NXP Semiconductors

User's Guide

Document Number: MBOOTQSPIUG

Rev. 4, December 2018

MCU Bootloader QuadSPI User's Guide

by: NXP Semiconductors

Contents

1 Introduction

The QuadSPI controller available on selected Kinetis devices supports execute-in-place (XIP) for external SPI flash memory devices. This document describes the usage of MCU bootloader (MCUBOOT) in configuring various features of QuadSPI block, including XIP, generating plaintext and encrypted bootable SB file image, and flashing QuadSPI memory with the SB file image.

QuadSPI features supported by MCU bootloader:

- Various types of SPI NOR flash memory devices available in the market.
- Flash memory booting from QuadSPI directly, using MCU bootloader.
- Single/Dual/Quad and Octal SPI NOR flash memory devices.
- High-performance read/write operation with parallel and DDR modes.
- Protecting intellectual property with AES-128 algorithm.

2 Overview

This document mainly focuses on the following topics:

- QuadSPI image boot procedure
- Creating an application image running on QuadSPI memory
- Configuring QuadSPI with MCU bootloader
- Programming QuadSPI memory with SB file
- Advanced usage: QuadSPI encrypted boot image
- Application requirements for re-configuring QuadSPI clock

In addition, the following topics are also covered in the appendix sections:

- QuadSPI configuration block (QCB)
- Re-enter MCU bootloader under direct boot mode
- Explore features supported in QCB
- Working around ROM issues in supported DDR mode devices

2.1 Terminology

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

Table 1. Terminology and abbreviations

Terminology	Description
MCUBOOT	MCU bootloader
BCA	Bootloader Configuration Area, which provides customization of bootloader options, such as enabledPeripherals, peripheralDetectionTimeout, and so on. See the MCU bootloader chapter in the silicon's reference manual for more details.
QCB	QuadSPI Configuration Block, a structure containing configurable parameters needed by the MCU bootloader to configure the QuadSPI controller. See the MCU bootloader chapter in the silicon's reference manual for more details.
KeyBlob	A data structure which holds the KeyBlob entries. Each keyblob entry defines the encrypted QuadSPI memory region, decryption key, and so on. See the MCU bootloader chapter in the silicon's reference manual for more details.
KEK	KeyBlob Encryption Key, an AES-128 key used for encrypting plaintext KeyBlob and decrypting encrypted KeyBlob. See the MCU bootloader chapter in the silicon's reference manual for more details.
SB file	The SB file is the NXP binary file format for bootable images. The file consists of sections and sequence of bootloader commands and data that assists MCU bootloader in programming the image to target memory. The image data in the SB file can be encrypted as well. The file can be downloaded to the target using the MCU bootloader receive-sb-file command. See the MCU bootloader chapter in silicon's reference manual for more details

Table continues on the next page...

Table 1. Terminology and abbreviations (continued)

Terminology	Description
OTFAD	On-the-fly AES Decryption is a powerful IP block in MK81F256 and MK82F256, which supports decryption of the encrypted QuadSPI image on-the-fly using KeyBlob. See the MCU bootloader chapter in the silicon's reference manual for more details

2.2 Requirements

2.2.1 Hardware requirements

- TWR-K80F150M Tower System module
- FRDM-K82F Freedom Development platform
- TWR-KL82 Tower System module
- FRDM-KL82 Freedom Development platform

2.2.2 Host tools

The following host tools are available with the release package. They assist in generating and provisioning of QuadSPI bootable image for the target device.

- blhost: command line host tool for MCU bootloader.
- elftosb: command line host tool for SB file generation.
- KinetisFlashTool: GUI host tool for MCU bootloader.

2.2.3 Demo application

Led_demo running in internal flash and QuadSPI memory, under <sdk_package>/boards/<board>/bootloader_examples/demo_apps

QCBGenerator, under <*sdk_package*>*/middleware/mcu-boot/apps/QCBGenerator/build*

2.2.4 Required toolchains

2.2.4.1 Firmware project

The following toolchains can be used to build the example led_demo firmware application provided with the release package.

- ARM® Keil® development tool v5.26 with corresponding device pack
- IAR Embedded Workbench for ARM® v8.30.1
- MCUXpresso IDE v10.3

2.2.4.2 Host project

The following toolchains can be used to build the example QCBGenerator application provided with the release package.

- Microsoft Visual Studio® Professional 2015 for Windows® OS Desktop
- Codeblocks
- GCC v5.4.0

2.3 QuadSPI image boot procedure

To understand how to boot a QuadSPI image with MCU bootloader, it is necessary to understand the QuadSPI image boot flow. There are two types of QuadSPI image boot flow:

- Boot from a plaintext QuadSPI image. This method can be used on all targets with QuadSPI support.
- Boot from an encrypted QuadSPI image. This method can only be used on K8x processors that include OTFAD support, such as MK81F256 and MK82F256.

2.3.1 Plaintext QuadSPI image boot flow

The figure below shows the boot flow of MCU bootloader in booting the device with a plaintext QuadSPI image.

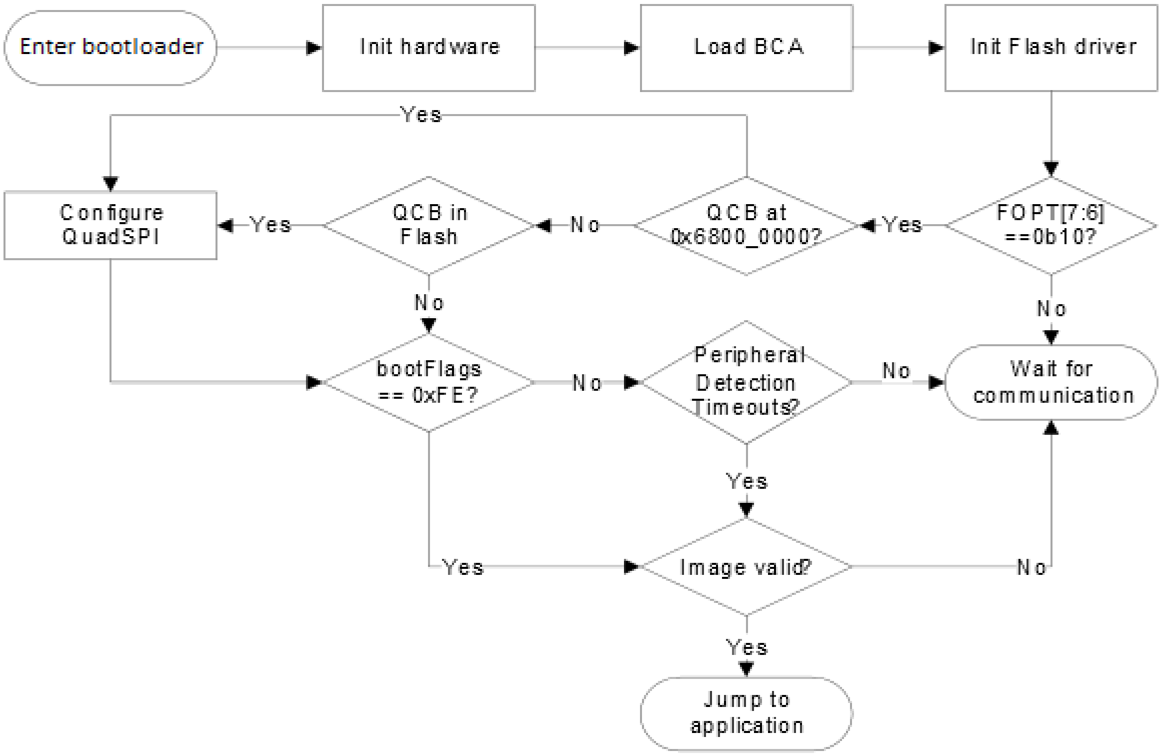


Figure 1. Plaintext QuadSPI image boot flow

2.3.2 Encrypted QuadSPI image boot flow

The below figure shows the boot flow of MCU bootloader in booting the device with an encrypted QuadSPI image.

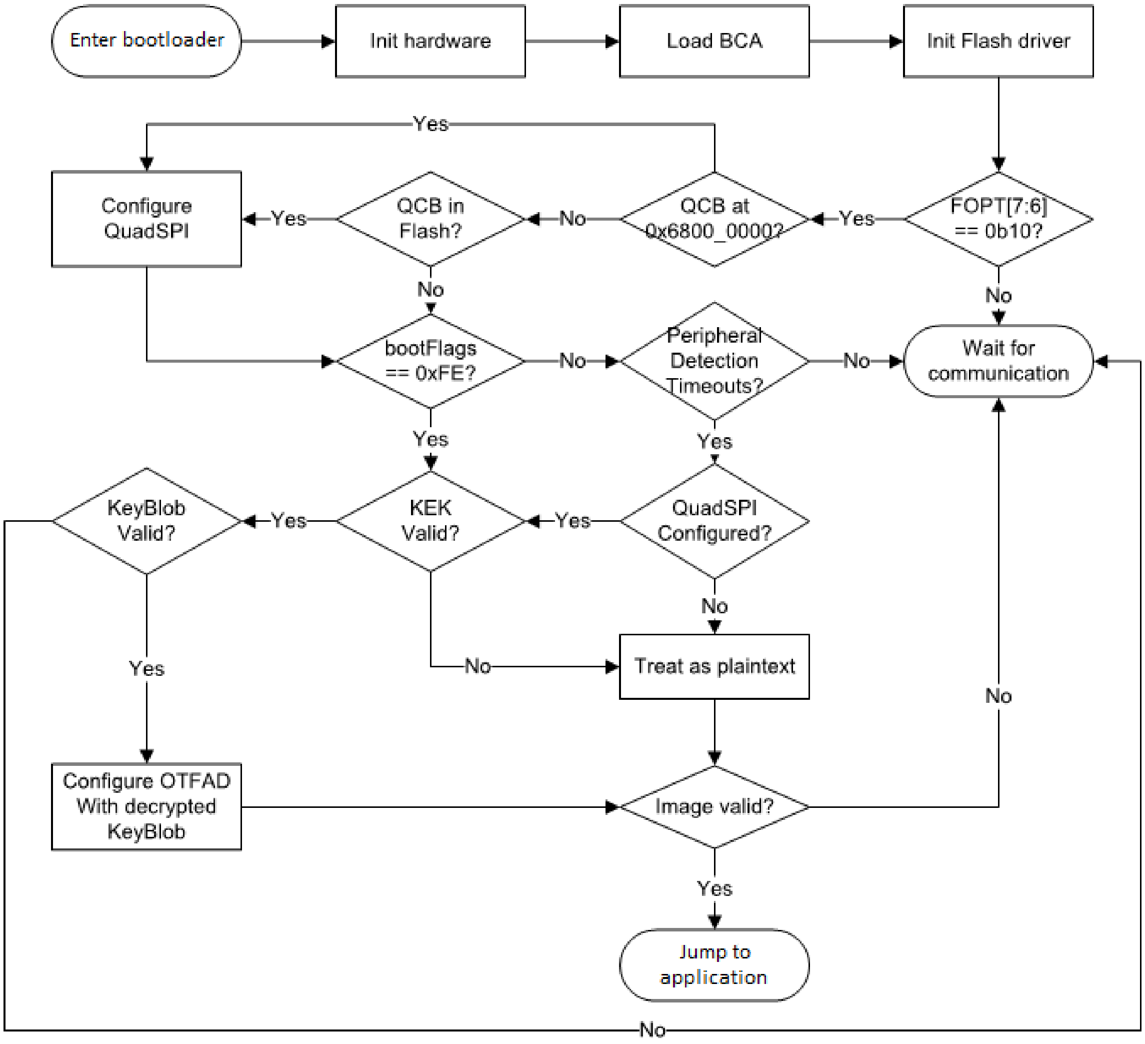


Figure 2. Encrypted QuadSPI image boot flow

3 Creating application for QuadSPI memory

This section describes how to modify a normal flash application (led_demo) to run from QuadSPI. The fully functional LED demo example for QuadSPI with source code can be found in <sdk_package>/boards/<board>/bootloader_examples/demo_apps. The chapter also discuss on how to create QCB data structure for a typical QuadSPI flash memory device.

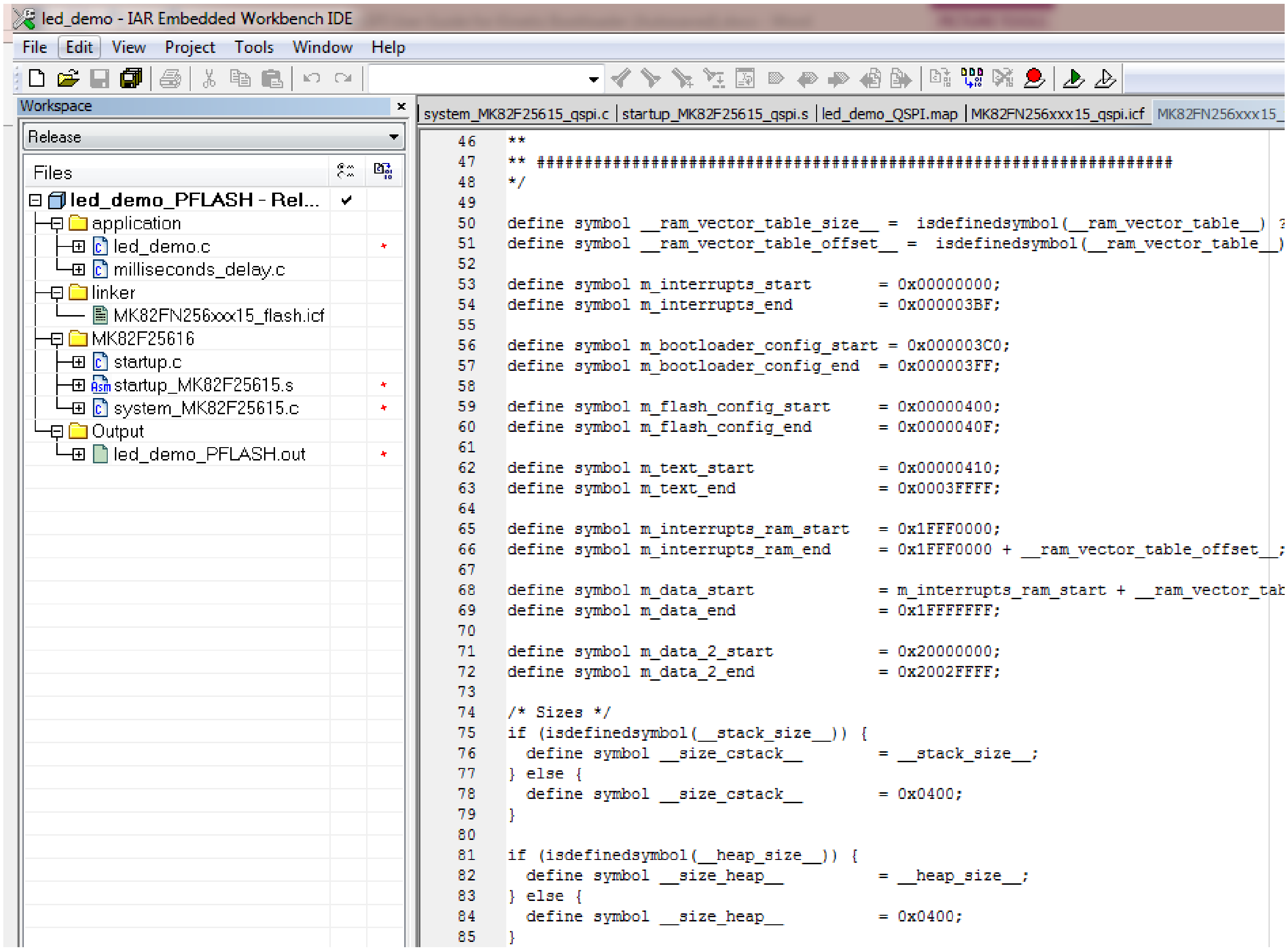
3.1 Starting point: Basics of internal flash memory mapped led-demo example project

Start from the LED demo example project code for the MK82F256 device. The example led-demo project files for each of the supported toolchains are available in *<sdk_package>/boards/<board>/bootloader_examples/demo_apps* folder of the package. This document focuses on the IAR project examples only. Open the *led_demo.eww* file from the *IAR* folder and select the led_demo_PFLASH project as the active project. See the following figure.

Note that the linker file for the led_demo_PFLASH project shows all sections located in the internal flash memory region, including the vector table, flash config area, and text sections.

When the led_demo_PFLASH image is built and flashed to the internal flash memory of the target device and begins its execution, it causes the blue and green LEDs to blink on the target board.

The subsequent sections show the changes needed to convert the led_demo_PLASH project to run on the QuadSPI memory for the target device.



```
**
** ###################################################################
*/

define symbol __ram_vector_table_size__  =  isdefinedsymbol(__ram_vector_table__) ?
define symbol __ram_vector_table_offset__  =  isdefinedsymbol(__ram_vector_table__)

define symbol m_interrupts_start       = 0x00000000;
define symbol m_interrupts_end         = 0x000003BF;

define symbol m_bootloader_config_start = 0x000003C0;
define symbol m_bootloader_config_end  = 0x000003FF;

define symbol m_flash_config_start     = 0x00000400;
define symbol m_flash_config_end       = 0x0000040F;

define symbol m_text_start             = 0x00000410;
define symbol m_text_end               = 0x0003FFFF;

define symbol m_interrupts_ram_start   = 0x1FFF0000;
define symbol m_interrupts_ram_end     = 0x1FFF0000 + __ram_vector_table_offset__;

define symbol m_data_start             = m_interrupts_ram_start + __ram_vector_tab
define symbol m_data_end               = 0x1FFFFFFF;

define symbol m_data_2_start           = 0x20000000;
define symbol m_data_2_end             = 0x2002FFFF;

/* Sizes */
if (isdefinedsymbol(__stack_size__)) {
  define symbol __size_cstack__        = __stack_size__;
} else {
  define symbol __size_cstack__        = 0x0400;
}

if (isdefinedsymbol(__heap_size__)) {
  define symbol __size_heap__          = __heap_size__;
} else {
  define symbol __size_heap__          = 0x0400;
}
```

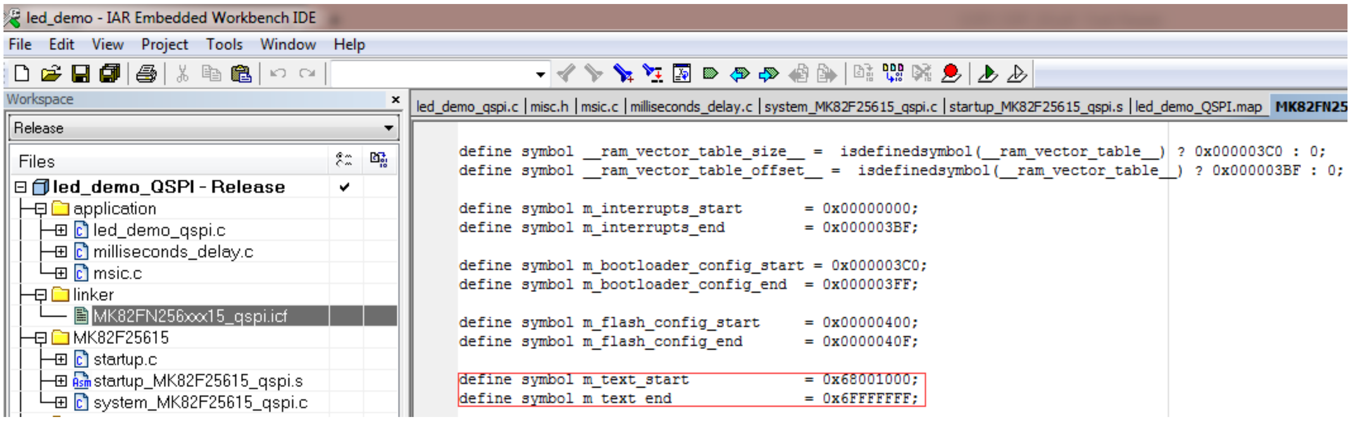
Figure 3. The led_demo_PFLASH project

3.2 Changes to the led-demo project

The following subsections describe the steps to map the led-demo to run from the external QuadSPI flash memory.

3.2.1 Changes to the linker file

The first step is to update the linker file. The `m_text_start`, and `m_text_end` symbol names must be updated. The address of `m_text_start` should be changed to 0x68001000, and `m_text_end` to 0x6FFFFFFF or the actual end address of the selected SPI flash device. See the changes in the following figure.

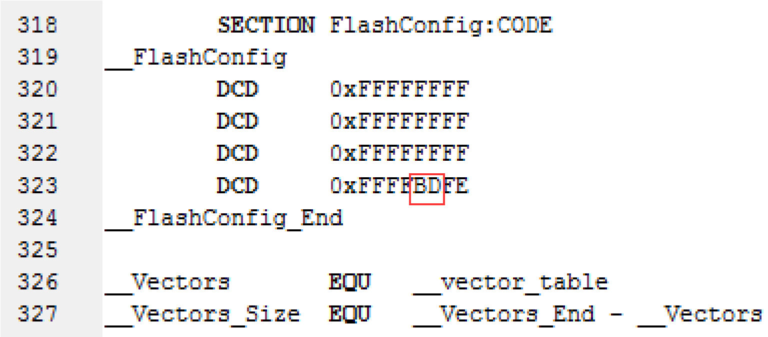

led_demo - IAR Embedded Workbench IDE
File Edit View Project Tools Window Help
Workspace
Release
Files
led_demo_QSPI - Release
application
led_demo_qspi.c
milliseconds_delay.c
msic.c
linker
MK82FN256xxx15_qspi.icf
MK82F25615
startup.c
startup_MK82F25615_qspi.s
system_MK82F25615_qspi.c
led_demo_qspi.c | misc.h | msic.c | milliseconds_delay.c | system_MK82F25615_qspi.c | startup_MK82F25615_qspi.s | led_demo_QSPI.map | MK82FN25

```
define symbol __ram_vector_table_size__ =  isdefinedsymbol(__ram_vector_table__) ? 0x000003C0 : 0;
define symbol __ram_vector_table_offset__ =  isdefinedsymbol(__ram_vector_table__) ? 0x000003BF : 0;

define symbol m_interrupts_start       = 0x00000000;
define symbol m_interrupts_end         = 0x000003BF;

define symbol m_bootloader_config_start = 0x000003C0;
define symbol m_bootloader_config_end  = 0x000003FF;

define symbol m_flash_config_start     = 0x00000400;
define symbol m_flash_config_end       = 0x0000040F;

define symbol m_text_start             = 0x68001000;
define symbol m_text_end               = 0x6FFFFFFF;
```

Figure 4. Linker file changes

3.2.2 Changes to flash config area

The bit 7-6 in the FOPT (0x40D) must be changed to 0b’10 to select the ROM as the boot source upon reset. The QuadSPI is configured after the ROM starts and when the QCB is present. After this operation, the flash config area is changed, as shown in the following figure.



```
        SECTION FlashConfig:CODE
__FlashConfig
        DCD     0xFFFFFFFF
        DCD     0xFFFFFFFF
        DCD     0xFFFFFFFF
        DCD     0xFFFFBDFE
__FlashConfig_End

__Vectors       EQU   __vector_table
__Vectors_Size  EQU   __Vectors_End - __Vectors
```

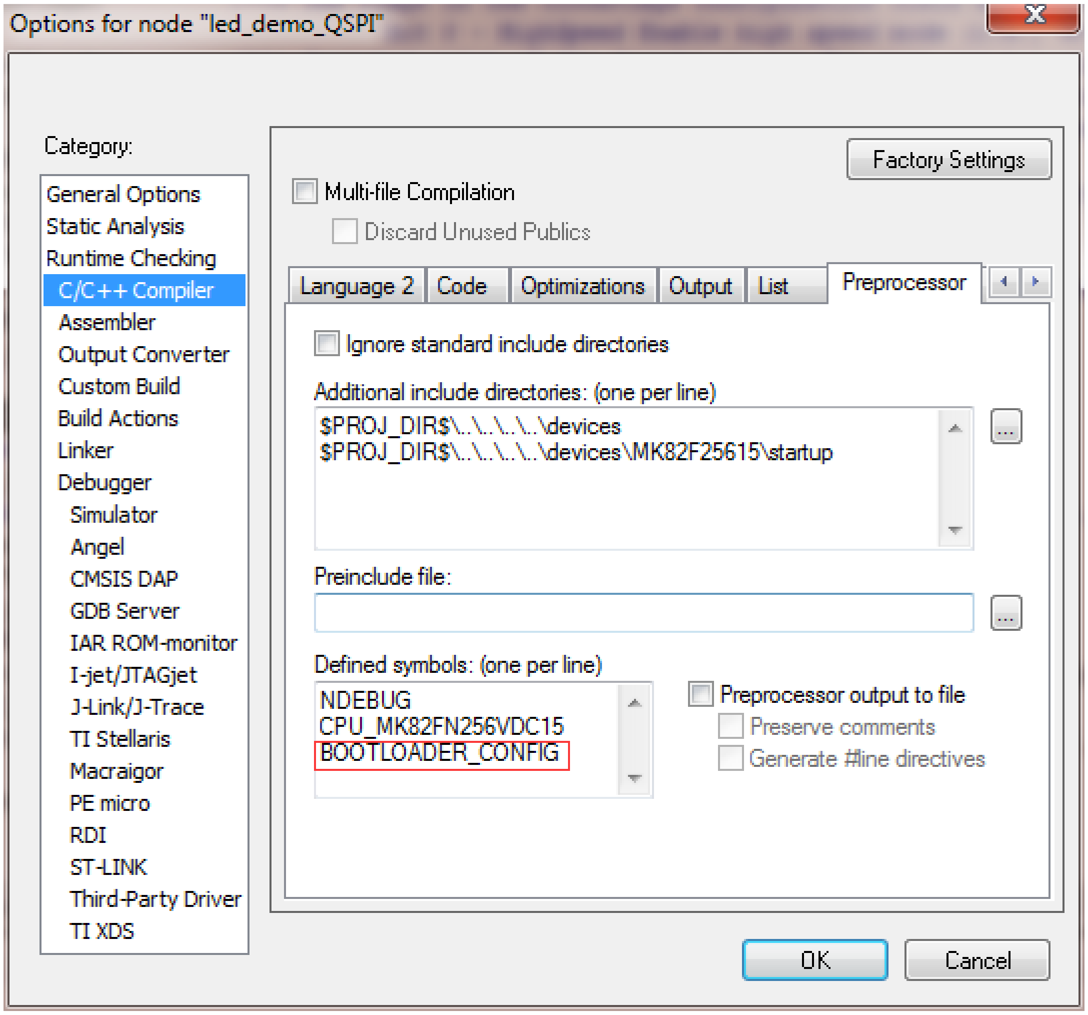
Figure 5. Change flash config area for QuadSPI image

See the *startup_MK82F2515_qspi.s* file in the led_demo-><*sdk_package*>*/middleware/mcu-boot/apps/demo_qspi/led_demo/devices/MK80F25615/startup/*<*toolchain*> folder for more details.

3.2.3 Configure BCA

After the previous step, the target is able to run the led-demo application once the active peripheral detection timeout occurs.

To customize the boot option for the QuadSPI image, the BCA is required. The first step to is to define BOOTLOADER_CONFIG in the project. Implement the operation shown in the following figure for IAR EWARM toolchain as an example.



□ Multi-file Compilation

□ Discard Unused Publics

□ Ignore standard include directories

Additional include directories: (one per line)

```
$PROJ_DIR$\..\..\..\..\devices
$PROJ_DIR$\..\..\..\..\devices\MK82F25615\startup
```

Preinclude file:

Defined symbols: (one per line)

```
NDEBUG
CPU_MK82FN256VDC15
BOOTLOADER_CONFIG
```

□ Preprocessor output to file

□ Preserve comments

□ Generate #line directives

Figure 6. Enable BCA in EWARM

There are two ways to configure the QuadSPI image boot option:

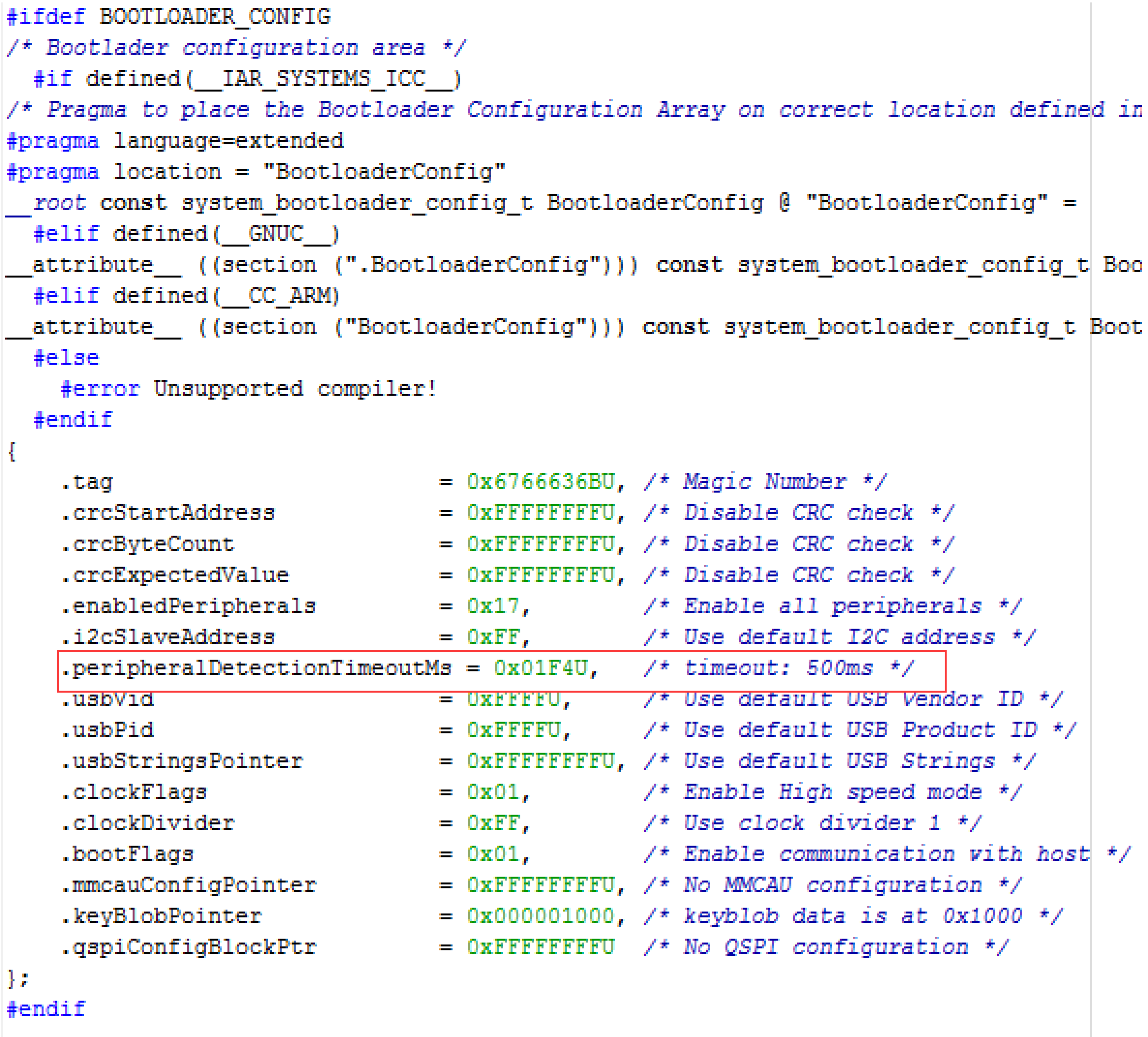
1. Change the peripheralDetectionTimeoutMs. For example, change it to 0x01F4 (500 ms).
2. Change the bootFlags to 0xFE, which means boot directly from application without delay. To re-enter MCU bootloader again, see Appendix B.

NOTE

The first way to configure the QuadSPI image boot option is recommended.

In this example, there is a BootloaderConfig constant variable defined in system_MK82F25615.c. It can be changed as shown in the following figure.

When the BCA change is complete, the target supports execution of led demo image if it has been programmed to internal flash or QuadSPI memory.



```
#ifdef BOOTLOADER_CONFIG
/* Bootlader configuration area */
  #if defined(__IAR_SYSTEMS_ICC__)
/* Pragma to place the Bootloader Configuration Array on correct location defined in
#pragma language=extended
#pragma location = "BootloaderConfig"
__root const system_bootloader_config_t BootloaderConfig @ "BootloaderConfig" =
  #elif defined(__GNUC__)
__attribute__ ((section (".BootloaderConfig"))) const system_bootloader_config_t Boo
  #elif defined(__CC_ARM)
__attribute__ ((section ("BootloaderConfig"))) const system_bootloader_config_t Boot
  #else
    #error Unsupported compiler!
  #endif
{
    .tag                            = 0x6766636BU, /* Magic Number */
    .crcStartAddress                = 0xFFFFFFFFU, /* Disable CRC check */
    .crcByteCount                   = 0xFFFFFFFFU, /* Disable CRC check */
    .crcExpectedValue               = 0xFFFFFFFFU, /* Disable CRC check */
    .enabledPeripherals             = 0x17,        /* Enable all peripherals */
    .i2cSlaveAddress                = 0xFF,        /* Use default I2C address */
    .peripheralDetectionTimeoutMs = 0x01F4U,       /* timeout: 500ms */
    .usbVid                         = 0xFFFFU,     /* Use default USB Vendor ID */
    .usbPid                         = 0xFFFFU,     /* Use default USB Product ID */
    .usbStringsPointer              = 0xFFFFFFFFU, /* Use default USB Strings */
    .clockFlags                     = 0x01,        /* Enable High speed mode */
    .clockDivider                   = 0xFF,        /* Use clock divider 1 */
    .bootFlags                      = 0x01,        /* Enable communication with host */
    .mmcauConfigPointer             = 0xFFFFFFFFU, /* No MMCAU configuration */
    .keyBlobPointer                 = 0x000001000, /* keyblob data is at 0x1000 */
    .qspiConfigBlockPtr             = 0xFFFFFFFFU  /* No QSPI configuration */
};
#endif
```

Figure 7. Set peripheralDetectionTimeoutMs to 500 ms

3.3 Generate QCB

QuadSPI Config Block (QCB) is required for MCU ROM bootloader to properly configure and access the QuadSPI device. This section shows the QCB structure, determines the QCB parameters for the specified SPI flash device, and generates the QCB with a simple project.

3.3.1 The QCB structure

The QCB is a data structure containing the most common used parameters for QuadSPI module. See the MCU bootloader chapter in the silicon's reference manual for more details. The QCB is organized as follows.

Table 2. QuadSPI configuration block

Offset	Size (bytes)	Configuration field	Description
0x00 - 0x03	4	tag	Magic number to verify whether QCB is valid. Must be set to 'kqcf'. [31:24] - 'f' (0x66) [23:16] - 'c' (0x63) [15: 8] - 'q'(0x71) [7: 0] - 'k'(0x6B)
0x04 - 0x07	4	version	Version number of QuadSPI config block. [31:24] - name: must be 'Q'(0x51) [23:16] - major: must be 1 [15: 8] - minor: must be 1 [7: 0] - bugfix: must be 0
0x08 - 0x0b	4	lengthInBytes	Size of QuadSPI config block, in terms of bytes. Must be 512.
0x0c - 0x0f	4	dqs_loopback	Enable DQS loopback support: 0 DQS loopback is disabled. 1 DQS loopback is enabledd, the DQS loopback mode is determined by subsequent **'*dqs_loopback_internal*'** field.
0x10 - 0x13	4	data_hold_time	Serial flash data hold time. Valid value 0/1/2. See the QuadSPI Chapter for details.
0x14 - 0x1b	8	-	Reserved.
0x1c - 0x1f	4	device_mode_config_en	Configure work mode enable for external flash devices: 0 Disabled - ROM does not configure work mode of external flash devices. 1 Enabled - ROM configures work mode of external flash devices based on "device_cmd" and LUT entries indicated by "write_cmd_ipcr".
0x20 - 0x23	4	device_cmd	Command to configure work mode of external flash devices. Effective only if "device_mode_config_en" is set to 1. This command is device-specific.
0x24 - 0x27	4	write_cmd_ipcr	IPCR pointed to LUT index for the command sequence of configuring the device to work mode. Value = index<<24

Table continues on the next page...

Table 2. QuadSPI configuration block (continued)

Offset	Size (bytes)	Configuration field	Description
0x28 - 0x2b	4	word_addressable	Word addressable: 0 Byte addressable serial flash mode. 1 Word addressable serial flash mode.
0x2c - 0x2f	4	cs_hold_time	Serial flash CS hold time in terms of flash clock cycles.
0x30 - 0x33	4	cs_setup_time	Serial flash CS setup time in terms of flash clock cycles.
0x34 - 0x37	4	sflash_A1_size	Size of external flash connected to ports of QSPI0A and QSPI0A_CS0, in terms of bytes.
0x38 - 0x3b	4	sflash_A2_size	Size of external flash connected to ports of QSPI0B and quadSPI0A_CS1, in terms of bytes. This field must be set to 0 if the serial flash devices are not present.
0x3c - 0x3f	4	sflash_B1_size	Size of external flash connected to ports of QSPI0B and quadSPI0B_CS0, in terms of bytes. This field must be set to 0 if the serial flash devices are not present.
0x40 - 0x43	4	sflash_B2_size	Size of external flash connected to ports of QSPI0B and quadSPI0B_CS1, in terms of bytes. This field must be set to 0 if the serial flash devices are not present.
0x44 - 0x47	4	sclk_freq	Frequency of QuadSPI serial clock: 0 Low frequency 1 Mid frequency 2 High frequency See the MCU bootloader chapter in silicon's reference manual for the definition of low-frequency, mid-frequency and high-frequency. In MK82F256, they are 24 MHz, 48 MHz, and 96 MHz.

Table continues on the next page...

Table 2. QuadSPI configuration block (continued)

Offset	Size (bytes)	Configuration field	Description
0x48 - 0x4b	4	busy_bit_offset	Busy bit offset in status register of Serial flash [31:16]: 0 - Busy flag in status register is 1 when flash devices are busy. 1 - Busy flag in status register is 0 when flash devices are busy. [15:0]: The offset of busy flag in status register, valid range 0-31.
0x4c - 0x4f	4	sflash_type	Type of serial flash: 0 Single-pad 1 Dual-pad 2 Quad-pad 3 Octal-pad
0x50 - 0x53	4	sflash_port	Port enablement for QuadSPI module: 0 Only pins for QSPI0A are enabled. 1 Pins for both QSPI0A and QSPI0B are enabled.
0x54 - 0x57	4	ddr_mode_enable	Enable DDR mode: 0 DDR mode is disabled. 1 DDR mode is enabled.
0x58 - 0x5b	4	dqs_enable	Enable DQS: 0 DQS is disabled. 1 DQS is enabled.
0x5c - 0x5f	4	parallel_mode_enable	Enable Parallel Mode: 0 Parallel mode is disabled. 1 Parallel mode is enabled.
0x60 - 0x63	4	portA_cs1	Enable QuadSPI0A_CS1: 0 QuadSPI0A_CS1 is disabled. 1 QuadSPI0A_CS1 is enabled. This field must be set to 1 if sflash_A2_size is not equal to 0.

Table continues on the next page...

Table 2. QuadSPI configuration block (continued)

Offset	Size (bytes)	Configuration field	Description
0x64 - 0x67	4	portB_cs1	Enable QuadSPI0B_CS1 0 QuadSPI0B_CS1 is disabled 1 QuadSPI0B_CS1 is enabled This field must be set to 1 if sflash_B2_size is not equal to 0.
0x68 - 0x6b	4	fsphs	Full Speed Phase selection for SDR instructions: 0 Select sampling at non-inverted clock. 1 Select sampling inverted clock.
0x6c - 0x6f	4	fsdly	Full Speed Delay selection for SDR instructions: 0 One clock cycle delay. 1 Two clock cycles delay.
0x70 - 0x73	4	ddrsmp	DDR sampling point: Valid range: 0 - 7.
0x74 - 0x173	256	look_up_table	Look-up-table for sequences of instructions. See the QuadSPI chapter in silicon's reference manual for more details.
0x174 - 0x177	4	column_address_space	Column Address Space: The parameter defines the width of the column address.
0x178 - 0x17b	4	config_cmd_en	Enable additional configuration command: 0 Additional configuration command is not needed. 1 Additional configuration command is needed.
0x17c - 0x18b	16	config_cmds	IPCR arrays for each connected SPI flash. "config_cmds[n]" provides IPCR value, namely seq_id << 24. All fields must be set to 0 if config_cmd_en is not set.
0x18c - 0x19b	16	config_cmds_args	Command arrays needed to be transferred to external SPI flash. "config_cmds_args[n]" provides commands to be written. All fields must be set to 0 if config_cmd_en is not asserted.

Table continues on the next page...

Table 2. QuadSPI configuration block (continued)

Offset	Size (bytes)	Configuration field	Description
0x19c - 0x19f	4	differential_clock_pin_enable	Enable differential flash clock pin: 0 Differential flash clock pin is disabled. 1 Differential flash clock pin is enabled.
0x1a0 - 0x1a3	4	flash_CK2_clock_pin_enable	Enable flash CK2 clock pin: 0 Flash CK2 clock pin is disabled. 1 Flash CK2 clock pin is enabled.
0x1a4 - 0x1a7	4	dqs_inverse_sel	Select clock source for internal DQS generation: 0 Use 1x internal reference clock for DQS generation. 1 Use inverse 1x internal reference clock for the DQS generation.
0x1a8 - 0x1ab	4	dqs_latency_enable	DQS Latency Enable: 0 DQS latency disabled. 1 DQS feature with latency included enabled.
0x1ac - 0x1af	4	dqs_loopback_internal	DQS loop back from internal DQS signal or DQS Pad: 0 DQS loop back is sent to DQS pad first and then looped back to QuadSPI. 1 DQS loop back from internal DQS signal directly.
0x1b0 - 0x1b3	4	dqs_phase_sel	Select Phase Shift for internal DQS generation: 0 No Phase shift. 1 Select 45 degree phase shift. 2 Select 90 degree phase shift. 3 Select 135 degree phase shift.
0x1b4 - 0x1b7	4	dqs_fa_delay_chain_sel	Delay chain tap number selection for QuadSPI0A DQS: Valid range: 0-63
0x1b8 - 0x1bb	4	dqs_fb_delay_chain_sel	Delay chain tap number selection for QuadSPI0B DQS: Valid range: 0-63
0x1bc - 0x1c3	8	-	Reserved.
0x1c4 - 0x1c7	4	page_size	Page size of external flash. Page size of all SPI flash devices must be the same.

Table continues on the next page...

Table 2. QuadSPI configuration block (continued)

Offset	Size (bytes)	Configuration field	Description
0x1c8 - 0x1cb	4	sector_size	Sector size of exernal SPI in flash. Sector size of all SPI flash devices must be the same.
0x1cc - 0x1cf	4	timeout_milliseconds	Timeout in terms of milliseconds: 0 Timeout check is disabled. Other: QuadSPI Driver returns timeout if the time that external SPI devices are busy lasts more than this value.
0x1d0 - 0x1d3	4	ips_cmd_second_divider	Second driver for IPs command based on QSPI_MCR[SCLKCFG], the maximum value of QSPI_MCR[SCLKCFG] depends on specific devices.
0x1d4 - 0x1d7	4	need_multi_phase	0 Only one phase is needed to access external flash devices. 1 Multiple phases are needed to erase/program external flash devices.
0x1d8 - 0x1db	4	is_spansion_hyperflash	0 External flash devices do not belong to Cypress HyperFlash family. 1 External flash devices belong to Cypress HyperFlash family.
0x1dc - 0x1df	4	pre_read_status_cmd_address_offset	Additional address for the PreReadStatus command. Set this field to 0xFFFFFFFF if it is not required.
0x1e0 - 0x1e3	4	pre_unlock_cmd_address_offset	Additional address for PreWriteEnable command. Set this field to 0xFFFFFFFF if it is not required.
0x1e4 - 0x1e7	4	unlock_cmd_address_offset	Additional address for WriteEnable command. Set this field to 0xFFFFFFFFF if it is not required.
0x1e8 - 0x1eb	4	pre_program_cmd_address_offset	Addtional address for PrePageProgram command. Set this field to 0xFFFFFFFF if it is not required.
0x1ec - 0x1ef	4	pre_erase_cmd_address_offset	Additional address for PreErase command. Set this field to 0xFFFFFFFF if it is not required.
0x1f0 - 0x1f3	4	erase_all_cmd_address_offset	Additional address for EraseAll command. Set this field to 0xFFFFFFFF if it is not required.
0x1f4 - 0x1ff	12	-	Reserved.

NOTE

1. Though there are several parameters in QCB, only a few parameters need to be configured for most SPI flash devices available on the market. See the example QCB for more details.
2. While using MCU ROM bootloader, make sure the "ips_cmd_second_divider" is not greater than 0x08 under SDR mode, and it is not greater than 2 under DDR mode.
3. It is recommended to configure QSPI to SDR mode and switch to DDR mode in the application where possible to achieve higher program performance with MCU bootloader.

In the QCB, the most important field is the Lookup Table (LUT), which contains command sequence for QuadSPI instructions, such as erase, read, and program. The command sequence in the LUT should appear in the order as shown in the following table:

Table 3. Look-up table entries for MCU bootloader

Index	Field	Description
0	Read	Sequence for read instructions.
1	WriteEnable	Sequence for WriteEnable instructions.
2	EraseAll	Sequence for EraseAll instructions, optional.
3	ReadStatus	Sequence for ReadStatus instructions.
4	PageProgram	Sequence for Page Program instructions.
6	PreErase	Sequence for Pre-Erase instructions.
7	SectorErase	Sequence for Sector Erase.
8	Dummy	Sequence for dummy operation if needed For example, if continuous read is configured in index 0, the dummy LUT should be configured to force external SPI flash to exit continuous read mode. If it is not required, this LUT entry must be set to 0.
9	PreWriteEnable	Sequence for Pre-WriteEnable instructions.
10	PrePageProgram	Sequence for Pre-PageProgram instructions.
11	PreReadStatus	Sequence for Pre-ReadStatus instructions.
5, 12, 13, 14, 15	Undefined	All of these sequences are free to be used for other purposes. I.e., index 5 can be used for enabling Quad mode of SPI flash devices. For more details, see Section 3.3.2, "Example QCB for MX25U3235F device on TWR-K80F150M Tower System module".

For most types of SPI, flash devices are available in the market. Only index 0, 1, 3, 4, 7, and 8 are required. However, for other types of high-end SPI flash devices, such as Cypress HyperFlash, additional indexes listed above may be required.

3.3.2 Example QCB for MX25U3235F device on TWR-K80F150M Tower System module

This section creates an example QCB data structure for TWR-K80F150M Tower System module. There are two MX25U3235F QuadSPI flash devices connected to QuadSPI0A and QuadSPI0B ports, respectively, on the board. The datasheet for MX25U3235F are available on the MXIC website, and the schematics for the TWR-K80F150M Tower System module is available on the NXP website.

The following are some attributes which are essential to create the QCB for the MX25U3235F flash device. The same (but not limited to the following) information can be found in its data sheet as well:

Table 4. MX25U3235F features for QuadSPI configuration

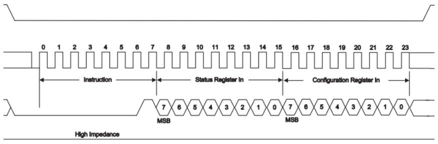
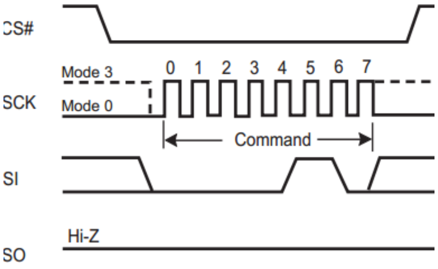
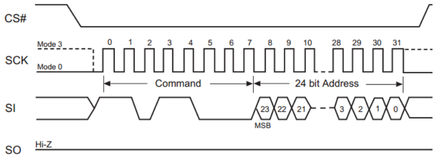
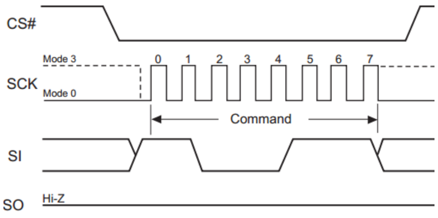
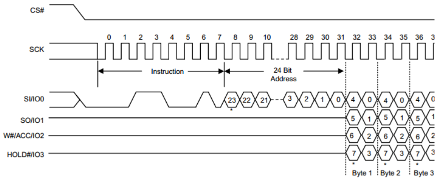
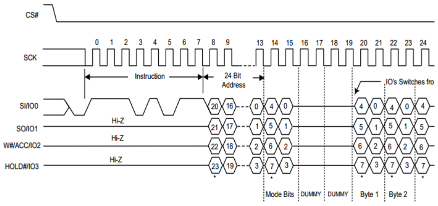
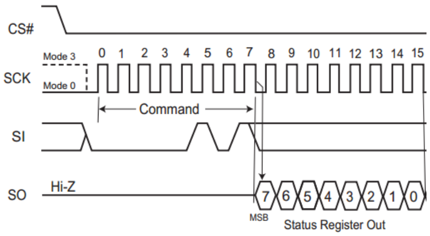
Attribute	Value/timing	Description
Maximum supported frequency (4 I/O)	104 MHz (6 dummy cycles)	-
Page size	256 bytes	-
Sector size	4 KB/32 KB/64 KB	4 KB is selected in this guide.
Chip size	4 MB	-
Busy/WriteInProgress bit in status register	Bit 0	Bit 0 in status registers is called busy flag. 1 means SPI flash device is busy. 0 means it is idle. The value needs to be set to 'busy_bit_offset' in QCB.
Enable Quad mode	 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 Instruction Status Register In Configuration Register In 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 MSB MSB High Impedance	Write status register, bit6 must be set to 1 in order to enable Quad mode. Following the QuadSPI chapter, the command sequence for this operation is: 1. CMD: 01, single pad 2. Write: length=1, single pad The data to be written is 0x40, and is configured to 'device_cmd' in QCB.
Write Enable	 CS# Mode 3 0 1 2 3 4 5 6 7 SCK Mode 0 Command SI Hi-Z SO	This is required before issuing any write/ erase operations to SPI flash devices. The command sequence for this operation is: 1. CMD: 0x06, single pad
Sector Erase	 CS# Mode 3 0 1 2 3 4 5 6 7 8 9 10 28 29 30 31 SCK Mode 0 Command 24 bit Address SI 23 22 21 3 2 1 0 MSB SO Hi-Z	Each sector must be erased before doing any program operation. The command sequence for this operation is: 1. CMD: 0x20, single pad 2. ADDR: 0x18 (24-bit address), single pad

Table continues on the next page...

Table 4. MX25U3235F features for QuadSPI configuration (continued)

Attribute	Value/timing	Description
Chip Erase	 CS# SCK Mode 3 Mode 0 0 1 2 3 4 5 6 7 Command SI SO Hi-Z	This command can be used to erase the entire content on SPI flash device. The command sequence for this operation is: 1. CMD: 0x60, single pad
4 x I/O Page program	 CS# SCK 0 1 2 3 4 5 6 7 8 9 10 28 29 30 31 32 33 34 35 36 Instruction 24 Bit Address SI/IO0 SO/IO1 WP#/ACC/IO2 HOLD#/IO3 Byte 1 Byte 2 Byte 3	This command is used to program the desired data to SPI flash device. Here, we use 4 x I/O page program command in order to improve the program performance. The command sequences for this operation are: 1. CMD: 0x38, single pad 2. ADDR: 0x18 (24 bit address) quad pads 3. WRITE: 0x40 (ignore this value) quad pads
4 I/O Read	 CS# SCK 0 1 2 3 4 5 6 7 8 9 13 14 15 16 17 18 19 20 21 22 23 24 Instruction 24 Bit Address IO's Switches fro SI/IO0 SO/IO1 WP#/ACC/IO2 HOLD#/IO3 Hi-Z Mode Bits DUMMY DUMMY Byte 1 Byte 2	This command is used to read data from SPI flash device. Here, we use 4 x I/O Read in order to improve read performance. The command sequence for this operation is: 1. CMD: 0xEB, single pad 2. ADDR: 0x18 (24 bit address) quad pads 3. DUMMY: 0x06 (6 cycles) quad pads 4. READ: 0x80 (128 byte at one pass) quad pads 5. JUMP_ON_CS: 0 (single pad)
Read Status	 CS# SCK Mode 3 Mode 0 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 Command SI SO Hi-Z 7 6 5 4 3 2 1 0 MSB Status Register Out	This command is used to check if the SPI flash device is busy after having issued a program/erase command to it. The command sequence for this operation is: 1. CMD: 0x05, single pad 2. READ: 1 (byte) single pad

The information needed for QCB creation for the TWR-K80F150M Tower System module is summarized in Table 4. The “Programmable Sequence Engine” and "Example Sequences" sections within the QuadSPI chapter of the MK80F256 Reference Manual can be referenced to create customized QCBs. The “Description” column in Table 4 also provides the LUT instructions for each command.

Based on the above summary, the ‘qspi_config_block_generator’ project is provided with the package as an example along with this user's guide. The example project can be used as a basis to generate customized QCBs.

3.3.3 Generate the QCB with a simple example project

The project can be found in the package at location *<sdk_package>/middleware/mcu-boot/apps/QCBGenerator/build*. Currently, two projects are provided to build from toolchains Microsoft Visual Studio 2013 and codeblocks. Launch Microsoft Visual Studio example project available in the Visual Studio folder. Edit the file qspi_config_block_generator.c to configure qspi_config_block in the main function.

There are two examples using different ways to enable Quad mode. The first one enables Quad mode using the device mode config feature, while the second one enables Quad mode using the config cmd feature. See the below examples for more details.

1. QCB using the device mode config feature:

```
const qspi_config_t qspi_config_block = {
    .tag = kQspiConfigTag,
    .version = { .version = kQspiVersionTag },
    .lengthInBytes = 512,
    .sflash_A1_size = 0x400000,                 // 4MB - MX25U3235F connected to QSPI0A
    .sflash_B1_size = 0x400000,                 // 4MB - MX25U3235F connected to QSPI0B
    // In K80 ROM bootloader, QSPI serial clock frequency is 96MHz
    .sclk_freq = kQspiSerialClockFreq_High,     // High frequency, 96MHz / 1 = 96MHz
    .sflash_type = kQspiFlashPad_Quad,          // SPI Flash devices work under quad-pad mode
    .sflash_port = kQspiPort_EnableBothPorts,   // Both QSPI0A and QSPI0B are enabled.
    .busy_bit_offset = 0,                       // Busy offset is 0
    .ddr_mode_enable = 0,                       // disable DDR mode
    .dqs_enable = 0,                            // Disable DQS feature
    .parallel_mode_enable = 0,                  // QuadSPI module work under serial mode
    .pagesize = 256,                            // Page Size: 256 bytes
    .sectorsize = 0x1000,                       // Sector Size: 4KB
    .device_mode_config_en = 1,                 // configure quad mode for SPI flash device
    .device_cmd = 0x40,                         // Enable quad mode
    .write_cmd_ipcr = 0x05000000U,              // IPCR indicating enable seqid (5<<24)
    // Set second divider for QSPI serial clock to 3 if K80 ROM Bootloader cannot program
    // SPI flash at 96 MHz, in this configuration, the program speed is 96MHz/4 = 24MHz
    .ips_command_second_divider = 3,
    .look_up_table = {
        // Seq0: Quad Read (maximum supported freq: 104MHz)
        /*
         * CMD: 0xEB - Quad Read, Single pad
         * ADDR: 0x18 - 24bit address, Quad pads
         * DUMMY: 0x06 - 6 clock cycles, Quad pads
         * READ: 0x80 - Read 128 bytes, Quad pads
         * JUMP_ON_CS: 0
         */
        [0] = 0x0A1804EB,
        [1] = 0x1E800E06,
        [2] = 0x2400,

        // Seq1: Write Enable (maximum supported freq: 104MHz)
        /* CMD: 0x06 - Write Enable, Single pad */
        [4] = 0x406,
        // Seq2: Erase all (maximum supported freq: 104MHz)
        /* CMD: 0x60 - Erase All chip, Single pad */
        [8] = 0x460,

        // Seq3: Read Status (maximum supported freq: 104MHz)
        /*
         * CMD: 0x05 - Read Status, single pad
         * READ: 0x01 - Read 1 byte
         */
        [12] = 0x1c010405,

        // Seq4: 4 I/O Page Program (maximum supported freq: 104MHz)
        /*
         * CMD: 0x38 - 4 I/O Page Program, Single pad
         * ADDR: 0x18 - 24bit address, Quad pad
         * WRITE: 0x40 - Write 64 bytes at one pass, Quad pad,
         * (Ignore the 64, because it will be overwritten by page size)
         */
        [16] = 0x0A180438,
        [17] = 0x2240,

        // Seq5: Write status register to enable quad mode
        /*
         * CMD: 0x01 - Write Status Register, single pad
         * WRITE: 0x01 - Write 1 byte of data, single pad
         */
        [20] = 0x20010401,

        // Seq7: Erase Sector
        /*
         * CMD: 0x20 - Sector Erase, single pad
         * ADDR: 0x18 - 24 bit address, single pad
         */
        [28] = 0x08180420,

        // Seq8: Dummy
        /*
         * CMD: 0 - Dummy command, used to force SPI flash to exit continuous read mode.
         * Unnecessary here because the continuous read mode isn't enabled.
         */
        [32] = 0,
    },
};
```

2. QCB using the config cmd feature:

```
const qspi_config_t qspi_config_block = {
    .tag = kQspiConfigTag,
    .version = {.version = kQspiVersionTag},
    .lengthInBytes = 512,
    .sflash_A1_size = 0x400000,               // 4MB
    .sclk_freq = kQspiSerialClockFreq_High,   // High frequency, in K82, it means 96MHz/1 = 96MHz
    .sflash_type = kQspiFlashPad_Quad,        // SPI Flash devices work under quad-pad mode
    .sflash_port = kQspiPort_EnableBothPorts, // Both QSPI0A and QSPI0B are enabled.
    .busy_bit_offset = 0,                     // Busy offset is 0
    .ddr_mode_enable = 0,                     // disable DDR mode
    .dqs_enable = 0,                          // Disable DQS feature
    .parallel_mode_enable = 0,                // QuadSPI module work under serial mode
    .pagesize = 256,                          // Page Size : 256 bytes
    .sectorsize = 0x1000,                     // Sector Size: 4KB
    .config_cmd_en = 1,                       // Enable config cmd feature
    .config_cmds = {[0] = 5UL << 24},         // IPCR indicating seq id for Quad Mode Enable
    .config_cmds_args = {[0] = 0x40},         // Enable quad mode
    .ips_command_second_divider = 3,          // Set second divider for QSPI serial clock to 3
    .look_up_table = {
        // Seq0 : Quad Read (maximum supported freq: 104MHz)
        /*
         * CMD: 0xEB - Quad Read, Single pad
         * ADDR: 0x18 - 24bit address, Quad pads
         * DUMMY: 0x06 - 6 clock cycles, Quad pads
         * READ: 0x80 - Read 128 bytes, Quad pads
         * JUMP_ON_CS: 0
         */
        [0] = 0x0A1804EB, [1] = 0x1E800E06, [2] = 0x2400,

        // Seq1: Write Enable (maximum supported freq: 104MHz)
        /* CMD: 0x06 - Write Enable, Single pad */
        [4] = 0x406,

        // Seq2: Erase All (maximum supported freq: 104MHz)
        /* CMD: 0x60 - Erase All chip, Single pad */
        [8] = 0x460,

        // Seq3: Read Status (maximum supported freq: 104MHz)
        /*
         * CMD: 0x05 - Read Status, single pad
         * READ: 0x01 - Read 1 byte
         */
        [12] = 0x1c010405,

        // Seq4: 4 I/O Page Program (maximum supported freq: 104MHz)
        /*
         * CMD: 0x38 - 4 I/O Page Program, Single pad
         * ADDR: 0x18 - 24bit address, Quad pad
         * WRITE: 0x40 - Write 64 bytes at one pass, Quad pad
         */
        [16] = 0x0A180438, [17] = 0x2240,
        // Seq5: Write status register to enable quad mode
        /*
         * CMD: 0x01 - Write Status Register, single pad
         * WRITE: 0x01 - Write 1 byte of data, single pad
         */
        [20] = 0x20010401,

        // Seq7: Erase Sector
        /*
         * CMD: 0x20 - Sector Erase, single pad
         * ADDR: 0x18 - 24 bit address, single pad
         */
        [28] = 0x08180420,

        // Seq8: Dummy
        /*
         * CMD: 0 - Dummy command, used to force SPI flash to exit continuous read mode.
         * unnecessary here because the continuous read mode is not enabled.
         */
        [32] = 0,
    },
};
```

After modifying the qspi_config_block variable, right-click the QCBGenerator project and choose to build.

If the project successfully builds, run QCBGenerator.exe from the Debug folder. The output file named 'qspi_config_block.bin' is generated under the Debug folder.

Both the QuadSPI project and QCB are ready. The next chapter describes how to flash the QuadSPI image to the target device with MCU bootloader.

NOTE

For some SPI Flash devices, the Quad mode configuration bit is non-volatile. It is recommended to set "device_mode_config_en" to 0 after the first configuration completes.

4 Configure QuadSPI with MCU bootloader

QuadSPI can be configured using the MCU bootloader by:

1. Configure QuadSPI at runtime.
2. Configure QuadSPI at start-up.

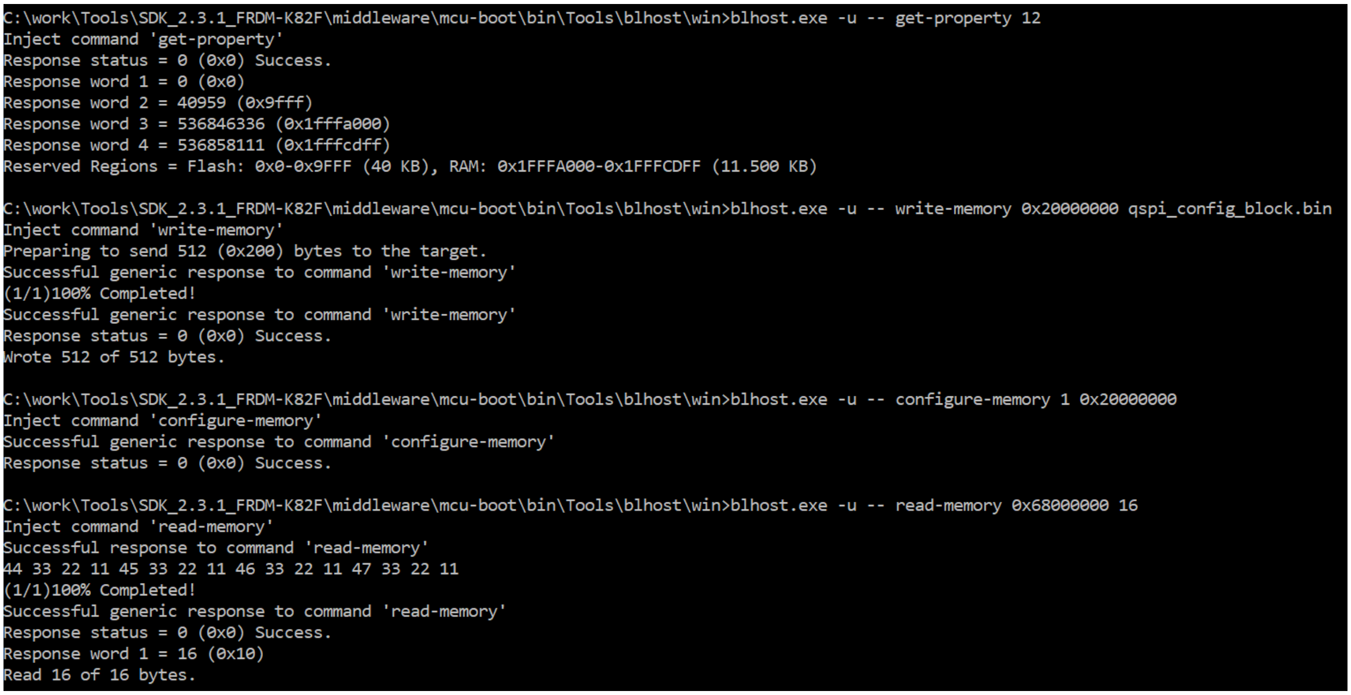
4.1 Configure QuadSPI at runtime

The TWR-K80F150M Tower System module is shipped without any pre-programmed QCB in QuadSPI memory or in internal flash memory. The following figure shows a simple example demonstrating steps to write and configure QCB. See the following figure.

1. Hold the NMI button, press the reset button, then release the reset button and NMI button, in that order.
2. Use the blhost property command to get the Reserved Region property value from MCU bootloader. This provides the RAM region reserved by MCU bootloader.
3. Choose a free RAM region, and using blhost, write QCB to that region.
4. Configure the QuadSPI with the "configure-memory" command.

NOTE

The first command line parameter to configure-memory command is "1" to represent the QuadSPI0, and the second parameter "0x2000_0000" to represent the start address of the QCB.



```
C:\work\Tools\SDK_2.3.1_FRDM-K82F\middleware\mcu-boot\bin\Tools\blhost\win>blhost.exe -u -- get-property 12
Inject command 'get-property'
Response status = 0 (0x0) Success.
Response word 1 = 0 (0x0)
Response word 2 = 40959 (0x9fff)
Response word 3 = 536846336 (0x1fffa000)
Response word 4 = 536858111 (0x1fffcdff)
Reserved Regions = Flash: 0x0-0x9FFF (40 KB), RAM: 0x1FFFA000-0x1FFFCDFF (11.500 KB)

C:\work\Tools\SDK_2.3.1_FRDM-K82F\middleware\mcu-boot\bin\Tools\blhost\win>blhost.exe -u -- write-memory 0x20000000 qspi_config_block.bin
Inject command 'write-memory'
Preparing to send 512 (0x200) bytes to the target.
Successful generic response to command 'write-memory'
(1/1)100% Completed!
Successful generic response to command 'write-memory'
Response status = 0 (0x0) Success.
Wrote 512 of 512 bytes.

C:\work\Tools\SDK_2.3.1_FRDM-K82F\middleware\mcu-boot\bin\Tools\blhost\win>blhost.exe -u -- configure-memory 1 0x20000000
Inject command 'configure-memory'
Successful generic response to command 'configure-memory'
Response status = 0 (0x0) Success.

C:\work\Tools\SDK_2.3.1_FRDM-K82F\middleware\mcu-boot\bin\Tools\blhost\win>blhost.exe -u -- read-memory 0x68000000 16
Inject command 'read-memory'
Successful response to command 'read-memory'
44 33 22 11 45 33 22 11 46 33 22 11 47 33 22 11
(1/1)100% Completed!
Successful generic response to command 'read-memory'
Response status = 0 (0x0) Success.
Response word 1 = 16 (0x10)
Read 16 of 16 bytes.
```

Figure 15. configure-memory at runtime with blhost

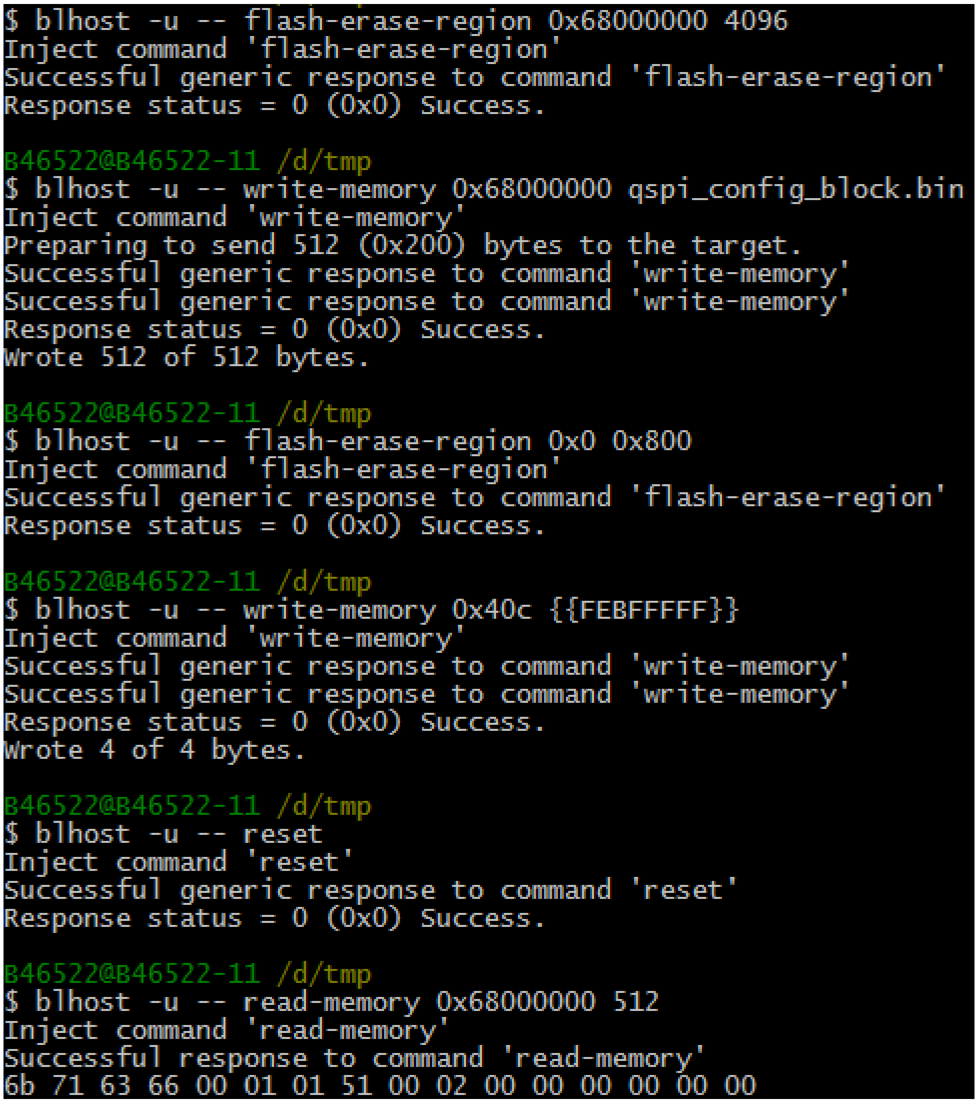
4.2 Configure QuadSPI at start-up

The previous sections show how to configure QCB when there is no QCB pre-programmed on the device. For subsequent boots, it makes sense to save the QCB to non-volatile memory, such as internal flash pointed by the BCA member field, 'qspiConfigBlockPtr', or at the start offset of QuadSPI memory. Next time the device boots from the ROM, the MCU bootloader in ROM detects the presence of the QCB and configure the QuadSPI automatically at start-up. The following steps are the recommended procedure based on the previous section. To program QCB at the start address of QuadSPI memory, see the following figure for the blhost command sequence.

1. Erase the first sector in QuadSPI memory before programming the QCB.
2. Write the QCB to the start of QuadSPI memory.
3. Erase the flash config area.
4. Program the FOPT with the desired value. Make sure FOPT[7:6] (0x40D address in internal flash) is set to 0b10 to default to boot from MCU bootloader in ROM.
5. Reset the target device and use the "read-memory command" to check and ensure if QuadSPI is configured successfully at start-up, as shown in the following figure.

When all of the above operations are completed, the QuadSPI is configured at start-up.

So far, we understand the basic steps of creating QCB and configuring QuadSPI using the MCU bootloader. The next sections describe how to program the QuadSPI image.



```
$ blhost -u -- flash-erase-region 0x68000000 4096
Inject command 'flash-erase-region'
Successful generic response to command 'flash-erase-region'
Response status = 0 (0x0) Success.

B46522@B46522-11 /d/tmp
$ blhost -u -- write-memory 0x68000000 qspi_config_block.bin
Inject command 'write-memory'
Preparing to send 512 (0x200) bytes to the target.
Successful generic response to command 'write-memory'
Successful generic response to command 'write-memory'
Response status = 0 (0x0) Success.
Wrote 512 of 512 bytes.

B46522@B46522-11 /d/tmp
$ blhost -u -- flash-erase-region 0x0 0x800
Inject command 'flash-erase-region'
Successful generic response to command 'flash-erase-region'
Response status = 0 (0x0) Success.

B46522@B46522-11 /d/tmp
$ blhost -u -- write-memory 0x40c {{FEBFFFFF}}
Inject command 'write-memory'
Successful generic response to command 'write-memory'
Successful generic response to command 'write-memory'
Response status = 0 (0x0) Success.
Wrote 4 of 4 bytes.

B46522@B46522-11 /d/tmp
$ blhost -u -- reset
Inject command 'reset'
Successful generic response to command 'reset'
Response status = 0 (0x0) Success.

B46522@B46522-11 /d/tmp
$ blhost -u -- read-memory 0x68000000 512
Inject command 'read-memory'
Successful response to command 'read-memory'
6b 71 63 66 00 01 01 51 00 02 00 00 00 00 00 00
```

Figure 16. Configure QuadSPI at start-up

5 Flash QuadSPI image via SB file

Generally, the QuadSPI image contains separate segments. For example, the vector table and the flash config area are in the internal flash and the executable code is located in the QuadSPI memory. Additionally, the corresponding regions must be erased before programming. It is inconvenient to use separate commands to finish this task. Here, we introduce the SB files and the “receive-sb-file” command to simplify the programming procedure.

5.1 Brief introduction of SB file

The MCU bootloader supports loading of the SB files. The SB file is a NXP-defined boot file format designed to ease the boot process. The file is generated using the NXP elftosb tool. The format supports loading of elf or srec files in a controlled manner, using boot commands such as load, jump, fill, erase, and so on. The boot commands are prescribed in the input command file (boot descriptor .bd) to the elftosb tool. The format supports encryption of the boot image using AES-128 input key.

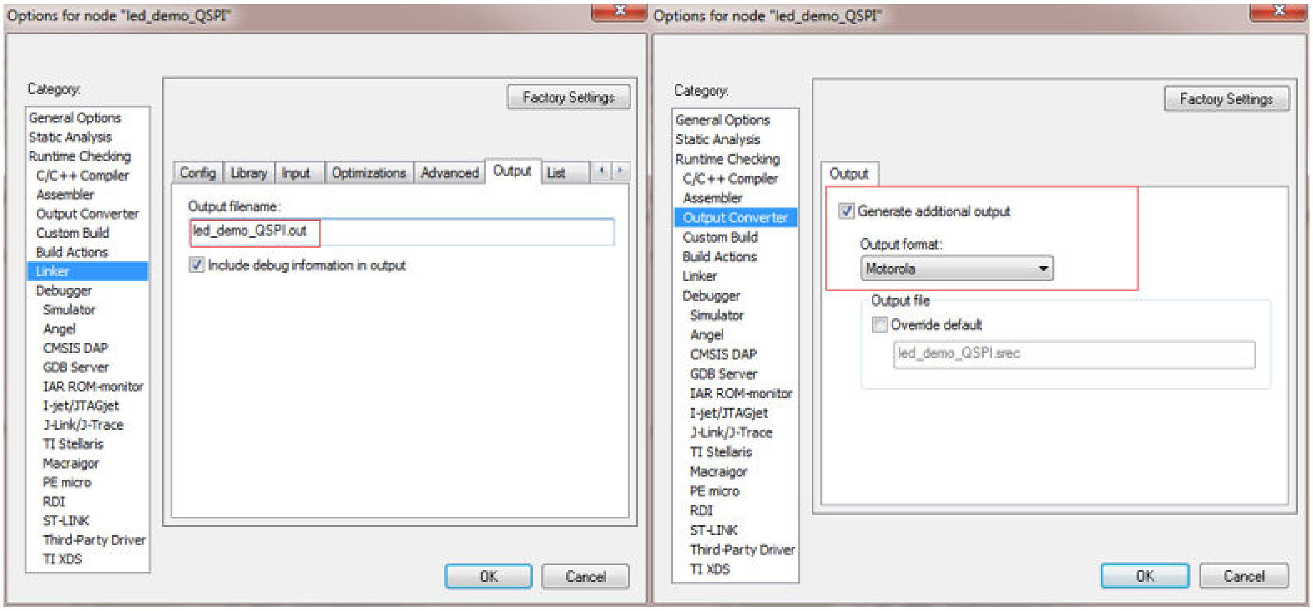
elftosb and SB file formats are described in greater detail in the accompanying documentation in the package.

In this user's guide, the typical use case is provided to demonstrate the usage of elftosb host tool and how to download the SB file with MCU bootloader.

5.2 Generate SB file for QuadSPI image

This section describes the generation of the SB file. The output led-demo srec file is used to generate the SB file (for KEIL, a similar approach can be followed).

- Open the led_demo_qspi project using the IAR EWARM toolchain. Using the project options dialog, select the "linker" and make sure the extension of the output file is ".out".
- Select the "Output Converter" and change the output format to "Motorola" for outputting the *.srec* format image. See the following figure.



Output filename: led_demo_QSPI.out

☑ Include debug information in output

☑ Generate additional output

Output format: Motorola

Output file

☐ Override default

led_demo_QSPI.srec

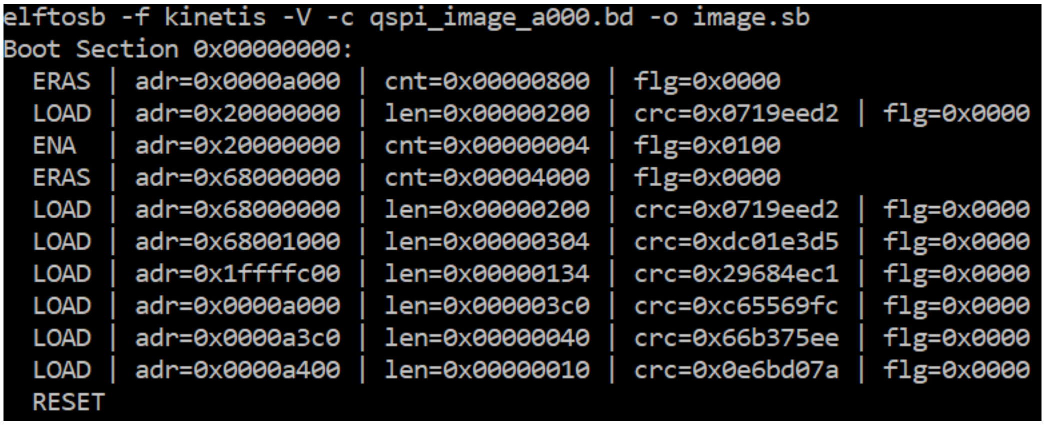
Figure 17. Generate led_demo_qspi.srec with EWARM

- Build either the Debug or the Release configuration of the project. When the build is completed, the *led_demo_QSPI.srec* file should be available in the *output/Debug* or *output/Release* folders.

The next step is to generate the SB file using a command-line host tool, elftosb. The boot descriptor file, *qspi_image_a000.bd*, is passed as the input to the elftosb tool on the command line. The following figure shows the BD file content, the "Sources" section provides the path to the input srec and QCB files and "Section (0)" shows the flow of the boot commands.

After creating the BD file shown in the following codeblock, copy the "qspi_config_block.bin", elftosb.exe, "led_demo_QSPI.srec", and BD files into the same directory. Then open the window with the command prompt and invoke the elftosb as “elftosb -f kinetis –V –c qspi_image_a000.bd –o image.sb”. The elftosb processes the *qspi_image_a000.bd* file and generates the *image.sb* file. The elftosb also outputs the commands list, as shown in Figure 12. Notice that the list corresponds to the BD file Section(0) statements.

```
# The sources block assigns file names to identifiers
sources {
    # SREC File path
    mySrecFile = "led_demo_qspi_a000.srec";
    # QCB file path
    qspiConfigBlock = "qspi_config_block.bin";
}

# The section block specifies the sequence of boot commands to be written to
# the SB file
section (0) {
    #1. Erase the vector table and flash config field.
    erase 0xa000..0xa800;

    # Step 2 and Step 3 are optional if the QuadSPI is configured at startup.
    #2. Load the QCB to RAM
    load qspiConfigBlock > 0x20000000;

    #3. Configure QuadSPI with the QCB above
    enable qspi 0x20000000;

    #4. Erase the QuadSPI memory region before programming.
    erase 0x68000000..0x68004000;

    #5.Load the QCB above to the start address of QuadSPI memory
    load qspiConfigBlock > 0x68000000;

    #6,7. Load all the RO data from srec file
    #including vector table, flash config field and codes.
    load mySrecFile;

    #8. Reset target.
    reset;
}
```

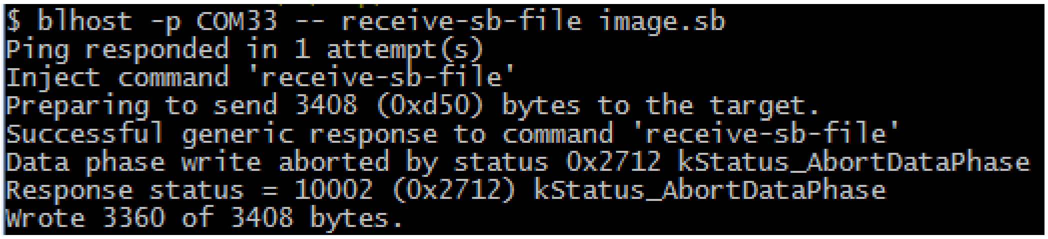


```
elftosb -f kinetis -V -c qspi_image_a000.bd -o image.sb
Boot Section 0x00000000:
  ERAS | adr=0x0000a000 | cnt=0x00000800 | flg=0x0000
  LOAD | adr=0x20000000 | len=0x00000200 | crc=0x0719eed2 | flg=0x0000
  ENA  | adr=0x20000000 | cnt=0x00000004 | flg=0x0100
  ERAS | adr=0x68000000 | cnt=0x00004000 | flg=0x0000
  LOAD | adr=0x68000000 | len=0x00000200 | crc=0x0719eed2 | flg=0x0000
  LOAD | adr=0x68001000 | len=0x00000304 | crc=0xdc01e3d5 | flg=0x0000
  LOAD | adr=0x1ffffc00 | len=0x00000134 | crc=0x29684ec1 | flg=0x0000
  LOAD | adr=0x0000a000 | len=0x000003c0 | crc=0xc65569fc | flg=0x0000
  LOAD | adr=0x0000a3c0 | len=0x00000040 | crc=0x66b375ee | flg=0x0000
  LOAD | adr=0x0000a400 | len=0x00000010 | crc=0x0e6bd07a | flg=0x0000
  RESET
```

Figure 18. elftosb command line usage example and output text

5.3 Flash QuadSPI image via MCU bootloader

When the SB file image is generated, either the blhost or KinetisFlashTool can be used to program the image to the target. The following figure shows an example of programming the SB file with blhost.



```
$ blhost -p COM33 -- receive-sb-file image.sb
Ping responded in 1 attempt(s)
Inject command 'receive-sb-file'
Preparing to send 3408 (0xd50) bytes to the target.
Successful generic response to command 'receive-sb-file'
Data phase write aborted by status 0x2712 kStatus_AbortDataPhase
Response status = 10002 (0x2712) kStatus_AbortDataPhase
Wrote 3360 of 3408 bytes.
```

Figure 19. Flash SB file with blhost

6 Advanced Usage: Encrypted QuadSPI image

The SB file generated in Section 5.2 is in plaintext form and not encrypted. This section focuses on several aspects of encrypted boot with MCU bootloader.

To use the encrypted boot feature, user must have basic knowledge of the SB key, KeyBlob Block, and KeyBlob Encryption Key (KEK), SB Key, AES-128 CTR, AES-128 CBC-MAC, and so on. See the MCU bootloader chapter in the silicon's reference manual for a detailed description. The following is a brief introduction to these terms:

- The KeyBlob Block is a data structure that contains up to four groups of KeyBlob entries. Each entry consists of the start address, length, decryption key, and counter of an encrypted QuadSPI memory region.
- The KeyBlob Block itself is encrypted by another AES key, called Key encryption key (KEK). KEK needs to be pre-programmed in flash's IFR region. In MK82F256, the Flash IFR index for KEK is from index 0x20 to 0x23. With the Key Blob and KEK, sections belonging to encrypted QuadSPI memory region (QuadSPI image data) can be encrypted using elftosb tools. The generated SB file has encrypted image data for the encrypted QuadSPI memory region.
- For devices with flash security enabled, only encrypted SB file images are allowed to be provisioned. MCU bootloader decrypts the encrypted SB image as it receives from the host using a separate SB key. The SB key is an AES-128 key pre-programmed into flash's IFR region at word offsets 0x30 to 0x33. The elftosb tool allows generation of encrypted SB file image using the SB key.

In general, the QuadSPI image is encrypted using the parameters in the KeyBlob with AES-128 CTR mode, the KeyBlob Block itself encrypted with KEK, and the SB file is encrypted via SB key with AES-128 CBC-MAC. The following figure shows an SB file containing plaintext QuadSPI image data. The vector table and other regions are in plaintext.

Based on the application type, the user can choose to have plaintext or encrypted QuadSPI image or encrypted SB file image solution.

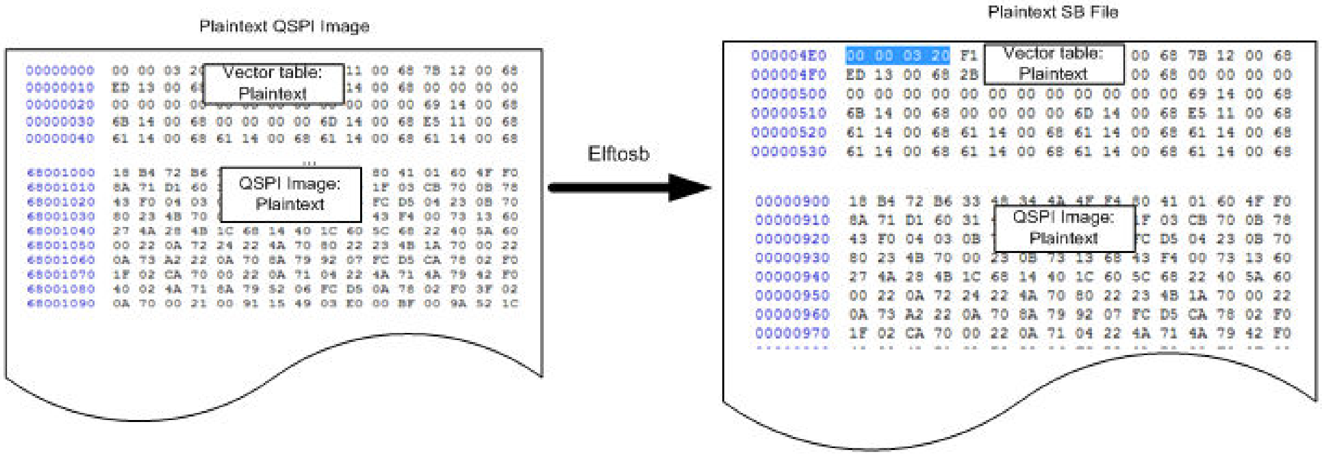


Figure 20. Plaintext SB file with Plaintext QuadSPI image

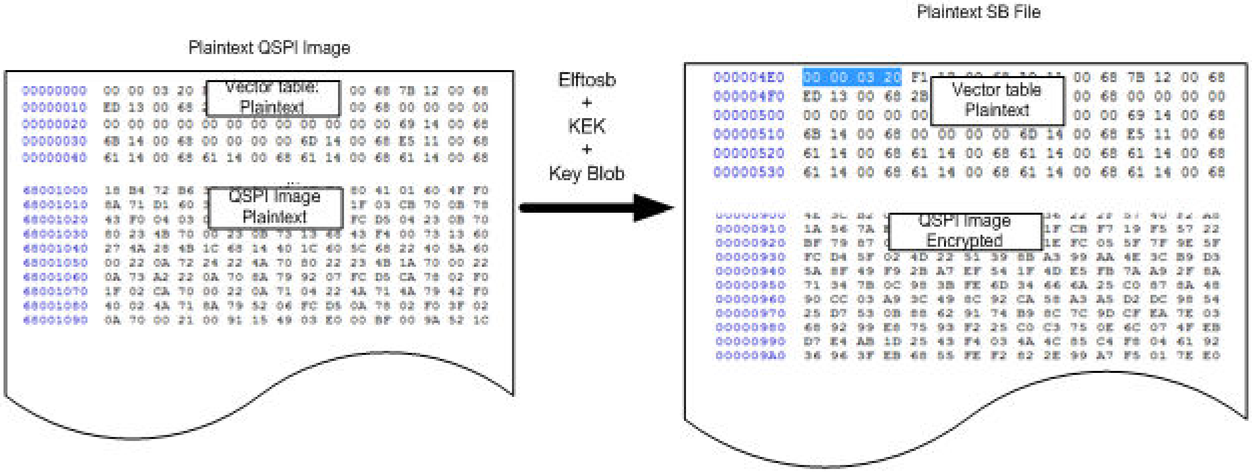


Figure 21. Plaintext SB file with Encrypted QuadSPI image

The following figure provides an encrypted SB file containing an encrypted QuadSPI image. The entire content of the SB file is obfuscated.

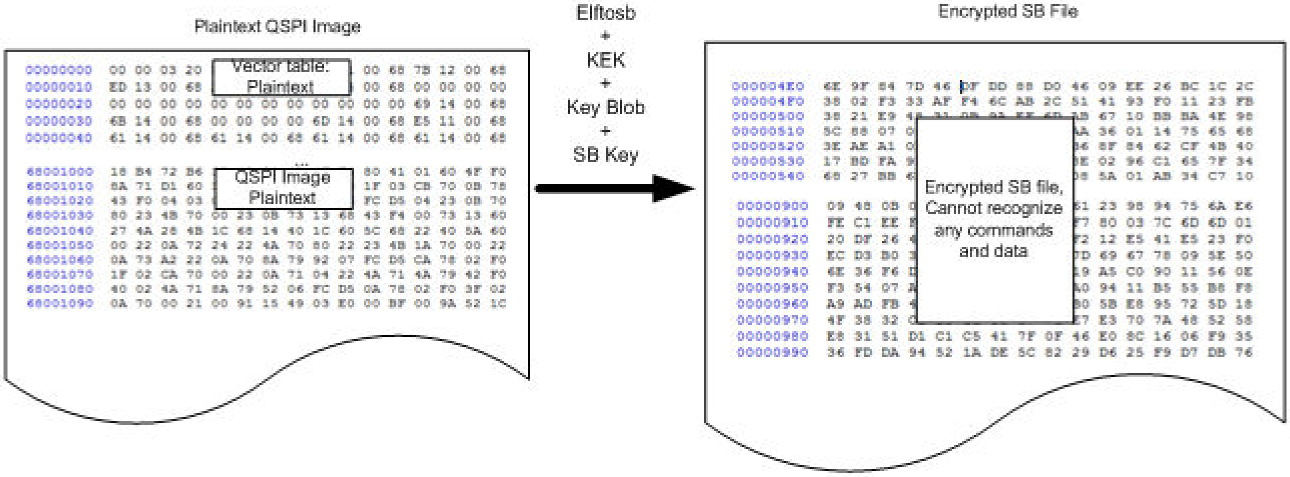


Figure 22. Encrypted SB file with Encrypted QuadSPI image

The rest of the sections in this chapter provide step-by-step instructions on programming keys, generating encrypted QuadSPI image data in the SB file, and encrypting the entire SB file image with the SB key.

6.1 Generate an SB file with KEK and SB KEY

Here is an example of generating an SB file with just the KEK and SB KEY. The generated SB file can be provisioned using MCU bootloader to program the keys into IFR region of the device.

The SB KEY is a 16 byte array. For example:

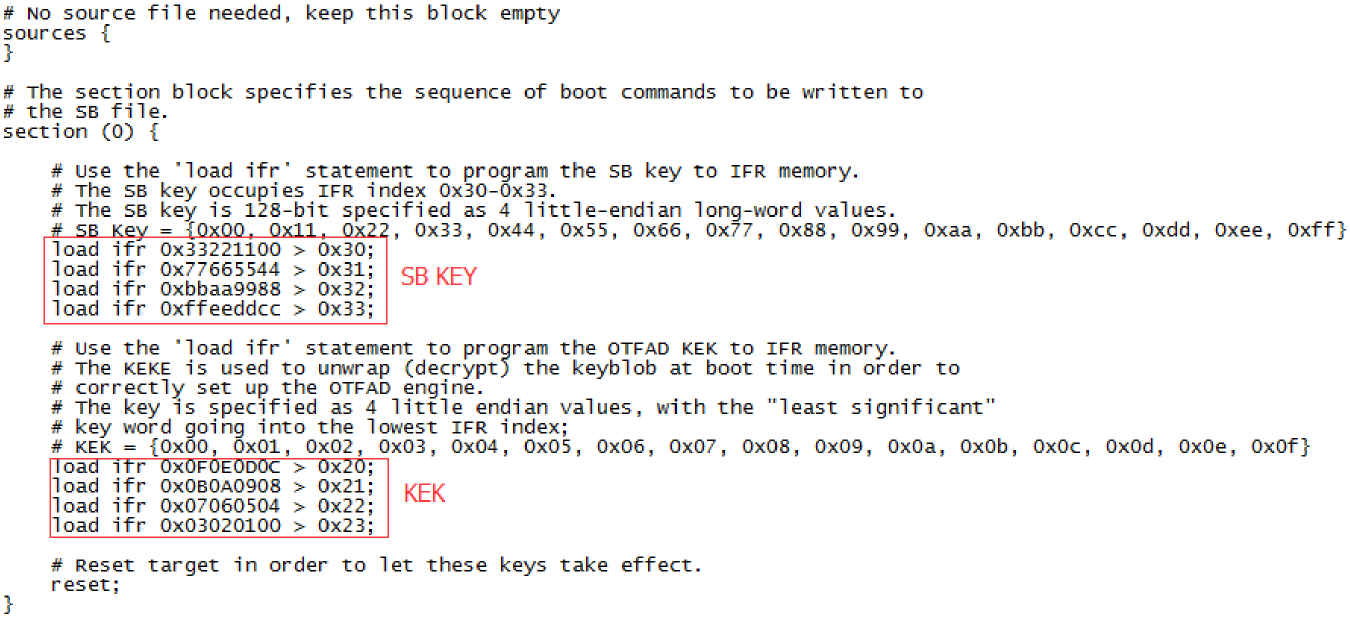
```
uint8_t sbKey[16] = {0x00, 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77, 0x88, 0x99, 0xaa, 0xbb, 0xcc,
0xdd, 0xee, 0xff}.
```

The KEK is also a 16 byte array. For example:

```
uint8_t kek[16] = {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c,
0x0d, 0x0e, 0x0f}.
```

Pay attention to the correct order of the data to be programmed to Flash IFR, because each IFR field needs to be programmed with 32-bit little-endian data. See the example BD file content provided in the following figure to understand how to specify the SB key and KEK to generate SB file image to program the keys.

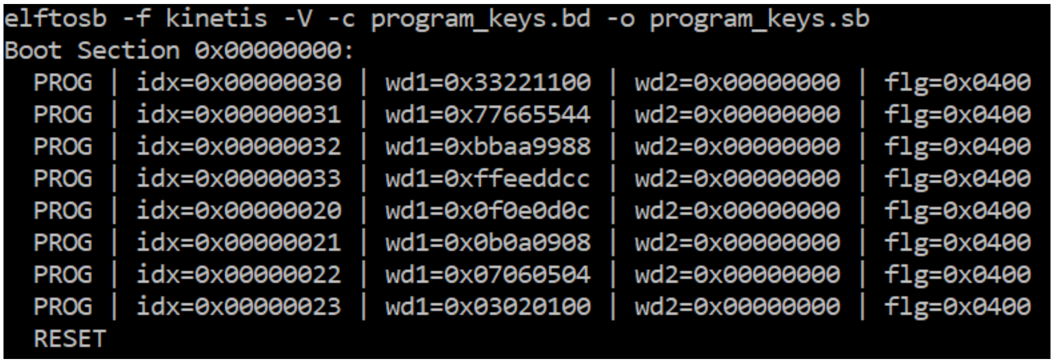
To generate SB file, a specified BD file needs to be generated first, assuming the BD file is called "program_keys.bd".



```
# No source file needed, keep this block empty
sources {
}

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

    # Use the 'load ifr' statement to program the SB key to IFR memory.
    # The SB key occupies IFR index 0x30-0x33.
    # The SB key is 128-bit specified as 4 little-endian long-word values.
    # SB Key = {0x00, 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77, 0x88, 0x99, 0xaa, 0xbb, 0xcc, 0xdd, 0xee, 0xff}
    load ifr 0x33221100 > 0x30;
    load ifr 0x77665544 > 0x31;
    load ifr 0xbbaa9988 > 0x32;
    load ifr 0xffeeddcc > 0x33;

    # Use the 'load ifr' statement to program the OTFAD KEK to IFR memory.
    # The KEKE is used to unwrap (decrypt) the keyblob at boot time in order to
    # correctly set up the OTFAD engine.
    # The key is specified as 4 little endian values, with the "least significant"
    # key word going into the lowest IFR index;
    # KEK = {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f}
    load ifr 0x0F0E0D0C > 0x20;
    load ifr 0x0B0A0908 > 0x21;
    load ifr 0x07060504 > 0x22;
    load ifr 0x03020100 > 0x23;

    # Reset target in order to let these keys take effect.
    reset;
}
```

Figure 23. Specified BD file for SB key and KEK

Using elftosb, the desired SB file is generated. The elftosb command line and output is shown in the following figure.



```
elftosb -f kinetis -V -c program_keys.bd -o program_keys.sb
Boot Section 0x00000000:
  PROG | idx=0x00000030 | wd1=0x33221100 | wd2=0x00000000 | flg=0x0400
  PROG | idx=0x00000031 | wd1=0x77665544 | wd2=0x00000000 | flg=0x0400
  PROG | idx=0x00000032 | wd1=0xbbaa9988 | wd2=0x00000000 | flg=0x0400
  PROG | idx=0x00000033 | wd1=0xffeeddcc | wd2=0x00000000 | flg=0x0400
  PROG | idx=0x00000020 | wd1=0x0f0e0d0c | wd2=0x00000000 | flg=0x0400
  PROG | idx=0x00000021 | wd1=0x0b0a0908 | wd2=0x00000000 | flg=0x0400
  PROG | idx=0x00000022 | wd1=0x07060504 | wd2=0x00000000 | flg=0x0400
  PROG | idx=0x00000023 | wd1=0x03020100 | wd2=0x00000000 | flg=0x0400
  RESET
```

Figure 24. Generate program_keys.sb

Either blhost or KinetisFlashTool can be used to flash the SB file to the target device.

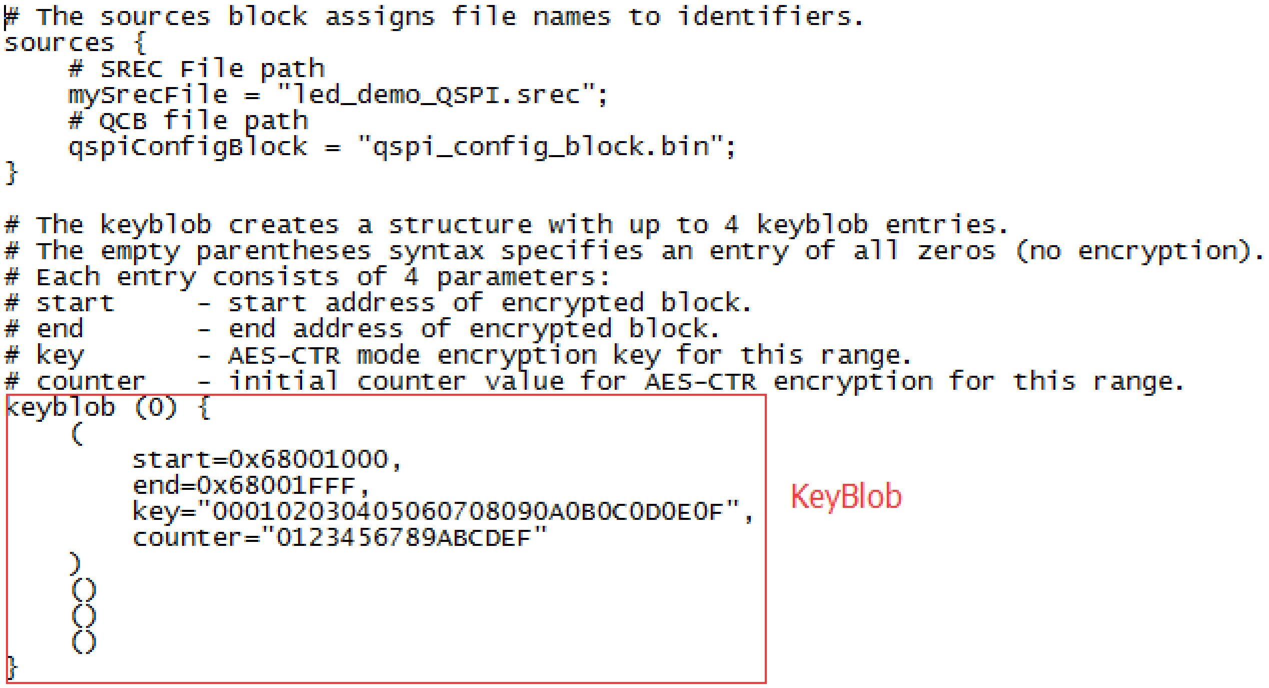
6.2 Generate an SB file with encrypted QuadSPI image

After the previous operation, another SB file (which contains the encrypted QuadSPI image) is still needed. Similar to how the SB file was generated in the previous section, the BD file is needed to describe all the operations in this SB file. Besides the operations listed in Chapter 4, it also contains the Key Blob Block, encrypted QuadSPI image, and Key Blob encryption wrapper.

6.2.1 The KeyBlob Block

This section shows the syntax of the keyBlob entry in the BD file with an example in the following figure. The example shows one QuadSPI memory region identified by the counter value.

```
keyblob (0) {
    #key blob 0
    (
        start = address1,
        end = address2,
        key = keystring,
        counter = counterstring
    )
    # key blob 1, keep this blank if this key blob isn’t needed.
    ()
    # key blob 2, keep this blank if this key blob isn’t needed.
    ()
    # key blob 3, keep this blank if this key blob isn’t needed.
    ()
}
```



```
# The sources block assigns file names to identifiers.
sources {
    # SREC File path
    mySrecFile = "led_demo_QSPI.srec";
    # QCB file path
    qspiConfigBlock = "qspi_config_block.bin";
}

# The keyblob creates a structure with up to 4 keyblob entries.
# The empty parentheses syntax specifies an entry of all zeros (no encryption).
# Each entry consists of 4 parameters:
# start     - start address of encrypted block.
# end       - end address of encrypted block.
# key       - AES-CTR mode encryption key for this range.
# counter   - initial counter value for AES-CTR encryption for this range.
keyblob (0) {
    (
        start=0x68001000,
        end=0x68001FFF,
        key="000102030405060708090A0B0C0D0E0F",
        counter="0123456789ABCDEF"
    )
    ()
    ()
    ()
}
```

Figure 25. KeyBlob definition

6.2.2 Encrypt QuadSPI image

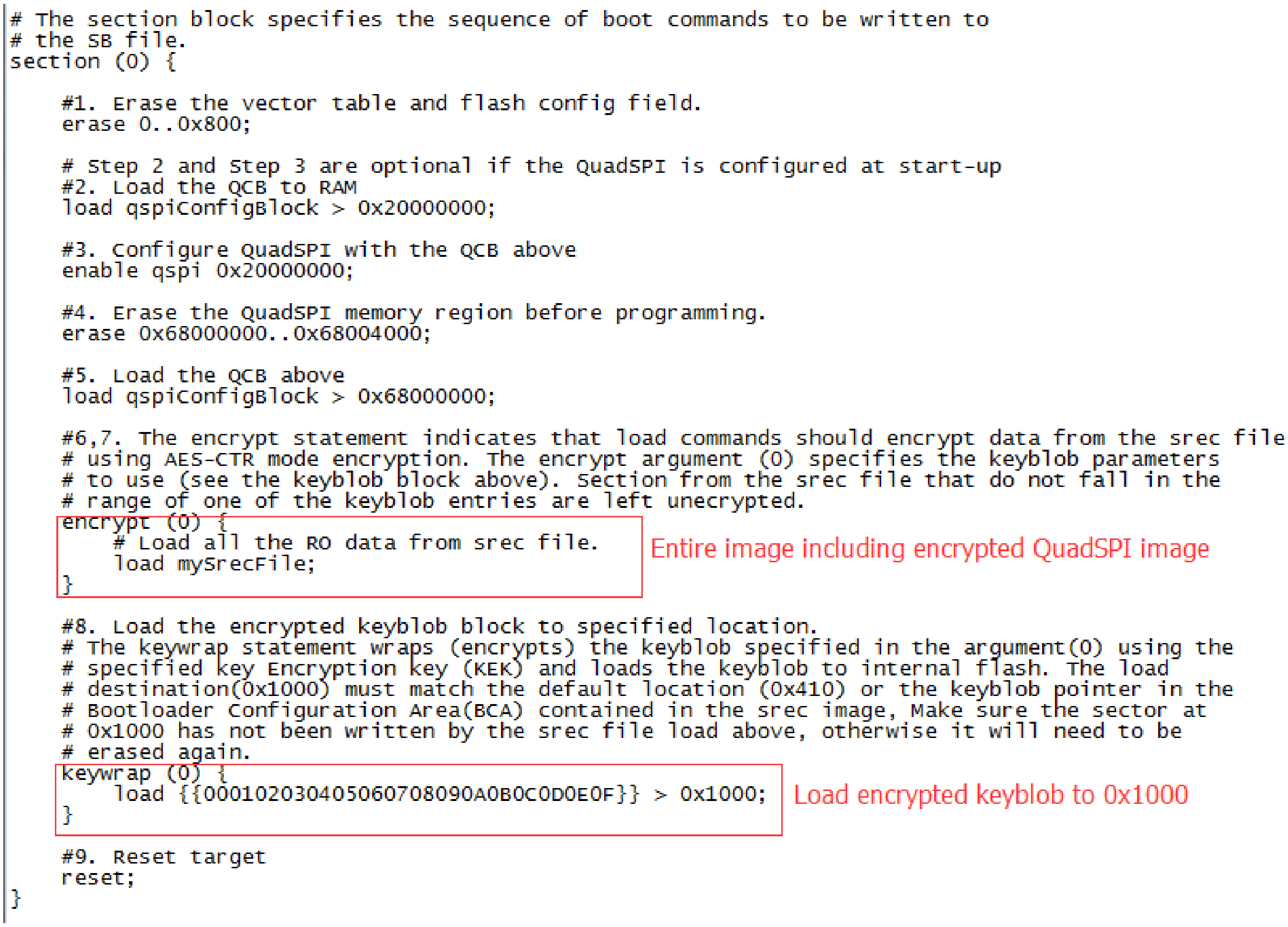
This section shows BD file changes required to encrypt the QuadSPI image using the KeyBlob. The encrypt (0) section in the BD file, shown in the following figure, causes elftosb to encrypt the QuadSPI image data falling in the QuadSPI memory regions pointed by the keyBlob counter.

The keyBlob itself is encrypted with the KEK. The keywrap (0) section in the BD file causes elftosb to wrap the keyBlob using the KEK specified in the load command of keywrap section.

The syntax for the keywrap section of BD file is as follows:

```
keywrap (0) {
    load {{KEK hex string}} > destination of encrypted key blob block;
}
```

The memory address 0x1000 in the example shown in the following figure is where the wrapped keyBlob is loaded during provisioning of SB file to the target device using MCU bootloader.



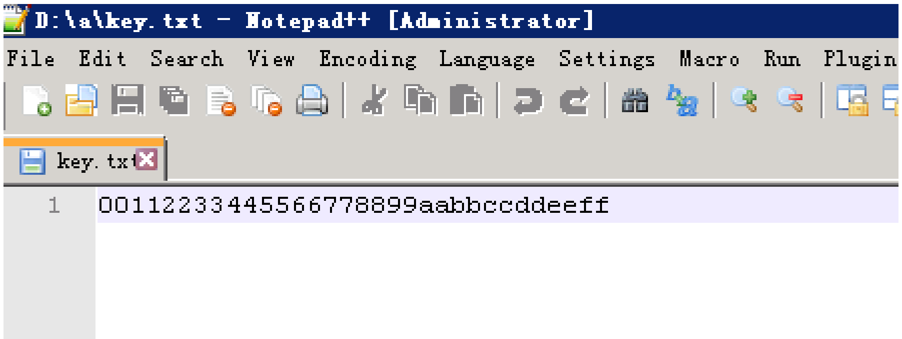
```
# The section block specifies the sequence of boot commands to be written to
# the SB file.
section (0) {

    #1. Erase the vector table and flash config field.
    erase 0..0x800;

    # Step 2 and Step 3 are optional if the QuadSPI is configured at start-up
    #2. Load the QCB to RAM
    load qspiConfigBlock > 0x20000000;

    #3. Configure QuadSPI with the QCB above
    enable qspi 0x20000000;

    #4. Erase the QuadSPI memory region before programming.
    erase 0x68000000..0x68004000;

    #5. Load the QCB above
    load qspiConfigBlock > 0x68000000;

    #6,7. The encrypt statement indicates that load commands should encrypt data from the srec file
    # using AES-CTR mode encryption. The encrypt argument (0) specifies the keyblob parameters
    # to use (see the keyblob block above). Section from the srec file that do not fall in the
    # range of one of the keyblob entries are left unecrypted.
    encrypt (0) {
        # Load all the RO data from srec file.
        load mySrecFile;
    }

    #8. Load the encrypted keyblob block to specified location.
    # The keywrap statement wraps (encrypts) the keyblob specified in the argument(0) using the
    # specified key Encryption key (KEK) and loads the keyblob to internal flash. The load
    # destination(0x1000) must match the default location (0x410) or the keyblob pointer in the
    # Bootloader Configuration Area(BCA) contained in the srec image, Make sure the sector at
    # 0x1000 has not been written by the srec file load above, otherwise it will need to be
    # erased again.
    keywrap (0) {
        load {{000102030405060708090A0B0C0D0E0F}} > 0x1000;
    }

    #9. Reset target
    reset;
}
```

Entire image including encrypted QuadSPI image
Load encrypted keyblob to 0x1000

Figure 26. Encrypt QuadSPI image and KeyBlob

6.2.3 Encrypting SB file with the SB key

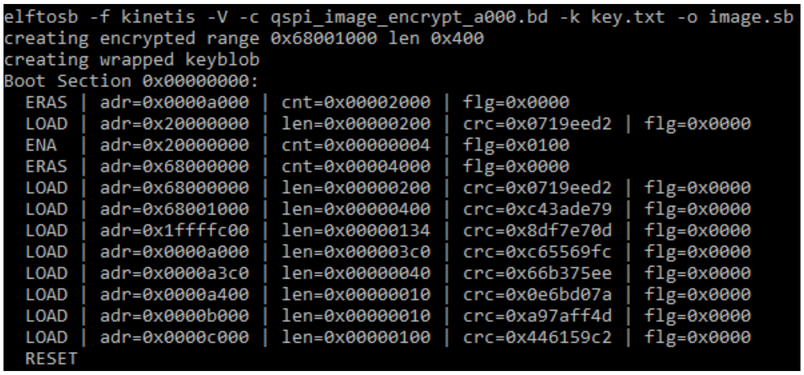
To encrypt the SB file with the elftosb, a file containing the SB key must be created, as shown in the following figure.


D:\a\key.txt - Notepad++ [Administrator]
File Edit Search View Encoding Language Settings Macro Run Plugin
key.txt

```
1   00112233445566778899aabbccddeeff
```

Figure 27. Create key.txt containing SB key

The following figure shows the generation of the encrypted SB file using the BD file drafted in the previous sections. The SB key is passed on the command line to the elftosb using the -k option.



```
elftosb -f kinetis -V -c qspi_image_encrypt_a000.bd -k key.txt -o image.sb
creating encrypted range 0x68001000 len 0x400
creating wrapped keyblob
Boot Section 0x00000000:
  ERAS | adr=0x0000a000 | cnt=0x00002000 | flg=0x0000
  LOAD | adr=0x20000000 | len=0x00000200 | crc=0x0719eed2 | flg=0x0000
  ENA  | adr=0x20000000 | cnt=0x00000004 | flg=0x0100
  ERAS | adr=0x68000000 | cnt=0x00004000 | flg=0x0000
  LOAD | adr=0x68000000 | len=0x00000200 | crc=0x0719eed2 | flg=0x0000
  LOAD | adr=0x68001000 | len=0x00000400 | crc=0xc43ade79 | flg=0x0000
  LOAD | adr=0x1ffffc00 | len=0x00000134 | crc=0x8df7e70d | flg=0x0000
  LOAD | adr=0x0000a000 | len=0x000003c0 | crc=0xc65569fc | flg=0x0000
  LOAD | adr=0x0000a3c0 | len=0x00000040 | crc=0x66b375ee | flg=0x0000
  LOAD | adr=0x0000a400 | len=0x00000010 | crc=0x0e6bd07a | flg=0x0000
  LOAD | adr=0x0000b000 | len=0x00000010 | crc=0xa97aff4d | flg=0x0000
  LOAD | adr=0x0000c000 | len=0x00000100 | crc=0x446159c2 | flg=0x0000
  RESET
```

Figure 28. Generate encrypted SB file with encrypted QuadSPI image

The output *image.sb* file can be programmed to the target device using the blhost or KinetisFlashTool, as shown in the earlier examples. Based on the example BD file, the *image.sb* file has the wrapped keyBlob, the keyBlob encrypted QuadSPI image data, and the entire content of the SB file encrypted with the SB key.

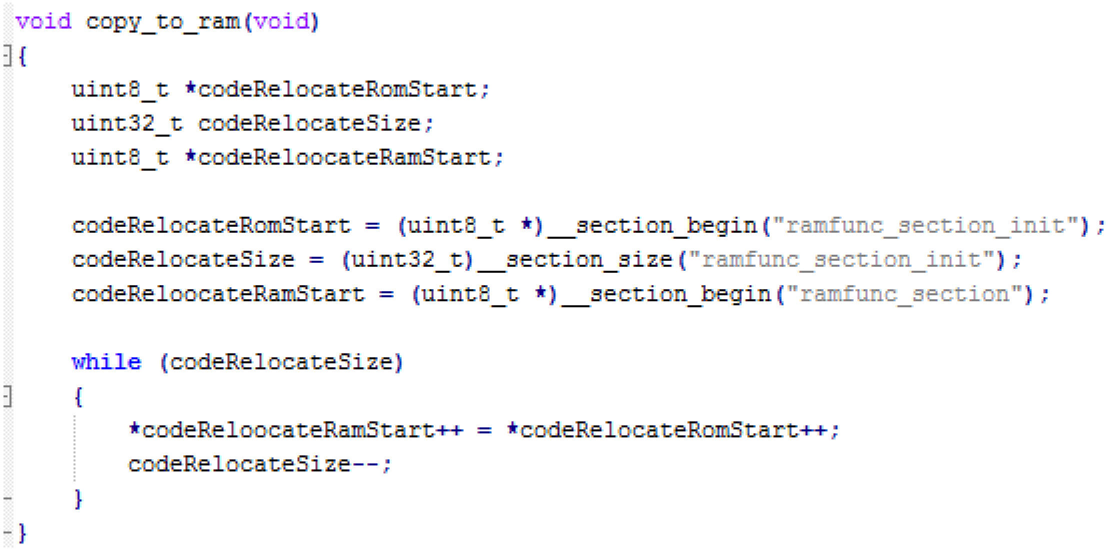
7 Change QuadSPI clock in QuadSPI image

When using MCU bootloader, if the target is booted from the QuadSPI image, both the QuadSPI serial clock and core clock are from MCGFLL. MCG is under FEE mode, using the IRC48M as the clock source. In some cases, this may not meet the system's accuracy and performance requirement. The MCG mode needs to be switched from FEE to PEE, with an external OSC as clock source. Be aware that this operation has great impact on the QuadSPI serial clock, so avoid running the clock switch function on the QuadSPI image directly. A relatively safer way to avoid this is to either copy this function to SRAM, or place this function in internal flash.

This chapter provides an example for how to create a clock switch function running on RAM.

7.1 Create a RAM function via IAR EWARM

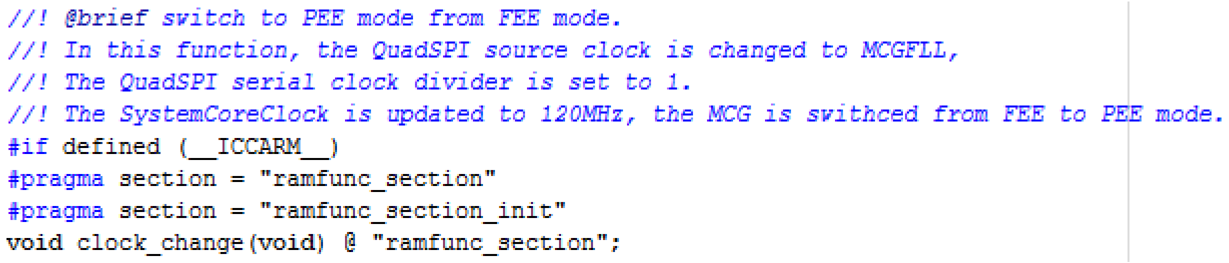
In order to create a RAM function with IAR EWARM, two sections need to be defined. The first is "ramfunc_section_init", which is used to store the data of a RAM function, and a "ramfunc_section", which is the actual execution section of the RAM function. The following code snippets provide an example of how to define and place code to these sections.



```
void copy_to_ram(void)
{
    uint8_t *codeRelocateRomStart;
    uint32_t codeRelocateSize;
    uint8_t *codeReloocateRamStart;

    codeRelocateRomStart = (uint8_t *)__section_begin("ramfunc_section_init");
    codeRelocateSize = (uint32_t)__section_size("ramfunc_section_init");
    codeReloocateRamStart = (uint8_t *)__section_begin("ramfunc_section");

    while (codeRelocateSize)
    {
        *codeReloocateRamStart++ = *codeRelocateRomStart++;
        codeRelocateSize--;
    }
}
```

Figure 29. Declare ram function section in EWARM project

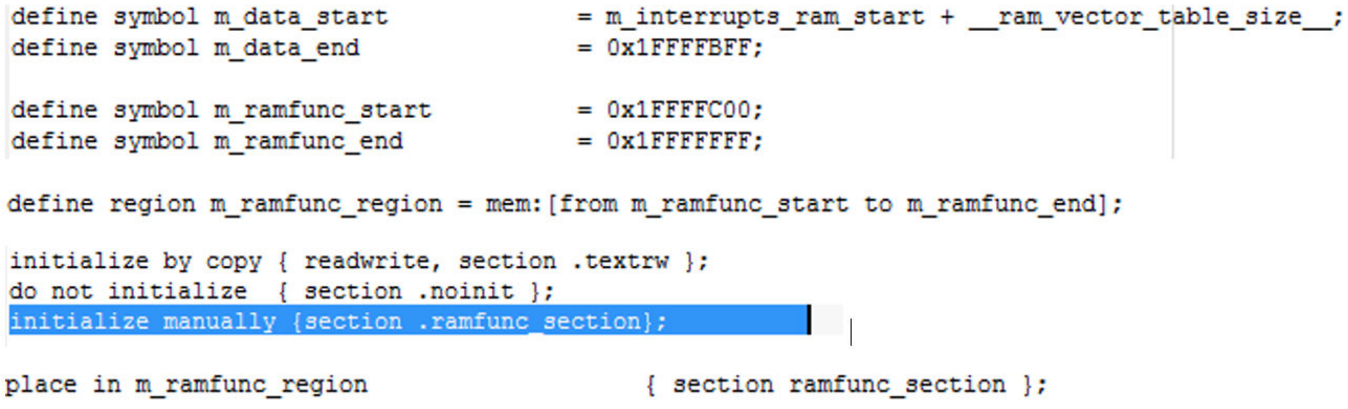
After the previous operation, we still need to define another function. For example, *copy_to_ram()* to copy the RAM func codes from QuadSPI memory to RAM. The following figure provides an example.



```
//! @brief switch to PEE mode from FEE mode.
//! In this function, the QuadSPI source clock is changed to MCGFLL,
//! The QuadSPI serial clock divider is set to 1.
//! The SystemCoreClock is updated to 120MHz, the MCG is swithced from FEE to PEE mode.
#if defined (__ICCARM__)
#pragma section = "ramfunc_section"
#pragma section = "ramfunc_section_init"
void clock_change(void) @ "ramfunc_section";
```

Figure 30. Implement copy_to_ram() function in EWARM project

Finally, change the linker file in order to let the linker know a RAM function section has been defined. The location to place this section, and the section, need to be copied to RAM manually.



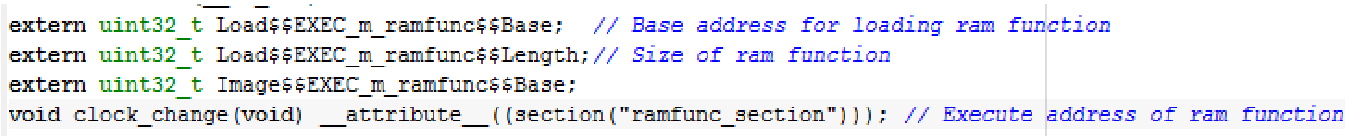
```
define symbol m_data_start              = m_interrupts_ram_start + __ram_vector_table_size__;
define symbol m_data_end                = 0x1FFFFBFF;

define symbol m_ramfunc_start           = 0x1FFFFC00;
define symbol m_ramfunc_end             = 0x1FFFFFFF;

define region m_ramfunc_region = mem:[from m_ramfunc_start to m_ramfunc_end];

initialize by copy { readwrite, section .textrw };
do not initialize  { section .noinit };
initialize manually {section .ramfunc_section};

place in m_ramfunc_region                    { section ramfunc_section };
```

Figure 31. Linker file changes for ram function in EWARM project

A complete project for this example can be found in *<sdk_package>/boards/<board>/bootloader_examples/demo_apps*.

7.2 Create a RAM function via Keil MDK

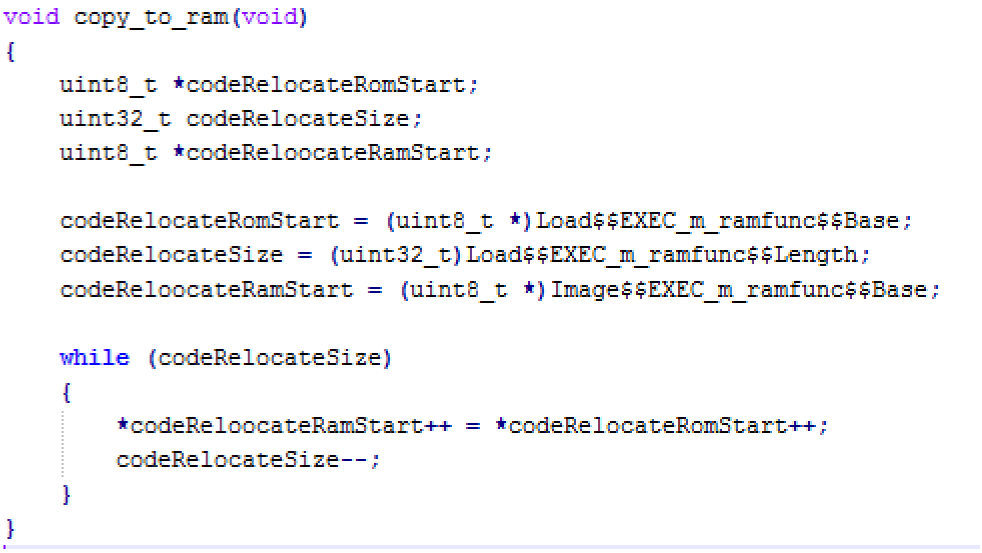
Keil also supports the creation of a RAM function, using a similar method as described for IAR EWARM. To create a RAM function via KEIL, declare a section. In this example, "ramfunc_section" has been declared. See the following figure.



```
extern uint32_t Load$$EXEC_m_ramfunc$$Base;  // Base address for loading ram function
extern uint32_t Load$$EXEC_m_ramfunc$$Length;// Size of ram function
extern uint32_t Image$$EXEC_m_ramfunc$$Base;
void clock_change(void) __attribute__((section("ramfunc_section"))); // Execute address of ram function
```

Figure 32. Declare RAM function in MDK project

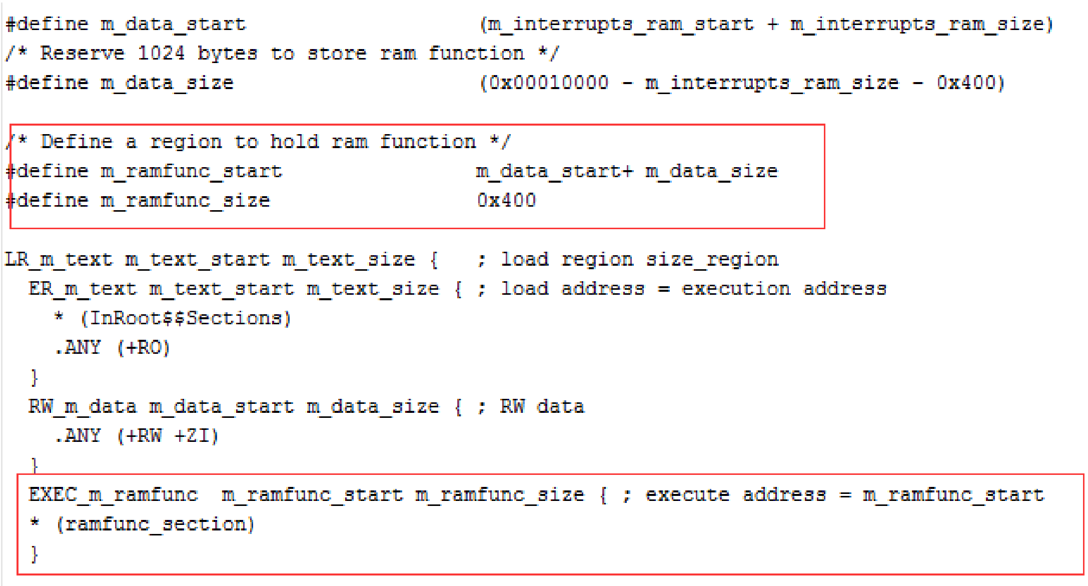
A copy_to_ram function is still necessary to copy the data from ROM to an actual execution address. See the following figure.



```
void copy_to_ram(void)
{
    uint8_t *codeRelocateRomStart;
    uint32_t codeRelocateSize;
    uint8_t *codeReloocateRamStart;

    codeRelocateRomStart = (uint8_t *)Load$$EXEC_m_ramfunc$$Base;
    codeRelocateSize = (uint32_t)Load$$EXEC_m_ramfunc$$Length;
    codeReloocateRamStart = (uint8_t *)Image$$EXEC_m_ramfunc$$Base;

    while (codeRelocateSize)
    {
        *codeReloocateRamStart++ = *codeRelocateRomStart++;
        codeRelocateSize--;
    }
}
```

Figure 33. Implement copy_to_ram() function in MDK project

To let the linker know a RAM function has been defined, add some information to the linker file. For example:



```
#define m_data_start                    (m_interrupts_ram_start + m_interrupts_ram_size)
/* Reserve 1024 bytes to store ram function */
#define m_data_size                     (0x00010000 - m_interrupts_ram_size - 0x400)

/* Define a region to hold ram function */
#define m_ramfunc_start                 m_data_start+ m_data_size
#define m_ramfunc_size                  0x400

LR_m_text m_text_start m_text_size {   ; load region size_region
  ER_m_text m_text_start m_text_size { ; load address = execution address
    * (InRoot$$Sections)
    .ANY (+RO)
  }
  RW_m_data m_data_start m_data_size { ; RW data
    .ANY (+RW +ZI)
  }
  EXEC_m_ramfunc  m_ramfunc_start m_ramfunc_size { ; execute address = m_ramfunc_start
  * (ramfunc_section)
  }
```

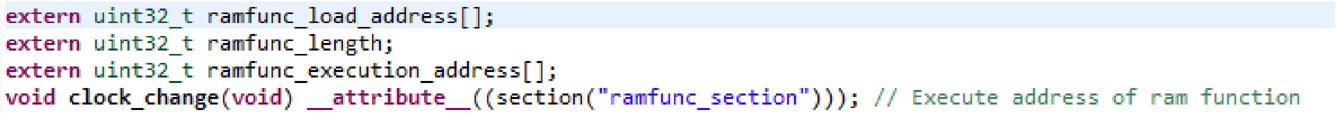
Figure 34. Linker file changes for RAM function in MDK project

A complete project for this example can be found in the *<sdk_package>/boards/<board>/bootloader_examples/demo_apps*.

7.3 Create a RAM function with MCUXpresso IDE

This section shows the steps required for the MCUXpresso IDE to create the RAM function.

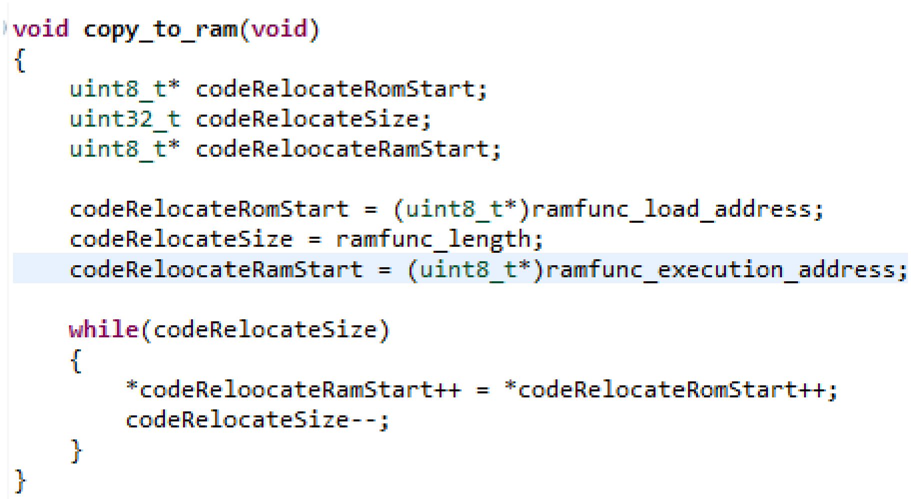
First, declare a section to place the RAM function codes to. In this example, a section called “ramfunc_section” is declared as follows:



```
extern uint32_t ramfunc_load_address[];
extern uint32_t ramfunc_length;
extern uint32_t ramfunc_execution_address[];
void clock_change(void) __attribute__((section("ramfunc_section"))); // Execute address of ram function
```

Figure 35. Declare a RAM function in MCUXpresso IDE

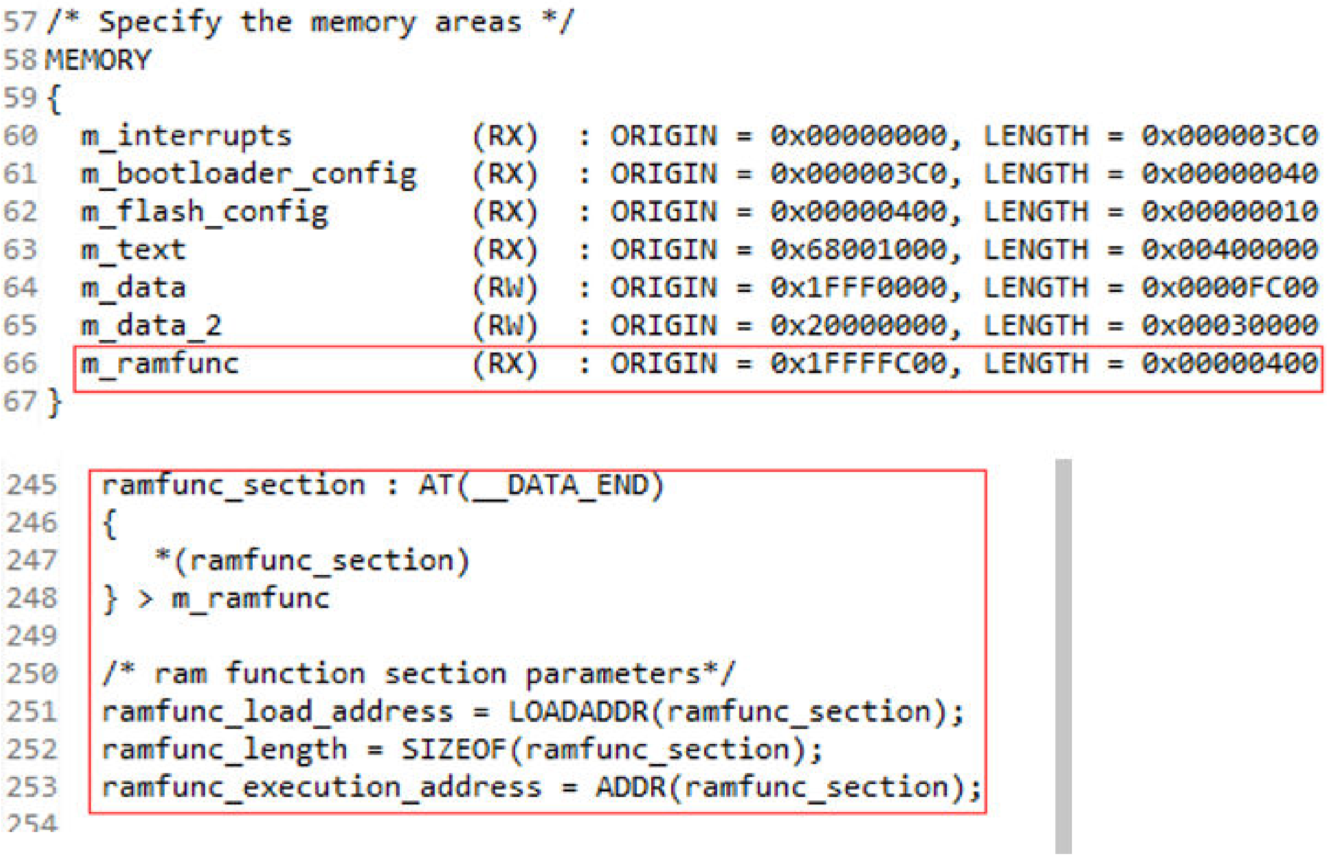
Then, implement the copy_to_ram() function in the MCUXpresso IDE project. An example is shown in the following figure:



```
void copy_to_ram(void)
{
    uint8_t* codeRelocateRomStart;
    uint32_t codeRelocateSize;
    uint8_t* codeReloocateRamStart;

    codeRelocateRomStart = (uint8_t*)ramfunc_load_address;
    codeRelocateSize = ramfunc_length;
    codeReloocateRamStart = (uint8_t*)ramfunc_execution_address;

    while(codeRelocateSize)
    {
        *codeReloocateRamStart++ = *codeRelocateRomStart++;
        codeRelocateSize--;
    }
}
```

Figure 36. Implement copy_to_ram() function in MCUXpresso IDE project

Finally, the linker file must be updated to let the MCUXpresso IDE realize that the RAM function is defined, and must be placed somewhere. The following figure demonstrates the changes for the RAM function in the linker file. A complete project for this example can be found in the *<sdk_package>/boards/<board>/bootloader_examples/demo_apps* folder.



```
/* Specify the memory areas */
MEMORY
{
  m_interrupts          (RX)  : ORIGIN = 0x00000000, LENGTH = 0x000003C0
  m_bootloader_config   (RX)  : ORIGIN = 0x000003C0, LENGTH = 0x00000040
  m_flash_config        (RX)  : ORIGIN = 0x00000400, LENGTH = 0x00000010
  m_text                (RX)  : ORIGIN = 0x68001000, LENGTH = 0x00400000
  m_data                (RW)  : ORIGIN = 0x1FFF0000, LENGTH = 0x0000FC00
  m_data_2              (RW)  : ORIGIN = 0x20000000, LENGTH = 0x00030000
  m_ramfunc             (RX)  : ORIGIN = 0x1FFFFC00, LENGTH = 0x00000400
}
```

```
ramfunc_section : AT(__DATA_END)
{
   *(ramfunc_section)
} > m_ramfunc

/* ram function section parameters*/
ramfunc_load_address = LOADADDR(ramfunc_section);
ramfunc_length = SIZEOF(ramfunc_section);
ramfunc_execution_address = ADDR(ramfunc_section);
```

Figure 37. Linker file changes for RAM function in MCUXpresso IDE

7.4 Ensure no timing issue after clock change

After performing the changes listed in the previous section, the clock switch function can be implemented. Note that the clock switch function must not violate the timing requirements for the QuadSPI module and the external SPI flash device. For example, if the external SPI flash is working in the SDR mode and you plan to switch the QuadSPI clock source to PLL 120 MHz, it is required to set the QuadSPI_MCR [SCLKCFG] to at least 1 (which means that the QuadSPI serial clock frequency is 120 MHz/2 = 60 MHz), because the maximum supported clock for the SDR mode is 100 MHz. See the clock_change() function in the example for more details.

8 Application running on QuadSPI alias area

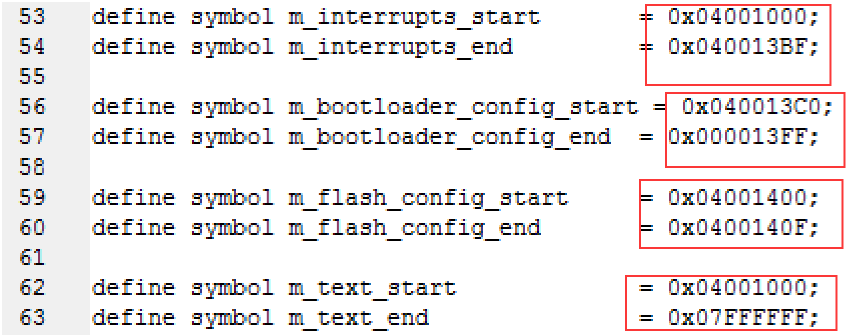
For reasons such as performance improvements, the application should be addressed to run from QuadSPI alias area (0x0400_0000 to 0x07FF_FFFF on MK82F256) instead of physical addresses. MCU ROM bootloader does not support downloading the application running on the alias area directly. However, a workaround solution is described in this section to allow application to run from the alias region. Here we use the led_demo demonstrated before as an example and show how to download and run such application from the alias memory region.

NOTE

This section is only applicable to MCU ROM Bootloader in K80, K81, and K82.

8.1 Create an application to run on QuadSPI Alias Area

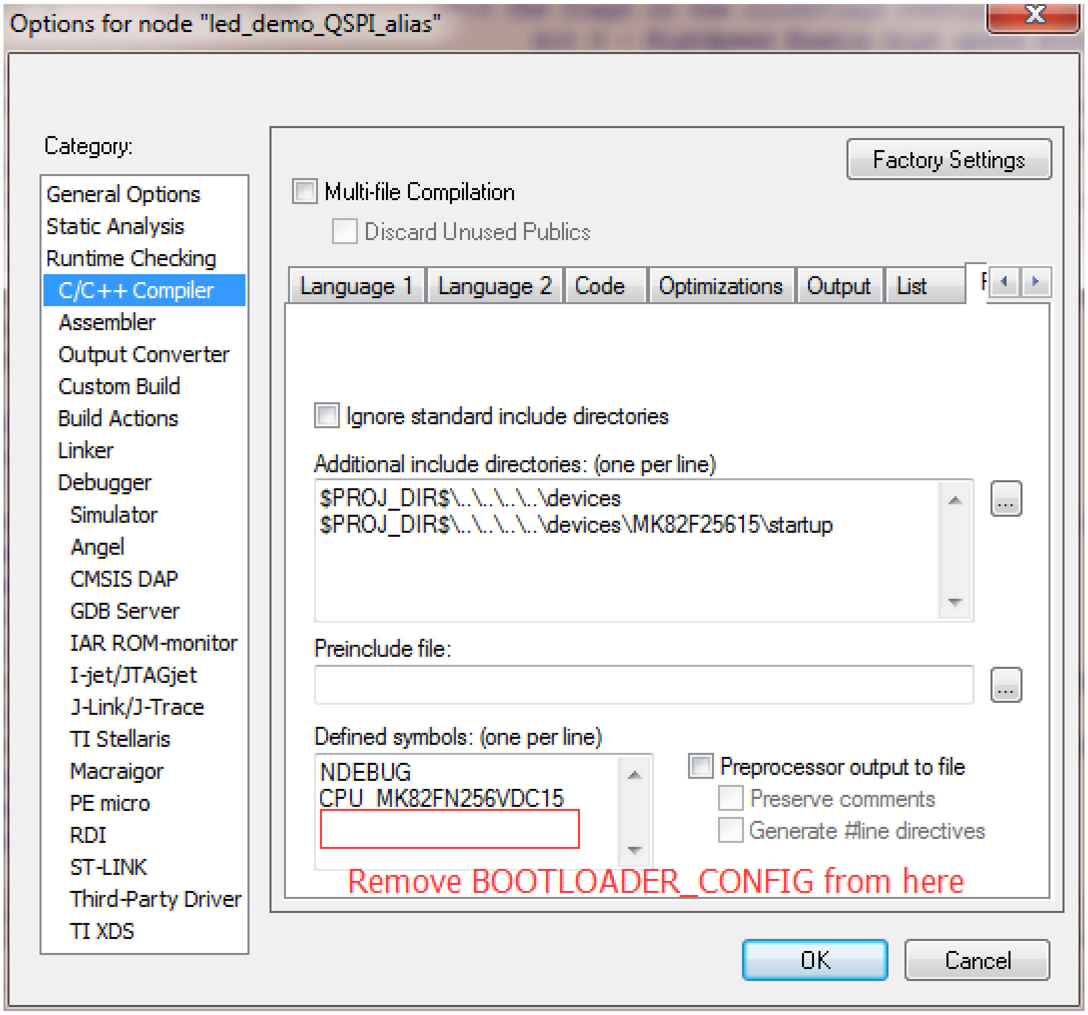
Using the led_demo_QSPI as a starting point, modify the linker file using the IAR project as an example. The following figure shows the changes to the address symbols in the linker file to allocate the sections to the QuadSPI alias memory.



```
define symbol m_interrupts_start       = 0x04001000;
define symbol m_interrupts_end         = 0x040013BF;

define symbol m_bootloader_config_start =  0x040013C0;
define symbol m_bootloader_config_end  = 0x000013FF;

define symbol m_flash_config_start     = 0x04001400;
define symbol m_flash_config_end       = 0x0400140F;

define symbol m_text_start             = 0x04001000;
define symbol m_text_end               = 0x07FFFFFF;
```

Figure 38. Linker file changes for QuadSPI Alias image in IAR project

Next, remove the BOOTLOADER_CONFIG macro from the IAR project, because the BCA is placed in the internal flash memory. In this example, the application is placed in the QuadSPI alias memory. See the following figure for details.



□ Multi-file Compilation

□ Discard Unused Publics

□ Ignore standard include directories

Additional include directories: (one per line)

```
$PROJ_DIR$\..\..\..\..\devices
$PROJ_DIR$\..\..\..\..\devices\MK82F25615\startup
```

Preinclude file:

Defined symbols: (one per line)

```
NDEBUG
CPU_MK82FN256VDC15
```

□ Preprocessor output to file

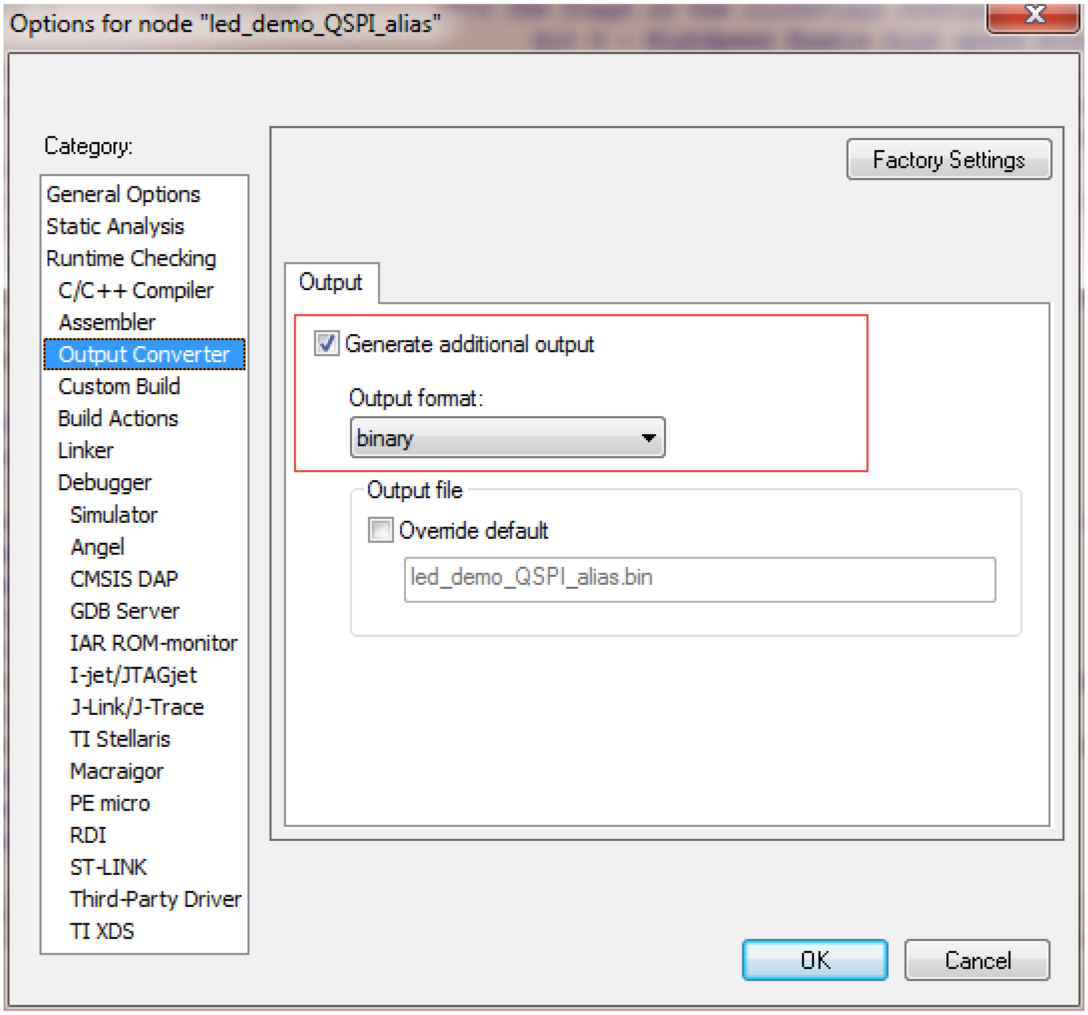
□ Preserve comments

□ Generate #line directives

Remove BOOTLOADER_CONFIG from here

Figure 39. Remove BOOTLOADER_CONFIG macro from IAR project

Finally, change the "Output Converter" option, and let the IAR generate a binary file. See the following figure.



☑ Generate additional output

Output format: binary

Output file

□ Override default

led_demo_QSPI_alias.bin

Figure 40. Let IAR output binary file

8.2 Create a simple boot application

As previously mentioned, MCU boootloader does not support boot from QuadSPI alias memory directly, and as such the host tool should command MCU bootloader to write the led_demo_QSPI application image to the physical address of QuadSPI memory starting with 0x6800_0000 address range. The workaround to make the QuadSPI application run out of alias memory is to create a simple boot application that, when invoked at boot, causes the PC to jump to the alias address where led_demo_QSPI application is linked. The boot application functionality includes:

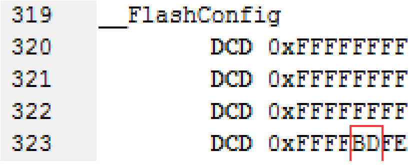
- Change the VTOR to the actual base address of the vector table in the led_demo_QuadSPI application.
- Change the stack pointed to the actual address pointed to in the start of the vector table for the led_demo_QuadSPI application.
- Jump to the led_demo_QuadSPI application.

In addition, the BCA and keyBlob also need to be included in the boot application. The example boot application is provided along with the led_demo_QuadSPI in *<sdk_package>/boards/<board>/bootloader_examples/demo_apps*. The following steps demonstrate how to generate the project for the boot application:

First, use the led_demo_PFLASH as a starting point, and replace the main() function with the code snippet from the following codeblock.

```
typedef void (*application_handler_t)(void);

enum
{
    QuadSPI_Image_Start = 0x04001000ul,
};

int main(void)
{
    static uint32_t s_stackPointer = 0;
    static application_handler_t runApplication;

    // Set the VTOR to the application vector table address
    SCB->VTOR = QuadSPI_Image_Start;

    s_stackPointer = *(uint32_t *)QuadSPI_Image_Start;
    runApplication = *(application_handler_t *)(QuadSPI_Image_Start + 4);

    __set_MSP(s_stackPointer);
    __set_PSP(s_stackPointer);

    runApplication();

    // Never run here
    while (1)
    {
    }
}
```

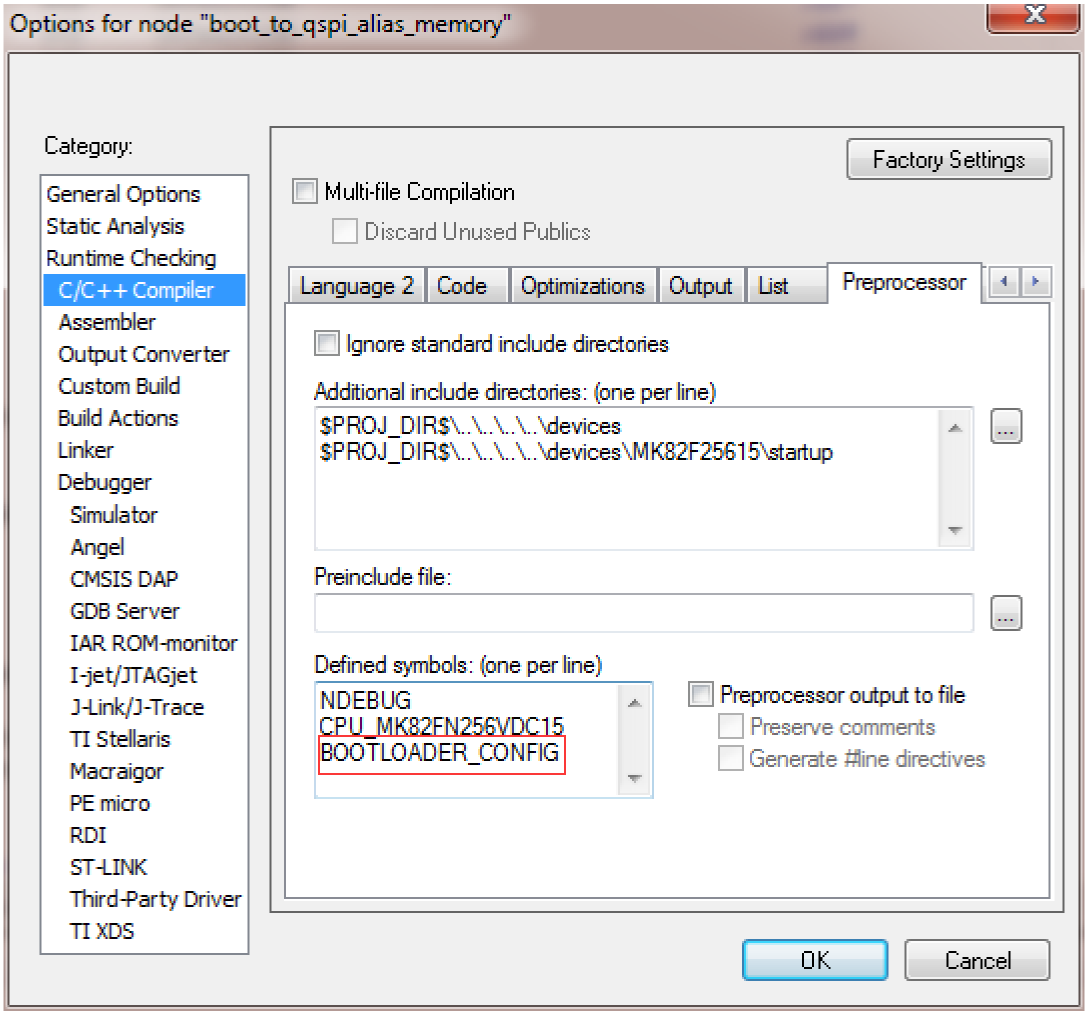
Next, change the startup_MK82F25616.s file. Ensure that FOPT [7:6] (loaded from address 0x40D) is set to 0b10. See the following figure.



```
__FlashConfig
        DCD 0xFFFFFFFF
        DCD 0xFFFFFFFF
        DCD 0xFFFFFFFF
        DCD 0xFFFFBDFE
```

Figure 41. Change FOPT to 0xBD

Enable BCA in the boot project by defining BOOTLOADER_CONFIG macro. See the following figure.



□ Multi-file Compilation

□ Discard Unused Publics

□ Ignore standard include directories

Additional include directories: (one per line)

```
$PROJ_DIR$\..\..\..\..\devices
$PROJ_DIR$\..\..\..\..\devices\MK82F25615\startup
```

Preinclude file:

Defined symbols: (one per line)

```
NDEBUG
CPU_MK82FN256VDC15
BOOTLOADER_CONFIG
```

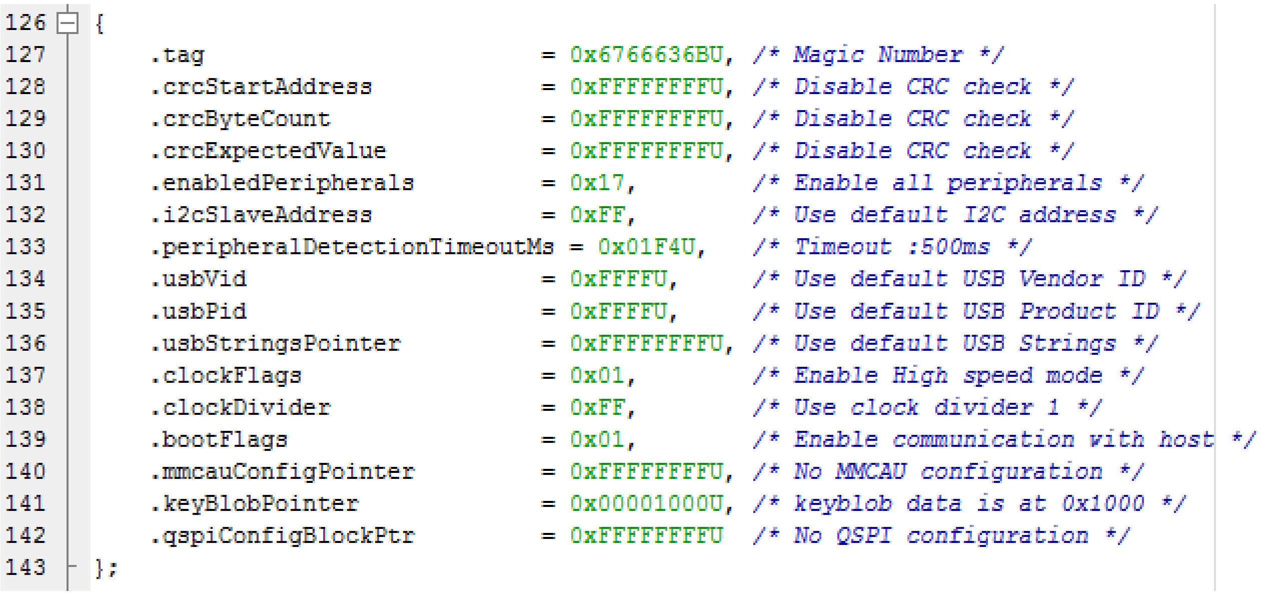
□ Preprocessor output to file

□ Preserve comments

□ Generate #line directives

Figure 42. Change Enable BCA in IAR project

Change the BCA fields as needed. For example, if 'peripheralDetectionTimeoutMs' needs to be changed to 500 and the 'keyBlobPointer' to 0x1000. The example BCA structure is shown in the following figure.



```
{
    .tag                           = 0x6766636BU, /* Magic Number */
    .crcStartAddress               = 0xFFFFFFFFU, /* Disable CRC check */
    .crcByteCount                  = 0xFFFFFFFFU, /* Disable CRC check */
    .crcExpectedValue              = 0xFFFFFFFFU, /* Disable CRC check */
    .enabledPeripherals            = 0x17,        /* Enable all peripherals */
    .i2cSlaveAddress               = 0xFF,        /* Use default I2C address */
    .peripheralDetectionTimeoutMs = 0x01F4U,      /* Timeout :500ms */
    .usbVid                        = 0xFFFFU,     /* Use default USB Vendor ID */
    .usbPid                        = 0xFFFFU,     /* Use default USB Product ID */
    .usbStringsPointer             = 0xFFFFFFFFU, /* Use default USB Strings */
    .clockFlags                    = 0x01,        /* Enable High speed mode */
    .clockDivider                  = 0xFF,        /* Use clock divider 1 */
    .bootFlags                     = 0x01,        /* Enable communication with host */
    .mmcauConfigPointer            = 0xFFFFFFFFU, /* No MMCAU configuration */
    .keyBlobPointer                = 0x00001000U, /* keyblob data is at 0x1000 */
    .qspiConfigBlockPtr            = 0xFFFFFFFFU  /* No QSPI configuration */
};
```

Figure 43. Update BCA

Finally, change the "Output Converter" option, and let the IAR output SREC file.

8.3 Downloading application running on QuadSPI alias memory with SB file

Assume that the application running on QuadSPI alias memory is called “led_demo_qspi_alias.bin”, the boot application is called “boot_to_qspi_alias_memory.srec”, and the QCB is called “qspi_config_block.bin”. An example BD file to generate the required SB file is shown in the following figure. Note that only one SB file is needed to load both boot application "boot_to_qspi_alias_memory.srec" and led_demo_QuadSPI_alias.bin.

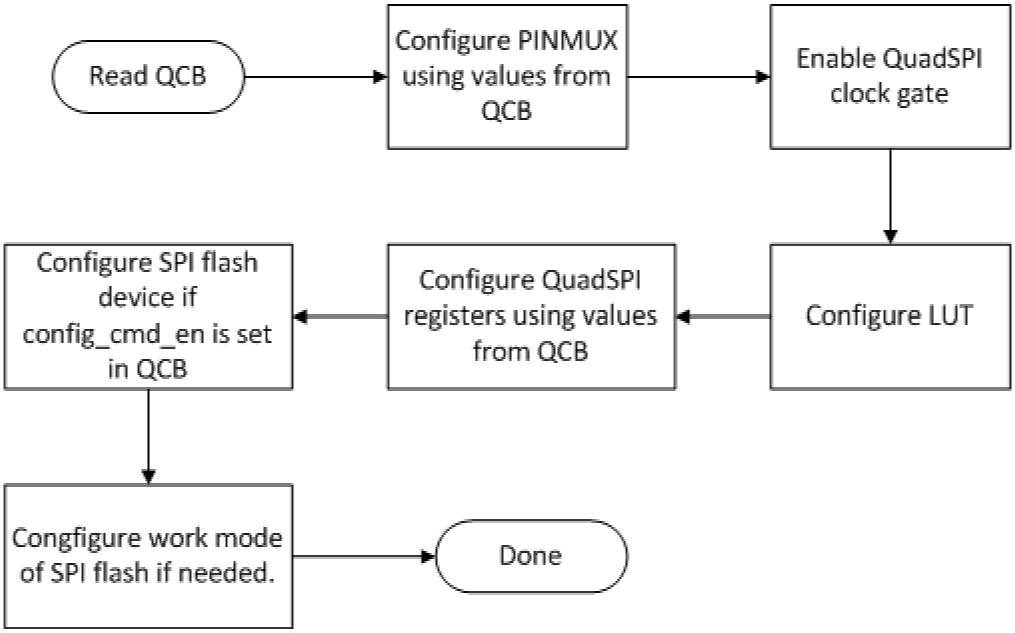

Read QCB
Configure PINMUX using values from QCB
Enable QuadSPI clock gate
Configure LUT
Configure QuadSPI registers using values from QCB
Configure SPI flash device if config_cmd_en is set in QCB
Congfigure work mode of SPI flash if needed.
Done

Figure 44. QuadSPI configurations flow in MCU bootloader

As previously mentioned, MCU bootloader does not recognize the QuadSPI alias memory addresses. Therefore, in the BD file the QuadSPI physical memory addresses should be specified for load and erase commands as shown in the following codeblock.

Generate the SB file and download it to the target device following instructions provided in Section 5.3.

```
# The sources block assigns file names to identifiers
sources {
    # SREC File path
    mySrecFile = "boot_to_qspi_alias_memory_rom.srec";
    # QCB file path
    qspiConfigBlock = "qspi_config_block.bin";
    # Alias QSPI image file path
    myBinFile = "led_demo_qspi_alias_rom.bin";
}

# The section block specifies the sequence of boot commands to be written to
# the SB file
section (0) {
    #1. Erase the vector table and flash config field.
    erase 0..0x800;

    # Step 2 and Step 3 are optional if the QuadSPI is configured at startup.
    #2. Load the QCB to RAM
    load qspiConfigBlock > 0x20000000;

    #3. Configure QuadSPI with the QCB above
    enable qspi 0x20000000;

    #4. Erase the QuadSPI memory region before programming.
    erase 0x68000000..0x68004000;

    #5.Load the QCB above to the start address of QuadSPI memory
    load qspiConfigBlock > 0x68000000;

    #. Load all boot_to_qspi_alias_memory applcation
    load mySrecFile;

    #7. Load Alias QSPI image
    load myBinFile > 0x68001000;

    #8. Reset target.
    reset;
}
```

8.4 Creating encrypted QuadSPI application running on QuadSPI Alias memory with SB file

Using the steps mentioned in Section 6.1 and Section 6.2 and using the same SB key, KEK, and KeyBlob, an encrypted SB file containing the encrypted QuadSPI alias image can be generated. See the following BD file for more details.

NOTE

1. The application is linked to the alias address range (0x0400_0000).
2. The application is loaded to the physical address range (see BD file step #7).
3. In the KeyBlob block, the OTFAD range is programmed to the physical address range.

```
# The sources block assigns file names to identifiers
sources {
    # SREC File path
    mySrecFile = "boot_to_qspi_alias_memory_rom.srec";
    # QCB file path
    qspiConfigBlock = "qspi_config_block.bin";
    # Alias QSPI image file path
    myBinFile = "led_demo_qspi_alias_rom.bin";
}

# The keyblob creates a structure with up to 4 keyblob entries.
# The empty parentheses syntax specifies an entry of all zeros (no encryption)
# Each entry consists of 4 parameters:
# start     - start address of encrypted block.
# end       - end address of encrypted block.
# key       - AES-CTR mode encryption key for this range.
# counter   - initial counter value for AES-CTR encryption for this range.
keyblob (0) {
    (
        start=0x68001000,
        end = 0x68001fff,
        key="000102030405060708090a0b0c0d0e0f",
        counter="0123456789abcdef"
    )
    ()
    ()
    ()
}

# The section block specifies the sequence of boot commands to be written to
# the SB file
section (0) {
    #1. Erase the vector table and flash config field.
    erase 0..0x800;

    # Step 2 and Step 3 are optional if the QuadSPI is configured at startup.
    #2. Load the QCB to RAM
    load qspiConfigBlock > 0x20000000;

    #3. Configure QuadSPI with the QCB above
    enable qspi 0x20000000;

    #4. Erase the QuadSPI memory region before programming.
    erase 0x68000000..0x68004000;

    #5.Load the QCB above to the start address of QuadSPI memory
    load qspiConfigBlock > 0x68000000;

    #6. Load all boot_to_qspi_alias_memory applcation
    load mySrecFile;

    #7. Encrypt QuadSPI Alias Application and load it to QuadSPI memory
    encrypt(0) {
        # Load alias QSPI image
        load myBinFile > 0x68001000;
    }

    #8. Encrypt KeyBlob structure with KEK and load it to 0x1000
    keywrap (0) {
        load {{000102030405060708090a0b0c0d0e0f}} > 0x1000;
    }

    #9. Reset target.
    reset;
}
```

9 Appendix A - QuadSPI configuration procedure

For MCU bootloader, follow the below steps to perfrom QuadSPI configuration using the QCB data. The following figure depicts the corresponding flow chart:

- Detect the location of QCB from either start address of QuadSPI memory or internal flash
- Configure QuadSPI pinmux based on the information from QCB
- Enable QuadSPI clock gate, prepare to configure QuadSPI registers
- Configure look-up table
- Configure QuadSPI registers such as AHB buffer size and DDR mode as needed
- Configure work mode of external SPI flash device, for example, Quad Mode or Octal Mode
- Additional configuration for external SPI flash device, if required in the QCB

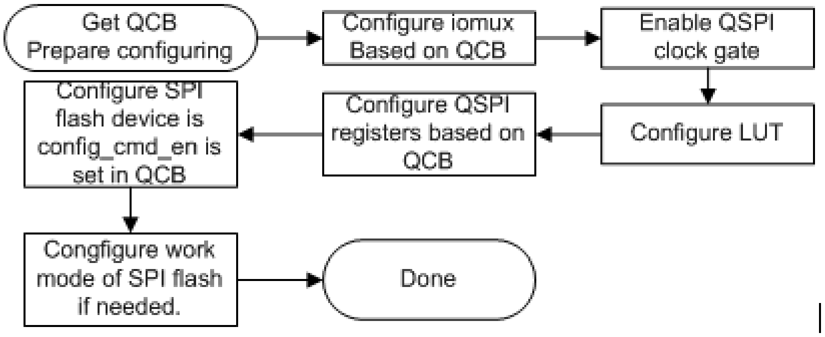


Figure 45. QuadSPI Configuration Flow in MCU bootloader

10 Appendix B - Re-enter MCU bootloader under direct boot mode

When direct boot is enabled in the BCA with bootFlags field set to 0xFE, ROM configures the QCB and jumps to the QuadSPI application image directly, bypassing the detection of active peripherals for firmware update from host. In this case, the QuadSPI application has to implement a workaround to invoke MCU bootloader when the host needs to update the application image. The QuadSPI application has to detect boot pin (NMI pin) assertion by the user and if asserted can follow below procedure to invoke MCU bootloader:

1. Erase the first sector of the internal flash to clear the BCA. Note that the flash configuration field of the BCA may have to be restored back, as shown in the code snippet in Figure 43.
2. Jump to the runBootloader() ROM API using the bootloader API tree pointer.

The following codeblock shows sample implementation of re-entry into bootloader from application code. The example code with the package contains the implementation of this feature in the led_demo_QuadSPI application.

```
void app_enter_bootloader(void)
{
    // Get Bootloader Tree pointer.
    const bootloader_tree_t *bootloaderTree = *(const bootloader_tree_t **)BOOTLOADER_TREE_ROOT;
    // Initialize Flash Driver
    flash_driver_t flashInstance;
    uint32_t status = bootloaderTree->flashDriver->flash_init(&flashInstance);
    // Save the flash config field before erase
    uint32_t flashConfigField[4];
    const uint32_t *flashConfigFieldStart = (const uint32_t *)(kApplicationBase + 0x400);
    for (uint32_t i = 0; i < sizeof(flashConfigField) / sizeof(flashConfigField[0]); i++)
    {
        flashConfigField[i] = *flashConfigFieldStart++;
    }
    // Erase the first sector.
    status = bootloaderTree->flashDriver->flash_erase(&flashInstance, kApplicationBase,
                                                      0x800, kFlashEraseKey);
    // Write the flash config field back.
    status = bootloaderTree->flashDriver->flash_program(&flashInstance,
             kApplicationBase + 0x400, flashConfigField, sizeof(flashConfigField));
    bootloaderTree->runBootloader(0);
}
```

11 Appendix C - Explore more features in QCB

Several more features of QuadSPI are supported by MCU bootloader such as parallel mode, continuous read mode, and so on. The following sections provide examples of generating QCB with these modes enabled.

11.1 Parallel mode

This section provides an example of generating a QCB with the parallel mode support. Pay attention to these key points:

- The sector size and page size should be twice the actual size for the parallel mode.
- The '*parallel_mode_enable*' field in QCB must be set to 1.
- The Program command should be replaced by the Page Program command, as the QuadSPI module only supports single-pad parallel programming.

This is the example:

```
// This is the QCB for the use case that
// two MX25U3235F are connected to QuadSPI0A and QuadSPI0B ports.
// Work under parallel mode
const qspi_config_t qspi_config_block = {
    .tag = kQspiConfigTag,
    .version = { .version = kQspiVersionTag },
    .lengthInBytes = 512,
    .sflash_A1_size = 0x400000,             // 4MB
    .sflash_B1_size = 0x400000,             // 4MB
    .sclk_freq = kQspiSerialClockFreq_High, // High frequency
    .sflash_type = kQspiFlashPad_Quad,      // SPI Flash devices work under quad-pad mode
    .sflash_port = kQspiPort_EnableBothPorts, // Both QSPI0A and QSPI0B are enabled.
    .busy_bit_offset = 0,                   // Busy offset is 0
    .ddr_mode_enable = 0,                   // disable DDR mode
    .dqs_enable = 0,                        // Disable DQS feature
    .parallel_mode_enable = 1,              // QuadSPI module work under parallel mode
    .pagesize = 512,                        // Page Size : 256 *2 = 512 bytes
    .sectorsize = 0x2000,                   // Sector Size: 4KB * 2 = 8KB
    .device_mode_config_en = 1,             // configure quad mode for spi flash
    .device_cmd = 0x40,                     // Enable quad mode
    .write_cmd_ipcr = 0x05000000U,          // IPCR indicating enable seqid (5<<24)
    .ips_command_second_divider = 3,     // Set second divider for QSPI serial clock to 3
    .look_up_table = {
        // Seq0: Quad Read (maximum supported freq: 104MHz)
        /*
         * CMD: 0xEB - Quad Read, Single pad
         * ADDR: 0x18 - 24bit address, Quad pads
         * DUMMY: 0x06 - 6 clock cycles, Quad pads
         * READ: 0x80 - Read 128 bytes, Quad pads
         * JUMP_ON_CS: 0
         */
        [0] = 0x0A1804EB,
        [1] = 0x1E800E06,
        [2] = 0x2400,

        // Seq1: Write Enable (maximum supported freq: 104MHz)
        /* CMD: 0x06 - Write Enable, Single pad */
        [4] = 0x406,

        // Seq2: Erase All (maximum supported freq: 104MHz)
        /* CMD: 0x60 - Erase All chip, Single pad */
        [8] = 0x460,

        // Seq3: Read Status (maximum supported freq: 104MHz)
        /*
         * CMD: 0x05 - Read Status, single pad
         * READ: 0x01 - Read 1 byte
         */
        [12] = 0x1c010405,

        // Seq4: Page Program (maximum supported freq: 104MHz)
        /*
         * CMD: 0x02 - Page Program, Single pad
         * ADDR: 0x18 - 24bit address, Single pad
         * WRITE: 0x40 - Write 64 bytes at one pass, Single pad
         *       (0x40 is ignored, because it will be overwritten by page size)
         */
        [16] = 0x08180402,
        [17] = 0x2040,

        // Seq5: Write status register to enable quad mode
        /*
         * CMD: 0x01 - Write Status Register, single pad
         * WRITE: 0x01 - Write 1 byte of data, single pad
         */
        [20] = 0x20010401,
        // Seq7: Erase Sector
        /*
         * CMD:  0x20 - Sector Erase, single pad
         * ADDR: 0x18 - 24 bit address, single pad
         */
        [28] = 0x08180420,

        // Seq8: Dummy
        /*
         * CMD: 0 - Dummy command, used to force SPI flash to exit continuous read mode.
         * unecessary here because the continous read mode isn't enabled.
         */
        [32] = 0,
    },
};
```

NOTE

The previous example must be placed in the *<sdk_package>/middleware/mcu-boot/apps/QCBGenerator/src* folder.

11.2 Continuous read mode

The MX25U3235F supports the continuous read mode (performance enhance mode) to provide high performance reads. The important item to be configure ford this use case is:

- The Dummy LUT entry must be configured according to the condition of the exiting continuous read mode. Otherwise, the device fails to perform an erase or a program operation as it cannot exit this mode correctly.

The following is an example:

NOTE

Only the flash device connected on the QuadSPI0 A1 supports this module.

```
// This is the QCB for when two MX25U3235F are connected to QuadSPI0A and QuadSPI0B ports.
// Work under parallel mode
const qspi_config_t qspi_config_block = {
    .tag = kQspiConfigTag,
    .version = { .version = kQspiVersionTag },
    .lengthInBytes = 512,
    .sflash_A1_size = 0x400000,              // 4MB
    .sclk_freq = kQspiSerialClockFreq_High,  // High frequency
    .sflash_type = kQspiFlashPad_Quad,       // SPI Flash devices work under quad-pad mode
    .sflash_port = kQspiPort_EnableBothPorts, // Both QSPI0A and QSPI0B are enabled.
    .busy_bit_offset = 0,                    // Busy offset is 0
    .ddr_mode_enable = 0,                    // disable DDR mode
    .dqs_enable = 0,                         // Disable DQS feature
    .parallel_mode_enable = 1,               // QuadSPI module work under parallel mode
    .pagesize = 512,                         // Page Size : 256 *2 = 512 bytes
    .sectorsize = 0x2000,                    // Sector Size: 4KB * 2 = 8KB
    .device_mode_config_en = 1,              // configure quad mode for spi flash
    .device_cmd = 0x40,                      // Enable quad mode
    .write_cmd_ipcr = 0x05000000U,           // IPCR indicating enable seqid (5<<24)
    .ips_command_second_divider = 3,     // Set second divider for QSPI serial clock to 3
    .look_up_table = {
        // Seq0: Quad Read (maximum supported freq: 104MHz)
        /*
         * CMD: 0xEB - Quad Read, Single pad
         * ADDR: 0x18 - 24bit address, Quad pads
         * MODE: 0xA5 - Continuous read mode, Quad Pads
         * DUMMY: 0x04 - 4 clock cycles, Quad pads
         * READ: 0x80 - Read 128 bytes, Quad pads
         * JUMP_ON_CS: 1
         */
        [0] = 0x0A1804EB,
        [1] = 0x0E04012A5,
        [2] = 0x24011E80,

        // Seq1: Write Enable (maximum supported freq: 104MHz)
        /* CMD: 0x06 - Write Enable, Single pad */
        [4] = 0x406,

        // Seq2: Erase All (maximum supported freq: 104MHz)
        /* CMD: 0x60 - Erase All chip, Single pad */
        [8] = 0x460,

        // Seq3: Read Status (maximum supported freq: 104MHz)
        /*
         * CMD: 0x05 - Read Status, single pad
         * READ: 0x01 - Read 1 byte
         */
        [12] = 0x1c010405,

        // Seq4: Page Program (maximum supported freq: 104MHz)
        /*
         * CMD: 0x02 - Page Program, Single pad
         * ADDR: 0x18 - 24bit address, Single pad
         * WRITE: 0x40 - Write 64 bytes at one pass, Single pad
         *        (0x40 is ignored, because it will be overwritten by page size)
         */
        [16] = 0x08180402,
        [17] = 0x2040,

        // Seq5: Write status register to enable quad mode
        /*
         * CMD: 0x01 - Write Status Register, single pad
         * WRITE: 0x01 - Write 1 byte of data, single pad
         */
        [20] = 0x20010401,

        // Seq7: Erase Sector
        /*
         * CMD: 0x20 - Sector Erase, single pad
         * ADDR: 0x18 - 24 bit address, single pad
         */
        [28] = 0x08180420,

        // Seq8: Dummy
        /*
         * CMD: 0xFF - Dummy command,
         * used to force SPI flash to exit continuous read mode.
         * Unnecessary here because the continuous read mode isn't enabled.
         */
        [32] = 0xFF,
    },
};
```

NOTE

See the example from the *<sdk_package>/middleware/mcu-boot/apps/QCBGenerator/src* folder.

12 Revision history

The following table contains a history of changes made to this user's guide.

Table 5. Revision history

Revision number	Date	Substantive changes
0	09/2015	Initial release
1	04/2016	Kinetis Bootloader v2.0 release
2	05/2018	MCU Bootloader v2.5.0 release
3	09/2018	MCU Bootloader v2.6.0 release
4	11/2018	MCU Bootloader v2.7.0 release

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.

NXP, the NXP logo, NXP SECURE CONNECTIONS FOR A SMARTER WORLD, All other product or service names are the property of their respective owners. ARM, AMBA, ARM Powered, are registered trademarks of ARM Limited (or its subsidiaries) in the EU and/or elsewhere. All rights reserved.

For more information, please visit: http://www.nxp.com

For sales office addresses, please send an email to: salesaddresses@nxp.com

Date of release: December 2018
Document identifier: MBOOTQSPIUG