Freescale Semiconductor
Application Note

Document Number: AN4547
Rev. 0, 10/2012

Secure Boot on i.MX25, i.MX35, and i.MX51 using HABv3

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 3 (HABv3). 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 HABv3. 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 HABv3: i.MX25, i.MX35, and i.MX51. On other processors, such as i.MX28, i.MX50, i.MX53, and i.MX 6 Series, secure boot is performed using HABv4. Secure boot on i.MX28 is documented in *Secure Boot with i.MX28 HAB v4* application note (AN4555). Secure boot on i.MX50, i.MX53, and i.MX 6 Series is documented in *Secure Boot on i.MX50, i.MX53, and i.MX 6 Series using HAB4* application note (AN4581).

NOTE

Secure boot using HAB is not 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.
- 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.
- 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 HABv3.
- 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.
- WTLS: Wireless Transport Layer Security. HABv3 uses certificates defined by this standard.

- 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.MX25 Reference Manual*, *i.MX35 Reference Manual*, and *i.MX51 Reference Manual*, available on freescale.com for free download. Search for IMX25RM, IMX35RM, or MCIMX51RM.
- *i.MX258 Security Reference Manual*, *i.MX358 Security Reference Manual*, and *i.MX51 Security Reference Manual*, available on freescale.com for free download. Search for MCIMX258SRM, IMX358SRM, or IMX51SRM.
- *HAB CST User Guide* available in the *Code Signing Tool* package downloadable on freescale.com. Search for IMX_CST_TOOL.
- *i.MX25 IC Identification Module (IIM) Fusebox* and *i.MX35 IC Identification Module (IIM) Fusebox* application notes, available on freescale.com for free download. Search for AN3682 (i.MX25) or AN3652 (i.MX35).

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, 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 (for example, RTIC) to improve boot performance. HABv3 makes exclusive use of the RSA algorithm, with all signatures following the PKCS#1 version 1.5 format, and all certificates following the WTLS format.

For more details, see the reference manual of the processor you are using, 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 HABv3 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.

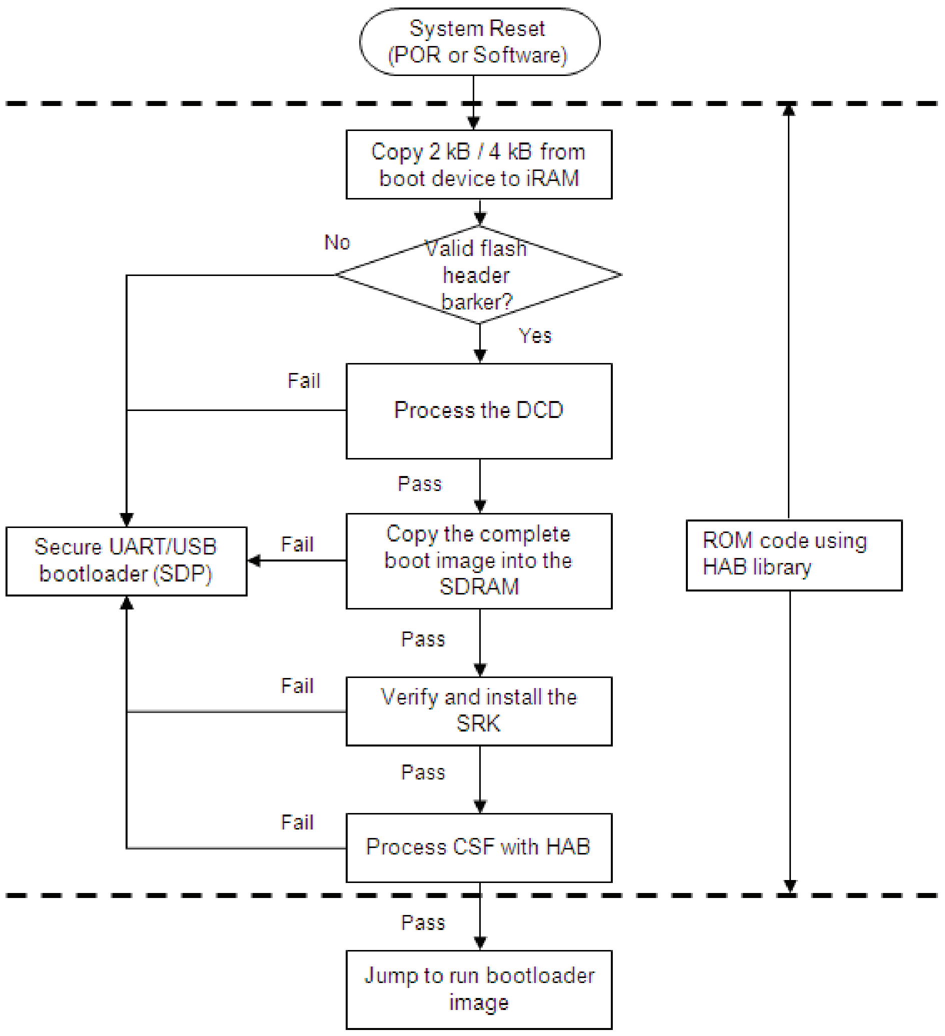
System Reset (POR or Software)
Copy 2 kB / 4 kB from boot device to iRAM
Valid flash header barker?
No
Yes
Process the DCD
Fail
Pass
Copy the complete boot image into the SDRAM
Fail
Pass
Verify and install the SRK
Fail
Pass
Process CSF with HAB
Fail
Pass
Jump to run bootloader image
Secure UART/USB bootloader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

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.

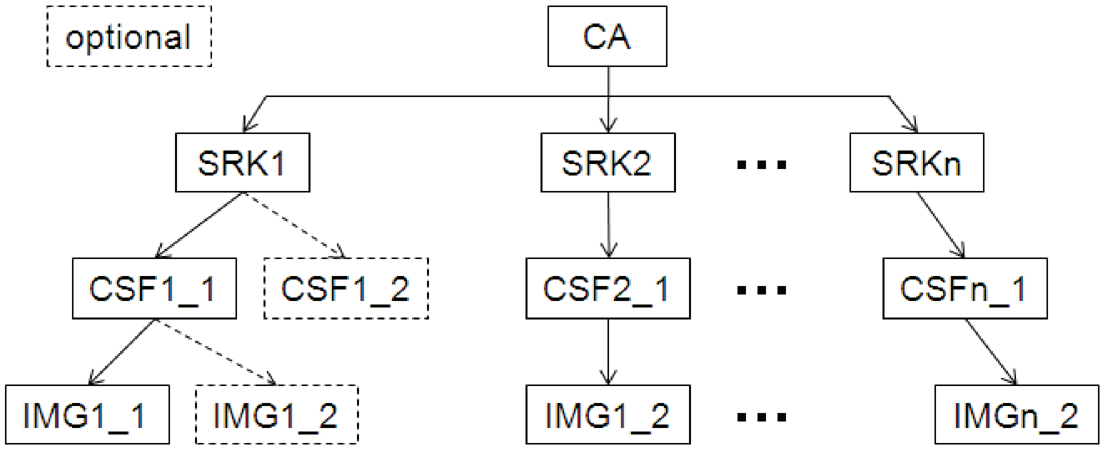

Figure 2. HABv3 Enabled Devices Typical PKI Tree

For further information on the PKI tree used for HABv3, 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. It illustrates how the structures are organized with each other, and what are the dependencies.

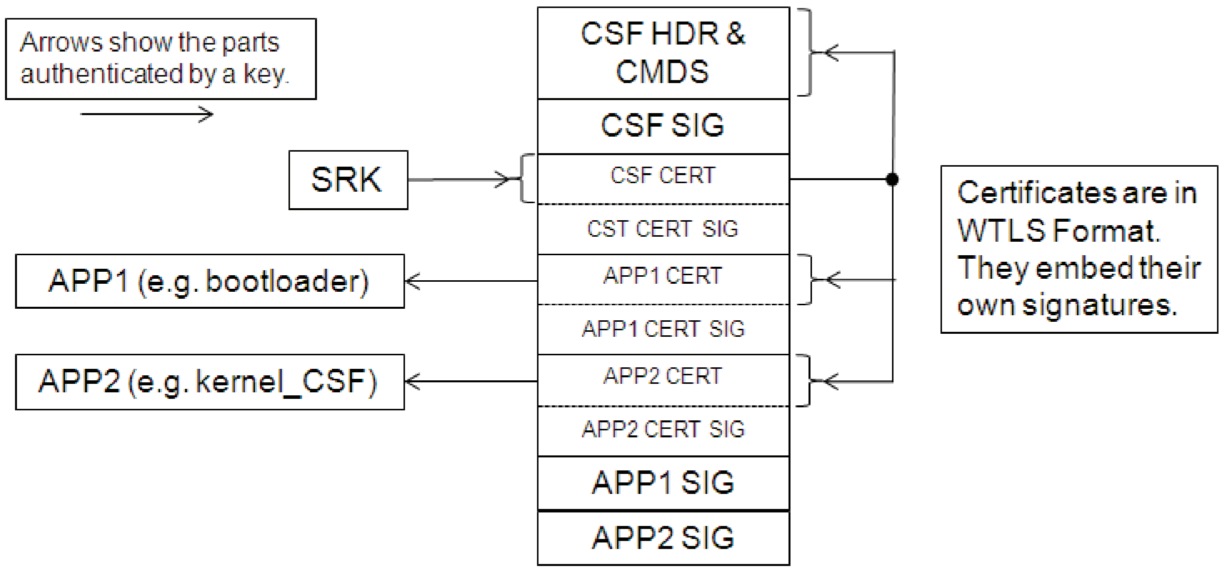

Figure 3. HABv3 Enabled Devices Detailed Authentication Chain

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 and comparing the result with a pre-computed hash that is provisioned in OTP fuses. This ensures that the integrity of the SRK 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 WTLS certificates.

The arrows show the authentication flow. For example, the CSF key is used to authenticate both the CSF and the application keys. The special nature of the arrow from CSF to Application Key is explained below.

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 application key can certify multiple applications. This means that those 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 an authentic one. It is, therefore, prudent to plan the CSF so it does not need to be changed 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 the application key through the CSF is a little different to the other links shown in the diagram. In order to provide better key separation, the application key is bound to the CSF using a hash fingerprint embedded in the CSF. This means that it is not possible to use the CSF with other application keys, even if they have been certified by the same CSF key.

Thirdly, each key type can authenticate only specific data types. The SRK can authenticate only the CSF key, not CSFs or applications. A CSF key can authenticate only CSFs and application keys. An application key can authenticate only applications or further application keys.

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:

- Flash header: A table of pointers used by the boot ROM to locate other required data components. See the System Boot chapter of the reference manual for more details.
- Device Configuration Data (DCD): A list of registers that the boot ROM will program with the provided data to perform an early initialization of the system. DCD is typically used to initialize the SDRAM. See the System Boot chapter of the reference manual for more details.
- 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.
- SRK: The root public key used in the authentication procedure.

3.1.1 Flash Header

The flash header is a mandatory part of the boot image, and its structure can be defined as follows:

```
typedef struct
{
    uint32_t            *app_code_jump_vector;
    uint32_t             app_code_barker;
    uint32_t            *app_code_csf;
    DCD_T               **dcd_ptr_ptr;
    hab_rsa_public_key  *super_root_key;
    DCD_T               *dcd_ptr;
    uint32_t            *app_dest_ptr;
} image_hdr_t;
```

Where:

- `uint32_t` is a type representing a 32 bit unsigned integer.
- `DCD_T` is a structure that defines the device configuration table (see Section 3.1.2, "Device Configuration Data,").
- `hab_rsa_public_key` is a structure that defines the SRK.

The elements of the flash header are described below:

- `*app_code_jump_vector`: Entry point for the image. Points to the address of the first instruction of the boot image.
- `app_code_barker`: A value that the ROM uses to determine that the boot device has been programmed. This must be 0000 00B1h.
- `*app_code_csf`: Points to the CSF data. This data is used by HAB to verify that the application is authentic. When performing a non-secure boot with HAB_TYPE=Engineering configuration, this field must be set to NULL.
- `**dcd_ptr_ptr`: This pointer must be set to point to the real pointer `*dcd_ptr` also contained in the flash header structure.
- `*super_root_key`: Pointer to the SRK structure. When performing a non-secure boot with HAB_TYPE=Engineering configuration, this field must be set to NULL.
- `*dcd_ptr`: Pointer to the Device Configuration Data (DCD) table.
- `*app_dest_ptr`: Pointer used by the ROM for most boot devices. During boot, the ROM copies the image data from boot device memory to destination RAM. This pointer defines the location of destination memory where the ROM copies the application.

The flash header is a block of data that must reside at a specific fixed address that depends on the boot device. The fixed addresses corresponding to the boot devices and additional details on the flash header are 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.

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 HAB type to which this CSF is bound (comparison with the fuse value HAB_TYPE: engineering/non-secure or production/secure).
- The HAB customer code to which this CSF is bound (comparison with the fuse value HAB_CUS).
- The UID (if any) to which this CSF is bound (comparison with the fuse value of UID).
- The PKI tree to be used in authentication operations.
- The physical memory regions to be authenticated, along with the authentication method and reference data.
- Additional device configuration operations.

The 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.

The certificates for the PKI tree and the associated digital signatures are attached to the CSF. They provide the reference data for authenticating the image data.

With HAB, multiple 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:

- Two for RTIC
- Eight for SAHARA (supported on i.MX51 only)
- Sixteen for the software implementation included in HAB

3.1.4 Super Root Key Structure

The private key of the SRK is held securely by the CST off-chip, and is used to certify public keys at the next level (CSF level) in the HAB PKI tree. Confidentiality and access control for the private keys are paramount to ensuring that only authentic images can be run on secure production devices—these properties are protected by the CST tool and the access policies set by device manufacturers.

The public key of the SRK is embedded within the image to be authenticated. Confidentiality of the SRK public key is not important, but data integrity and authenticity are crucial, which is why OTP fuses are used to keep a reference finger print or hash of the SRK.

At boot, the SRK structure is loaded from the boot device. A SHA-256 digest is computed from that structure and compared to the reference value obtained from fuses. It is essential that this test is passed to continue with the boot process.

The SRK data must conform to the structure shown below:

```
typedef struct

{

    uint8_t   rsa_exponent[MAX_EXP_SIZE]; /* RSA public exponent */

    uint8_t   *rsa_modulus;               /* RSA modulus pointer */

    uint16_t exponent_size;               /* Exponent size in bytes */

    uint16_t modulus_size;                /* Modulus size in bytes */

    bool init_flag;                  /* Reserved- should be FALSE (0) */

} hab_rsa_public_key;
```

Where:

- `uint8_t` is an 8-bit unsigned integer.
- `uint16_t` is a 16-bit unsigned integer.
- `bool` is an 8-bit flag indicating TRUE (1) or FALSE (0).
- `rsa_exponent`: Exponent of the RSA key. The maximum exponent size is 4.
- `rsa_modulus`: Pointer to the RSA key modulus.
- `exponent_size`: Exponent size in bytes. Must be less than or equal to the maximum exponent size.
- `modulus_size`: Modulus size in bytes. Must be greater than or equal to 128 bytes and less than or equal to 256 bytes.

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

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 HABv3, the image must contain a correctly formatted flash header with a valid barker code 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 iRAM. 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 flash header that is placed at a known offset. The ROM code starts by comparing the barker code against the expected value to detect that a boot image is present. If the barker code is not validated, then, boot ROM execution enters the SDP. This signifies that the boot device was not provided.

If the barker code is validated, then, the pointer to the DCD table `*dcd_ptr` is extracted and the DCD commands are processed. Note that when the processor is in secure boot state, only memories and certain peripherals are programmable by the DCD at this stage. The complete list is provided in the System Boot chapter of the reference manual.

If there are no errors at this point, then, the ROM reads the total image size in bytes. This value must be placed immediately after the end of the DCD table. It is used by the ROM to determine how many bytes of the image to copy from the boot device to the destination memory, pointed to by `*app_dest_ptr`.

Keep in mind that all the pointers included in the flash header are with respect to the final destination in memory `*app_dest_ptr`.

At this stage, the rest of the boot process depends on the boot configuration—Engineering or Production—which is determined by the HAB_TYPE fuse field.

3.2.2 Non-Secure Boot Image Layout

When performing a non-secure boot with the HAB_TYPE fuse field set to Engineering, providing the CSF and SRK data as part of the image is optional. If this data is not provided, NULL pointers must be used for CSF and SRK entries in the flash header to make the boot ROM skip the steps related to security. Eventually, the ROM code simply jumps into the boot code pointed to by `*app_code_jump_vector`.

If the CSF and SRK are provided, all HAB failures are not considered fatal and the boot process is allowed to continue. This is the final configuration for Non-Secure products as described above. The Engineering configuration should also be used for development purposes of secure products where CSFs and other data components for secure boot can be debugged.

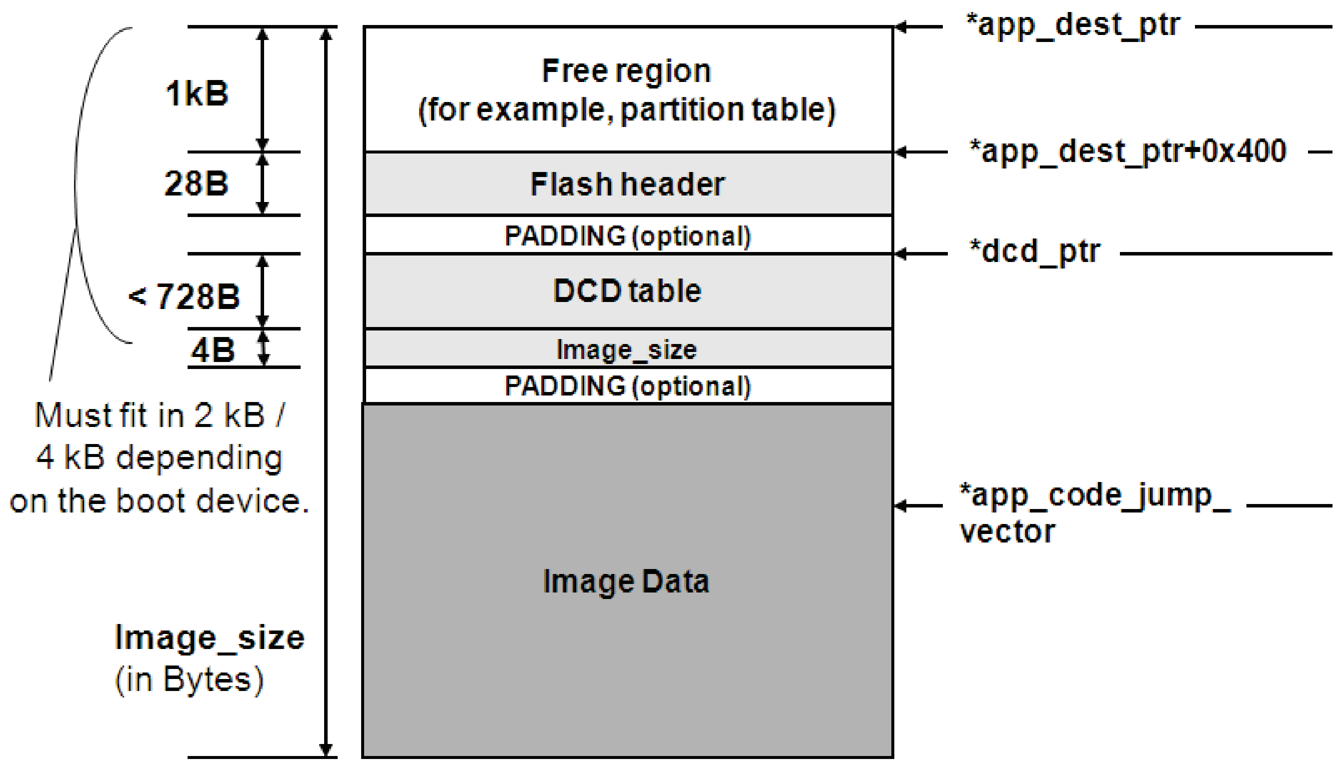

Figure 4. Typical Memory Layout of an Unsigned Image

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

The free region is a region of the boot device that is not used by the boot ROM. The size of the free region and the flash header 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 HAB_TYPE fuse field tells HAB whether the device is to boot securely or not. By setting this field to Production configuration, the i.MX processor will only allow a properly signed image to execute. The CSF and SRK data components are mandatory and must be included in the image, along with valid pointers in the flash header structure.

As described previously, the ROM locates the SRK structure with the help of the `*super_root_key` pointer. It is important to have the SRK tied to the processor to avoid it being replaced by another non-trusted key. Therefore, the ROM computes a SHA-256 hash of the concatenated exponent and modulus of the SRK public key. The result is compared to the reference value blown into the OTP fuses. After the SRK is verified, the ROM locates the CSF data by using the `*csf_ptr` pointer from the flash header.

This step is where the application authentication takes place. The principal steps (not necessarily in order) involved in processing the CSF are:

1. If the UID record in the CSF indicates a specific UID, check if it matches the UID of the processor.
2. Check if the TYPE record in the CSF matches the processor HAB type.
3. Check if the CODE record in the CSF matches the processor customer code.
4. Verify the CSF key certificate using the SRK.
5. Verify the CSF signature using the CSF key.
6. Verify the application key certificates using the CSF key.
7. Verify the application signature(s) using the application keys.
8. Perform any device configuration operations specified in the CSF.

Note that not all steps apply to every CSF.

The HAB offers two different application signature verification options:

- Generic signature verification: A digital signature across the given image region is verified using a specified application key. The image and signature can be loaded onto multiple devices.
- Bound signature verification: A digital signature, across both the given image region and the processor UID, is verified using a specified application key. With this processor-specific signature, the image is bound to a single device only, and cannot be executed on other devices.

Note that the bound signature verification option applies to the given image region, whereas the specific UID record in the CSF applies to the CSF as a whole. The two methods of binding to a device can be used independently or together. For example, to bind a boot image to a batch of field-test devices, it would be possible to use a CSF containing a bound signature verification command for the application. The same CSF can be used for all units, but the application signature would have to be generated separately for each unit.

Alternatively, to unlock a single production device to allow application development without repeated code-signing, a processor-specific CSF can be generated that verifies the bare minimum amount of code, leaving the external memory essentially free for arbitrary application loads.

If signature verifications are successful, the ROM code jumps into the boot code pointed to by `*app_code_jump_vector`.

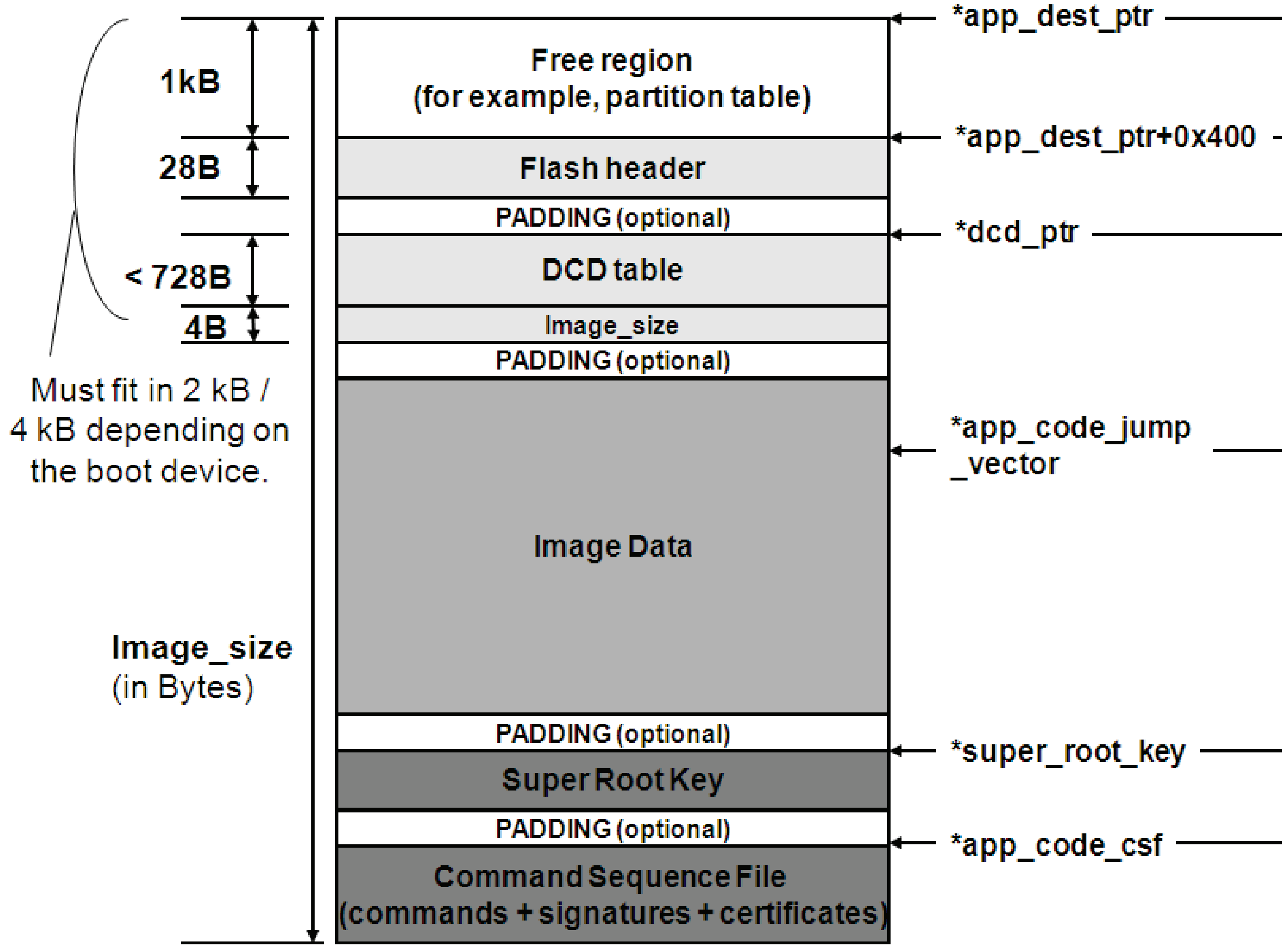

Figure 5. Typical Memory Layout of a Signed Image

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

The free region is a region of the boot device that is not used by the boot ROM. The size of the free region and the flash header offset vary with the boot device. See the System Boot chapter of the reference manual for more details.

NOTE

The HAB requires that the flash header and DCD table must be covered by a digital signature. It is also recommended to cover as much of the image as possible by a digital signature.

3.3 Generating CSF and SRK Data

3.3.1 Generating Command Sequence File (CSF) Data

The CSF contains all the commands that the ROM will execute during the secure boot. These commands instruct HAB on which memory areas of the image to authenticate, which keys to 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 are attached to the CSF.

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 output from the CST is a binary block of data that can readily be attached to an unsigned image turning it into a signed image. As mentioned already, this data must be located at the address pointed to by `*csf_ptr`, in the flash header data structure.

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.

NOTE

For the i.MX25, there is a specific CSF command that can be used to unlock the DryIce module, allowing image software to configure DryIce. An example of CSF with that specific command is provided in Section 9, “i.MX25 CSF Example with Unlock DryIce Command.”

3.3.2 Generating Super Root Key (SRK) Data

The SRK is the top level key used for the code signing process with HAB. The SRK is generated by the scripts included with the Freescale reference CST tool. The following command, when run from the keys directory of the reference CST, will generate the complete PKI tree needed for code signing with HAB:

```
./hab3_pki_tree.sh
```

The private keys are located in the keys directory and care must be taken to ensure that the confidentiality of these keys is maintained. However, for assembling a signed image, the contents of the crts directory is of more interest. After completing the above command, the script will have generated one or more C language source files, containing the SRK data in the format required by HAB. For example:

```
./crts/SRK1_sha256_2048_65537_v3_ca_crt.der.c
```

Depending on the number of SRKs the script is asked to generate, there may be more than one C file generated. That is, there will be one C file for each SRK generated, but only one SRK may be used in an image. This code must then be compiled and linked to the appropriate address that is defined by the `*super_root_key` pointer in the flash header.

The data structure for the SRK is described in Section 3.1.4, "Super Root Key Structure." To assist with including the SRK data structure in the image software build process, the reference CST includes the following file:

```
./hdr/hab_super_root.h
```

There are two methods in which the SRK can be built into the image. These methods are described in the sections that follow.

3.3.2.1 Building SRK with Boot Image

Starting from C code is the most convenient method to integrate the SRK C file with the other boot image source files. That is, the code is compiled with the rest of the image, and the linker is used to resolve the pointer address of the SRK in the flash header.

Here is an example based on u-boot, but the method should be adapted for the bootloader being used. The steps are as follows:

1. Copy the two files mentioned above, SRK1_sha256_2048_65537_v3_ca_crt.der.c and hab_super_root.h, to the board directory, such as ./board/freescale/mx35_3stack.
2. Add the SRK source file in the list of objects to build, in the Makefile of the same directory:

```
COBJS := mx35_3stack.o SRK1_sha256_2048_65537_v3_ca_crt.der.o
```

3. Change the SRK pointer from 0 to the following, in the flash_header.S file:

```
super_root_key: .long hab_super_root_key
```

By following this strategy, the SRK is converted to binary and integrated automatically during the build of the boot image. All the addresses are resolved during the linking stage. The main drawback is that when the SRK changes, the whole image must be re-built.

NOTE

If the SRK is in-built with the boot image, it does not need to be manually assembled with the image as described in Section 3.4, "Assembling CSF and SRK with Boot Image," and Section 4, "Signed U-Boot Example."

3.3.2.2 Building SRK Manually

It is also possible to manually build the SRK from the C code generated by the srktool. Any compiler for ARM core can be used for this purpose. This example uses the CodeSourcery "arm-none-eabi" cross compiler tool chain. The steps are as follows:

1. Compile code. This requires access to the header file hab_super_root.h, hence the include option (-I../hdr).

```
arm-none-eabi-gcc -c SRK1_sha256_2048_65537_v3_ca_crt.der.c -I../hdr -o
SRK1_sha256_2048_65537_v3_ca_crt.der.o
```

2. Link it at address of the first element in the SRK structure. The generated SRK C file contains only two constants: SRK modulus + SRK structure. When compiled and linked, the modulus is placed before the structure. So, it is necessary to take into account that offset within the SRK binary.

   The link address should be `Ttext` = `*super_root_key` - `modulus_size`.

   In this example, the address is `Ttext` = 0x83F3D100 - 2048-bit/8.

```
arm-none-eabi-ld SRK1_sha256_2048_65537_v3_ca_crt.der.o -Ttext 0x83F3D000 -o
SRK1_sha256_2048_65537_v3_ca_crt.der.elf
```

   Ignore any warning about the entry point (which is expected as there is no entry point in this code).

3. Generate a binary from the elf file. That binary will have to be combined with the application at the defined offset `*super_root_key`.

```
arm-none-eabi-objcopy -O binary SRK1_sha256_2048_65537_v3_ca_crt.der.elf
SRK1_sha256_2048_65537_v3_ca_crt.der.bin
```

The next section describes how the generated binary can be merged with the boot image.

NOTE

The *HAB CST User Guide* describes how to program the SRK hash value in the OTP fuses.

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. Section 7, “Development and Debugging Tips,” includes an example to test the fused SRK value prior to blowing the fuses.

3.4 Assembling CSF and SRK with Boot Image

From the previous steps, a binary representing the CSF data (Section 3.3.1, “Generating Command Sequence File (CSF) Data,”) and a binary of the SRK (Section 3.3.2, “Generating Super Root Key (SRK) Data,”) were generated. This section explains how to assemble them to create a signed image. The purpose is to create a signed boot image that is organized as shown in Figure 5 and Figure 6.

The initial size of the unsigned boot image is `initial_Image_Size` as shown in Figure 4. It goes from the start address to the end of the image data.

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

- `boot_img.bin`: This is the unsigned boot image binary with the flash header configured for secure boot.
- `boot_img-signed.bin`: This is the signed boot image binary with the CSF and SRK included.
- `initial_Image_Size`: This is the image size of the unsigned boot image.
- `srk_data.bin`: This is the SRK binary, such as “SRK1_sha256_2048_65537_v3_ca_crt.der.bin.”

- `csf_data.bin`: This is the CSF binary.

1. First step is to pad the boot image from `initial_Image_Size` to `*super_root_key` address. For example, if `*super_root_key` = 9782_a000h and `*app_dest_ptr` = 9780_0000h, the binary has to be padded up to offset `*super_root_key` - `*app_dest_ptr` = 0002_A000h.

```
objcopy -I binary -O binary --pad-to 0x2A000 --gap-fill=0xff boot_img.bin
boot_img-pad.bin
```

2. Now, the SRK binary can be merged with the padded boot image. This makes it localized at the defined offset, with the help of flash header pointers:

```
cat boot_img-pad.bin srk_data.bin > boot_img_with_srk.bin
```

3. Then, the next step is to pad the boot image that included the SRK to `*csf_ptr` offset. For example, if `*csf_ptr` = 9782_A120h and `*app_dest_ptr` = 9780_0000h, the binary has to be padded up to offset `*csf_ptr` - `*app_dest_ptr` = 0002_A120h.

```
objcopy -I binary -O binary --pad-to 0x2A120 --gap-fill=0xff
boot_img_with_srk.bin boot_img_with_srk-pad.bin
```

4. Finally, the CSF binary can be merged with the previously padded boot image. This makes it localized at the defined offset, with the help of flash header pointers:

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

This signed boot image can now be used to boot an i.MX processor in secure mode.

4 Signed U-Boot Example

U-boot is a bootloader that is commonly used to boot a Linux device. It is provided in the Freescale Linux BSP. There are multiple i.MX processors and boards that can be used to illustrate a secure u-boot, but this example will focus on the i.MX51 Babbage board. The principle applied is exactly same when using other i.MX ICs or boards.

The signed u-boot will have the following memory map:

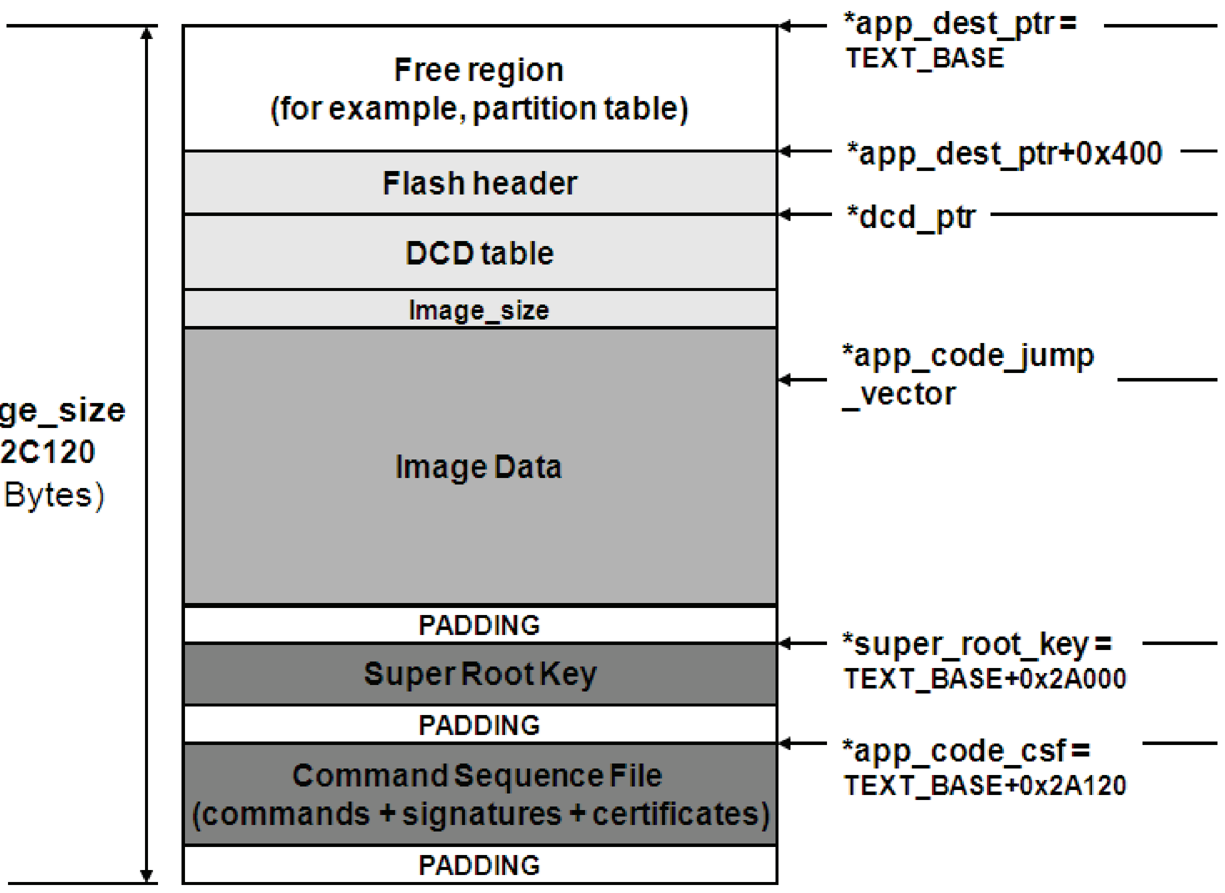
Free region
(for example, partition table)
Flash header
DCD table
Image_size
Image Data
PADDING
Super Root Key
PADDING
Command Sequence File
(commands + signatures + certificates)
PADDING
Image_size = 0x2C120 (in Bytes)
*app_dest_ptr = TEXT_BASE
*app_dest_ptr+0x400
*dcd_ptr
*app_code_jump _vector
*super_root_key = TEXT_BASE+0x2A000
*app_code_csf = TEXT_BASE+0x2A120

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

TEXT_BASE is the pointer, `*app_dst_ptr`, which points to the location where the code is copied from the boot device. This variable is defined in the file:

```
./u-boot/board/freescale/mx51_bbg/config.mk
```

If the boot is performed from an SD card, the offset of the flash header must equal 0000_0400h, so the flash header is placed at TEXT_BASE + 0000_0400h.

Knowing the size of the flash header (28 bytes), the DCD is necessarily located at `*dcd_ptr` = TEXT_BASE + 0000_041Ch.

This layout is chosen based on the unsigned u-boot binary size. By default, the size is around 160 kB, so the offset for the CSF/SRK data must be located beyond that limit.

Using an offset of 0002_A000h, which corresponds to 168 kB, leaves some space for u-boot to grow, if necessary. This empty area between the end of u-boot code and the `*super_root_key` is padded as explained previously.

This places the SRK at the offset `*super_root_key` = TEXT_BASE + 0002_A000h.

The largest keys size supported by HAB is 2048-bit RSA key with an exponent of 65537. This generates a structure of 272 bytes. The space reserved for the SRK data is fixed at 288 bytes to include some padding for convenience.

This places the CSF at the offset `*csf_ptr` = TEXT_BASE + 0002_A120h.

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 IMAGE_SIZE 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 IMAGE_SIZE bytes of data.

The following are the steps to generate the layout defined above:

1. Modify the linker script to reserve some space for the CSF and SRK 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 flash header can use them as pointers. The file is located at:

   ```
   ./u-boot/board/freescale/mx51_bbg/u-boot.lds
   ```

   The modified linker script is in Section 10.1, "./u-boot/board/freescale/mx51_bbg/u-boot.lds."
2. The flash header should be modified with the updated image size and the CSF and SRK pointers. Also, a second flash header is created at offset 0h to ease the test and development when using the Serial Download Protocol. That additional flash header is not needed for a device boot, but required when the code is loaded by using the SDP as described in the Section 7.2, "Using a Flash Header for Device Boot and SDP." The file is located at:

   ```
   ./u-boot/board/freescale/mx51_bbg/flash_header.S
   ```

   The modified linker script is in Section 10.2, "./u-boot/board/freescale/mx51_bbg/flash_header.S."
3. U-boot must be built to generate u-boot.bin. See the u-boot documentation for the detailed procedure.

   ```
   make mx51_bbg_config

   make
   ```
4. The obtained binary has to be padded up to offset 0000_2A00h, as shown in Section 3.4, "Assembling CSF and SRK with Boot Image:"

   ```
   objcopy -I binary -O binary --pad-to 0x2A000 --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. See this user guide for all the details about the pre-requisites and for a more complete step by step procedure.

   ```
   /install_path/keys/hab3_pki_tree.sh
   ```
6. The code can be signed, and the CSF instruction file is available in Section 10.3, "U-Boot CSF Example.

```
/install_path/linux/cst --output csf_u-boot.bin < csf_u-boot.txt
```

7. Finally, by following the step by step instructions from (2) provided in Section 3.4, “Assembling CSF and SRK with Boot Image,” the final signed binary image can be created. boot_img-pad.bin = u-boot-pad.bin and csf_data.bin= csf_u-boot.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.

This is the recommended solution for i.MX51, though it should be functional for i.MX25 and i.MX35 without guarantee and support.

5.1.2 Solution 2

Use u-boot commands with <bank> and <row> in hexadecimal format:

```
iim read <bank> <row>

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

This solution works for all i.MX processors covered by this application note, but requires using the latest u-boot code from the following git repository:

`http://opensource.freescale.com/git?p=imx/uboot-imx.git;a=summary`

5.1.3 Solution 3

Use the sample code provided in the application note, AN3682 for the i.MX25 and AN3652 for the i.MX35. Note that the controller managing the fuses (IIM) is the same for all i.MX variants covered in this note; therefore, this example code is compatible for all these chips, except that the registers addresses and fuse maps are different.

5.2 Fuse Configuration Recommendations

During production, it is suggested to change the HAB_TYPE of the chip to Production configuration only after the programming, provisioning, validation, and all other custom operations are done. Once a boot image is configured for secure boot, it must be signed and authenticated correctly prior its execution, regardless of the boot mode (SDP or internal). It is much easier to fix any encountered issues while a chip is still in Engineering configuration.

Some 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.MX25, once the SRK_HASH[255:0] is programmed, the SRK_LOCK96 and SRK_LOCK160 fuses must be programmed to disable any modification to the reference digest for the SRK.

See the i.MX fuse map and/or IIM Fusebox application notes mentioned in Section 1.5, “References,” for the list of available protections with lock fuses.

6 Using HAB API in Application Code

The trusted chain of executed code can be continued past the initial image launched by the boot ROM. The HAB API offers the possibility continuing the secure boot chain by authenticating the next software component that is going to be executed. For example, it could be the kernel of the operating system.

For that purpose, three HAB functions are available, and can be considered trustworthy as they are located in ROM and cannot be changed:

- `hab_result hab_csf_check(uint8_t csf_count, uint32_t *csf_list)`

  Performs integrity checks on software in programmable memory as instructed by CSFs.
- `hab_result hab_assert_verification(uint8_t *block_start, uint32_t block_length)`

  Determines if a block of data lies within the regions of the pre-authenticated block or the regions verified during CSF Check. This function should be run only after a CSF Check.
- `void pu_irom_boot_decision(void)`

  Indicates the entry point for the serial downloader mode.

Note that these APIs operate on physical addresses, which implies that the MMU must remain disabled, or the mapping of the ROM code and iRAM must be flat.

See the System Boot chapter of the reference manual for details about the HAB API functions. In particular, the address of these functions can be retrieved from there.

The following C source code is a usage example written for the i.MX25. Note that only the functions and variables addresses are different for the i.MX35 or i.MX51, the rest of the code remains the same.

```
/***************************************************************************

*

* Copyright (c) 2012 Freescale Semiconductor;

* All Rights Reserved

*

***************************************************************************

*
```

```
* THIS SOFTWARE IS PROVIDED BY FREESCALE "AS IS" AND ANY EXPRESSED OR
* IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES
* OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.
* IN NO EVENT SHALL FREESCALE OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT,
* INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
* (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
* SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
* HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
* STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING
* IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF
* THE POSSIBILITY OF SUCH DAMAGE.
*
***************************************************************************
*
* Comments:
*   This is an example to use the HAB API from boot code.
*
**************************************************************************/

typedef unsigned char uint8_t;
typedef unsigned short uint16_t;
typedef unsigned int uint32_t;

/************** HAB data ********************/

/* these addresses depend on the i.MX, here this is for i.MX25 */
#define JUMP_TABLE          (0x00000088)
#define SERIAL_DWLD_MODE    (0x00406969)
#define ERROR_LOGGING       0x780018D4

typedef uint8_t hab_result_t;
#define hab_result (*(hab_result_t*)ERROR_LOGGING)
#define HAB_PASSED  0xF0
```

```
#define HAB_FAILURE 0x39


/* According to the HAB API, the parameter csf_list points to a list of length
csf_count.

Let's use the following structure:

*/

typedef struct {

    uint8_t csf_list_length;/* Length of the CSF List */

    uint32_t *code_csf;/* Address of the CSF */

} csf_list_s;

typedef hab_result_t hab_csf_check_t(uint8_t csf_count, uint32_t *csf_list);

#define hab_csf_check ((hab_csf_check_t*)(JUMP_TABLE+0))


typedef hab_result_t hab_assert_verification_t(uint8_t *block_start, uint32_t
block_length);

#define hab_assert_verification ((hab_assert_verification_t*)(JUMP_TABLE+4))


typedef void hab_uart_usb_bootloader_t(void);

#define hab_uart_usb_bootloader
((hab_uart_usb_bootloader_t*)(SERIAL_DWLD_MODE))


/* address where the CSF binary was freely decided to be placed */

#define MY_CSF_ADDRESS  0x80040000

static const csf_list_s my_csf_list={1,(uint32_t *)MY_CSF_ADDRESS};


/* use typically the start address and length of the block defined in the CSF,

    and being authenticated by using hab_csf_check.

    Or use the code entry point address with the minimum length of 8bytes */

#define MY_DATA_ADDR    0x80000000

#define MY_DATA_LEN     0x10000


int main(void)

{

    /* clear HAB status */

    hab_result = HAB_FAILURE;
    /* process the CSF list */
    hab_result = hab_csf_check(my_csf_list.csf_list_length,
my_csf_list.code_csf);


    /* for example, go to serial download in case of error or into a custom
function */
    if(hab_result != HAB_PASSED)
        hab_uart_usb_bootloader();  /* does not return */


    /* ensures that the code entry point is located in a verified area */
   hab_result = hab_assert_verification((uint8_t*)(MY_DATA_ADDR), MY_DATA_LEN);


    /* for example, go to serial download in case of error or into a custom
function */
    if(hab_result != HAB_PASSED)
        hab_uart_usb_bootloader();  /* does not return */


    /* Finally, if it passes, then jump to the code entry point */



    return 0;
}
```

NOTE

Freescale also provides a more comprehensive reference software package, for extending the secure boot chain, called Loader Security. This Loader Security Library can be obtained with a software license agreement through a Freescale representative.

7 Development and Debugging Tips

7.1 Error Logging/Status Code

All along the boot procedure, and until the boot image is executed, the HAB returns the status code of the first encountered error at a specific address. It is essential to read the error logging address, which is located in the iRAM, to debug a problem during boot. It can be accessed like any 32-bit memory read, or through the “Get Status” command of the SDP protocol that returns the status code.

The list of status codes is given in the System Boot chapter of the security reference manual. The error logging addresses are provided in Table 1 for convenience.

Table 1. Error Logging Addresses

Processor	iRAM address
i.MX25	7800_18D4h
i.MX35	1000_18DCh
i.MX51	1FFE_1A98h

7.2 Using a Flash Header for Device Boot and SDP

During development, it is convenient to test some software without having to program it in a boot device. This can be achieved using the ROM Serial Download Protocol (SDP), which allows code to be downloaded and executed through one of the serial connections. As a reminder, the “flash header” is mandatory whether booting from a boot device or in SDP mode.

For most devices, the HAB expects to find the flash header at a typical offset 0x400; however, when using the SDP, the flash header is expected at the offset 0x0.

For convenience, it is simple to create a flash header at offset 0x0 with exactly the same information as the flash header at 0x400 for a standard device boot, except that the `*dcd_ptr_ptr` pointer points to the local `*dcd_ptr` within the same flash header in each case. With this flash header, the same boot image build can be executed through the SDP or from a boot device.

7.3 Using Serial Download Protocol to Test a Secure Boot Image

The Serial Download Protocol, also called UART/USB Serial Download Protocol, allows communicating with the ROM code with a defined set of commands, described in the System Boot chapter of the reference manual.

Typical usage is:

1. Send a “Get Status” command to ensure that the chip is ready. It typically returns 0xF0 0xF0 0xF0 0xF0 right out of reset.
2. Send “Write Memory” commands to program the memories and their controllers (SDRAM), IOMUX, clocks, etc, exactly like the DCD does. When in Production configuration, remember that the accessed address must lie within the authorized list of controllers as per the Device

Configuration Data (DCD) section of the System Boot chapter of the reference manual. Also, the modifications of the clock tree are not recommended at this stage, because this could change the frequency of the clock used by the UART, breaking the serial communication link.

3. Send a "Write File" command with file type FP=0xAA. The signed image containing the DCD, CSF, SRK, and a valid flash header is transmitted here. It is not recommended to send separately the CSF and DCD with other file types FP=0xCC and 0xEE.
4. Send a "Get Status" command right after step (3) to terminate the protocol, and start the authentication process prior to the execution of the code. It should respond 0x88 0x88 0x88 0x88.

7.3.1 Engineering Configuration

In this configuration, authentication failures are non-fatal and the image is always executed. Therefore, it is necessary to define a method to get the status code reported by the ROM, once the code was verified. It could be through a JTAG access, or through a printf command within the image downloaded, or using any other methods. A reminder about where this status code is located is available in Section 7.1, "Error Logging/Status Code."

7.3.2 Production Configuration

In this configuration, the code is only executed if the authentication passed. If the boot image runs, then the status will be 0xF0, because the authentication is successful.

If the boot image does not run, it is either because authentication passed but the boot image itself fails to execute properly, or because there was an issue during the authentication procedure (incorrect CSF, DCD not signed, incorrect DCD …). In such cases, the boot ROM jumps back into the SDP mode. This is the default ROM behavior when authentication failures occur or data aborts occur during boot.

The SDP "Get Status" command is used to get the status code. If this is 0xF0, the authentication passed, and it means that the image cannot execute properly. See the System Boot chapter of the security reference manual for a comprehensive list of status codes.

If the "Get Status" command does not return any data, it means, the chip did not enter SDP mode. This is probably because the code is stuck without having generated any errors.

The Advanced Tool Kit (ATK) can be used to download and execute a code through the SDP protocol as described above. Though, the tool does not return any error code, and does not allow executing a particular atomic command such "Get Status". Only high level macro functions, such as, function to program memory or download code are available for users.

7.4 Testing Generated SRK Digest with Fuse Cache Registers

This section describes a method of testing the SRK digest value, before actually burning the fuses.

When the processor exits from the Power On Reset state during the boot process, the physical state of the fuses is read and stored in cache registers for further reference. From that point on, every software or direct hardware access to the fuses is made through these registers.

For a fuse cache bit that is not protected against write access, it is possible to overwrite its value with a simple register write. This is true in all boot modes and security configurations (HAB types); hence, it is recommended to protect sensitive fuses by burning the fuse FBWP that prevent such direct modifications in a final product. For example, the SRK digest burned into the fuses must be protected against modification in Production configuration, but does not require special care during development in Engineering configuration.

If a signed boot image is created, but no digest is burned into the SRK_HASH fuses, then, the error status reported by the HAB library is 0x47 - HAB_FAIL_SUPER_ROOT_INSTALL. This is expected because there is no valid reference digest for HAB to compare against. Once a valid digest is provided, this will resolve the issue of the 0x47 status.

The following method of testing the SRK can be performed with the Advanced Tool Kit (ATK):

1. When using this method, the downloaded image must have a flash header at the offset 0x0 as described in Section 7.2, “Using a Flash Header for Device Boot and SDP.”
2. When launching the ATK, there is an option in the first interactive window that allows choosing a “Custom initial file.” The ATK documentation describes how to write a customized initialization file for a non-Freescale board.

   This file typically initializes the SDRAM as the DCD would do from a boot device. In addition, the SRK_HASH cache registers can be initialized with the digest value being tested. Note that this is only allowed in Engineering configuration, because those registers cannot be written through a “write register” command of the SDP in Production configuration. It has the same restrictions as a Production DCD.
3. The next step is to download the image with the “Flash tool.” The download address must be `*app_dest_ptr`. When the download button is pressed, the image is downloaded and its execution is triggered. The HAB code will proceed with the installation of the SRK using the SRK digest pattern contained in the fuse cache registers, then run the CSF, and launch the boot image. If the digest value provided is correct, the error status should be different than 0x47.

   For example, using the output file./crts/srk_wtls_cert_efuse_info.txt from the CST, suppose the digest of the SRK is:

```
SHA256(SRK1_sha256_2048_65537_v3_ca_crt.der_for_hash.bin)=
ab196cf9e50eca6a3fcab18e1340931d720b56a48151deda40651888db56ea6a
```

   Then, the ATK custom initialization file should include the following to initialize the IIM shadow registers for the i.MX25:

```
0x53ff0850  0xab  8

0x53ff1004  0x19  8

0x53ff1008  0x6c  8

…

0x53ff1078  0xea  8

0x53ff107c  0x6a  8
```

For other i.MX processors, see the relevant memory map to get the address of the IIM shadow registers. Once the digest is validated by the above method, it is normally safe to burn it into the fuses.

8 Troubleshooting

8.1 Reported Status 0x47—HAB_FAIL_SUPER_ROOT_INSTALL

During boot, the ROM code ensures that the SRK reference digest programmed in fuses and the generated digest from the SRK in the image are the same.

The comparison might fail for various reasons that could be:

- A missing digest in fuses.
- A wrong value blown in fuses, including provisioning the hash value bytes in the wrong order.
- A corrupted SRK modulus or structure in the boot device.
- Incorrect SRK information provided in the SRK structure.
- An incorrect pointer in the flash header.

8.2 Reported Status 0x55—HAB_FAIL_ASSERT

If this occurs during the authentication from the Serial Download Mode, this is usually due to the ROM attempting to verify the DCD table from an I2C EEPROM.

By default, the BT_EEPROM_CFG fuse or equivalent I/O pin sets this flag to 0, indicating that the DCD should be read from an I2C EEPROM. On platforms without a DCD in I2C EEPROM, this fuse or pin must be set to avoid that verification.

8.3 Reported Status 0x2E—HAB_INVALID_CSF_TYPE

Most of the development phase will use Engineering configuration with the final step of moving to Production configuration for the final product. This is defined by the state of the HAB_TYPE[1:0] fuses.

The TYPE field in the CSF must always match the value of those fuses. As a consequence, an image signed for an engineering chip cannot run on a production chip. The opposite is also true, although the error does not prevent the code signed for Production from running on an Engineering chip.

If the CSF does not reflect the state of these fuses, the returned status is 0x2E (HAB_INVALID_CSF_TYPE).

It is, therefore, essential to sign the boot image according to the HAB_TYPE of the chip.

9 i.MX25 CSF Example with Unlock DryIce Command

As mentioned in Section 3.3.1, “Generating Command Sequence File (CSF) Data,” there is a specific command to unlock some of the DryIce registers during boot. That command must be the first command to be executed when the CSF is processed. Here is an example:

```
[Header]
    Version = 3.0
    Security Configuration = Production
    Hash Algorithm = SHA256
    Engine = RTIC
    Certificate Format = WTLS
    Signature Format = PKCS1
    UID = Generic
    Code = 0x00
[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]

# below is the command that unlock the access to the DryIce registers
[Write Data]
    Width = 4
    Address Data = 0x53FFC03C 0xCA693569

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


 [Authenticate Data]
    Verification index = 2
    Blocks = 0x83F00000 0 0x0003D090 "/project/my_product/my_code_to_sign.bin"
```

10 U-Boot Modified Source Code and CSF Example for i.MX51

10.1 ./u-boot/board/freescale/mx51_bbg/u-boot.lds

The modifications are highlighted in blue.

```
/*
 * January 2004 - Changed to support H4 device
 * Copyright (c) 2004 Texas Instruments
 *
 * (C) Copyright 2002
 * Gary Jennejohn, DENX Software Engineering, <gj@denx.de>
 *
 * (C) Copyright 2009 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!     XXX FIXME XXX      */
       board/freescale/mx51_bbg/flash_header.o(.text.flasheader)
       cpu/arm_cortexa8/start.o
       board/freescale/mx51_bbg/libmx51_bbg.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 sigantures.
        * The SRK has a reserved space here too.
        * 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 + 0x2A000;
       __hab_data_start = .;
       /* max size of a 2048-bit SRK with exp 65537 is 272bytes */
       __srk_data = .;
       . = . + 0x120;
       /* 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);
        __bss_start = .;
        .bss : { *(.bss) }
        _end = .;
}
```

10.2 ./u-boot/board/freescale/mx51_bbg/flash_header.S

The modifications are highlighted in blue.

```
/*
 * (C) Copyright 2009-2010 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/mx51.h>
#include "board-imx51.h"

#ifdefCONFIG_FLASH_HEADER
#ifndef CONFIG_FLASH_HEADER_OFFSET
# error "Must define the offset of flash header"
#endif
#define MXC_DCD_ITEM(i, type, addr, val)\
dcd_node_##i:                                                          \
        .word type                                                    ;\
        .word addr                                                    ;\
        .word val                                                     ;\

.section ".text.flasheader", "x"

/* flash header for the Serial Download Protocol mode */
app_code_jump_v_sdp:        .word   _start
app_code_code_barker_sdp:  .word   CONFIG_FLASH_HEADER_BARKER
app_code_csf_sdp:           .word   __csf_data
dcd_ptr_ptr_sdp:            .word   dcd_ptr_sdp
super_root_key_sdp:         .word   __srk_data
dcd_ptr_sdp:                .word   dcd_array_start
app_dest_ptr_sdp:           .word   TEXT_BASE

/* flash header for the device boot */
  .org    CONFIG_FLASH_HEADER_OFFSET
app_code_jump_v:            .word   _start
app_code_code_barker:      .word   CONFIG_FLASH_HEADER_BARKER
app_code_csf:               .word   __csf_data
dcd_ptr_ptr:                .word   dcd_ptr
super_root_key:             .word   __srk_data
dcd_ptr:                    .word   dcd_array_start
app_dest_ptr:               .word   TEXT_BASE

dcd_array_start:
dcd_barker:          .word0xB17219E9
dcd_array_size:.word                  dcd_data_end - dcd_array_start - 8
/* DCD */
/* DDR2 IOMUX configuration */
MXC_DCD_ITEM(1, 4, IOMUXC_BASE_ADDR + 0x8a0, 0x200)
MXC_DCD_ITEM(2, 4, IOMUXC_BASE_ADDR + 0x50c, 0x20c5)
MXC_DCD_ITEM(3, 4, IOMUXC_BASE_ADDR + 0x510, 0x20c5)
MXC_DCD_ITEM(4, 4, IOMUXC_BASE_ADDR + 0x83c, 0x2)
MXC_DCD_ITEM(5, 4, IOMUXC_BASE_ADDR + 0x848, 0x2)
MXC_DCD_ITEM(6, 4, IOMUXC_BASE_ADDR + 0x4b8, 0xe7)
MXC_DCD_ITEM(7, 4, IOMUXC_BASE_ADDR + 0x4bc, 0x45)
MXC_DCD_ITEM(8, 4, IOMUXC_BASE_ADDR + 0x4c0, 0x45)
MXC_DCD_ITEM(9, 4, IOMUXC_BASE_ADDR + 0x4c4, 0x45)
MXC_DCD_ITEM(10, 4, IOMUXC_BASE_ADDR + 0x4c8, 0x45)
MXC_DCD_ITEM(11, 4, IOMUXC_BASE_ADDR + 0x820, 0x0)
MXC_DCD_ITEM(12, 4, IOMUXC_BASE_ADDR + 0x4a4, 0x3)
MXC_DCD_ITEM(13, 4, IOMUXC_BASE_ADDR + 0x4a8, 0x3)
MXC_DCD_ITEM(14, 4, IOMUXC_BASE_ADDR + 0x4ac, 0xe3)
MXC_DCD_ITEM(15, 4, IOMUXC_BASE_ADDR + 0x4b0, 0xe3)
MXC_DCD_ITEM(16, 4, IOMUXC_BASE_ADDR + 0x4b4, 0xe3)
MXC_DCD_ITEM(17, 4, IOMUXC_BASE_ADDR + 0x4cc, 0xe3)
MXC_DCD_ITEM(18, 4, IOMUXC_BASE_ADDR + 0x4d0, 0xe2)
/* Set drive strength to MAX */
MXC_DCD_ITEM(19, 4, IOMUXC_BASE_ADDR + 0x82c, 0x4)
MXC_DCD_ITEM(20, 4, IOMUXC_BASE_ADDR + 0x8a4, 0x4)
MXC_DCD_ITEM(21, 4, IOMUXC_BASE_ADDR + 0x8ac, 0x4)
MXC_DCD_ITEM(22, 4, IOMUXC_BASE_ADDR + 0x8b8, 0x4)
/* 13 ROW, 10 COL, 32Bit, SREF=4 Micron Model */
/* CAS=3,  BL=4 */
MXC_DCD_ITEM(23, 4, ESDCTL_BASE_ADDR + ESDCTL_ESDCTL0, 0x82a20000)
MXC_DCD_ITEM(24, 4, ESDCTL_BASE_ADDR + ESDCTL_ESDCTL1, 0x82a20000)
MXC_DCD_ITEM(25, 4, ESDCTL_BASE_ADDR + ESDCTL_ESDMISC, 0x000ad0d0)
MXC_DCD_ITEM(26, 4, ESDCTL_BASE_ADDR + ESDCTL_ESDCFG0, 0x333584ab)
MXC_DCD_ITEM(27, 4, ESDCTL_BASE_ADDR + ESDCTL_ESDCFG1, 0x333584ab)
/* Init DRAM on CS0 */
MXC_DCD_ITEM(28, 4, ESDCTL_BASE_ADDR + ESDCTL_ESDSCR, 0x04008008)
MXC_DCD_ITEM(29, 4, ESDCTL_BASE_ADDR + ESDCTL_ESDSCR, 0x0000801a)
MXC_DCD_ITEM(30, 4, ESDCTL_BASE_ADDR + ESDCTL_ESDSCR, 0x0000801b)
MXC_DCD_ITEM(31, 4, ESDCTL_BASE_ADDR + ESDCTL_ESDSCR, 0x00448019)
MXC_DCD_ITEM(32, 4, ESDCTL_BASE_ADDR + ESDCTL_ESDSCR, 0x07328018)
MXC_DCD_ITEM(33, 4, ESDCTL_BASE_ADDR + ESDCTL_ESDSCR, 0x04008008)
MXC_DCD_ITEM(34, 4, ESDCTL_BASE_ADDR + ESDCTL_ESDSCR, 0x00008010)
MXC_DCD_ITEM(35, 4, ESDCTL_BASE_ADDR + ESDCTL_ESDSCR, 0x00008010)
MXC_DCD_ITEM(36, 4, ESDCTL_BASE_ADDR + ESDCTL_ESDSCR, 0x06328018)
MXC_DCD_ITEM(37, 4, ESDCTL_BASE_ADDR + ESDCTL_ESDSCR, 0x03808019)
MXC_DCD_ITEM(38, 4, ESDCTL_BASE_ADDR + ESDCTL_ESDSCR, 0x00408019)
MXC_DCD_ITEM(39, 4, ESDCTL_BASE_ADDR + ESDCTL_ESDSCR, 0x00008000)
/* Init DRAM on CS1 */
MXC_DCD_ITEM(40, 4, ESDCTL_BASE_ADDR + ESDCTL_ESDSCR, 0x0400800c)
MXC_DCD_ITEM(41, 4, ESDCTL_BASE_ADDR + ESDCTL_ESDSCR, 0x0000801e)
MXC_DCD_ITEM(42, 4, ESDCTL_BASE_ADDR + ESDCTL_ESDSCR, 0x0000801f)
MXC_DCD_ITEM(43, 4, ESDCTL_BASE_ADDR + ESDCTL_ESDSCR, 0x0000801d)
MXC_DCD_ITEM(44, 4, ESDCTL_BASE_ADDR + ESDCTL_ESDSCR, 0x0732801c)
MXC_DCD_ITEM(45, 4, ESDCTL_BASE_ADDR + ESDCTL_ESDSCR, 0x0400800c)
MXC_DCD_ITEM(46, 4, ESDCTL_BASE_ADDR + ESDCTL_ESDSCR, 0x00008014)
MXC_DCD_ITEM(47, 4, ESDCTL_BASE_ADDR + ESDCTL_ESDSCR, 0x00008014)
MXC_DCD_ITEM(48, 4, ESDCTL_BASE_ADDR + ESDCTL_ESDSCR, 0x0632801c)
MXC_DCD_ITEM(49, 4, ESDCTL_BASE_ADDR + ESDCTL_ESDSCR, 0x0380801d)
MXC_DCD_ITEM(50, 4, ESDCTL_BASE_ADDR + ESDCTL_ESDSCR, 0x0040801d)
MXC_DCD_ITEM(51, 4, ESDCTL_BASE_ADDR + ESDCTL_ESDSCR, 0x00008004)
MXC_DCD_ITEM(52, 4, ESDCTL_BASE_ADDR + ESDCTL_ESDCTL0, 0xb2a20000)
MXC_DCD_ITEM(53, 4, ESDCTL_BASE_ADDR + ESDCTL_ESDCTL1, 0xb2a20000)
MXC_DCD_ITEM(54, 4, ESDCTL_BASE_ADDR + ESDCTL_ESDMISC, 0x000ad6d0)
MXC_DCD_ITEM(55, 4, ESDCTL_BASE_ADDR + ESDCTL_ESDCDLYGD, 0x90000000)
MXC_DCD_ITEM(56, 4, ESDCTL_BASE_ADDR + ESDCTL_ESDSCR, 0x00000000)
dcd_data_end:
image_len:                  .word   __hab_data_end - TEXT_BASE
#endif
```

10.3 U-Boot CSF Example

Assuming that TEXT_BASE = `*app_dst_ptr` = 9780_0000h, and that the size of the image to sign goes from the start of the boot image up to the SRK: `*super_root_key` - `*app_dst_ptr` = TEXT_BASE + 0002_A000h - TEXT_BASE = 0002_A000h

```
[Header]

    Version = 3.0

    Security Configuration = Engineering

    Hash Algorithm = SHA256

    Engine = RTIC

    Certificate Format = WTLS

    Signature Format = PKCS1

    UID = Generic

    Code = 0x00

[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"


[Authenticate Data]

    Verification index = 2

    Blocks = 0x97800000 0 0x0002A000 "/project/my_product/u-boot_to_sign.bin"
```

11 Revision History

Table 2 provides a revision history for this application note.

Table 2. Document Revision History

Rev. Number	Date	Substantive Change(s)
Rev. 0	10/2012	Initial public release.

How to Reach Us:

Home Page:
freescale.com

Web Support:
freescale.com/support

Document Number: AN4547
Rev. 0
10/2012