NXP Semiconductors
User's Guide

Document Number: IMXMCUMFUUG
Rev. 0, 10/2017

i.MX MCU Manufacturing User's Guide

1. Introduction

This document describes generation of bootable image for i.MX MCU devices. It also explains the process to interface i.MX MCU Boot ROM and KBOOT based Flashloader and to program a bootable image into the external flash including:

- QuadSPI NOR/Octal Flash / HyperFLASH
- Serial NAND
- eMMC
- SD
- Parallel NOR
- SLC raw NAND
- SPI NOR/EEPROM

The i.MX MCU Boot ROM application resides in the ROM and enables RAM loading. The Flashloader application is loaded into SRAM and facilitates loading of boot images from boot devices into RAM. It also authenticates and executes the boot image.

This document introduces the Flashloader application, a companion tool to i.MX Boot ROM, and a complete solution for programming bool images to bootable devices. The Flashloader application runs in SRAM, so it should be downloaded to SRAM typically via ROM serial download interface. The Flashloader prepares and configures the devices for boot. It creates boot configuration structure on the bootable media wherever required, assists in programming encrypted images, generates key blobs, communicates with master on serial peripherals like USB, UART, I2C, and SPI using kboot commands interface protocol in downloading boot images.

Contents

1. Introduction.. 1
2. Overview... 2
3. i.MX MCU bootable image... 4
4. Generate i.MX MCU bootable image 6
5. Generate SB file for bootable image programming.......... 19
6. Program bootable image 33
7. Appendix.. 37

It also introduces the Elftosb utility. The Elftosb utility converts an elf images to signed, encrypted, and bootable image for i.MX devices. It also creates all the boot structures like image vector table, boot data etc. It generates the input command sequence file required to code sign or encrypt the image using the NXP signing tool (cst). It also calls the cst to generate the signatures and pack everything together in order boot ROM expects the boot image.

The document describes the usage of MfgTool.exe (Manufacturing Tool) in a manufacturing environment for production of devices (programming boot images on the bootable media using all the available tools).

2. Overview

The i.MX MCU Boot ROM application resides in the ROM and enables RAM loading. The Flashloader application is loaded into SRAM and facilitates loading of boot images from boot devices into RAM. It also authenticates and executes the boot image.

This document introduces the Flashloader application, a companion tool to i.MX Boot ROM, and a complete solution for programming bool images to bootable devices. The Flashloader application runs in SRAM, so it should be downloaded to SRAM typically via ROM serial download interface. The Flashloader prepares and configures the devices for boot. It creates boot configuration structure on the bootable media wherever required, assists in programming encrypted images, generates key blobs, communicates with master on serial peripherals like USB, UART, I2C, and SPI using kboot commands interface protocol in downloading boot images.

It also introduces the Elftosb utility. The Elftosb utility converts an elf images to signed, encrypted, and bootable image for i.MX devices. It creates all the boot structures like image vector table, boot data etc. It generates the input command sequence file required to code sign or encrypt the image using the NXP signing tool (cst). It also calls the cst to generate the signatures and pack everything together in order boot ROM expects the boot image.

The document describes the usage of MfgTool.exe (Manufacturing Tool) in a manufacturing environment for production of devices (programming boot images on the bootable media using all the available tools).

2.1. i.MX MCU Boot ROM

The i.MX MCU Boot ROM is a standard bootloader for all i.MX MCU devices. It resides in ROM space and supports booting from external flash devices for both XIP and non-XIP boot cases. It also provides serial downloader feature via UART or USB-HID interface into the internal RAM of i.MX MCU devices.

The i.MX MCU Boot ROM is a specific implementation of the existing i.MX MPU ROM Bootloader. For Flash programming use case, the i.MX MCU Boot ROM provides serial downloader feature powered by SDP command interface. For additional information, refer to the System Boot chapter in the device reference manual. The Mfgtool then can load the KBOOT based Flashloader into internal SRAM and jump to the Flashloader to enable Flash programming features.

2.2. KBOOT based Flashloader

The KBOOT based Flashloader is a specific implementation of the Kinetis Bootloader. It is used as a one-time programming aid for manufacturing. Most of the KBOOT commands are supported in the Flashloader to enable external Flash programming. See MCUx Flashloader Reference Manual for more details.

2.3. Host utilities

The MfgTool is a GUI host program used to interface with devices running i.MX MCU Boot ROM under serial downloader mode. It can also be used to program an application image by interfacing with Flashloader application.

The blhost is a command-line host program used to interface with devices running KBOOT based bootloaders. It is part of MfgTool release.
The Elftosb utility is a command-line host program used to generate bootable images for i.MX MCU Boot ROM.

The cst is a command-line host program used to generate certificates, image signatures, and encrypt images for i.MX MCU Boot ROM.

NOTE

These applications are released as part of Flashloader release package available on www.nxp.com/KBOOT. They are available in the *<install_dir>/bin/Tools* folder of the release package.

2.4. Terminology

The following table summarizes the terms and abbreviations included in this user's guide document.

Table 1. Terminology and Abbreviations

Terminology	Description
KBOOT	Kinetis Bootloader
KeyBlob	KeyBlob is a data structure that wraps the DEK for image decryption using AES-CCM algorithm
DEK	"Key" used to decrypt the encrypted bootable image
SB file	The SB file is the NXP binary file format for bootable images. The file consists of sections, sequence of bootloader commands, and data that assists Kinetis Bootloader in programming the image to target memory. The image data can also be encrypted in the SB file. The file can be downloaded to the target using the Kinetis Bootloader receive-sb-file command.
CST	Code Signing Tool
XIP	Execute-In-Place

3. i.MX MCU bootable image

3.1. Bootable image layout in target flash device

There are two types of supported boot image:

- XIP (Execute-In-Place) boot image: This type of boot image is only applicable to Serial NOR devices connected to QuadSPI or FlexSPI interfaces and Parallel NOR devices connected to WEIM or SEMC interface. The boot device memory is identical to the destination memory. ROM can boot this boot image directly.
- Non-XIP boot image: This type of boot image is usually for the NAND, SD, and eMMC devices that does not support the XIP feature. The boot device memory is different from the destination memory. ROM loads the boot image from the Boot device memory to Destination memory and then boots from the Destination memory.

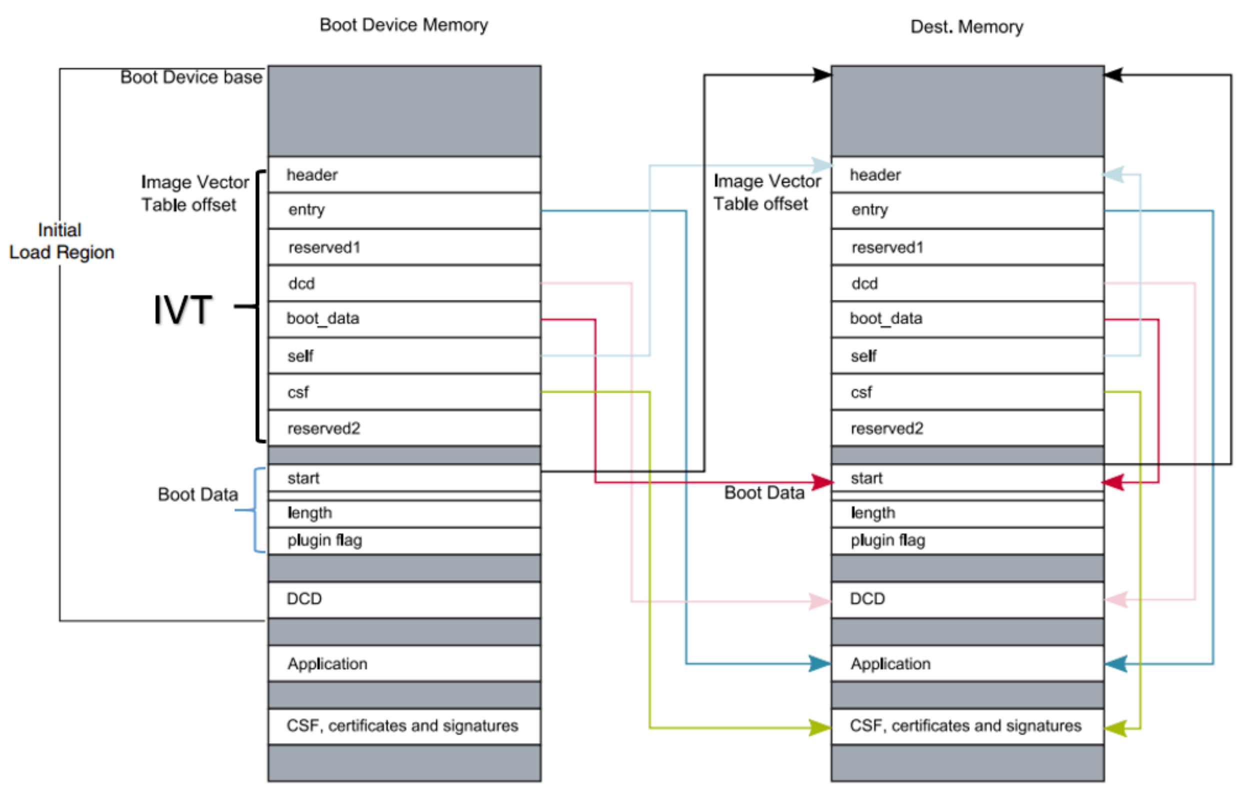

Figure 1. Bootable image layout

3.2. Boot image format

This section describes the boot image format and data structures. For ease-of-use, the Elftosb utility is provided to help customers automatically generate the boot image format file. The Elftosb utility usage is described later in this document.

Some data structures must be included in the bootable image. The bootable image consists of

- Image Vector Table (IVT): a list of pointers located at a fixed address that ROM examines to determine where the other components of the bootable image are located
- Boot Data: a table that indicates the bootable image location, image size in bytes and the plugin flag
- Device configuration data (DCD): IC configuration data, usually is used to configure DDR/SDRAM memory
- User application and data
- CSF (optional): signature block for Secure Boot, generated by CST
- KeyBlob (optional) – a data structure consists of wrapped DEK for encrypt boot

Each bootable image starts with appropriate IVT. In general, for the external memory devices that support XIP feature, the IVT offset is 0x1000 else it is 0x400. For example, for FlexSPI NOR on RT1052, the IVT must start at address 0x60001000 (start address is 0x6000_0000, IVT offset is 0x1000). Refer to the “Program Image” section in the System Boot Chapter of the device reference manual for more details.

3.2.1. IVT and boot data

The IVT is the data structure that the Boot ROM reads from the boot devices supplying the bootable image containing the required data components to perform a successful boot.

See the Program image section in the System Boot Chapter of the device reference manual for more details.

Table 2. IVT data structure

Offset	Field	Description
0x00 - 0x03	header	Byte 0 tag, fixed to 0xD1 Byte 1,2 length, bit endian format containing the overall length of the IVT in bytes, fixed to 0x00, 0x20 Byte 3: version, valid values: 0x40, 0x41, 0x42, 0x43
0x04 - 0x07	entry	Absolute address of the first instruction to execute from the image, or the vector address of the image
0x08 - 0x0b	reserved1	Reserved for future use, set to 0
0x0c - 0x0f	dcd	Absolute address of the image DCD. It is optional, so this field can be set to NULL if no DCD is required.
0x10 - 0x13	boot_data	Absolute address of the boot data
0x14 - 0x17	self	Absolute address of the IVT.
0x18 - 0x1b	csf	Absolute address of the Command Sequence File (CSF) used by the HAB library
0x1c - 0x1f	reserved2	Reserved, set to 0

3.2.2. Boot data structure

Table 3. Boot Data structure

Offset	Field	Description
0x00-0x03	start	Absolute address of the bootable image
0x04-0x07	length	Size of the bootable image
0x08-0x0b	plugin	Plugin flag, set to 0 if it is a normal boot image

3.3. Signed image

The bootable image can be signed by CST tool. The tool generates the CSF data in the binary file format that consists of command sequences and signatures based on given input command sequence file (csf file). Refer to the documentation in the CST release package for further details.

In this document, a simple method is introduced to generate signed images using Elftosb utility.

3.4. Encrypted image

There are two types of encrypted image formats:

- **Encrypted XIP image format**

The Flashloader generates the encrypted XIP image using the AES CTR algorithm when programming the image on the device. On execution, the hardware engine does on-the-fly decryption.

- **Encrypted image generated by CST**

To increase the security level, the bootable image can be signed and further encrypted by the CST. The KeyBlob must be generated on the device. The hardware deletes all sensitive keys if any security violation happens, so the sensitive keys cannot be cloned.

In this document, a simple method is introduced to generate signed images using Elftosb utility.

4. Generate i.MX MCU bootable image

There are two types of i.MX MCU bootable image.

- Normal boot image: This type of image can boot directly by boot ROM.
- Plugin boot image: This type of image can be used to load a boot image from devices that are not natively supported by boot ROM.

Both types of image can be unsigned, signed, and encrypted for different production phases and different security level requirements:

- Unsigned Image: The image does not contain authentication-related data and is used during development phase.

- Signed Image: The image contains authentication-related data (CSF section) and is used during production phase.
- Encrypted Image: The image contains encrypted application data and authentication-related data and is used during the production phase with higher security requirement.

The above types of bootable images can be generated by using the Elftosb utility. The detailed usage of the Elftosb utility is available in the “<install_dir>/doc” folder.

4.1. Description of the Elftosb utility

The Elftosb utility is a command-line host program used to generate the i.MX bootable image for the i.MX MCU boot ROM. The utility also generates wrapped binary file with command sequences and a bootable image. To create a SB file, use command-line options and an input text file (also called BD or command file).

4.1.1. The Elftosb utility options

The table shows the command line options used to create the i.MX MCU bootable image.

Table 4. Elftosb utility options

Option	Description
-f	Specify the bootable image format To create the i.MX MCU bootable image, the usage for family argument “-f” is: “-f imx” To create the SB file, the usage is: “-f kinetis”
-c	Command file to generate corresponding bootable image For example, “-c program_flexspi_nor_hyperflash.bd”
-o	Output file path For example, “-o ivt_flashloader.bin”
-V	Print extra detailed log information
-?	Print help info

4.1.2. BD file

Each BD file consists of the following four blocks: options, sources, constants, section

- The image paths are typically defined in the “sources” block.
- The constant variables are defined in the “constants” block.
- The memory configuration and programming-related operations are defined in the “section” block.

There are two types of BD files that are supported by the Elftosb utility. The first type of file is used for the i.MX bootable image generation. The “-f imx” option is mandatory during boot image generation using the Elftosb utility. The second type of file contains commands that are mainly used for memory programming. The “-f kinetis” flag is mandatory in this use case.

4.1.3. BD file for i.MX bootable image generation

The BD file for i.MX bootable image generation usually consists of four blocks. These are options, sources, constants, section.

4.1.3.1. Options block

The table shows the options used to generate a bootable image for the Options block.

Table 5. Supported options in the "Options" block

Options	Description
Flags	Generates unsigned, signed, encrypted boot images, and plugin images: • bit 2 - Encrypted image flag • bit 3 - Signed image flag • bit 4 - Plugin image flag For example: • 0x00 - unsigned image • 0x08 - signed image • 0x04 - encrypted image (encrypted image is always a signed image) • 0x18 - signed plugin image
startAddress	Provides the starting address of the target memory where image should be loaded by ROM.
ivtOffset	Provides offset where the IVT data structure must appear in the boot image. The default is 0x400 if not specified. The valid value is 0x400 or 0x1000 for i.MX boot image.
initialLoadSize	Defines the start of the executable image data from elf or the srec file. The default value is 0x2000 if not specified. In general, this value should be 0x1000 or 0x2000.
DCDFilePath	Defines the path to DCD file. If not specified, the DCD pointer in the IVT will be set to NULL (0) else the dcd file contents will be loaded at offset 0x40 from ivtOffset. The dcd file size is limited to (initialLoadSize - ivtOffset-0x40).

4.1.3.2. Sources block

Typically, all the application image paths are provided in this section. Currently, the ELF file and SREC file are supported for i.MX Bootable image generation, as shown in the following figure.

```
sources {
    elfFile = extern(0);
}
```

Figure 2. Example "sources" block for i.MX bootable image

4.1.3.3. Constants block

The Constants block provides a constant variable that is used to generate CSF data for image authentication and decryption. The Constants block is optional for an unsigned image. The supported constants are listed below.

```
constants {
    SEC_CSF_HEADER              = 20;
    SEC_CSF_INSTALL_SRK         = 21;
    SEC_CSF_INSTALL_CSFK        = 22;
    SEC_CSF_AUTHENTICATE_CSF    = 24;
    SEC_CSF_INSTALL_KEY         = 25;
    SEC_CSF_AUTHENTICATE_DATA   = 26;
    SEC_CSF_INSTALL_SECRET_KEY  = 27;
    SEC_CSF_DECRYPT_DATA        = 28;

}
```

Figure 3. Constants field for signed/encrypted image

4.1.3.4. Section blocks

The Section blocks are used to create the sections for an i.MX bootable image, for example, all sections for CSF data. For the unsigned image, the Section block is simple, just a fixed blank section, as shown in the following figure.

```
section (0)
{
}
```

Figure 4. Section blocks for an unsigned image

For the signed and encrypted image, the following sections are defined for Elftosb utility to generate the CSF descriptor file required by CST for CSF data generation, in which the first 6 sections are supported by both signed image and encrypted image generation, and the later 2 are only for encrypted image generation.

- SEC_CSF_HEADER

This section defines the necessary elements required for CSF Header generation as well as default values used for other sections throughout the remaining CSF.

Table 6. Elements for CSF Header section generation

Element	Description
Header_Version	HAB library version Valid values: 4.0, 4.1, 4.2, 4.3
Header_HashAlgorithm	Default Hash Algorithm Valid values: sha1, sha256, sha512
Header_Engine	Default Engine Valid values: ANY, DCP, CAAM, SW
Engine Configuration	Default Engine Configuration Recommended value: 0
Certificate Format	Default Certificate Format Must be “x509”
Signature Format	Default signature format Must be “CMS”

An example section block is shown as follows.

```
section (SEC_CSF_HEADER;
    Header_Version="4.2",
    Header_HashAlgorithm="sha256",
    Header_Engine="DCP",
    Header_EngineConfiguration=0,
    Header_CertificateFormat="x509",
    Header_SignatureFormat="CMS"
    )
{
}
```

Figure 5. Example CSF Header section block

- SEC_CSF_INSTALL_SRK

This section contains the elements to authenticate and install the root public key for use in subsequent sections, as shown in the following table.

Table 7. Elements for CSF Install SRK section generation

Element	Description
InstallSRK_Table	Path pointing to the Super Root Key Table file
InstallSRK_Source	SRK index with the SRK table

An example section block is shown as follows.

```
section (SEC_CSF_INSTALL_SRK;
    InstallSRK_Table="keys/SRK_1_2_3_4_table.bin", // "valid file path"
    InstallSRK_SourceIndex=0
    )
{
}
```

Figure 6. Example CSF Install SRK section block

- SEC_CSF_INSTALL_CSFK

This section consists of the elements used to authenticate and install a public key for use in subsequent sections.

Table 8. Elements for CSF Install CSFK section generation

Element	Description
InstallCSFK_File	File path pointing to CSFK certificate
InstallCSFK_CertificateFormat	CSFK certificate format Fixed to “x509”

An example section block is shown as follows.

```
section (SEC_CSF_INSTALL_CSFK;
    InstallCSFK_File="crts/CSF1_1_sha256_2048_65537_v3_usr_crt.pem", // "valid file path"
    InstallCSFK_CertificateFormat="x509" // "x509"
    )
{
}
```

Figure 7. Example CSF Install CSFK section block

- SEC_CSF_AUTHENTICATE_CSF

This section is used to authenticate the CSF from which it is executed using the CSFK mentioned in the section above. The default setting is enough. See the following example for more details.

```
section (SEC_CSF_AUTHENTICATE_CSF)
{
}
```

Figure 8. Example CSF Authenticate CSF section block

- SEC_CSF_INSTALL_KEY

This section consists of elements used to authenticate and install a public key for use in subsequent sections, as shown in the following table.

Table 9. Elements for CSF Install Key section generation

Element	Description
InstallKey_File	File path pointing to a Public key file
InstallKey_VerificationIndex	Verification key index in Key store Valid values: 0, 2, 3, 4
InstallKey_TargetIndex	Target key index in key store Valid values: 2, 3, 4

An example section block is shown as follows.

```
section (SEC_CSF_INSTALL_KEY;
    InstallKey_File="crts/IMG1_1_sha256_2048_65537_v3_usr_crt.pem",
    InstallKey_VerificationIndex=0, // Accepts integer or string
    InstallKey_TargetIndex=2) // Accepts integer or string
{
}
```

Figure 9. Example CSF Install KEY section block

- SEC_CSF_AUTHENTICATE_DATA

This section contains elements that are used to verify the authenticity of pre-loaded data in memory.

Table 10. Elements for CSF Authenticate Data section generation

Element	Description
AuthenticateData_VerificationIndex	Verification key index in key store
AuthenticateData_Engine	Data signature hash engine

	Valid values: ANY, DCP, CAAM, SW
AuthenticateData_EngineConfiguration	Configuration flags for the engine Default value: 0

An example section block is shown as follows.

```
section (SEC_CSF_AUTHENTICATE_DATA;
    AuthenticateData_VerificationIndex=2,
    AuthenticateData_Engine="DCP",
    AuthenticateData_EngineConfiguration=0)
{
}
```

Figure 10. Example CSF Authenticate Data section block

- SEC_CSF_INSTALL_SECRET_KEY

This section contains elements used to install the secret key to the MCU secret key store, which is used for Keyblob decryption. This section is required for encrypted image generation.

Table 11. Elements for CSF Install Secret Key section generation

Element	Description
SecretKey_Name	Specifies the file path used for CST to generate the random decryption key file
SecretKey_Length	Key length in bits Valid values: 128, 192, and 256
SecretKey_VerifyIndex	Master KEK index Valid values: 0 or 1
SecretKey_TargetIndex	Target secret key store index Valid value: 0-3

An example section block is shown as follows.

```
section (SEC_CSF_INSTALL_SECRET_KEY;
    SecretKey_Name="dek.bin",
    SecretKey_Length=128,
    SecretKey_VerifyIndex=0,
    SecretKey_TargetIndex=0)
{
}
```

Figure 11. Example CSF Secret Key section block

- SEC_CSF_DECRYPT_DATA

This section contains the necessary elements used to decrypt and authenticate a list of code/data blocks using the secret key stored in the secret key store, as shown in the following table.

Table 12. Elements for CSF Decrypt Data section generation

Element	Description
Decrypt_Engine	MAC engine Valid value: CAAM, DCP
Decrypt_EngineConfiguration	Configuration flags for the engine Default value: 0
Decrypt_VerifyIndex	Secret key index in the Secret key store Valid values: 0-3
Decrypt_MacBytes	Size of MAC in bytes If engine is CAAM, the valid value is even number between 4-16. The recommended value is 16. If engine is DCP, the valid value is 16.

```
section (SEC_CSF_DECRYPT_DATA;
    Decrypt_Engine="DCP",
    Decrypt_EngineConfiguration="0", // "valid engine configuration values"
    Decrypt_VerifyIndex=0,
    Decrypt_MacBytes=16)
{
}
```

Figure 12. Example CSF Decrypt Data section block

4.1.4. BD file for memory programming

Typically, “load”, “enable”, and “erase” commands are the most commonly used commands in a BD file for memory programming.

1. “load” command: It can be used to load raw binary, srec file, elf file, hex string, etc.. It also supports loading data to external memory devices, for example:
    - Load itcm_boot_image.bin > 0x8000; (Load data to ITCM)
    - Load flexspi_nor_boot_image.bin > 0x60001000; (Load data to the memory mapped memory device)
    - Load semc_nor_boot_image.bin > 0x80001000; (Load data to SEMC NOR, memory mapped memory device)
    - Load spinand boot_image.bin > 0x04; (Load data to SPI NAND)
    - Load sdcard boot_image.bin > 0x400; (Load data to the SD Card)
    - Load mmccard boot_image.bin > 0x400; (Load data to eMMC)
    - Load spieeprom boot_image.bin > 0x400; (Load data to SPI EEPROM/NOR)
    - Load semcnand boot_image.bin > 0x400; (Load data to SLC raw NAND via SEMC)
    - Load fuse 0x00000000 > 0x10; (Load data to the Fuse block)
2. “enable” command: It is used to configure external memory devices, for example:
    - Enable flexspinor 0x1000
    - Enable spinand 0x1000
    - Enable sdcard 0x1000
    - Enable mmccard 0x1000
    - Enable spieeprom 0x1000
    - Enable semcnor 0x1000
    - Enable semcnand 0x1000

3. “erase” command: It is used to erase a memory range in the selected memory device. For example:
    - Erase 0x60000000..0x60010000 (Erase 64 KB from FlexSPI NOR)
    - Erase spinand 0x4..0x08 (Erase 4 blocks from SPI NAND)
    - Erase sdcard 0x400..0x14000
    - Erase mmccard 0x400..0x14000
    - Erase spieeprom 0x400..0x14000
    - Erase semcnand 0x400..0x14000

4.2. Generate unsigned normal i.MX MCU bootable image

Typically, the unsigned bootable image is generated and programmed to the destination memory during the development phase.
Elftosb utility supports unsigned bootable image generation using options, BD file, and ELF/SREC file generated by toolchain.
Taking the Flashloader project as an example, here are the steps to create a bootable image for Flashloader.
Step 1: Create a BD file. For unsigned image creation, the “constants” block is optional, as shown in the following figure.

```
options {
    flags = 0x00;
    startAddress = 0x20000000;
    ivtOffset = 0x400;
    initialLoadSize = 0x2000;
}

sources {
    srecFile = extern(0);
}

section (0)
{
}
```

Figure 13. Example BD file for unsigned image generation

After the BD file is created, place it into the same folder that holds Elftosb utility executable.
Step 2: Copy Flashloader.srec provided in the release package into the same folder that holds Elftosb utility executable.
Step 3: Generate the Bootable image using Elftosb utility.

```
$ ./elftosb.exe -f imx -V -c Example_BD_Files/imx-unsigned.bd -o ivt_flashloader_unsigned.bin flashloader.srec
        Section: 0x0
```

Figure 14. Example command to generate unsigned boot image

Then, there are two bootable images generated by Elftosb utility. The first one is ivt_flashloader_unsigned.bin. The memory regions from 0 to ivt_offset are filled with padding bytes (all 0x00s).
The second one is ivt_flashloader_nopadding.bin, which starts from ivtdata directly without any padding before ivt.

4.3. Generate signed normal i.MX MCU bootable image

To generate a signed bootable image using Elftosb utility, perform the following steps:
Step 1: Create a BD file. The BD file can be as follows.

```
options {
    flags = 0x08;
    startAddress = 0x20000000;
    ivtOffset = 0x400;
    initialLoadSize = 0x2000;
}

sources {
    srecFile = extern(0);
}

constants {
    SEC_CSF_HEADER              = 20;
    SEC_CSF_INSTALL_SRK         = 21;
    SEC_CSF_INSTALL_CSFK        = 22;
    SEC_CSF_AUTHENTICATE_CSF    = 24;
    SEC_CSF_INSTALL_KEY         = 25;
    SEC_CSF_AUTHENTICATE_DATA   = 26;
}

section (SEC_CSF_HEADER;
    Header_Version="4.2",
    Header_HashAlgorithm="sha256",
    Header_Engine="DCP",
    Header_EngineConfiguration=0,
    Header_CertificateFormat="x509",
    Header_SignatureFormat="CMS"
    )
{
}

section (SEC_CSF_INSTALL_SRK;
    InstallSRK_Table="keys/SRK_1_2_3_4_table.bin", // "valid file path"
    InstallSRK_SourceIndex=0
    )
{
}
```

```
section (SEC_CSF_INSTALL_CSFK;
    InstallCSFK_File="crts/CSF1_1_sha256_2048_65537_v3_usr_crt.pem", // "valid file path'
    InstallCSFK_CertificateFormat="x509" // "x509"
    )
{
}

section (SEC_CSF_AUTHENTICATE_CSF)
{
}

section (SEC_CSF_INSTALL_KEY;
    InstallKey_File="crts/IMG1_1_sha256_2048_65537_v3_usr_crt.pem",
    InstallKey_VerificationIndex=0, // Accepts integer or string
    InstallKey_TargetIndex=2) // Accepts integer or string
{
}

section (SEC_CSF_AUTHENTICATE_DATA;
    AuthenticateData_VerificationIndex=2,
    AuthenticateData_Engine="DCP",
    AuthenticateData_EngineConfiguration=0)
{
}
```

Figure 15. Example BD file to generate a signed boot image

After the blank BD file is created, place it into the same folder that holds Elftosb utility executable.
Step 2: Copy Flashloader.srec provided in the release package into the same folder that holds Elftosb utility executable.
Step 3: Copy the “cst” executable, “crts” folder, and “keys” folder from “<cst_installation_dir>” to the same folder that holds Elftosb utility executable.
Step 4: Generate a Bootable image using Elftosb utility.

```
$ ./elftosb.exe -f imx -V -c Example_BD_Files/imx-signed.bd -o ivt_flashloader_signed.bin flashloader.srec
```

Figure 16. Example command to generate a signed boot image

Then, there are two bootable images generated by Elftosb utility. The first one is ivt_flashloader_signed.bin. The memory regions from 0 to ivt_offset is filled with padding bytes (all 0x00s).
The second one is ivt_flashloader_signed_nopadding.bin, which starts from ivt_offset directly. The CSF section is generated and appended to the unsigned bootable image successfully.

4.4. Generate encrypted normal i.MX MCU bootable image

To generate an encrypted image, perform the following steps:
Step 1: Create a BD file.

```
options {
    flags = 0x0c;
    startAddress = 0x20000000;
    ivtOffset = 0x400;
    initialLoadSize = 0x2000;
}

sources {
    elfFile = extern(0);
}

constants {
    SEC_CSF_HEADER              = 20;
    SEC_CSF_INSTALL_SRK         = 21;
    SEC_CSF_INSTALL_CSFK        = 22;
    SEC_CSF_AUTHENTICATE_CSF    = 24;
    SEC_CSF_INSTALL_KEY         = 25;
    SEC_CSF_AUTHENTICATE_DATA   = 26;
    SEC_CSF_INSTALL_SECRET_KEY  = 27;
    SEC_CSF_DECRYPT_DATA        = 28;

}

section (SEC_CSF_HEADER;
    Header_Version="4.3",
    Header_HashAlgorithm="sha256",
    Header_Engine="DCP",
    Header_EngineConfiguration=0,
    Header_CertificateFormat="x509",
    Header_SignatureFormat="CMS"
    )
{
}

section (SEC_CSF_INSTALL_SRK;
    InstallSRK_Table="keys/SRK_1_2_3_4_table.bin", // "valid file path"
    InstallSRK_SourceIndex=0
    )
{
}

section (SEC_CSF_INSTALL_CSFK;
    InstallCSFK_File="crts/CSF1_1_sha256_2048_65537_v3_usr_crt.pem", // "valid file path"
    InstallCSFK_CertificateFormat="x509" // "x509"
    )
{
}
section (SEC_CSF_AUTHENTICATE_CSF)
{
}

section (SEC_CSF_INSTALL_KEY;
    InstallKey_File="crts/IMG1_1_sha256_2048_65537_v3_usr_crt.pem",
    InstallKey_VerificationIndex=0, // Accepts integer or string
    InstallKey_TargetIndex=2) // Accepts integer or string
{
}

section (SEC_CSF_AUTHENTICATE_DATA;
    AuthenticateData_VerificationIndex=2,
    AuthenticateData_Engine="DCP",
    AuthenticateData_EngineConfiguration=0)
{
}

section (SEC_CSF_INSTALL_SECRET_KEY;
    SecretKey_Name="dek.bin",
    SecretKey_Length=128,
    SecretKey_VerifyIndex=0,
    SecretKey_TargetIndex=0)
{
}

section (SEC_CSF_DECRYPT_DATA;
    Decrypt_Engine="DCP",
    Decrypt_EngineConfiguration="0", // "valid engine configuration values"
    Decrypt_VerifyIndex=0,
    Decrypt_MacBytes=16)
{
}
```

Figure 17. Example BD file to generate an encrypt image

Step 2: Copy Flashloader.srec provided in the release package into the same folder that holds Elftosb utility executable.
Step 3: Copy the “cst” executable, “crts” folder, and “keys” folder from “<cst_installation_dir>” to the same folder that holds Elftosb utility executable.
Step 4: Generate an encrypted bootable image using Elftosb utility.

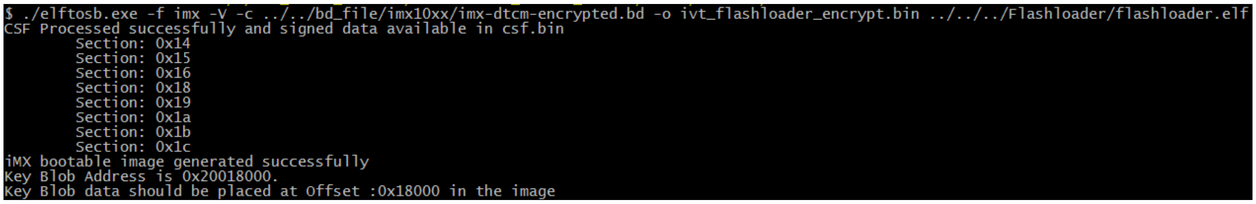

Figure 18. Example command to generate encrypt image

Then, there are two bootable images generated by Elftosb utility. The first one is ivt_flashloader_encrypt.bin. The memory regions from 0 to ivt_offset are filled with padding bytes (all 0x00s).
The Key Blob offset printed out in the example above is used in later section.
The second one is ivt_flashloader_encrypt_nopadding.bin, which starts from ivt_offset directly. The CSF section is generated and appended to the unsigned bootable image successfully.
Step 5: Generate the KeyBlob section using Flashloader.
The encrypted image generated by Elftosb utility is incomplete because the KeyBlob section must be generated on the SoC side only.
There are two methods to generate the KeyBlob block:

- Generate KeyBlob using the “generate-key-blob <dek_file> <blob_file>” command supported by Flashloader and blhost. See Appendix for more details.
- Generate KeyBlob during manufacturing and use the KeyBlob option block. See Chapter 5 for more details.

4.5. Generate Plugin boot image

The plugin boot image generation process is similar as the one for normal boot image. The only difference is that the bit 4 in the “flags” element within the “options” block must be set to 1, in other words, the valid flags value list for the plugin boot image is {0x10, 0x18, 0x1c}.
The following is an example BD file for plugin boot image generation.

```
options {
    flags = 0x10;
    startAddress = 0x60000000;
    ivtOffset = 0x1000;
    initialLoadSize = 0x2000;
}

sources {
    elfFile = extern(0);
}

section (0)
{
}
```

Figure 19. Example BD file for unsigned plugin boot image generation

5. Generate SB file for bootable image programming

To make the manufacturing process easier, all the commands supported by Flashloader and bootable image, can be wrapped into a single SB file. Even if there are any changes in the application, MfgTool still uses this SB file to manufacture. The SB file can be updated separately without updating scripts for MfgTool use.
In this section, a bootable image will be created using the method introduced in former chapter. Then corresponding SB file is generated using the bootable image. The corresponding BD file is prepared first to generate SB file for bootable image.

5.1. Generate SB file for FlexSPI NOR image programming

5.1.1. Generate Normal Bootable Image

For example, in RT1052, the FlexSPI NOR memory starts from address 0x6000_0000 and IVT from offset 0x1000. After following the steps in section 4.2, Generate unsigned normal i.MX MCU bootable image, and BD file generation, here is the usage of Elftosb utility to create bootable image for FlexSPI NOR. All the BD files are provided in the release package. The image below refers to the example command to generate signed image.

```
$ ./elftosb.exe -f imx -V -c ../../bd_file/imx-flexspinor-signed.bd -o ivt_flexspi_xip_signed.bin ../../../example_images/led_demo_evk_flexspi_nor.elf
CSF Processed successfully and signed data available in csf.bin
        Section: 0x14
        Section: 0x15
        Section: 0x16
        Section: 0x18
        Section: 0x19
        Section: 0x1a
```

Figure 20. Example command to generate signed flexspi boot image

After running above command, a file with suffix "_nopadding.bin" is programed into destination memory via subsequent SB file based on this binary.

5.1.2. Generate SB file for plaintext FlexSPI NOR image programming

Usually, a BD file for FlexSPI NOR boot consists of 7 parts.

1. The bootable image file path is provided in sources block
2. The FlexSPI NOR Configuration Option block is provided in section block
3. To enable FlexSPI NOR access, the "enable" command must be provided following above option block.
4. In case the Flash device is not erased, an "erase" command is required before programming data to Flash device. The erase operation is time consuming and is not required for a blank Flash device (factory setting) during Manufacturing.
5. The FlexSPI NOR Configuration Block (FNORCB) is required for FlexSPI NOR boot. To program the FNORCB generated by FlexSPI NOR Configuration Option block, a special magic number '0xF000000F" must load into RAM first
6. To notify the Flashloader to program the FNORCB, an "enable" command must be used following the magic number loading
7. After above operation, Flashloader can program the bootable image binary into Serial NOR Flash through FlexSPI module using load command

A simple example containing above steps is shown below.

```
# The source block assign file name to identifiers
sources {
 myBinFile = extern (0);          (1)
}

# The section block specifies the sequence of boot commands to be written to the SB file
section (0) {

    #1. Prepare Flash option
    # 0xc0233007 is the tag for Serial NOR parameter selection
    # bit [31:28] Tag fixed to 0x0C
    # bit [27:24] Option size fixed to 0
    # bit [23:20] Flash type option
    #             0 - QuadSPI SDR NOR
    #             1 - QUadSPI DDR NOR
    #             2 - HyperFLASH 1V8
    #             3 - HyperFLASH 3V
    #             4 - Macronix Octal DDR
    #             6 - Micron Octal DDR
    #             8 - Adesto EcoXIP DDR
    # bit [19:16] Query pads (Pads used for query Flash Parameters)
    #             0 - 1
    #             2 - 4
    #             3 - 8
    # bit [15:12] CMD pads (Pads used for query Flash Parameters)
    #             0 - 1
    #             2 - 4
    #             3 - 8
    # bit [11: 08] fixed to 0
    # bit [07: 04] fixed to 0
    # bit [03: 00] Flash Frequency, device specific
    #
    #
    # In this example, the 0xc0233007 represents:
    #     HyperFLASH 1V8, Query pads: 8 pads, Cmd pads: 8 pads, Frequency: 133MHz
    load 0xc0233007 > 0x2000;         (2)
    # Configure HyperFLASH using option a address 0x2000
    enable flexspinor 0x2000;         (3)

    #2 Erase flash as needed.
    erase 0x60000000..0x60010000;     (4)

    #3. Program config block
    # 0xf000000f is the tag to notify Flashloader to program FlexSPI NOR config block to the start of device
    load 0xf000000f > 0x3000;         (5)
    # Notify Flashloader to response the option at address 0x3000
    enable flexspinor 0x3000;         (6)

    #4. Program image
    load myBinFile > 0x60001000;      (7)
}
```

Figure 21. Example BD file for FlexSPI NOR programming

Here is an example to generate SB file using Elftosb utility, ivt_flexspi_nor_xip_nopadding.bin and BD file shown in Figure 5-2.

```
$ ./elftosb.exe -f imx -V -c ../../bd_file/imx10xx/imx-flexspinor-normal-unsigned.bd -o ivt_flexspi_nor_xip.bin ../../../example_images/led_demo_evk_flexspi_nor_0x60002000.srec
        Section: 0x0
iMX bootable image generated successfully
```

Figure 22. Example command to generate SB file for flexspi nor programming

After above command, a file named boot_image.sb will be created in the same folder that holds Elftosb utility executable.

5.1.3. Generate SB file for FlexSPI NOR Image encryption and programming

Usually, a BD file for FlexSPI NOR image encryption and programming consists of 7 parts.

1. The bootable image file path is provided in sources block
2. Enable FlexSPI NOR access using FlexSPI NOR Configuration Option block
3. Erase the Flash device if it is not blank. The erase operation is time consuming and is not required for a blank Flash device (factory setting) during Manufacturing.
4. Enable image encryption using PRDB option block
5. Program FNORCB using magic number
6. Program boot image binary into Serial NOR via FlexSPI module
7. Enable Encrypted XIP fuse bits.

A simple example containing above steps is shown below.

```
# The source block assign file name to identifiers
sources {
 myBinFile = extern (0);                                    (1)
}

# The section block specifies the sequence of boot commands to be written to the SB file
section (0) {

    #1. Prepare Flash option
    # In this example, the 0xc0233007 represents:
    #     HyperFLASH 1V8, Query pads: 8 pads, Cmd pads: 8 pads, Frequency: 133MHz
    load 0xc0233007 > 0x2000;                               (2)
    # Configure HyperFLASH using option a address 0x2000
    enable flexspinor 0x2000;

    #2 Erase flash as needed.
    erase 0x60000000..0x60010000;                           (3)

    #3 Prepare PRDB options
    # 0xe0120000 is an option for PRDB contruction and image encryption
    # bit[31:28] tag, fixed to 0x0E
    # bit[27:24] Key source, fixed to 0 for A0 silicon
    # bit[23:20] AES mode: 1 - CTR mode
    # bit[19:16] Encrypted region count
    # bit[15:00] reserved in A0
    load 0xe0120000 > 0x4000;
    # Region 0 start
    load 0x60001000 > 0x4004;                               (4)
    # Region 0 length
    load 0x00001000 > 0x4008;
    # Region 1 start
    load 0x60002000 > 0x400c;
    # Region 1 length
    load 0x0000e000 > 0x4010;
    # Program PRDB0 based on option
    enable flexspinor 0x4000;

    #4. Program config block
    # 0xf000000f is the tag to notify Flashloader to program FlexSPI NOR config block to the start of device
    load 0xf000000f > 0x3000;                               (5)
    # Notify Flashloader to response the option at address 0x3000
    enable flexspinor 0x3000;

    #5. Program image                                        (6)
    load myBinFile > 0x60001000;

    #6. Program BEE_KEY0_SEL and BEE_KEY1_SEL                 (7)
    load fuse 0x0000e000 > 0x06;
}
```

Figure 23. Example BD file for encrypted FlexSPI NOR image generation and programming

The steps to generate SB file is the same as above section.

5.2. Generate SB file for FlexSPI NAND image programming

For FlexSPI NAND boot, the IVT offset is always 0x400. However, to reduce effort in calculating the start address for each firmware region, the Flashloader supports programming the FlexSPI NAND boot image to corresponding firmware region in block granularity. So, the bootable image without "_nopadding" suffix will be used.

5.2.1. Generate SB file for FlexSPI NAND image programming

In general, a BD file for FlexSPI NAND image programming consists of 4 parts.

1. The bootable image file path is provided in sources block
2. Enable FlexSPI NAND access using FlexSPI NAND Configuration Option block
3. Erase SPI NAND device as needed
4. Program boot image binary into Serial NAND via FlexSPI module

```
# The source block assign file name to identifiers
sources {
myBootImageFile = extern (0);                    (1)
}

# The section block specifies the sequence of boot commands to be written to the SB file
section (0) {

    #1. Prepare FlexSPI NAND FCB option block
    #------------------------------------------------------------------------------------------------------------
    # Here is an example show how to create FCB using FlexSPI NAND FCB option block
    # option0: tag=0x0c, searchCount=2, searchStride = 64 pages, Address type: Block Address,
    # Option Block size: 4 long words
    load 0xc2000104 > 0x10000;
    # nandOptionAddr: 0x10000                                          (2)
    load 0x00010020 > 0x10004;
    # image info for firmware 0: start from block 4, max block count 4
    load 0x00040004 > 0x10008;
    # image info for firmware 1: start from block 8, max block count 4
    load 0x00080004 > 0x1000c;
    # FlexSPI NAND Cofniguration Option Block
    # Tag = 0x0c, Option size 1 long words, Flash size: 1Gbit, 1 Plane, Pages Per Block: 64, Page Size: 2KBytes
    # Max Freq:60MHz
    load 0xC0010023 > 0x10020; #(Addess must be equal to the value in nandOptionAddr)
    # Configure Serial NAND using option block at address 0x10000
    enable spinand 0x10000;

    #2 Erase flash as needed. (Unit: Block)
    # Erase firmware0 region
    erase spinand 0x4..0x8;                                        (3)
    # Erase firmware1 region
    erase spinand 0x8..0xc;

    #3. Program image
    # Load to firmware0 region
    load spinand myBootImageFile > 0x4;
                                                                  (4)
    # Load to firmware1 region
    load spinand myBootImageFile > 0x8;
}
```

Figure 24. Example BD file for FlexSPI NAND image programming

5.2.2. Generate SB file for encrypted FlexSPI NAND Image and KeyBlob programming

Generally, the BD file for FlexSPI NAND image programming with KeyBlob consists of 7 parts.

1. The bootable image file path is provided in sources block
2. Enable FlexSPI NAND access using FlexSPI NAND Configuration Option block
3. Erase SPI NAND device as needed
4. Program boot image binary into Serial NAND via FlexSPI module
5. Update KeyBlob information using KeyBlob Option block
6. Program KeyBlob block into SPI NAND for firmware 0
7. Program KeyBlob block into SPI NAND for firmware 1

An example BD file is shown as below.

```
# The source block assign file name to identifiers
sources {
myBootImageFile = extern (0);          (1)
dekFile = extern(1);
}

# The section block specifies the sequence of boot commands to be written to the SB file
section (0) {

    #1. Prepare FlexSPI NAND FCB option block
    load 0xc2000104 > 0x10000;
    load 0x00010020 > 0x10004;
    load 0x00040004 > 0x10008;                    (2)
    load 0x00080004 > 0x1000c;

    load 0xC1010023 > 0x10020; #(Address must be equal to the value in nandOptionAddr)
    load 0x00702000 > 0x10024;
    # Configure Serial NAND using option block at address 0x10000
    enable spinand 0x10000;

    #2 Erase flash as needed. (Unit: Block)
    erase spinand 0x4..0x8;                        (3)
    erase spinand 0x8..0xc;

    #3. Program image
    # Load to firmware0 region
    load spinand myBootImageFile > 0x4;
                                                       (4)
    # Load to firmware1 region
    load spinand myBootImageFile > 0x8;

    #4. Generate KeyBlob and program it to SPI NAND
    # Load DEK to RAM
    load dekFile > 0x10100;
    # Construct KeyBlob Option
    # tag = 0x0b, type=0, block size=3, DEK size=128bit      (5)
    load 0xb0300000 > 0x10200;
    # dek address = 0x10100
    load 0x00010100 > 0x10204;
    # keyblob offset in boot image
    # Note: this is only an example bd file, the value must be replaced with actual
    #       value in users project
    load 0x00004000 > 0x10208;
    enable spinand 0x10200;

    #5. Program KeyBlob to firmware0 region          (6)
    load 0xb1000000 > 0x10300;
    enable spinand 0x10300;

    #6. Program KeyBlob to firmware1 region
    load 0xb1000001 > 0x10400;                       (7)
    enable spinand 0x10400;
}
```

Figure 25. Example BD file for encrypted FlexSPI NAND image and KeyBlob programming

5.3. Generate SB file for SD image programming

The SD image always starts at offset 0x400. The i.MX boot image generated by Elftosb utility with “_nopadding.bin” will be used for programming.

5.3.1. 5.3.1 Generate SB file for SD image programming

In general, there are 6 steps in the BD file to program the bootable image to SD card.

1. The bootable image file path is provided in sources block
2. Prepare SDCard option block
3. Enable SDCard access using enable command
4. Erase SD card memory as needed
5. Program boot image binary into SD card

6. Program optimal SD boot parameters into Fuse (optional, remove it if it is not required in actual project)

An example is shown below.

```
# The source block assign file name to identifiers
sources {
myBootImageFile = extern (0);        (1)
}

# The section block specifies the sequence of boot commands to be written to the SB file
section (0) {

    #1. Prepare SDCard option block
    load 0xd0000000 > 0x100;                    (2)
    load 0x00000000 > 0x104;

    #2. Configure SDCard                (3)
    enable sdcard 0x100;

    #3. Erase blocks as needed.             (4)
    erase sdcard 0x400..0x14000;

    #4. Program SDCard Image                    (5)
    load sdcard myBootImageFile > 0x400;

    #5. Program Efuse for optimal read performance (optional)
    #load fuse 0x00000000 > 0x07;                               (6)
}
```

Figure 26. Example BD file for SD boot image programming

The steps to generate SB file for encrypted SD boot image and KeyBlob programming is similar to FlexSPI NAND. See below figure for more details.

```
# The source block assign file name to identifiers
sources {
 myBootImageFile = extern (0);
 dekFile = extern (1);
}

# The section block specifies the sequence of boot commands to be written to the SB file
section (0) {

    #1. Prepare SDCard option block
    load 0xd0000000 > 0x100;
    load 0x00000000 > 0x104;

    #2. Configure SDCard
    enable sdcard 0x100;

    #3. Erase blocks as needed.
    erase sdcard 0x400..0x14000;

    #4. Program SDCard Image
    load sdcard myBootImageFile > 0x400;

    #5. Generate KeyBlob and program it to SD Card
    # Load DEK to RAM
    load dekFile > 0x10100;
    # Construct KeyBlob Option
    #---------------------------------------------------------------------------
    # bit [31:28] tag, fixed to 0x0b
    # bit [27:24] type, 0 - Update KeyBlob context, 1 Program Keyblob to SPI NAND
    # bit [23:20] keyblob option block size, must equal to 3 if type =0,
    #             reserved if type = 1
    # bit [19:08] Reserved
    # bit [07:04] DEK size, 0-128bit 1-192bit 2-256 bit, only applicable if type=0
    # bit [03:00] Firmware Index, only applicable if type = 1
    # if type = 0, next words indicate the address that holds dek
    #              the 3rd word
    #---------------------------------------------------------------------------
    # tag = 0x0b, type=0, block size=3, DEK size=128bit
    load 0xb0300000 > 0x10200;
    # dek address = 0x10100
    load 0x00010100 > 0x10204;
    # keyblob offset in boot image
    # Note: this is only an example bd file, the value must be replaced with actual
    #       value in users project
    load 0x00004000 > 0x10208;
    enable sdcard 0x10200;

    #6. Program KeyBlob to firmware0 region
    load 0xb1000000 > 0x10300;
    enable sdcard 0x10300;

    #7. Program Efuse for optimal read performance (optional)
    #load fuse 0x00000000 > 0x07;
}
```

Figure 27. Example BD file for encrypted SD boot image and KeyBlob programming

5.4. Generate SB file for eMMC image programming

The eMMC image always starts at offset 0x400. The i.MX boot image generated by Elftosb utility with "_nopadding.bin" will be used for programming.
There are two types of eMMC boot mode: Normal boot and Fast boot.

5.4.1. Normal mode

There are 6 steps in the BD file to program the bootable image to eMMC for normal boot mode.

1. The bootable image file path is provided in sources block
2. Prepare eMMC option block
3. Enable eMMC access using enable command
4. Erase eMMC card memory as needed
5. Program boot image binary into eMMC
6. Program optimal eMMC boot parameters into Fuse (optional, remove it if it is not required in actual project).

```
# The source block assign file name to identifiers
sources {
myBootImageFile = extern (0);            (1)
}

# The section block specifies the sequence of boot commands to be written to the SB file
section (0) {

    #1. Prepare MMCCard option block
    load 0xc0000000 > 0x100;                 (2)
    load 0x00000000 > 0x104;

    #2. Configure MMCCard
    enable mmccard 0x100;                    (3)

    #3. Erase blocks as needed.
    erase mmccard 0x400..0x14000;            (4)

    #4. Program MMCCard Image
    load mmccard myBootImageFile > 0x400;    (5)

    #5. Program Efuse for optimal read performance (optional)
    #load fuse 0x00000000 > 0x07;            (6)

}
```

Figure 28. Example BD file for eMMC boot image programming for Normal boot mode

5.4.2. Fast Mode

There are 9 steps in the BD file to program the bootable image to eMMC for Fast boot mode.

1. The bootable image file path is provided in “sources” block
2. Prepare eMMC option block and enable eMMC access using “enable” command
3. Erase eMMC card memory as needed.
4. Program boot image binary into eMMC
5. Program optimal eMMC boot parameters into Fuse (optional, remove it if it is not required in actual project).
6. Prepare 2nd eMMC option block
7. Re-enable eMMC access using new option block
8. Erase data in User Data area as required
9. Load User Data file to User Data area

```
# The source block assign file name to identifiers
sources {
    myBootImageFile = extern (0);
    myUserDataFile = extern (1);
}

# The section block specifies the sequence of boot commands to be written to the SB file
section (0) {

    #1. Prepare 1st MMC Card option block.
    #   8bit DDR, High Speed, Boot partition 1 selected for access.
    #   Fast boot config: Boot partition 1, 8bit DDR, ACK.
    load 0xC1121625 > 0x100;
    load 0x00000000 > 0x104;

    #2. Configure MMC Card.
    enable mmccard 0x100;

    #3. Erase blocks at Boot partion 1 as needed.
    erase mmccard 0x400..0x14000;

    #4. Program Boot Image to MMC Card Boot partion 1.
    load mmccard myBootImageFile > 0x400;

    #5. Program Efuse according the fast boot config.(optional if use GPIO instead of Efuse)
    #load fuse 0x000006B3 > 0x05;

    #6 Prepare 2nd MMC Card option block.
    #  8bit DDR, High Speed, User data area is selected for access.
    load 0xC0001600 > 0x100;
    load 0x00000000 > 0x104;

    #7. Re-configure MMC Card
    enable mmccard 0x100;

    #8. Erase blocks at User data area as needed.
    erase mmccard 0x8000..0x100000;

    #9. Program User Data file to User data area.
    load mmccard myUserDataFile > 0x8000;
}
```

(1) (2) (3) (4) (5) (6) (7) (8) (9)

Figure 29. Example BD file for eMMC boot image programming for Fast boot mode

The BD file for encrypted eMMC boot image and KeyBlob programming is similar to SD.

5.5. Generate SB file for Serial NOR/EEPROM image programming

There are 5 steps in the BD file to program the bootable image to SD card.

1. The bootable image file path is provided in sources block
2. Prepare Serial NOR/EEPROM option block and enable Serial NOR/EEPROM access using enable command.
3. Erase Serial NOR/EEPROM memory as required
4. Program boot image binary into Serial NOR/EEPROM device
5. Enable Recovery Boot via Serial NOR/EEPROM as required

An example is shown below.

```
# The source block assign file name to identifiers
sources {
myBootImageFile = extern (0);                    (1)
}

# The section block specifies the sequence of boot commands to be written to the SB file
section (0) {

    #1. Prepare SPI NOR/EEPROM option block
    # bit [31:28] tag, fixed to 0x0c                       (2)
    # bit [27:24] Size, (bytes/4) - 1
    # bit [23:20] SPI instance
    # bit [19:16] PCS index
    # bit [15:12] Flash type, 0-SPI NOR, 1-SPI EEPROM
    # bit [11:08] Flash size(Bytes) 0 - 512K, 1-1M, 2-2M, 3-4M, 4-8M
    #             13-64K, 14-128K, 15-256K, etc.
    # bit [07:04] Sector size (Bytes), 0-4K, 1-8K, 2-32K, 3-64K,
    #             4-128K, 5-256K
    # bit [03:00] Page size (Bytes) 0-256, 1-512
    load 0xC0100300 > 0x100;

    #2. Configure SPI NOR/EEPROM
    enable spieeprom 0x100;

    #3. Erase blocks as needed.                    (3)
    erase spieeprom 0x400..0x14000;

    #4. Program SPI NOR/EEPROM Image                 (4)
    load spieeprom myBootImageFile > 0x400;

    #5. Enable Recovery boot
    # Note: this fuse field is SoC specific, need to be updated
    #       according to fusemap.                    (5)
    #load fuse 0x01000000 > 0x2d;

}
```

Figure 30. Example BD file for Serial NOR/EEPROM boot image programming

The BD file for encrypted SPI EEPRM/NOR boot image and KeyBlob programming is similar to SD.

5.6. Generate SB file for SEMC NOR image programming

In general, there are 5 steps in the BD file to program the bootable image to SD card.

1. The bootable image file path is provided in sources block
2. Prepare SEMC NOR option block and SEMC NOR access using enable command.
3. Erase SEMC NOR memory as required
4. Program boot image binary into SEMC NOR device
5. Program optimal SEMC NOR access parameters to Fuse as required

An example BD file is shown below.

```
# The source block assign file name to identifiers
sources {
 myBinFile = extern (0);
}

constants {
    kAbsAddr_Start= 0x90000000;
    kAbsAddr_Ivt = 0x90001000;
    kAbsAddr_App = 0x90002000;
}

# The section block specifies the sequence of boot commands to be written to the SB file
section (0) {

    #1. Prepare Flash option
    # Note: This is a template, need to be udpated to actual option in users' project
    load 0xD0002600 > 0x2000;
    # Configure to CSX2, ADV high active, 16bits IO, safe AC timing mode
    enable semcnor 0x2000;

    #2. Erase flash as needed.
    # Note: This is a template, need to be udpated to actual required memory range
    #       in users' project.
    erase 0x90000000..0x90010000;

    #3. Program image
    load myBinFile > kAbsAddr_Ivt;

    #4. Program Fuse as needed
    # Note: this is a template, need to be updated to actual required fuse value in users'
    #       project
    # load fuse 00000000 > 0x05;
}
```

Figure 31. Example BD file for SEMC NOR boot image programming

5.7. Generate SB file for SEMC NAND image programming

There are 5 steps in the BD file to program the bootable image to SD card.

1. The bootable image file path is provided in sources block
2. Prepare SEMC NAND FCB option block and SEMC NAND access using enable command
3. Erase SEMC NAND memory as required
4. Program boot image binary into SEMC NAND device
5. Program optimal SEMC NAND access parameters to Fuse as required

An example is shown below.

```
# The source block assign file name to identifiers
sources {                                              (1)
 myBinFile = extern (0);
}

# The section block specifies the sequence of boot commands to be written to the SB file
section (0) {

    #1. Prepare SEMC NAND FCB option
    # Note: This is a template, need to be udpated to actual option in users' project
    # See MCUX Flashloader RM for more details
    # ONFI 1.0, non-EDO, Timing mode 0, 8bit IO, CSX0
    load 0xD0030501 > 0x2000;                          (2)
    #  image copy = 1, search stride = 1, search count = 1
    load 0x00010101 > 0x2004;
    #  block index = 2, block count = 1
    load 0x00020001 > 0x2008;
    enable semcnand 0x2000;

    #2. Erase flash as needed.
    # Note: This is a template, need to be udpated to actual required memory range
    #       in users' project.                               (3)
    # byte adress = page size * pages per block * block index
    erase semcnand 0x100000..0x180000;

    #3. Program image (byte address)                          (4)
    # byte adress = page size * pages per block * block index
    load semcnad myBinFile > 0x100000;

    #4. Program Fuse as needed
    # Note: this is a template, need to be updated to actual required fuse value in users'
    #       project
    # load fuse 00000000 > 0x05;                     (5)
}
```

Figure 32. Example BD file for SEMC NAND boot image programming

5.8. Generate SB file for fuse program

In certain cases, the fuse must be programmed first to enable specific features for selected boot devices or security levels. For example, to enable Fast boot mode for eMMC, enable HAB closed mode, the fuse must be programmed first.

The Elftosb utility can support programming Fuse using built-in supported “load fuse” command, below is an example to program SRK table and enable HAB closed mode.

```
# The source block assign file name to identifiers
sources {
}

constants {
}

# The section block specifies the sequence of boot commands to be written to the SB file
# Note: this is just a template, please update it to actual values in users' project
section (0) {

    # Program SRK table
    load fuse 0xD132E7F1 > 0x18;
    load fuse 0x63CD795E > 0x19;
    load fuse 0x8FF38102 > 0x1A;
    load fuse 0x22A78E77 > 0x1B;
    load fuse 0x01019c82 > 0x1C;
    load fuse 0xFC3AC699 > 0x1D;
    load fuse 0xF2C327A3 > 0x1E;
    load fuse 0xDAC9214E > 0x1F;

    # Program SEC_CONFIG to enable HAB closed mode
    load fuse 0x00000002 > 0x06;

}
```

Figure 33. Example BD file for Fuse programming

6. Program bootable image

Bootable image programming is supported by MfgTool only.

6.1. MfgTool

The MfgTool supports i.MX MCU Boot ROM and KBOOT based Flashloader. It can be used in factory production environment. The Mfgtool can detect i.MX MCU Boot ROM devices connected to PC and invokes “blhost” to program the image on target memory devices connected to i.MX MCU device. The MfgTool and blhost utilities are documented in detail at www.nxp.com/KBOOT.

The template of MfgTool configuration profile is provided along with this document. It is applicable to most use cases without any modifications.

6.1.1. Mfgtool Directory structure

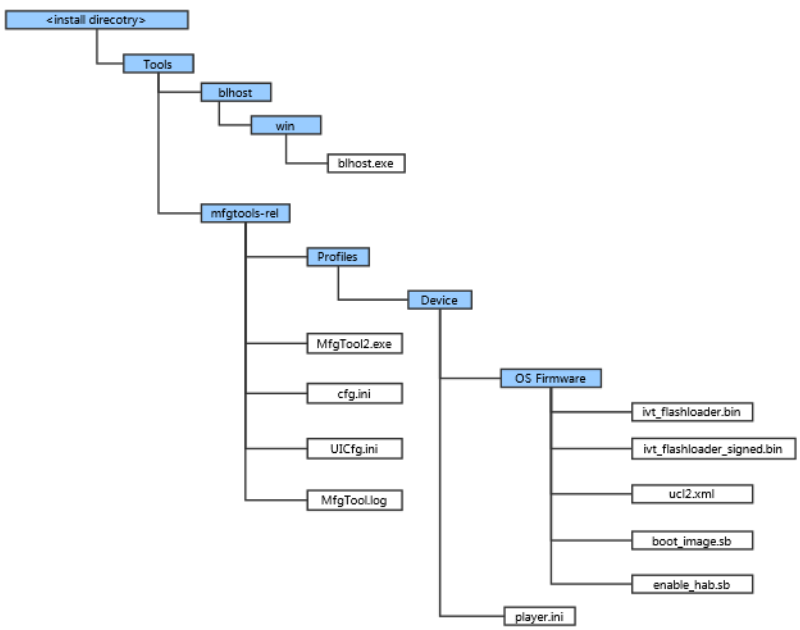

Figure 34. MfgTool organization

1. In the release package, the mfgtools-rel folder appears in the tools folder along with blhost folder
2. The blhost.exe appears in the blhost/win folder and the MfgTools executable “MfgTool2.exe” appears in the mfgtools-rel folder
3. The Profiles folder contains the profile for the supported devices that include an “OS Firmware” folder and player.ini file
4. The ucl2.xml file in the OS Firmware folder is the main XML that Mfgtool processes. It contains the flow of the manufacturing process for the device. The process includes identification parameters for the device and blhost commands parameter to identify the device connected to the

PC host and a set of blhost commands required for updating the image. The ucl2.xml file can be customized to suit custom setup or manufacturing process flow. The folder contains an example xml files for user's reference. An example ucl2.xml is shown below. In general, it defines the supported states and lists.

```xml
<UCL>

<CFG>
    <STATE name="BootStrap" dev="MXRT105X" vid="1FC9" pid="0130"/> <!-- I.MX SDP USB-HID -->
    <STATE name="Blhost" dev="KBL-HID" vid="15A2" pid="0073"/>   <!--KIBBLE USB-HID-->
</CFG>

<LIST name="MXRT105x-DevBoot" desc="Boot Flashloader">
<!-- Stage 1, load and execute Flashloader -->
    <CMD state="BootStrap" type="boot" body="BootStrap" file="ivt_flashloader.bin" > Loading Flashloader. </CMD>
    <CMD state="BootStrap" type="jump"  onError = "ignore"> Jumping to Flashloader. </CMD>

<!-- Stage 2, Program boot image into external memory using Flashloader -->
    <CMD state="Blhost" type="blhost" body="get-property 1" > Get Property 1. </CMD> <!--Used to test if flashloader runs successfully-->
    <CMD state="Blhost" type="blhost" timeout="15000" body="receive-sb-file \"Profiles\\MXRT105X\\OS Firmware\\boot_image.sb\"" > Program Boot Image. </CMD>
    <CMD state="Blhost" type="blhost" body="Update Completed!">Done</CMD>

</LIST>

<LIST name="MXRT105X-SecureBoot" desc="Boot Signed Flashloader">
<!-- Stage 1, load and execute Flashloader -->
    <CMD state="BootStrap" type="boot" body="BootStrap" file="ivt_flashloader_signed.bin" > Loading Flashloader. </CMD>
    <CMD state="BootStrap" type="jump" onError="ignore"> Jumping to Flashloader. </CMD>

<!-- Stage 2, Enable HAB closed mode using Flashloader -->
    <CMD state="Blhost" type="blhost" body="get-property 1" ifhab="Open" > Get Property 1. </CMD> <!--Used to test if flashloader runs successfully-->
    <CMD state="Blhost" type="blhost" body="receive-sb-file \"Profiles\\MXRT105X\\OS Firmware\\enable_hab.sb\"" ifhab="Open" > Program Boot Image. </CMD>
    <CMD state="Blhost" type="blhost" body="reset" ifhab="Open"> Reset. </CMD> <!--Reset device to enable HAB Close Mode-->

<!-- Stage 3, Program signed image into external memory using Flashloader -->
    <CMD state="Blhost" type="blhost" body="get-property 1" ifhab="Close"> Get Property 1. </CMD> <!--Used to test if flashloader runs successfully-->
    <CMD state="Blhost" type="blhost" timeout="15000" body="receive-sb-file \"Profiles\\MXRT105X\\OS Firmware\\boot_image_signed.sb\"" ifhab="Close" > Program Boot Image. </CMD>
    <CMD state="Blhost" type="blhost" body="Update Completed!" ifhab="Close" >Done</CMD>
</LIST>

</UCL>
```

Figure 35. Example UCL2.xml settings

5. The "ivt_flashloader.bin" file under the "OS firmware" is the Flashloader released to support image programming in development phase
6. The "ivt_flashloader_signed.bin" file under the "OS firmware" is the Bootable Flashloader image file generated by users for SecureBoot solution in production phase
7. The "boot_image.sb" file under the "OS firmware" is the wrapped file with command sequences and bootable images generated using Elftosb utility by users
8. The "enable_hab.sb" file under the "OS firmware" is the wrapped file with command sequences that programs Fuses to enable HAB closed mode, which is generated using elftosb utility by users
9. The play.ini in the "<Device>" profile folder contains configurable parameters for the manufacturing tool application
10. The cfg.ini and UICfg.ini files provide customizable parameters for the look and feel of the tool's GUI. The cfg.ini in tool's GUI is used to select "chip", "platform" and "name" in list. Refer to the example below
    Note: Select appropriate "chip" from <Device> list, "name" from list in ucl2.xml in <Device>/OS Firmware folder.

```
[profiles]
chip = MXRT105X

[platform]
board =

[LIST]
name = MXRT105X-DevBoot
```

Figure 36. Example cfg.ini for RT105x

11. UlCfg.ini is used to select the number of instances supported by MfgTool UI. The valid instance range is 1-4
12. The MfgTool.log text file is a useful tool to debug failures reported on MfgTool UI. The MfgTool logs the entire command line string which was used to invoke blhost and collects the output response text the blhost puts out on stdout into the MfgTool log file. The log file should be the considered first while troubleshooting

6.1.2. Preparation before image programming using MfgTool

See Chapter 4 and 5 for more details.

6.2. Connect to the i.MX MCU Platform

The i.MX MCU platform can be connected to a host computer to interface with the i.MX MCU Boot ROM application. After the platform is connected in serial downloader mode, use the Mfgtool to program bootable image into the target flash memory. If the connection establishes successfully and the cfg.ini, UlCfg.ini are configured appropriately, the device will be recognized by MfgTool like the one below.

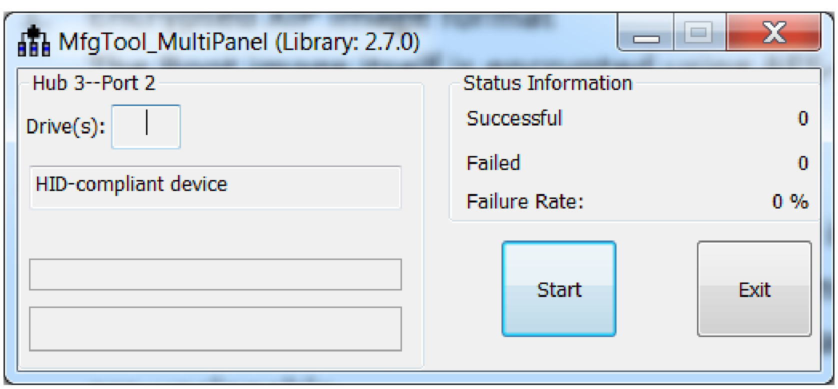

Figure 37. MfgTool GUI with device connected

6.3 Program bootable image during development

In development phase, the device may be in HAB open mode for most use cases. Users can configure the “name” field in cfg.ini file as <Device>-DevBoot, then prepare the boot_image.sb file using elftosb utility. After the “boot_image.sb” is generated, place it into “<Device>/OS Firmware/” folder. Then put device into serial downloader mode and connect it to host PC. After opening the MfgTool.exe and click “Start” to trigger a programming sequence. When the programming completes, the below window appears. To exit MfgTool, click “Stop” and “Exit” in turn.

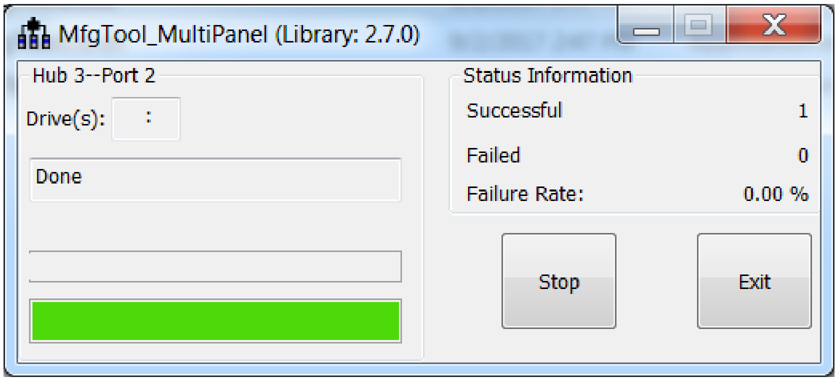

Figure 38. Successful result for programming with MfgTool for DevBoot

6.4 Program bootable image for production

In production phase, the device can be in HAB closed mode for most use cases. Users can configure the “name” field in cfg.ini file as <Device>-SecureBoot, then prepare the boot_image.sb file, enable_hab.sb and ivt_flashloader_signed.bin using Elftosb utility. After all are generated, place them into “<Device>/OS Firmware/” folder, then put device in serial downloader, connect it to host PC. Open MfgTool.exe and click “Start” to trigger a programming sequence. After the programming completes, the below window will be seen. To exit MfgTool, click “Stop” and “Exit” in turn.

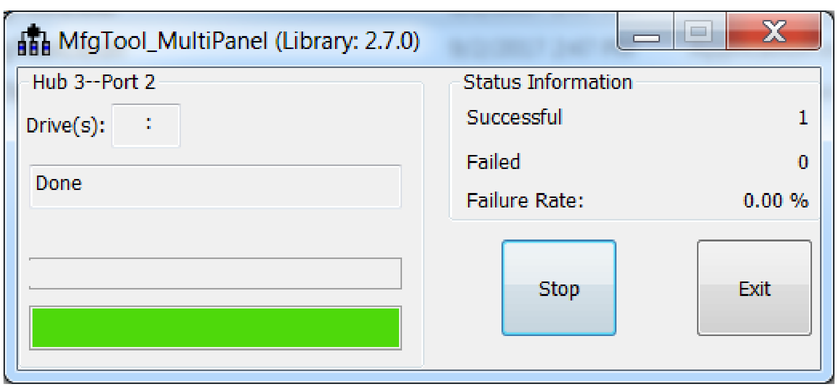

Figure 39. Successful result for programming with MfgTool for SecureBoot

7. Appendix

7.1. Plugin boot application

The plugin boot application is usually used to enable boot features that are not natively supported by Boot ROM, for example,

- Boot from USB disk
- Boot from Ethernet,
- DDR/SDRAM configuration
- Redundant boot/reliable boot

The prototype of plugin boot application is:

```
bool (*plugin_download)(void **start, size_t *bytes, uint32_t *ivt_offset);
```

7.1.1. Principles for Plugin boot application design

The Boot ROM needs to jump between Plugin boot image and the normal boot image that is loaded by the plugin boot application. To avoid any impact on the ROM boot flow, here are some recommended principles for plugin boot application design.

1. The plugin boot application must not use the memory that is currently reserved for ROM use
2. The plugin boot application should use minimum stack spaces to avoid the possibility of stack overflow caused by plugin boot application
3. The plugin boot application must consider Watchdog service, if the WDOG enable bit is enabled in the Fuse block

7.1.2. Boot Flow of Plugin boot application

The boot flow for Plugin boot application is as follows

1. Boot ROM loads the XIP plugin boot image, does authentication and execution and then jump to plugin boot application
2. The plugin boot application loads the signed Non-XIP image from address 0x60008000 and jumps back to Boot ROM
3. Boot ROM does authentication/decryption based on the parameters output by plugin boot application and jumps to the non-XIP boot image after authenticating successfully

7.1.3. Example Plugin boot application to enable non-XIP boot on FlexSPI NOR

The Non-XIP boot case is not natively supported by some i.MX MCU Boot ROM devices. In this case, a simple plugin boot image can be created to enable non-XIP boot case for these boot devices.
The basic flow of how Plugin boot works is as follows:

Here are the example codes for plugin boot application for RT105x FlexSPI NOR boot.

```
#define BOOT_IMAGE_LOAD_BASE        0x60008000
```

Figure 40. Example plugin boot application (Part 1)

```
enum
{
    kTag_HAB_IVT = 0xd1,
};

typedef struct  _hab_hdr
{
    uint8_t tag;
    uint8_t len[2];
    uint8_t version;
}hab_hdr_t;

typedef struct  _hab_ivt
{
   hab_hdr_t hdr;
   uint32_t entry;
   uint32_t reserved1;
   uint32_t dcd;
   uint32_t boot_data;
   uint32_t self;
   uint32_t csf;
   uint32_t reserved2;
}hab_ivt_t;

//!@brief Boot data structure
typedef struct _boot_data
{
  uint32_t start;
  uint32_t length;
  uint32_t plugin;
  uint32_t reserved;
} boot_data_t;

//!@brief Boot Image header, including both IVT and BOOT_DATA
typedef struct  _boot_imag_hdr
{
  hab_ivt_t ivt;
  boot_data_t boot_data;
} boot_image_hdr_t;
```

Figure 41. Examples code for plugin boot application (Part 2)

```
/*!@brief Plugin Download function
 *
 * This function is used to copy non-xip boot image from Flash to RAM
 *
 */
bool plugin_download(void **start, size_t *bytes, uint32_t *ivt_offset)
{
  bool result = false;
  const boot_image_hdr_t *boot_hdr;

  // Search IVT
  uint32_t ivt_offset_list[3] = {0, 0x400, 0x1000};

  uint32_t search_index = 0;
  while (search_index < sizeof(ivt_offset_list) / sizeof(ivt_offset_list[0]))
  {
    boot_hdr = (const boot_image_hdr_t *)(ivt_offset_list[search_index] +
                                          BOOT_IMAGE_LOAD_BASE);
    if (boot_hdr->ivt.hdr.tag != kTag_HAB_IVT)
    {
      search_index++;
      continue;
    }

    *start = (void *)boot_hdr->boot_data.start;
    *bytes = boot_hdr->boot_data.length;
    *ivt_offset = boot_hdr->ivt.self - boot_hdr->boot_data.start;

    uint32_t *dst = (uint32_t *)boot_hdr->boot_data.start;
    uint32_t *src = (uint32_t *)((uint32_t)boot_hdr - *ivt_offset);
    size_t remaining_length = ((*bytes + 3) & ~0x03) / 4;
    while (remaining_length--)
    {
      *dst++ = *src++;
    }

    result = true;
    break;
  }

  return result;
}
```

Figure 42. Examples code for plugin boot application (Part 3)

The ELF file and SREC file for above example plugin boot application are also included in the release package. Both, can be found in “<install_dir>/example_images” folder.

7.1.4. Images loaded by plugin boot application

The image loaded by Plugin boot application can be either XIP image or the non-XIP image. Refer to Chapter 4 for more details.

7.2. Example of the whole manufacturing flow

Taking a pre-compiled led-demo as an example, a quick start guide is provided in this section to demonstrate the complete manufacturing flow for FlexSPI NOR boot based on RT1050-EVK board.

7.2.1. Manufacturing process in Development phase

In development phase, mostly the image is unsigned and it is mainly for functionality test.

7.2.1.1. Create i.MX bootable image

Following Chapter 4, based on the memory map of FlexSPI assigned for RT105x SoC, here are the steps to generate i.MX bootable image using Elftosb utility.

1. Generate the BD file for boot image generation. The BD file content is showed in below figure. It is also available in the release package in “<install_dir>/Tools/bd_file/rt10xx” folder.

```
options {
    flags = 0x00;
    startAddress = 0x60000000
    ivtOffset = 0x1000;
    initialLoadSize = 0x2000;
}

sources {
    elfFile = extern(0);
}

section (0)
{
}
```

Figure 43. Example DB file for FlexSPI Boot image creation

2. Generate the i.MX Bootable image using elftosb file.

Here is the example command:

```
$ ./elftosb.exe -f imx -V -c ../../bd_file/imx10xx/imx-flexspinor-normal-unsigned.bd -o ivt_flexspi_nor_xip.bin ../../../example_images/led_demo_evk_flexspi_nor_0x60002000.srec
        Section: 0x0
iMX bootable image generated successfully
```

Figure 44. Example command to generate FlexSPI NOR boot image

After above command, two bootable images generated are:

- ivt_flexspi_nor_xip.bin
- ivt_flexspi_nor_xip_nopadding.bin

The ivt_flexspi_nor_xip_nopadding.bin will be used to generate SB file for HyperFLASH programming in subsequent section.

7.2.1.2. Create SB file for HyperFLASH programming

Following chapter 5, here is an example to generate the SB file for HyperFLASH programming for RT1050-EVK board.

```
# The source block assign file name to identifiers
sources {
 myBinFile = extern (0);
}

constants {
    kAbsAddr_Start= 0x60000000;
    kAbsAddr_Ivt = 0x60001000;
    kAbsAddr_App = 0x60002000;
}

# The section block specifies the sequence of boot commands to be written to the SB file
section (0) {

    #1. Prepare Flash option
    # 0xc0233007 is the tag for Serial NOR parameter selection
    # bit [31:28] Tag fixed to 0x0C
    # bit [27:24] Option size fixed to 0
    # bit [23:20] Flash type option
    #             0 - QuadSPI SDR NOR
    #             1 - QUadSPI DDR NOR
    #             2 - HyperFLASH 1V8
    #             3 - HyperFLASH 3V
    #             4 - Macronix Octal DDR
    #             6 - Micron Octal DDR
    #             8 - Adesto EcoXIP DDR
    # bit [19:16] Query pads (Pads used for query Flash Parameters)
    #             0 - 1
    #             2 - 4
    #             3 - 8
    # bit [15:12] CMD pads (Pads used for query Flash Parameters)
    #             0 - 1
    #             2 - 4
    #             3 - 8
    # bit [11: 08] fixed to 0
    # bit [07: 04] fixed to 0
    # bit [03: 00] Flash Frequency, device specific
    #
    # In this example, the 0xc0233007 represents:
    #     HyperFLASH 1V8, Query pads: 8 pads, Cmd pads: 8 pads, Frequency: 133MHz
    load 0xc0233007 > 0x2000;
    # Configure HyperFLASH using option a address 0x2000
    enable flexspinor 0x2000;

    #2 Erase flash as needed.
    erase 0x60000000..0x60010000;

    #3. Program config block
    # 0xf000000f is the tag to notify Flashloader to program FlexSPI NOR config block to the start of device
    load 0xf000000f > 0x3000;
    # Notify Flashloader to response the option at address 0x3000
    enable flexspinor 0x3000;

    #5. Program image
    load myBinFile > kAbsAddr_Ivt;
}
```

Figure 45. Example BD file for FlexSPI NOR programming

After the BD file is ready, the next step is to generate the boot_image.sb file that is for MfgTool use later.
Here is the example command:

```
$ ./elftosb.exe -f kinetis -V -c ../../bd_file/imx10xx/program_flexspinor_image_hyperflash.bd -o boot_image.sb ivt_flexspi_nor_xip_nopadding.bin
Boot Section 0x00000000:
  FILL | adr=0x00002000 | len=0x00000004 | ptn=0xc0233007
  ENA  | adr=0x00002000 | cnt=0x00000004 | flg=0x0900
  ERAS | adr=0x60000000 | cnt=0x00010000 | flg=0x0000
  FILL | adr=0x00003000 | len=0x00000004 | ptn=0xf000000f
  ENA  | adr=0x00003000 | cnt=0x00000004 | flg=0x0900
  LOAD | adr=0x60001000 | len=0x00001884 | crc=0xaf7132cf | flg=0x0000
```

Figure 46. Example command to generate SB file for FlexSPI NOR programming

After performing above command, the boot_image.sb is generated in Elftosb utility folder.

7.2.1.3. Program Image to Flash using MfgTool

Use the following steps to program boot image into flash device

1. Copy the boot_image.sb file to “<mfgtool_root_dir>/Profiles/MXRT105X/OS Firmware” folder
2. Change the “name” under “[List]” to “MXRT105x-DevBoot” in cfg.ini file in <mfgtool_root_dir> folder
3. Put the RT1050-EVK board to Serial Downloader mode by setting SW7 to “1-OFF, 2-ON, 3-OFF, 4-ON”
4. Power up RT1050-EVK board and insert USB cable to J9
5. Open MfgTool, it will show as the detected device like the one shown in Figure 6-4.
6. Click “Start”, Mfgtool will do manufacturing process. After completion, it will show the status as success as shown in Figure 6-5. Click “Stop” and close the Mfgtool
7. Put the RT1050-EVK board to internal boot mode and select HyperFLASH as boot device by setting SW7 to “1-OFF, 2-ON, 3-ON, 4-OFF”. Then reset the device, the LED above the ethernet interface starts blinking

7.2.2. Manufacturing process in Production phase

In production phase, the image requires to be signed and even encrypted. In this case, the device must be configured to HAB closed mode.
Assuming the PKI tree is ready for cst use, copy “ca”, “crts”, and “keys” folder and cst executable to the folder that holds Elftosb utility executable, as shown below.

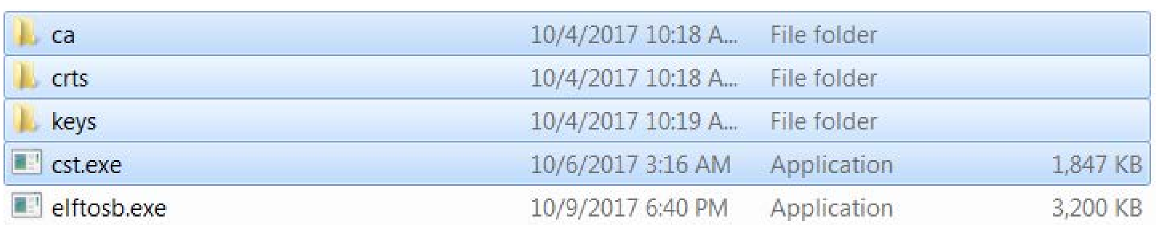

Figure 47. Copy required key and certs for signed image generation

7.2.2.1. Generate signed i.MX bootable image

Following chapter 4, based on the memory map of FlexSPI assigned for RT105x SoC, here are the steps to generate signed i.MX bootable image using Elftosb utility.

1. Generate the BD file for boot image generation. The BD file content is showed in figure below. It is also available in the release package in “<install_dir>/Tools/bd_file/rt10xx” folder.

```
options {
    flags = 0x08;
    startAddress = 0x60000000;
    ivtOffset = 0x1000;
    initialLoadSize = 0x2000;
}

sources {
    elfFile = extern(0);
}

constants {
    SEC_CSF_HEADER              = 20;
    SEC_CSF_INSTALL_SRK         = 21;
    SEC_CSF_INSTALL_CSFK        = 22;
    SEC_CSF_AUTHENTICATE_CSF    = 24;
    SEC_CSF_INSTALL_KEY         = 25;
    SEC_CSF_AUTHENTICATE_DATA   = 26;
}

section (SEC_CSF_HEADER;
    Header_Version="4.2",
    Header_HashAlgorithm="sha256",
    Header_Engine="DCP",
    Header_EngineConfiguration=0,
    Header_CertificateFormat="x509",
    Header_SignatureFormat="CMS"
    )
{
}

section (SEC_CSF_INSTALL_SRK;
    InstallSRK_Table="keys/SRK_1_2_3_4_table.bin", // "valid file path"
    InstallSRK_SourceIndex=0
    )
{
}

section (SEC_CSF_INSTALL_CSFK;
    InstallCSFK_File="crts/CSF1_1_sha256_2048_65537_v3_usr_crt.pem", // "valid file path"
    InstallCSFK_CertificateFormat="x509" // "x509"
    )
{
}

section (SEC_CSF_AUTHENTICATE_CSF)
{
}

section (SEC_CSF_INSTALL_KEY;
    InstallKey_File="crts/IMG1_1_sha256_2048_65537_v3_usr_crt.pem",
    InstallKey_VerificationIndex=0, // Accepts integer or string
    InstallKey_TargetIndex=2) // Accepts integer or string
{
}

section (SEC_CSF_AUTHENTICATE_DATA;
    AuthenticateData_VerificationIndex=2,
    AuthenticateData_Engine="DCP",
    AuthenticateData_EngineConfiguration=0)
{
}
```

Figure 48. Example BD file for signed image generation

2. Generate the i.MX Bootable image using Elftosb utility file.

Here is the example command:

```
$ ./elftosb.exe -f imx -V -c ../../bd_file/imx10xx/imx-flexspinor-normal-signed.bd -o ivt_flexspi_nor_xip_signed.bin ../../../example_images/led_demo_evk_flexspi_nor_0x60002000.srec
CSF Processed successfully and signed data available in csf.bin
        Section: 0x14
        Section: 0x15
        Section: 0x16
        Section: 0x18
        Section: 0x19
        Section: 0x1a
iMX bootable image generated successfully
```

Figure 49. Example command to generate signed boot image

After above command, two bootable images are generated:

- ivt_flexspi_nor_xip_signed.bin
- ivt_flexspi_nor_xip_signed_nopadding.bin

The ivt_flexspi_nor_xip_signed_nopadding.bin will be used to generate SB file for HyperFLASH programming in subsequent section.

7.2.2.2. Create SB file for Fuse programming

In the keys folder, there is a file named “SRK_1_2_3_4_fuse.bin”. This is the HASH table for SRK authentication during boot. It must be programmed to fuses to enable secure boot mode.
Below is an example file.

```
00000000  F1 E7 32 D1 5E 79 CD 63 02 81 F3 8F 77 8E A7 22  ñç2Ñ^yÍc..ó.wŽ§"
00000010  82 9C 01 01 99 C6 3A FC A3 27 C3 F2 4E 21 C9 DA  ,œ..™Æ:ü£'ÃòN!ÉÚ
```

Figure 50. Example SRK_1_2_3_4_fuse.bin file

Below is an example BD file which shows the procedure to program fuses. The fuse field is a 32-bit long word data. It will be programmed into fuses by Flashloader in little-endian mode.

```
# The source block assign file name to identifiers
sources {
}

constants {
}

# The section block specifies the sequence of boot commands to be written to the SB file
# Note: this is just a template, please update it to actual values in users' project
section (0) {

    # Program SRK table
    load fuse 0xD132E7F1 > 0x18;
    load fuse 0x63CD795E > 0x19;
    load fuse 0x8FF38102 > 0x1A;
    load fuse 0x22A78E77 > 0x1B;
    load fuse 0x01019c82 > 0x1C;
    load fuse 0xFC3AC699 > 0x1D;
    load fuse 0xF2C327A3 > 0x1E;
    load fuse 0xDAC9214E > 0x1F;
    
    # Program SEC_CONFIG to enable HAB closed mode
    load fuse 0x00000002 > 0x06;
    
}
```

Figure 51. Example BD file to enable HAB closed mode

The last command in above BD file is used to enable HAB closed mode by setting SEC_CONFIG [1] bit in the fuse to 1.
After BD file is ready, the next step is to create SB file for Fuse programming to enable HAB closed mode.

An example command is shown below:

```
$ ./elftosb.exe -f kinetis -V -c ../../bd_file/imx10xx/enable_hab.bd -o enable_hab.sb
Boot Section 0x00000000:
  PROG | idx=0x00000018 | wd1=0xd132e7f1 | wd2=0x00000000 | flg=0x0400
  PROG | idx=0x00000019 | wd1=0x63cd795e | wd2=0x00000000 | flg=0x0400
  PROG | idx=0x0000001a | wd1=0x8ff38102 | wd2=0x00000000 | flg=0x0400
  PROG | idx=0x0000001b | wd1=0x22a78e77 | wd2=0x00000000 | flg=0x0400
  PROG | idx=0x0000001c | wd1=0x01019c82 | wd2=0x00000000 | flg=0x0400
  PROG | idx=0x0000001d | wd1=0xfc3ac699 | wd2=0x00000000 | flg=0x0400
  PROG | idx=0x0000001e | wd1=0xf2c327a3 | wd2=0x00000000 | flg=0x0400
  PROG | idx=0x0000001f | wd1=0xdac9214e | wd2=0x00000000 | flg=0x0400
  PROG | idx=0x00000006 | wd1=0x00000002 | wd2=0x00000000 | flg=0x0400
```

Figure 52. Example command to generate SB file for Fuse programming

After the above command is executed, a file named "enable_hab.sb" gets generated. It is required in Mfgtool for SecureBoot solution.

7.2.2.3. Create SB file for Image encryption and programming for HyperFLASH

Following chapter 5, here is an example to generate the SB file for image encryption and programming on HyperFLASH for RT1050-EVK board.
Refer to the BD file in section 5.1.3.
After the BD file is ready, the next step is to generate the SB file. Refer below for an example command.

```
$ ./elftosb.exe -f kinetis -V -c ../../bd_file/imx10xx/program_flexspinor_image_hyperflash_encrypt.bd -o boot_image.sb ivt_flexspi_nor_xip_signed_nopadding.bin
Boot Section 0x00000000:
  FILL | adr=0x00002000 | len=0x00000004 | ptn=0xc0233007
  ENA  | adr=0x00002000 | cnt=0x00000004 | flg=0x0900
  ERAS | adr=0x60000000 | cnt=0x00010000 | flg=0x0000
  FILL | adr=0x00004000 | len=0x00000004 | ptn=0xe0120000
  FILL | adr=0x00004004 | len=0x00000004 | ptn=0x60001000
  FILL | adr=0x00004008 | len=0x00000004 | ptn=0x00001000
  FILL | adr=0x0000400c | len=0x00000004 | ptn=0x60002000
  FILL | adr=0x00004010 | len=0x00000004 | ptn=0x0000e000
  ENA  | adr=0x00004000 | cnt=0x00000004 | flg=0x0900
  FILL | adr=0x00003000 | len=0x00000004 | ptn=0xf000000f
  ENA  | adr=0x00003000 | cnt=0x00000004 | flg=0x0900
  LOAD | adr=0x60001000 | len=0x00004000 | crc=0xbae0b661 | flg=0x0000
  PROG | idx=0x00000006 | wd1=0x0000e000 | wd2=0x00000000 | flg=0x0400
```

Figure 53. Example command to generate SB file for FlexSPI NOR image encryption and programming

After above command, a file named "boot_image.sb" is generated in the folder that contains Elftosb utility executable.

7.2.2.4. Create signed Flashloader image

The BD file for signed Flashloader image generation is similar as the one in section 7.2.2.1.
The only difference is that the startAddress is 0x20000000 and IVTOffset is 0x400.
After the BD file is ready, the next step is to generate i.MX boot image using Elftosb utility. The example command is as below:

```
$ ./elftosb.exe -f imx -V -c ../../bd_file/imx10xx/imx-dtcm-signed.bd -o ivt_flashloader_signed.bin ../../../Flashloader/flashloader.elf
CSF Processed successfully and signed data available in csf.bin
        Section: 0x14
        Section: 0x15
        Section: 0x16
        Section: 0x18
        Section: 0x19
        Section: 0x1a
iMX bootable image generated successfully
```

Figure 54. Example command for Signed Flashloader image generation

After above command, two bootable images are generated:

- ivt_flashloader_signed.bin
- ivt_flashloader_signed_nopadding.bin

The first one is required by MfgTool for Secure Boot.

7.2.2.5. Program Image to Flash using MfgTool

Here are the steps to program boot image into Flash device

1. Copy the boot_image.sb file, ivt_flashloader_signed.bin and enable_hab.sb to "<mfgtool_root_dir>/Profiles/MXRT105X/OS Firmware" folder
2. Change the "name" under "[List]" to "MXRT105x-SecureBoot" in cfg.ini file in <mfgtool_root_dir> folder
3. Put the RT1050-EVK board to Serial Downloader mode by setting SW7 to "1-OFF, 2-ON, 3-OFF, 4-ON"
4. Power up RT1050-EVK board, and insert USB cable to J9
5. Open MfgTool, it will show the detected device like the one shown in Figure 6-4.
6. Click "Start", Mfgtool will do manufacturing process and after completion, it will show the status as success as shown in Figure 6-5. Click "Stop" and close the Mfgtool
7. Put the RT1050-EVK board to internal boot mode and select HyperFLASH as boot device by setting SW7 to "1-OFF, 2-ON, 3-ON, 4-OFF". Reset the device. The LED above the ethernet interface starts blinking which indicates that the image is running.

7.3 Generate KeyBlob manually

Users may need to generate the Keyblob manually in some cases. Flashloader supports such usage with blhost.

The KeyBlob must be generated when the device works under HAB closed mode with signed Flashloader application.

Assuming the dek.bin is ready (generated by Elftosb utility during encrypted image generation). Here is an example command to generate KeyBlob block.

```
$ ../../blhost/win/blhost.exe -u -- generate-key-blob dek.bin keyblob.bin
Inject command 'generate-key-blob'
Preparing to send 16 (0x10) bytes to the target.
Successful response to command 'generate-key-blob'
(1/1)100% Completed!
Response status = 0 (0x0) Success.
Response word 1 = 0 (0x0)
Response word 2 = 72 (0x48)
Read 72 of 72 bytes.
```

Figure 55. Generate KeyBlob using Flashloader

After above command is executed, a "keyblob.bin" file gets generated.

```
00000000  81 00 38 43 66 55 10 00 A6 7D 63 D7 E9 26 7F 80
00000010  66 94 3D 90 80 3A A4 66 BE AD C9 53 1C 40 2E 5F
00000020  0F A6 6A D5 E3 F2 97 BB A4 E7 67 6F 8E 74 61 65
00000030  78 90 E1 33 B1 03 EF E2 00 00 00 00 00 00 00 00
00000040  00 00 00 00 00 00 00 00
```

Figure 56. Example KeyBlob

With the encrypted image generated by Elftosb utility and keyblob.bin generated by flashloader, it is also feasible to combine the encrypted image and keyblob.bin. Then create a complete encrypted boot image with a Hex Editor. In this example, the KeyBlob offset is 0x18000 in the boot image.
Below is an example piece of encrypted image combined by Hex Editor.

```
00017FD0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00017FE0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00017FF0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00018000  81 00 38 43 66 55 10 00 A6 7D 63 D7 E9 26 7F 80
00018010  66 94 3D 90 80 3A A4 66 BE AD C9 53 1C 40 2E 5F
00018020  0F A6 6A D5 E3 F2 97 BB A4 E7 67 6F 8E 74 61 65
00018030  78 90 E1 33 B1 03 EF E2 00 00 00 00 00 00 00 00
00018040  00 00 00 00 00 00 00 00
```

Figure 57. Create complete encrypted image using Hex editor

How to Reach Us:

Home Page:
nxp.com

Web Support:
nxp.com/support

Information in this document is provided solely to enable system and software implementers to use NXP products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits based on the information in this document. NXP reserves the right to make changes without further notice to any products herein.

NXP makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does NXP assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. "Typical" parameters that may be provided in NXP data sheets and/or specifications can and do vary in different applications, and actual performance may vary over time. All operating parameters, including "typicals," must be validated for each customer application by customer's technical experts. NXP does not convey any license under its patent rights nor the rights of others. NXP sells products pursuant to standard terms and conditions of sale, which can be found at the following address: nxp.com/SalesTermsandConditions.

Document Number IMXMCUMFUUG
Revision 0, 10/2017