c0 47 42  |H..I?J7.H...O.GB|
00000020
[104]>
```

Figure 9. SRK Fuse Values for i.MX50 and i.MX 6 Series

NOTE

The -C option for `hexdump` is essential; otherwise, the SRK bytes will not be in the correct order.

Before theses values are placed in an XML script for the Freescale Manufacturing Tool, they need to be converted to words, as illustrated below:

```
0xfdf28547 0x270d6ac6 0xee44ad7b 0x058b0724

0x49da1948 0xb4374a3f 0xffefed48 0x4247c04f
```

These values can now be placed in an `ucl.xml` script to use them with the Freescale Manufacturing Tool, as shown in the below example for the i.MX 6 Series Sabre Lite board. The hash result for different keys may be different from what is shown here, but the mapping or ordering of the bytes will be the same. Failing to follow this byte ordering will cause a HAB failure, when processing the CSF Installation SRK command.

```
<LIST name="MX6Q Sabre-lite SRK Hash" desc="SRK hash fuse programming">
    <CMD type="find" body="Recovery" timeout="180"/>
    <CMD type="boot" body="Recovery" file ="u-boot-mx6q-sabrelite.bin" >Loading
uboot.</CMD>
    <CMD type="load" file="uImage" address="0x10800000"
        loadSection="OTH" setSection="OTH" HasFlashHeader="FALSE" >Doing Ker-
```

```
nel.</CMD>
    <CMD type="load" file="initramfs.cpio.gz.uboot" address="0x10C00000"
        loadSection="OTH" setSection="OTH" HasFlashHeader="FALSE" >Doing Init-
ramfs.</CMD>
    <CMD type="jump" > Jumping to OS image. </CMD>
    <CMD type="find" body="Updater" timeout="180"/>
    <!-- ***** Caution - running this xml script with the fuse burning commands
uncommented
        ***** in the Mfg tool permanently burns fuses. Once completed this operation
cannot
         ***** be undone!
    -->
    <CMD type="push" body="$ echo 0xfdf28547 > /sys/fsl_otp/HW_OCOTP_SRK0">Burn Word
0 of SRK hash field in OTP </CMD>
    <CMD type="push" body="$ echo 0x270d6ac6 > /sys/fsl_otp/HW_OCOTP_SRK1">Burn Word
1 of SRK hash field in OTP </CMD>
    <CMD type="push" body="$ echo 0xee44ad7b > /sys/fsl_otp/HW_OCOTP_SRK2">Burn Word
2 of SRK hash field in OTP </CMD>
    <CMD type="push" body="$ echo 0x058b0724 > /sys/fsl_otp/HW_OCOTP_SRK3">Burn Word
3 of SRK hash field in OTP </CMD>
    <CMD type="push" body="$ echo 0x49da1948 > /sys/fsl_otp/HW_OCOTP_SRK4">Burn Word
4 of SRK hash field in OTP </CMD>
    <CMD type="push" body="$ echo 0xb4374a3f > /sys/fsl_otp/HW_OCOTP_SRK5">Burn Word
5 of SRK hash field in OTP </CMD>
    <CMD type="push" body="$ echo 0xffefed48 > /sys/fsl_otp/HW_OCOTP_SRK6">Burn Word
6 of SRK hash field in OTP </CMD>
    <CMD type="push" body="$ echo 0x4247c04f > /sys/fsl_otp/HW_OCOTP_SRK7">Burn Word
7 of SRK hash field in OTP </CMD>
    <CMD type="push" body="$ cat /sys/fsl_otp/HW_OCOTP_SRK0"/>
    <CMD type="push" body="$ cat /sys/fsl_otp/HW_OCOTP_SRK1"/>
    <CMD type="push" body="$ cat /sys/fsl_otp/HW_OCOTP_SRK2"/>
    <CMD type="push" body="$ cat /sys/fsl_otp/HW_OCOTP_SRK3"/>
    <CMD type="push" body="$ cat /sys/fsl_otp/HW_OCOTP_SRK4"/>
    <CMD type="push" body="$ cat /sys/fsl_otp/HW_OCOTP_SRK5"/>
    <CMD type="push" body="$ cat /sys/fsl_otp/HW_OCOTP_SRK6"/>
    <CMD type="push" body="$ cat /sys/fsl_otp/HW_OCOTP_SRK7"/>
</LIST>
</UCL>
```

This script can also be extended to burn the SRK Hash lock fuse.

5.5 SRK Revocation on i.MX 6 Series

The i.MX 6 Series supports revocation of SRKs. Before explaining SRK revocation, it is helpful to know a little more about how the SRK table, generated by the `srktool` of the CST, is structured. The SRK table generated by the `srktool` of the CST may contain up to four separate public keys, with only one being selected at boot time through an Install SRK CSF command. In case, an SRK in use is compromised, then, it is possible to revoke that key on i.MX 6 Series devices. The revocation is performed by burning the corresponding bit in the SRK_REVOKE eFuse field.

Figure 10 illustrates an example.

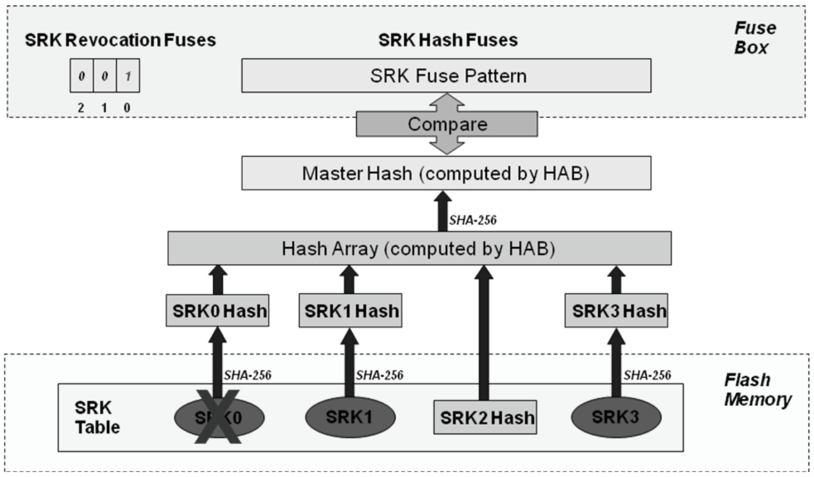


Figure 10. SRK Revocation using HABv4

In this example, an SRK table with four public keys has been generated. To revoke SRK0 from a bootloader or another stage after the boot ROM, it is necessary to blow the SRK_REVOKE[0] eFuse. However, in Closed configuration, HAB, by default, sets the SRK_REVOKE_LOCK sticky bit in the OCOTP controller to write protect this field. To instruct HAB not to lock the SRK_REVOKE field requires the use of the Unlock CSF command, with the command flag indicating to unlock the SRK_REVOKE field. Including this command allows the SRK0 fuse to be blown by a trusted bootloader that has been validated by HAB, during the boot process. Below is an example CSF description file that unlocks the SRK_REVOKE eFuse field, allowing u-boot or a later stage to update the field.

```
[Header]
    Version = 4.1  # i.MX 6 Series uses HABv4.1
    Hash Algorithm = sha256
    Engine Configuration = 0
    Certificate Format = X509
    Signature Format = CMS
[Install SRK]
    File = "../crts/SRK_1_2_3_4_table.bin"
    Source index = 0
[Install CSFK]
    File = "../crts/CSF1_1_sha256_2048_65537_v3_usr_crt.pem"
[Authenticate CSF]
```

```
[Unlock]
    Engine = OCOTP
    Features = SRK REVOKE
[Install Key]
    Verification index = 0
    Target index = 2
    File = "../crts/IMG1_1_sha256_2048_65537_v3_usr_crt.pem"
# Sign padded u-boot starting at the IVT through to the end with
# length = 0x2C000 (padded u-boot length) - 0x400 (IVT offset) = 0x2BC00
# This covers the essential parts: IVT, boot data and DCD.
# Blocks have the following definition:
#    Image block start address on i.MX, Offset from start of image file,
#    Length of block in bytes, image data file
[Authenticate Data]
    Verification index = 2
    Blocks = 0x27800400 0x400 0x2BC00 "u-boot-pad.bin"
```

HAB allows only the first three SRKs to be revoked. This ensures that there is at least one SRK available for use. If all SRKs were to be revoked in Closed configuration, then, the i.MX processor would not be able to boot.

Note that a hash of SRK2 has been placed in the table rather than the entire key. This allows reducing the table size without generating a new SRK hash value, when the SRK2 key is added to the table later.

6 Development and Debugging Tips

6.1 Error Logging

In development phase, it is recommended to test the device in Open configuration. All along the boot procedure and until the boot image is executed, the HAB logs the encountered errors and warnings. It is essential because this information may be required to debug problems during boot. This information can be accessed from the OCRAM buffer by using the HAB API.

The `report_event` function helps to browse through the events created during HAB authentication. The `report_status` function can be used to determine the security configuration and security state of the system. For detailed information on HAB Event Data, contact your Freescale representative. All FAILURE events reported by HAB in Open configuration must be resolved before moving on to Closed configuration. Although warning events do not prevent booting in Closed configuration, though they should be analyzed. It is recommended to fix all warnings before final production of the image because they slow down the boot process.

The HAB API is actually accessed through the ROM Vector Table, which consists of a header, followed by a list of addresses to certain functions. Its start address is 0000_0094h, as per the System Boot chapter of the reference manual.

At 0000_0094h:

```
hab_hdr_t hdr
```

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

At 0000_0098h:

```
hab_status_t(* entry )(void)
```

Enter and initialize HAB library.

At 0000_009Ch:

```
hab_status_t(* exit )(void)
```

Finalize and exit HAB library.

At 0000_00A0h:

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

Check target address.

At 0000_00A4h:

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

Authenticate image.

At 0000_00A8h:

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

Execute a boot configuration script.

At 0000_00ACh:

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

Execute an authentication script.

At 0000_00B0h:

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

Test an assertion against the audit log.

At 0000_00B4h:

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

Report an event from the audit log.

At 0000_00B8h:

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

Report security status.

At 0000_00BCh:

```
void(* failsafe )(void)
```

Enter failsafe boot mode which is actually the SDP.

In the development phase, with a chip in Open configuration, the boot image can use the following example code to get and display the encountered errors, during the boot process. In Closed configuration, if there is any error, the boot image is not allowed to execute, so, that code is useless.

This example can be modified to also report all warning events, by using HAB_STS_ANY in `hab_rvt_report_event`, instead of HAB_FAILURE.

```
/* -------- start of HAB API updates ------------*/
/* The following defines and structures are taken from HAB4 SIS */

/* Status definitions */
typedef enum hab_status {
        HAB_STS_ANY = 0x00,
        HAB_FAILURE = 0x33,
        HAB_WARNING = 0x69,
        HAB_SUCCESS = 0xf0
  }     hab_status_t;

/* Security Configuration definitions */
typedef enum hab_config {
    HAB_CFG_RETURN = 0x33,      /**< Field Return IC */
    HAB_CFG_OPEN = 0xf0,        /**< Non-secure IC */
    HAB_CFG_CLOSED = 0xcc       /**< Secure IC */
} hab_config_t;
```

```
/* State definitions */
typedef enum hab_state {
    HAB_STATE_INITIAL = 0x33,   /**< Initialising state (transitory) */
    HAB_STATE_CHECK = 0x55,     /**< Check state (non-secure) */
    HAB_STATE_NONSECURE = 0x66, /**< Non-secure state */
    HAB_STATE_TRUSTED = 0x99,   /**< Trusted state */
    HAB_STATE_SECURE = 0xaa,    /**< Secure state */
    HAB_STATE_FAIL_SOFT = 0xcc, /**< Soft fail state */
    HAB_STATE_FAIL_HARD = 0xff, /**< Hard fail state (terminal) */
    HAB_STATE_NONE = 0xf0,      /**< No security state machine */
    HAB_STATE_MAX
} hab_state_t;

typedef hab_status_t hab_rvt_report_event_t(hab_status_t, uint32_t, uint8_t* ,
size_t*);
typedef hab_status_t hab_rvt_report_status_t(hab_config_t *, hab_state_t *);


#define HAB_RVT_REPORT_EVENT  (*(uint32_t *) 0x000000B4)
#define hab_rvt_report_event  ((hab_rvt_report_event_t*)HAB_RVT_REPORT_EVENT)
#define HAB_RVT_REPORT_STATUS (*(uint32_t *) 0x000000B8)
#define hab_rvt_report_status ((hab_rvt_report_status_t*)HAB_RVT_REPORT_STATUS)

void display_event(uint8_t *event_data, size_t bytes)
{
        uint32_t i;
        if ((event_data) && (bytes > 0))
        {
                for (i = 0; i < bytes; i++)
                {
                        if (i == 0)
                        {
                                printf("    0x%02x", event_data[i]);
                        }
                        else if ((i % 8) == 0)
                        {
                                printf("\n    0x%02x", event_data[i]);
                        }
                        else
                        {
                                printf(" 0x%02x", event_data[i]);
                        }
                }
        }
}
```

```
int get_hab_status(void)
{
        uint32_t     index = 0;                    /* Loop index */
        uint8_t      event_data[128];              /* Event data buffer */
        size_t       bytes = sizeof(event_data);  /* Event size in bytes */
        hab_config_t config = 0;
        hab_state_t  state = 0;

        /* Check HAB status */
        if (hab_rvt_report_status(&config, &state) != HAB_SUCCESS)
        {
            printf("\nHAB Configuration: 0x%02x  HAB State: 0x%02x\n",
                   config, state);

            /* Display HAB Error events */
            while (hab_rvt_report_event(HAB_FAILURE, index, event_data, &bytes)
                    == HAB_SUCCESS)
            {
                printf("\n");
                printf("--------- HAB Event %d -----------------\n", index + 1);
                printf("event data:\n");
                display_event(event_data, bytes);
                printf("\n");
                bytes = sizeof(event_data);
                index++;
            }
        }

        /* Display message if no HAB events are found */
        else
        {
            printf("\nHAB Configuration: 0x%02x  HAB State: 0x%02x\n",
                   config, state);
            printf("No HAB Events Found!\n\n");
        }
}
/* ----------- end of HAB API updates ------------*/
```

6.2 Signing Code using Manufacturing Tool

The Freescale manufacturing tool can be used to download and execute code when in Close configuration mode.

The steps to download the code are as follows:

1. Parse the file to load in order to find the IVT and its DCD table pointer.
2. If there is a DCD table, it is loaded to the address, 0x00910000, in the OCRAM with the SDP command, DCD_WRITE. The DCD table must always be signed, which implies that this area in OCRAM must be signed.
3. The pointer to the DCD table is cleared in the IVT, in order to prevent the HAB library from processing the DCD table again during the authentication process. There is no need to re-initialize some memory, such as DDR3, when it already contains valid data.
4. The code is loaded to the boot_data address defined in the boot image structure.

It is necessary to consider the below two points in the signature process:

- The CSF description file should contain a command to sign the DCD table, located at the address, 0x00910000.

  A typical CSF authenticate data command is given below:

  ```
  [Authenticate Data]
      Verification index = 2
      Blocks = 0x77800400 0x400 0x2BC00 "u-boot-pad.bin"
  ```

  For example, the new command is as follows:

  ```
  [Authenticate Data]
      Verification index = 2
      Blocks = 0x77800400 0x400 0x2BC00 "u-boot-pad.bin", \
      Blocks = 0x00910000 0x430 0x2E0 "u-boot-pad.bin"
  ```

  The second parameter is the offset of the DCD table in the binary file, and the third parameter is the size of the table. These parameters can vary according to the memory layout defined by the user.

- Since the IVT is modified when downloading to the target, so the code must be signed with a cleared DCD pointer. However, the code must be provided with a valid pointer to allow the manufacturing tool to locate the DCD table.

  For example, a script can be used to store the DCD address, which needs to be erased before calling the cst executable at step 6 in Section 4, "Signed U-Boot Example." After that, the DCD address can be restored to continue the steps that generate the final signed binary.

Here is an example of bash script used to generate the signed code:

```
#!/bin/bash
```

```
PROG_NAME=my_code

# ${PROG_NAME} padded up to 0x2C000 where the CSF will be added later
objcopy -I binary -O binary --pad-to 0x2C000 --gap-fill=0xff ${PROG_NAME}.bin
${PROG_NAME}_padded.bin

# DCD address must be cleared for signature, as mfgtool will clear it.
./mod_4_mfgtool.sh clear_dcd_addr ${PROG_NAME}_padded.bin

# generate the signatures, certificates, … in the CSF binary
../linux/cst --o ${PROG_NAME}_csf.bin < ${PROG_NAME}.csf

# DCD address must be set for mfgtool to localize the DCD table.
./mod_4_mfgtool.sh set_dcd_addr ${PROG_NAME}_padded.bin

# gather ${PROG_NAME} + its CSF
cat ${PROG_NAME}_padded.bin ${PROG_NAME}_csf.bin > ${PROG_NAME}_tmp.bin

# padding to get a file with size like specified in the IVT
objcopy -I binary -O binary --pad-to 0x22000 --gap-fill=0xff ${PROG_NAME}_tmp.bin
${PROG_NAME}_signed.bin

# remove temporary file
rm ${PROG_NAME}_tmp.bin
```

Here is an example mod_4_mfgtool.sh script used to handle the DCD address:

```
#!/bin/bash

# DCD address must be cleared for signature, as mfgtool will clear it.
if [ "$1" == "clear_dcd_addr" ]; then
        # store the DCD address
        dd if=$2 of=dcd_addr.bin bs=1 count=4 skip=1036
        # generate a NULL address for the DCD
        dd if=/dev/zero of=zero.bin bs=1 count=4
        # replace the DCD address with the NULL address
        dd if=zero.bin of=$2 seek=1036 bs=1 conv=notrunc
fi
# DCD address must be set for mfgtool to localize the DCD table.
if [ "$1" == "set_dcd_addr" ]; then
        # restore the DCD address with the original address
        dd if=dcd_addr.bin of=$2 seek=1036 bs=1 conv=notrunc
        rm zero.bin
fi
```

6.3 Unlocking the RNG on i.MX 6 Series

When performing a secure boot that includes the CAAM hardware cryptographic accelerator on i.MX 6 Series devices in Closed configuration, the boot ROM will automatically initialize the random number generator internal to the CAAM. However, there may be situations when this is not desired, for example, when testing for RNG certification or when minimizing the time taken by the ROM in bootloader stage.

For such situations, HAB supports the Unlock CSF command to prevent the ROM from initializing the RNG. Using the Unlock CSF command does not prevent the use of the RNG, rather it simply leaves the initialization task for the OS driver.

Below is an example CSF that illustrates the use of the Unlock CSF command for the CAAM RNG. This is very similar to the Unlock CSF command for SRK revocation, discussed in Section 5.5, "SRK Revocation on i.MX 6 Series:"

```
[Header]
    Version = 4.1
    Hash Algorithm = sha256
    Engine Configuration = 0
    Certificate Format = X509
    Signature Format = CMS
[Install SRK]
    File = "../crts/SRK_1_2_3_4_table.bin"
    Source index = 0
[Install CSFK]
    File = "../crts/CSF1_1_sha256_2048_65537_v3_usr_crt.pem"
[Authenticate CSF]
[Unlock]
    Engine = CAAM
    Features = RNG
[Install Key]
    Verification index = 0
    Target index = 2
    File = "../crts/IMG1_1_sha256_2048_65537_v3_usr_crt.pem"

# Sign padded u-boot starting at the IVT through to the end with
# length = 0x2C000 (padded u-boot length) - 0x400 (IVT offset) = 0x2BC00
# This covers the essential parts: IVT, boot data and DCD.
# Blocks have the following definition:
# Image block start address on i.MX, Offset from start of image file,
# Length of block in bytes, image data file
```

```
[Authenticate Data]
    Verification index = 2
    Blocks = 0x27800400 0x400 0x2BC00 "u-boot-pad.bin"
```

Note that when i.MX 6 Series is in the Open configuration, the boot ROM does not initialize the RNG, by default. In this case, the Unlock CSF command is not necessary, and even if included in the CSF, it will have no effect.

7 Troubleshooting

7.1 SRK Authentication for i.MX 6 Series in Open Configuration

There is a known limitation about the verification of the SRK table in the ROM of i.MX 6 Series devices. In these devices, the intent was to only verify the SRK table hash, when the SRK fuse field was non-zero for Open configuration. However, for i.MX 6 Series in Open configuration, the HAB always skips the verification of the SRK table, regardless of whether the SRK fuse field has been provisioned or not.

This means that it is necessary to ensure that the SRK field is correctly programmed, prior to moving the i.MX 6 Series security configuration to Closed. It is highly recommended to use the `srktool` included as part of the CST release. The byte ordering of the SRK table hash value should be correct to ensure proper operation.

NOTE

Failing to follow the steps in provisioning the SRK hash eFuses correctly results in a device that will not boot in Closed configuration.

7.2 SRK Authentication for i.MX50 and i.MX53 in Open Configuration

On i.MX50 and i.MX53 devices, it is required to blow the SRK even for Open configuration, when developing a secure product. On these devices, HAB enforces SRK authentication even for Open configuration. This means that if the SRK fuses are not properly provisioned, the Install SRK CSF command will fail and HAB will stop processing the CSF.

8 U-Boot Modified Source Code and CSF

8.1 ./u-boot/board/freescale/mx53_loco/u-boot.lds

The modifications are highlighted in blue.

```
/*
 * (C) Copyright 2010 Freescale Semiconductor, Inc.
 *
 * See file CREDITS for list of people who contributed to this
 * project.
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License as
 * published by the Free Software Foundation; either version 2 of
 * the License, or (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston,
 * MA 02111-1307 USA
 */

OUTPUT_FORMAT("elf32-littlearm", "elf32-littlearm", "elf32-littlearm")
OUTPUT_ARCH(arm)
ENTRY(_start)
SECTIONS
{
        . = 0x00000000;

        . = ALIGN(4);
        .text      :
        {
          /* WARNING - the following is hand-optimized to fit within */
          /* the sector layout of our flash chips! */
          board/freescale/mx53_loco/flash_header.o(.text.flasheader)
          cpu/arm_cortexa8/start.o
          board/freescale/mx53_loco/libmx53_loco.a(.text)
          lib_arm/libarm.a              (.text)
          net/libnet.a                  (.text)
          drivers/mtd/libmtd.a          (.text)
          drivers/mmc/libmmc.a          (.text)
          . = DEFINED(env_offset) ? env_offset : .;
          common/env_embedded.o(.text)

          *(.text)
        }

        . = ALIGN(4);
        .rodata : { *(.rodata) }

        . = ALIGN(4);
        .data : { *(.data) }

        . = ALIGN(4);
        .got : { *(.got) }

        . = .;
        __u_boot_cmd_start = .;
        .u_boot_cmd : { *(.u_boot_cmd) }
        __u_boot_cmd_end = .;

    /* reserve this area to store HAB related data such
     * CSF commands, certificates, and signatures.
     * The offset from TEXT_BASE is chosen to leave some space
     * for u-boot to grow if necessary. It should anyway be bigger
     * than the size of the "non-secure" u-boot binary.
     */
        . = TEXT_BASE + 0x2C000;
    __hab_data_start = .;
    /* leave 8kB for the CSF */
    __csf_data = .;
        . = . + 0x2000;
    __hab_data_end = .;
    /* place this __hab_data memory region before the .bss
     * region to avoid being over written at runtime by the
     * zero initialized region below
     */

        . = ALIGN(4);
        _end_of_copy = .; /* end_of ROM copy code here */
        __bss_start = .;
        .bss : { *(.bss) }
        _end = .;
}
```

8.2 ./u-boot/board/freescale/mx53_loco/flash_header.S

Depending on the u-boot version, the device configuration is done through the recommended DCD method or the plug-in method. Even though some versions use a plug-in code to perform the device configuration, this method must not be used for a secure boot. The DCD method must be used instead, which provides the highest level of security and performance at boot.

It is relatively simple to move that configuration from the plug-in to DCD method. Typically, a single IVT is kept, the plug-in code is removed, and the configuration is done through the DCD table.

The below example shows how the source code should appear.

```
/*
 * Copyright (C) 2010-2011 Freescale Semiconductor, Inc.
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License as
 * published by the Free Software Foundation; either version 2 of
 * the License, or (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston,
 * MA 02111-1307 USA
 */

#include <config.h>
#include <asm/arch/mx53.h>

#ifdef  CONFIG_FLASH_HEADER
#ifndef CONFIG_FLASH_HEADER_OFFSET
# error "Must define the offset of flash header"
#endif

#define CPU_2_BE_32(l) \
       ((((l) & 0x000000FF) << 24) | \
        (((l) & 0x0000FF00) << 8)  | \
        (((l) & 0x00FF0000) >> 8)  | \
        (((l) & 0xFF000000) >> 24))

#define MXC_DCD_ITEM(i, addr, val)    \
dcd_node_##i:                         \
        .word CPU_2_BE_32(addr) ;      \
        .word CPU_2_BE_32(val)  ;      \
.section ".text.flasheader", "x"
        b       _start
        .org    CONFIG_FLASH_HEADER_OFFSET

ivt_header:       .word 0x402000D1 /* Tag=0xD1, Len=0x0020, Ver=0x40 */
app_code_jump_v:  .word _start
reserv1:          .word 0x0
dcd_ptr:          .word dcd_hdr
boot_data_ptr:    .word boot_data
self_ptr:         .word ivt_header
app_code_csf:     .word __csf_data
reserv2:          .word 0x0

boot_data:        .word TEXT_BASE
image_len:        .word __hab_data_end - TEXT_BASE
plugin:           .word 0x0

dcd_hdr:          .word 0x40A001D2 /* Tag=0xD2, Len=51*8 + 4 + 4, Ver=0x40 */
write_dcd_cmd:    .word 0x049C01CC /* Tag=0xCC, Len=51*8 + 4, Param=4 */

/* DCD */
MXC_DCD_ITEM(1, IOMUXC_BASE_ADDR + 0x554, 0x00300000)
MXC_DCD_ITEM(2, IOMUXC_BASE_ADDR + 0x558, 0x00300040)
MXC_DCD_ITEM(3, IOMUXC_BASE_ADDR + 0x560, 0x00300000)
MXC_DCD_ITEM(4, IOMUXC_BASE_ADDR + 0x564, 0x00300040)
MXC_DCD_ITEM(5, IOMUXC_BASE_ADDR + 0x568, 0x00300040)
MXC_DCD_ITEM(6, IOMUXC_BASE_ADDR + 0x570, 0x00300000)
MXC_DCD_ITEM(7, IOMUXC_BASE_ADDR + 0x574, 0x00300000)
MXC_DCD_ITEM(8, IOMUXC_BASE_ADDR + 0x578, 0x00300000)
MXC_DCD_ITEM(9, IOMUXC_BASE_ADDR + 0x57c, 0x00300040)
MXC_DCD_ITEM(10, IOMUXC_BASE_ADDR + 0x580, 0x00300040)
MXC_DCD_ITEM(11, IOMUXC_BASE_ADDR + 0x584, 0x00300000)
MXC_DCD_ITEM(12, IOMUXC_BASE_ADDR + 0x588, 0x00300000)
MXC_DCD_ITEM(13, IOMUXC_BASE_ADDR + 0x590, 0x00300040)
MXC_DCD_ITEM(14, IOMUXC_BASE_ADDR + 0x594, 0x00300000)
MXC_DCD_ITEM(15, IOMUXC_BASE_ADDR + 0x6f0, 0x00300000)
MXC_DCD_ITEM(16, IOMUXC_BASE_ADDR + 0x6f4, 0x00000000)
MXC_DCD_ITEM(17, IOMUXC_BASE_ADDR + 0x6fc, 0x00000000)
MXC_DCD_ITEM(18, IOMUXC_BASE_ADDR + 0x714, 0x00000000)
MXC_DCD_ITEM(19, IOMUXC_BASE_ADDR + 0x718, 0x00300000)
MXC_DCD_ITEM(20, IOMUXC_BASE_ADDR + 0x71c, 0x00300000)
MXC_DCD_ITEM(21, IOMUXC_BASE_ADDR + 0x720, 0x00300000)
MXC_DCD_ITEM(22, IOMUXC_BASE_ADDR + 0x724, 0x04000000)
MXC_DCD_ITEM(23, IOMUXC_BASE_ADDR + 0x728, 0x00300000)
MXC_DCD_ITEM(24, IOMUXC_BASE_ADDR + 0x72c, 0x00300000)
MXC_DCD_ITEM(25, ESDCTL_BASE_ADDR + 0x088, 0x35343535)
MXC_DCD_ITEM(26, ESDCTL_BASE_ADDR + 0x090, 0x4d444c44)
MXC_DCD_ITEM(27, ESDCTL_BASE_ADDR + 0x07c, 0x01370138)
MXC_DCD_ITEM(28, ESDCTL_BASE_ADDR + 0x080, 0x013b013c)
MXC_DCD_ITEM(29, ESDCTL_BASE_ADDR + 0x018, 0x00011740)
MXC_DCD_ITEM(30, ESDCTL_BASE_ADDR + 0x000, 0xc3190000)
MXC_DCD_ITEM(31, ESDCTL_BASE_ADDR + 0x00c, 0x9f5152e3)
MXC_DCD_ITEM(32, ESDCTL_BASE_ADDR + 0x010, 0xb68e8a63)
MXC_DCD_ITEM(33, ESDCTL_BASE_ADDR + 0x014, 0x01ff00db)
MXC_DCD_ITEM(34, ESDCTL_BASE_ADDR + 0x02c, 0x000026d2)
MXC_DCD_ITEM(35, ESDCTL_BASE_ADDR + 0x030, 0x009f0e21)
MXC_DCD_ITEM(36, ESDCTL_BASE_ADDR + 0x008, 0x12273030)
MXC_DCD_ITEM(37, ESDCTL_BASE_ADDR + 0x004, 0x0002002d)
MXC_DCD_ITEM(38, ESDCTL_BASE_ADDR + 0x01c, 0x00008032)
MXC_DCD_ITEM(39, ESDCTL_BASE_ADDR + 0x01c, 0x00008033)
MXC_DCD_ITEM(40, ESDCTL_BASE_ADDR + 0x01c, 0x00028031)
MXC_DCD_ITEM(41, ESDCTL_BASE_ADDR + 0x01c, 0x052080b0)
MXC_DCD_ITEM(42, ESDCTL_BASE_ADDR + 0x01c, 0x04008040)
MXC_DCD_ITEM(43, ESDCTL_BASE_ADDR + 0x01c, 0x0000803a)
MXC_DCD_ITEM(44, ESDCTL_BASE_ADDR + 0x01c, 0x0000803b)
MXC_DCD_ITEM(45, ESDCTL_BASE_ADDR + 0x01c, 0x00028039)
MXC_DCD_ITEM(46, ESDCTL_BASE_ADDR + 0x01c, 0x05208138)
MXC_DCD_ITEM(47, ESDCTL_BASE_ADDR + 0x01c, 0x04008048)
MXC_DCD_ITEM(48, ESDCTL_BASE_ADDR + 0x020, 0x00005800)
MXC_DCD_ITEM(49, ESDCTL_BASE_ADDR + 0x040, 0x04b80003)
MXC_DCD_ITEM(50, ESDCTL_BASE_ADDR + 0x058, 0x00022227)
MXC_DCD_ITEM(51, ESDCTL_BASE_ADDR + 0x01C, 0x00000000)

#endif
```

8.3 U-Boot CSF Example for i.MX53

This example assumes that TEXT_BASE = *start = 7780_0000h, and that the size of the image to be signed goes from the IVT up to the CSF: length = *csf - ivt_addr = TEXT_BASE + 0002_C000h - (TEXT_BASE + 0400h) = 0002_BC00h.

```
[Header]
    Version = 4.0
    Hash Algorithm = sha256
    Engine Configuration = 0
    Certificate Format = X509
    Signature Format = CMS
[Install SRK]
    File = "../crts/SRK_1_2_3_4_table.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"
# Sign padded u-boot starting at the IVT through to the end with
# length = 0x2C000 (padded u-boot length) - 0x400 (IVT offset) = 0x2BC00
# This covers the essential parts: IVT, boot data and DCD.
# Blocks have the following definition:
#    Image block start address on i.MX, Offset from start of image file,
#    Length of block in bytes, image data file
[Authenticate Data]
    Verification index = 2
    Blocks = 0x77800400 0x400 0x2BC00 "u-boot-pad.bin"
```

9 Revision History

Table 1 provides a revision history for this application note.

Table 1. Document Revision History

Rev. Number	Date	Substantive Change(s)
Rev. 0	09/2012	Initial public release.

Document Number: AN4581
Rev. 0
09/2012