AN12829
Firmware Upgrade Using KM35Z512

Rev. 0 — 25 April 2020

Application Note

Contents

1 Introduction... 1
2 MKM35Z512 series MCU...................... 1
3 Basic theory... 2
4 Hardware and setup.............................. 2
5 Software design.................................... 5
6 Summary... 21
7 References.. 22
8 Revision history.................................. 22

1 Introduction

This document describes a firmware upgrade procedure using KM35Z512 Kinetis microcontroller. Firmware upgrade of the MCU-based embedded products is a very useful feature as it eliminates the need of any programming tool to reprogram the MCU, which is difficult to be done with the products in field. Using only a suitable communication channel for example, serial port or remote communication module like GPRS communication, the products in field can be upgraded with software patches or new features. KM35Z512 device has 2 x 256 KB program flash memory. This means, although the program code can execute from entire 512 KB flash memory, alternatively program code when executing only from one bank of the flash memory, the other bank can be used to store new version of the application firmware, which can be copied to the first bank to upgrade the software/firmware. Here the second bank is used only to store the new version of firmware and not to execute at place. Firmware upgrade can be of full replacement of the application software or only partially, if the application is divided into code sections and/or data sections.

The process described in this document is useful to utilize as an example to implement firmware upgrade of applications in different embedded solutions using Kinetis microcontrollers with dual bank flash memory. Using dual bank memory does not put any constrains on the application flow during firmware upgrade as the second flash bank in KM35Z512 is read-while-write (rww) which means, read/execution of code can happen in first flash bank while programming on the second bank is ongoing.

The process described in this document can also be utilized with little update to store and upgrade firmware within single program flash bank. In that case, without the read-while-write facility we may need to temporarily disable the code to execute from the same bank where another partition is attempted to be erased/programmed. In such case, executing code from RAM temporarily solves the issue. KM35Z512 do not have such limitation if the product is designed appropriately that the active application is executed from one bank and new application is stored in the other bank.

2 MKM35Z512 series MCU

NXPs MKM35Z512 series MCU is based on the 90 nm process technology. It has on-chip peripherals, computational performance, and power capabilities to enable the development of a low-cost and highly integrated power meter, see Figure 1. It is based on the 32-bit Arm Cortex-M0+ core, with CPU clock rated up to 75 MHz. The analog measurement frontend is integrated on all devices; it includes a highly accurate 24-bit Sigma Delta ADC, PGA, high-precision internal 1.2 V voltage reference (Vref), phase shift compensation block, 16-bit SAR ADC, a peripheral crossbar (XBAR), Programmable Delay Block (PDB), and a Memory-Mapped Arithmetic Unit (MMAU). The XBAR module acts as a programmable switch matrix, enabling multiple simultaneous connections of internal and external signals. An accurate Independent Real-Time Clock (IRTC) with passive and active tamper detection capabilities is also available on all devices.

In addition to high-performance analog and digital blocks, the MKM35Z512 series MCU was designed with an emphasis on achieving the required software separation. It integrates hardware blocks, supporting the distinct separation of the legally relevant software from other software functions.

The hardware blocks controlling and/or checking the access attributes include:

- Arm Cortex-M0+ core
- DMA controller module

- Miscellaneous control module
- Memory protection unit
- Peripheral bridge
- General-purpose input / output module

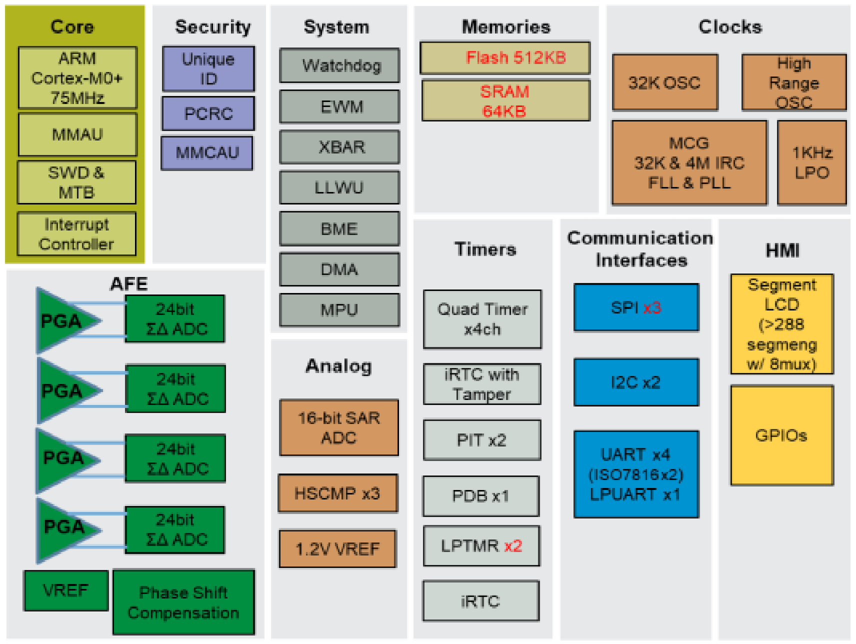


Figure 1. KM35Z512 MCUs block diagram

The MKM35Z512 devices are highly capable and fully programmable MCUs with application software driving the differentiation of the product. Currently, the necessary peripheral software drivers, metering algorithms, communication protocols, and a vast number of complementary software routines are available directly from semiconductor vendors or third parties. Because the MKM35Z512 MCUs integrate a high-performance analog frontend, communication peripherals, hardware blocks for software separation, and capable of executing various Arm Cortex-M0+ compatible software, they are ideal components for development of residential, commercial, and light industrial electronic power meter applications and other similar applications.

3 Basic theory

Typically, firmware update is done by the bootloader (flash resident in this case) only. In such case, after power on reset, bootloader can switch to download process and accept the new application firmware, and then can activate it. Another way of application firmware upgrade is done by current application to store the new application software to a reserved memory, and after the download is done, based on a schedule time or immediate action, can switch to its bootloader to activate the new application firmware. The later process is useful as it does not disrupt the application execution while the time-consuming firmware download and storing process can be carried on and once finished, can be scheduled to activate later as per convenience. Also, this process is more suitable and similar for remote download using a GPRS or similar module in for example, a smart energy meter. In such cases, the GPRS module running in the context of application software is utilized to download and store the new firmware. The application can schedule the activation for later time, using off course the help of bootloader when a downtime or backup time is identified.

4 Hardware and setup

The setup consists of a TWR-KM35Z7M tower board standalone and connected to a host machine by micro-B USB cable. The host machine can be a personal computer or a handheld unit able to communicate to the serial port of the tower card as emulated through its USB OpenSDA as described in the quick start guide of TWR-KM35Z7M.

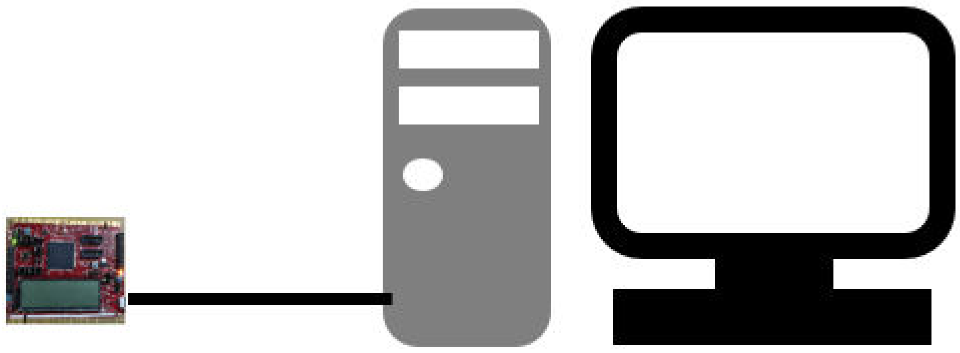


Figure 2. Firmware upgrade setup

During the development of this document, a personal computer is used as host machine with Windows OS 10 to set up the communication and a software executable to communicate as per the communication process described in this document.

4.1 Get to know the TWR-KM35Z7M

To use the TWR-KM35Z7M for the first time, download the Quick Start Guide and follow the step-by-step instructions to setup for programming and communicate with the board. Normally, no jumper setting needs to be changed and the board can use the default jumper settings.

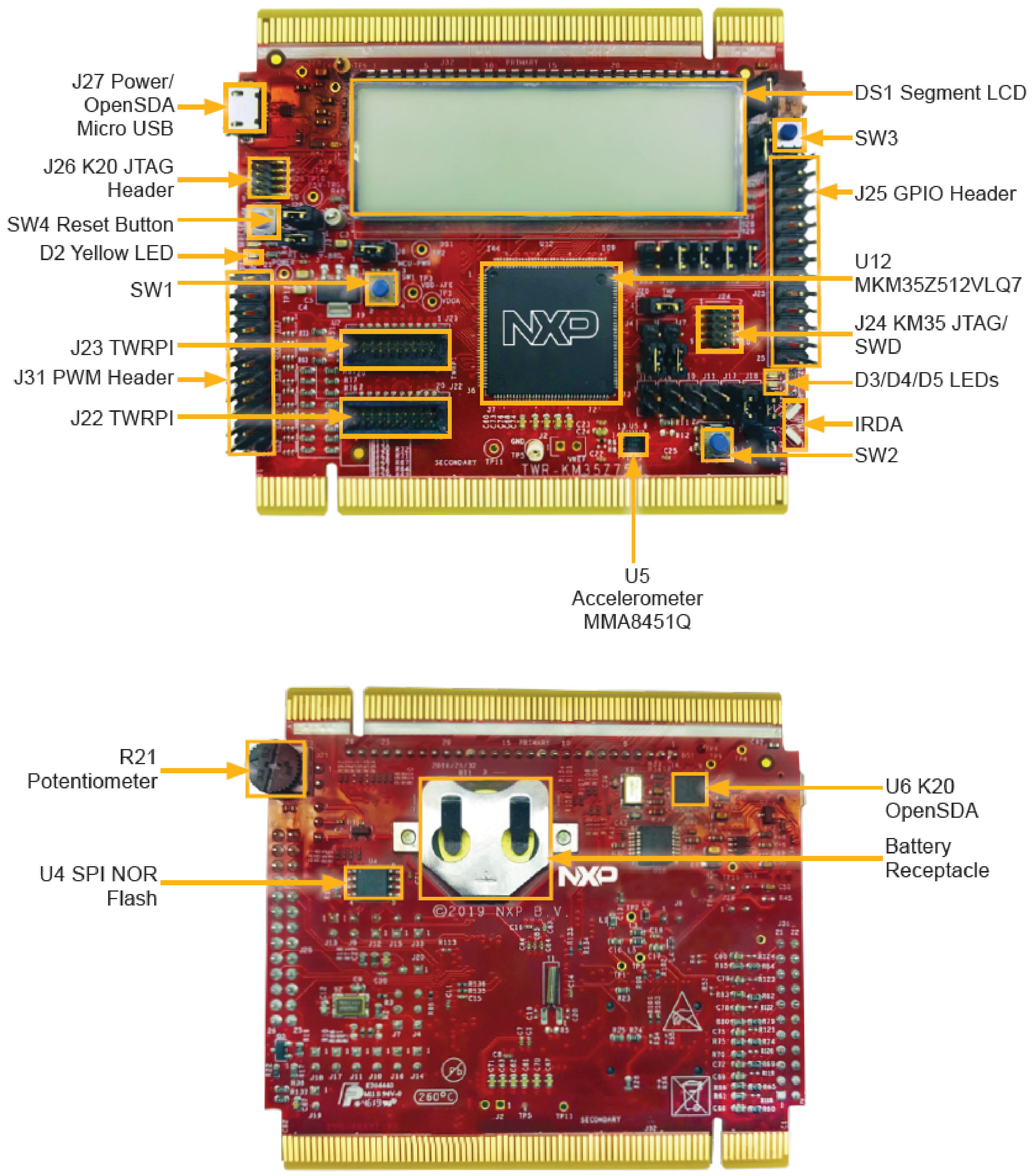


Figure 3. TWR_KM35Z512 and its components

5 Software design

This section describes the bootloader and software application of the MKM35Z512 firmware upgrade process. The bootloader is a small footprint boot code and facilitator to copy new application firmware version to the active code and data sections. The software application, apart from doing an application-specific job, also does the new firmware transfer job from the firmware provider host.

Below flash memory partitions done for –

- Bootloader – initial 6 KB, address range 0x0000_0000 to 0x0000_17FF
- Application software/firmware – maximum 250 KB, address range 0x0000_1800 to 0x0003_FFFF
- Firmware storage – the whole 256 KB of the second memory bank of MKM35Z512 MCU, address range 0x0004_0000 to 0x0007_FFFF . Initial 2 KB of memory (size of a flash sector) is used to store useful metadata about the new firmware, for example, the firmware file size, block size, total number of blocks. This metadata can include more information about the firmware. The rest of the memory in this bank is used for actual application firmware.

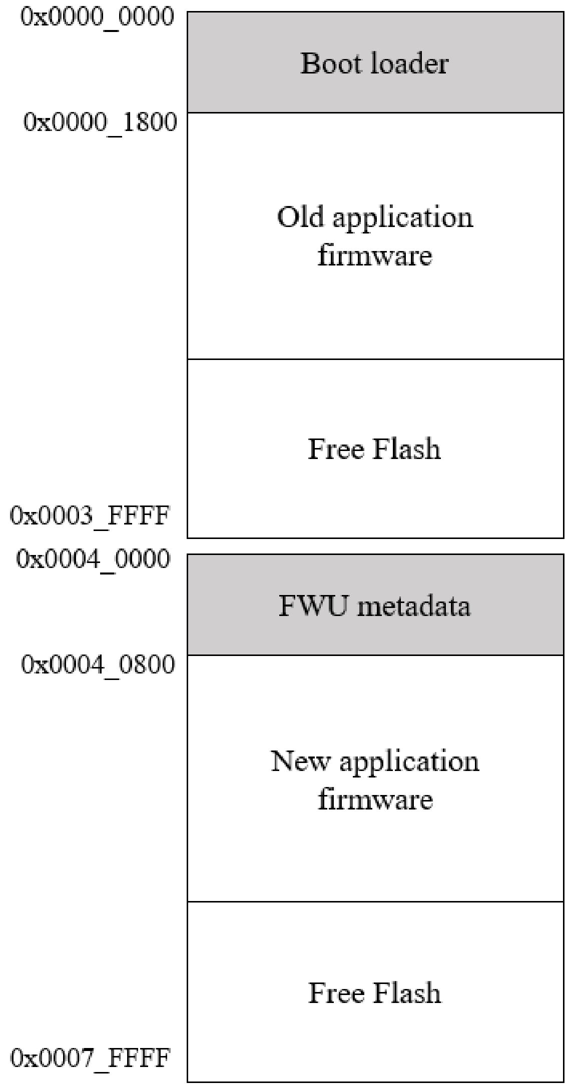


Figure 4. Flash partition scheme

5.1 Block diagram

Both the bootloader and the application software are written in the C language and compiled using the IAR® Embedded Workbench for Arm (version 8.42 or higher). The software application is based on the MKM35Z512 bare-metal SDK software drivers.

The application can do different type of jobs, but for simplification, this application firmware only blinks an LED on board at a constant rate. Other than this, the application does the communication and flash program activities which are essential to facilitate and demonstrate the firmware transfer from the host side.

Figure 5 shows the simple software architecture of the application program, including interactions of the software peripheral drivers. All tasks executed by the MKM35Z512 application software are briefly explained in the following subsections.

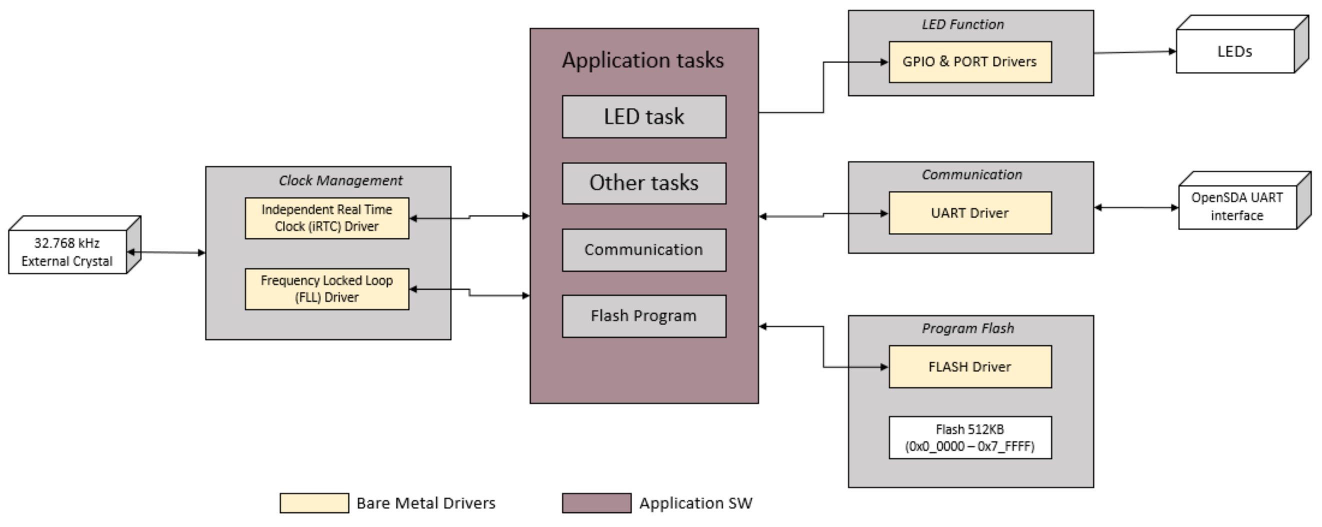


Figure 5. Application software architecture

5.1.1 Bootloader

Boot loader is the first program that executes after MCU is reset. So, its exception vectors are available at the reset vector address 0x0000_0000 as per Arm Cortex-M0+ processor architecture. This bootloader does one of two jobs –

1. Jump to application entry point located at a fixed flash memory location 0x0000_1800, or,
2. Upon receiving a trigger from the application with a unique code 0xACAC being written in the iRTC RAM, activate the new firmware by copying from the second flash bank partition to the application partition in the first program flash bank.

5.1.1.1 Bootloader program flow

Bootloader apart from just branching to the application software after reset, also compliments the user application to facilitate the new firmware activation as described in this document.

Figure 6 describes the boot loader flow.

NOTE

This boot loader does not accept or transfer new application firmware from the host but just facilitates the activation of a pre-loaded new application firmware from the storage area to the active area from where the application is supposed to execute.

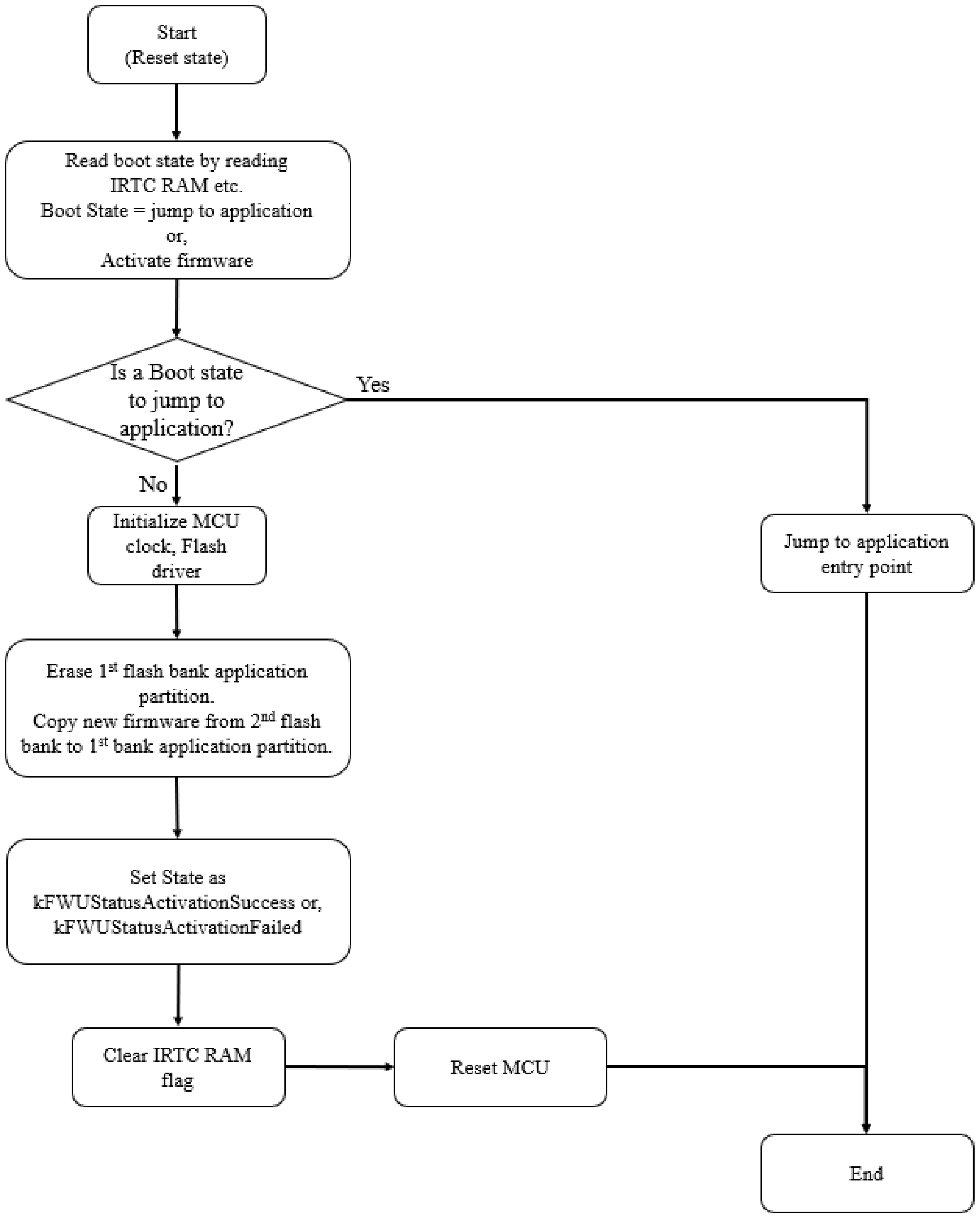
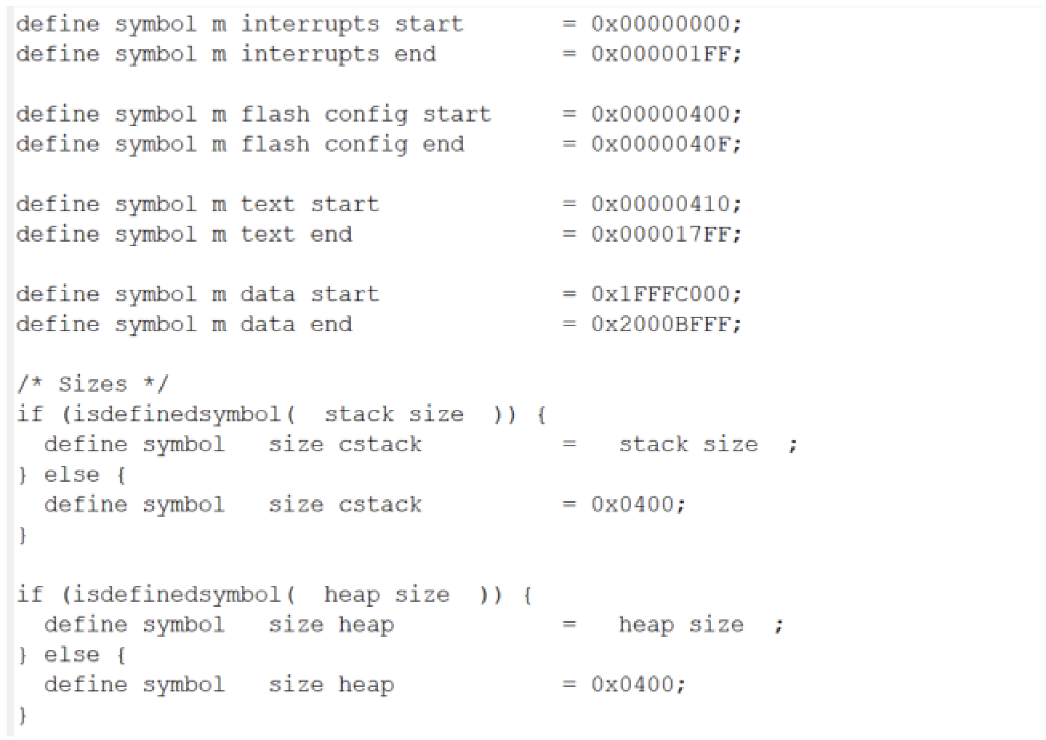

Start
(Reset state)
Read boot state by reading IRTC RAM etc.
Boot State = jump to application or,
Activate firmware
Is a Boot state to jump to application?
Yes
No
Initialize MCU clock, Flash driver
Jump to application entry point
Erase 1st flash bank application partition.
Copy new firmware from 2nd flash bank to 1st bank application partition.
Set State as kFWUStatusActivationSuccess or, kFWUStatusActivationFailed
Clear IRTC RAM flag
Reset MCU
End

Figure 6. Bootloader program flow

5.1.1.2 Bootloader linker file setup

Boot loader linker script file is restricted to only 0x0000_0000 to 0x0000_017FF that is, 6 KB only for program flash.



```
define symbol m interrupts start       = 0x00000000;
define symbol m interrupts end         = 0x000001FF;

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               = 0x000017FF;

define symbol m data start             = 0x1FFFC000;
define symbol m data end               = 0x2000BFFF;

/* 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 7. Bootloader linker file

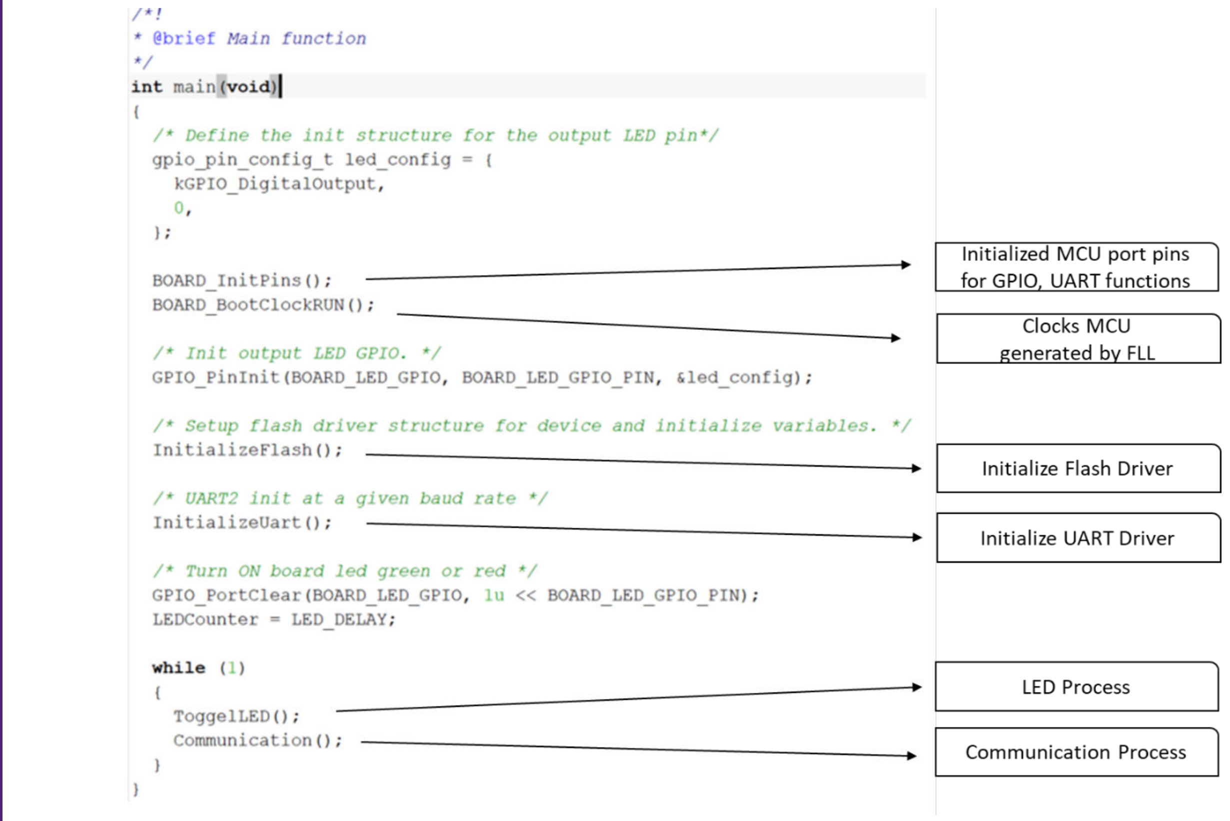
5.1.2 Application

Application firmware does the embedded software functions. For example, in case of an energy meter application, can run metrology, load survey, tariff, billing, display, handle user inputs, communication and firmware upgrade. For simplification, this application only does a job of glowing an LED on the board and then runs a communication process only to handle the new firmware download to a pre-defined flash bank area.

5.1.2.1 Application tasks

Like any other embedded application, the application software initializes the MCU hardware and peripherals for use.

The application initially glows a green or red LED in the board.



```
/*!
* @brief Main function
*/
int main(void)
{
  /* Define the init structure for the output LED pin*/
  gpio_pin_config_t led_config = {
    kGPIO_DigitalOutput,
    0,
  };

  BOARD_InitPins();
  BOARD_BootClockRUN();

  /* Init output LED GPIO. */
  GPIO_PinInit(BOARD_LED_GPIO, BOARD_LED_GPIO_PIN, &led_config);

  /* Setup flash driver structure for device and initialize variables. */
  InitializeFlash();

  /* UART2 init at a given baud rate */
  InitializeUart();

  /* Turn ON board led green or red */
  GPIO_PortClear(BOARD_LED_GPIO, 1u << BOARD_LED_GPIO_PIN);
  LEDCounter = LED_DELAY;

  while (1)
  {
    ToggelLED();
    Communication();
  }
}
```

Initialized MCU port pins for GPIO, UART functions
Clocks MCU generated by FLL
Initialize Flash Driver
Initialize UART Driver
LED Process
Communication Process

Figure 8. Application program initialization and tasks C code

After this, Communication task waits for any *command* from the host machine. If any command is received and identified correctly, then respective actions are taken by the software and *response* is sent to the host that sent the command.

5.1.2.2 Initialization of hardware

Application software initializes the MCU hardware and peripherals for use. These include MCU port pins for GPIO function to glow LED, UART function for communication purpose. Clock module is initialized to run the MCU at higher clock rate. As the application is going to erase/program flash sectors, flash drivers are also initialized.

The application initially glows a green or red LED in the board.

5.1.2.3 Communication process

Communication task waits for any *command* from the host machine. If any command is received and identified correctly, then respective actions are taken by the software and *response* is sent to the host that sent the command.

Below is the list of possible command codes the Communication task can interpret and act accordingly –

- nxpfwutx – this 8 bytes long command is interpreted as the beginning of new firmware transfer. After receiving this command, the application will expect the command parameters which includes 4 bytes FW file size and 4 bytes block size. Number of blocks in the file are determined automatically by MCU software by dividing file size by the block size.

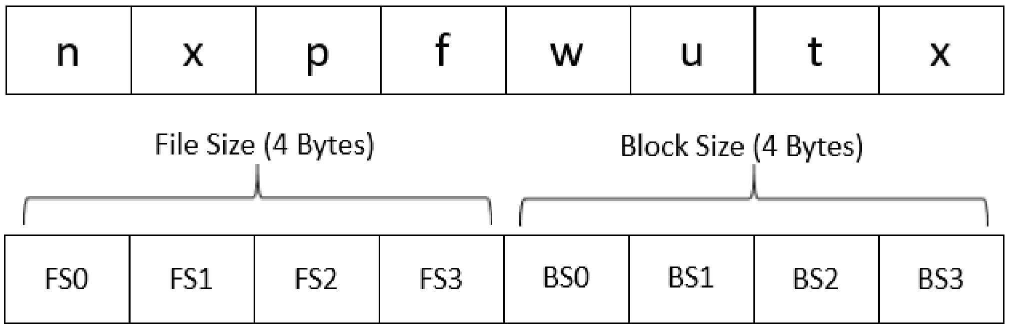


Figure 9. Firmware transfer command and parameter syntax

FS0-FS3 4 bytes contain the file size, BS0-BS3 4 bytes contains the block size.

- nxpfwuact – this 9 bytes long command is interpreted as the request to transfer new FW from the stored flash region to the active program flash region. In this demo, the new FW is stored in second flash bank range 0x0004_0000 to 0x0007_FFFF. The working active application always resides in the second partition of the first flash bank starting range 0x0000_1800 to 0x0003_FFFF.

Communication process is triggered after receiving a command or block of firmware binary file data by the asynchronous UART interrupts. The reception of commands and actions are identified into steps as shown below –

kFWUStatusIdle	Idle state
kFWUStatusCmd	Received a command
kFWUStatusInitiated	Received the firmware header information
kFWUStatusTransfer	Firmware transfer in progress
kFWUStatusActivationInitiated	Firmware activation has been initiated
kFWUStatusActivationSuccess	Firmware activation was successful
kFWUStatusActivationFailed	Firmware activation failed

kFWUStatusIdle – when no command/data transaction is active.

kFWUStatusCmd – when a command as mentioned above has been received. After this state is reached, next action depends on the type of command, i.e., FWU transfer command or FWU activation command as mentioned in this document.

kFWUStatusInitiated – the firmware header information has been received. This indicates that the FWU transfer command was initiated. To prepare for new firmware program in the second flash bank transfer, this firmware destination area is first prepared by erasing the second flash bank partition. The firmware file size and the block size are available in the firmware header and is stored in the firmware metadata flash area.

```
typedef struct
{
  uint32_t fileSize;
  uint32_t blockSize;
} fwu_header_t;

typedef struct
{
  fwu_header_t fwuHeader;
  uint32_t blockNumber;
  fwu_status_e fwuStatus;
} fwu_info_t;
```

kFWUStatusTransfer – transfer of firmware binary image is in progress. The firmware binary file is transferred in small blocks with fixed size, 256 bytes (but not limited to) only. To identify the firmware block, 4 bytes long block number is transferred along with the block. The current block count is updated and saved every time a new file block is received and programmed to the destination flash bank. Below diagram shows the command and block exchange sequence –

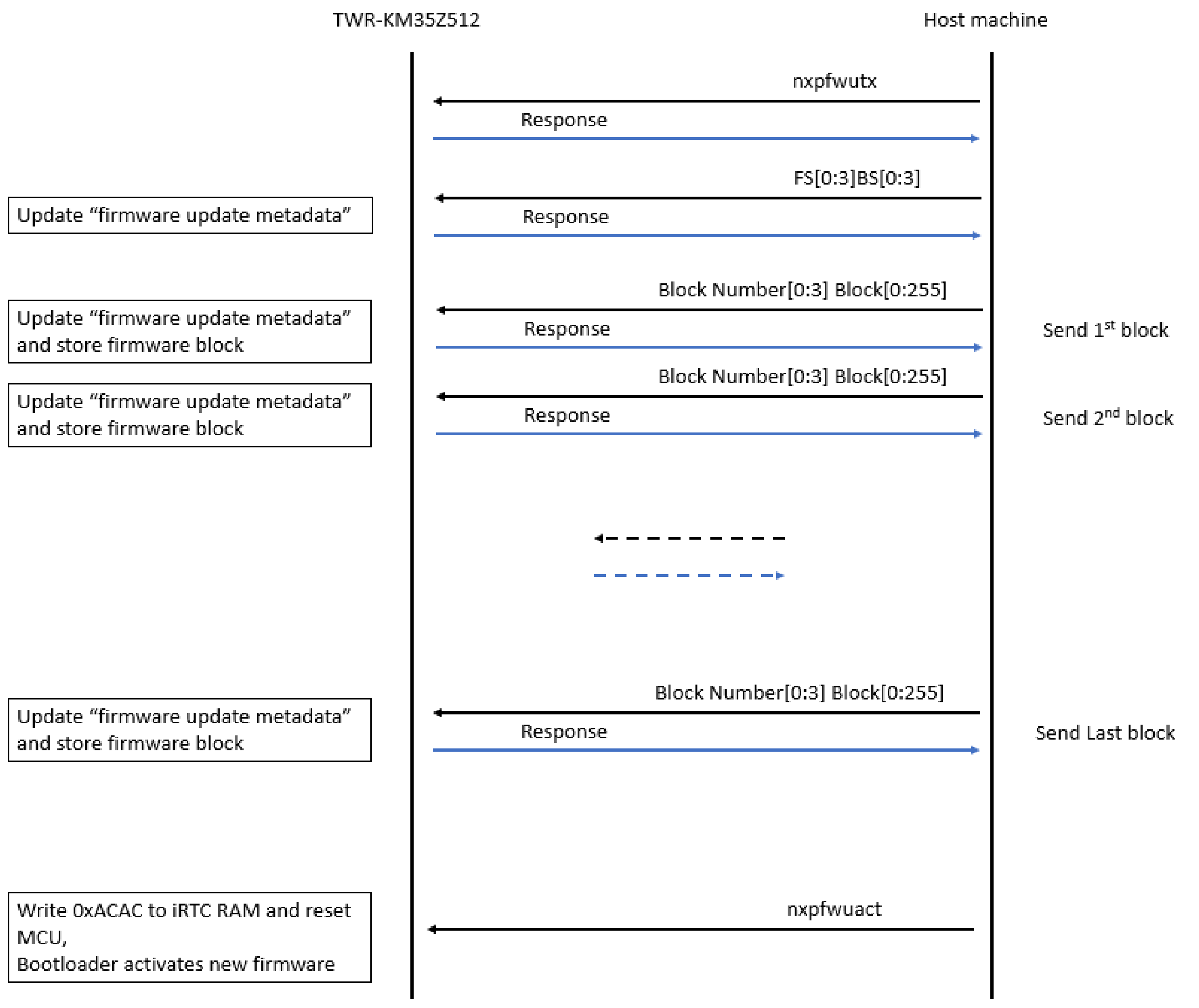
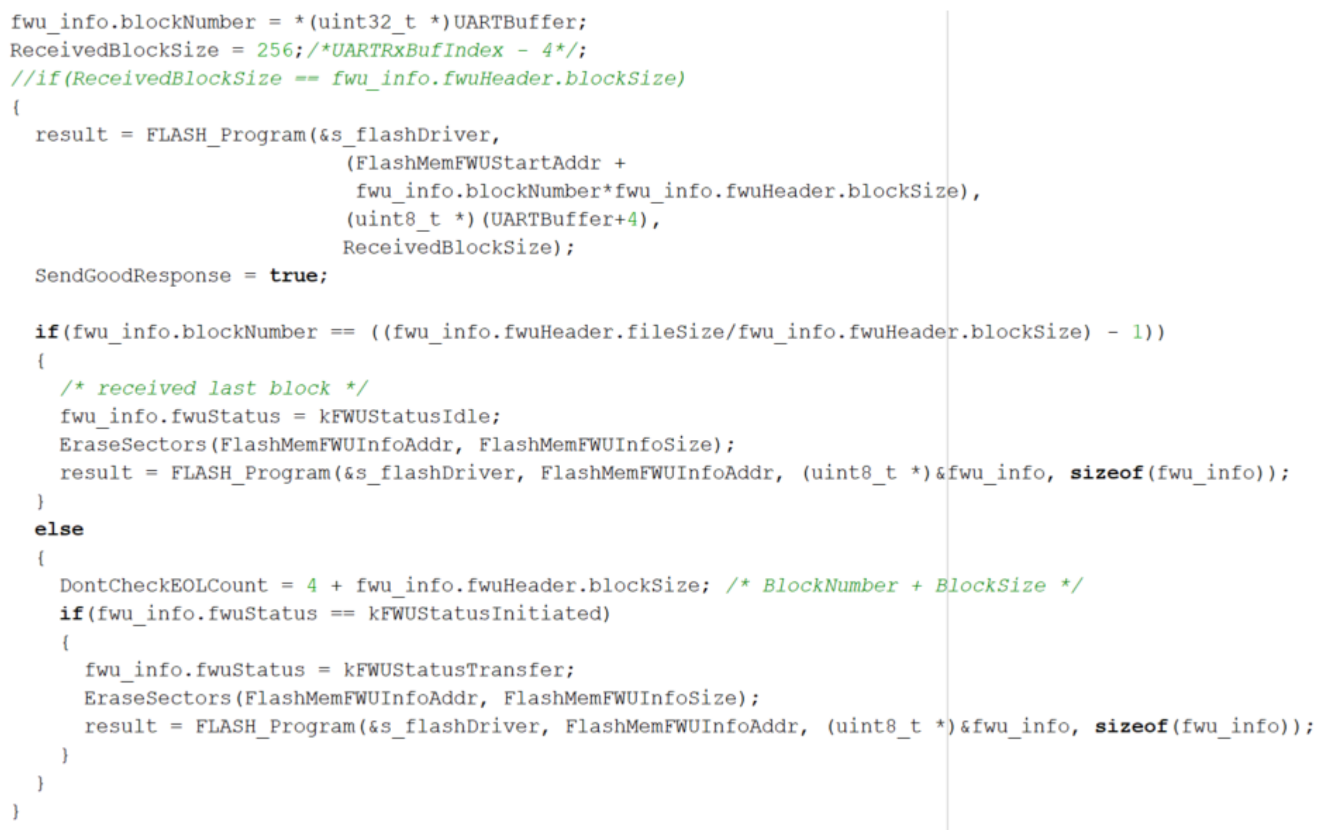

TWR-KM35Z512
Host machine
nxpfwutx
Response
FS[0:3]BS[0:3]
Update "firmware update metadata"
Response
Block Number[0:3] Block[0:255]
Update "firmware update metadata" and store firmware block
Response
Send 1st block
Block Number[0:3] Block[0:255]
Update "firmware update metadata" and store firmware block
Response
Send 2nd block
Block Number[0:3] Block[0:255]
Update "firmware update metadata" and store firmware block
Response
Send Last block
nxpfwuact
Write 0xACAC to iRTC RAM and reset MCU,
Bootloader activates new firmware

Figure 10. Communication flow control between tower board and host

Figure 11 is a C code snapshot that performs block transfer operations –



```c
fwu_info.blockNumber = *(uint32_t *)UARTBuffer;
ReceivedBlockSize = 256;/*UARTRxBufIndex - 4*/;
//if(ReceivedBlockSize == fwu_info.fwuHeader.blockSize)
{
  result = FLASH_Program(&s_flashDriver,
                          (FlashMemFWUStartAddr +
                           fwu_info.blockNumber*fwu_info.fwuHeader.blockSize),
                          (uint8_t *)(UARTBuffer+4),
                          ReceivedBlockSize);
  SendGoodResponse = true;

  if(fwu_info.blockNumber == ((fwu_info.fwuHeader.fileSize/fwu_info.fwuHeader.blockSize) - 1))
  {
    /* received last block */
    fwu_info.fwuStatus = kFWUStatusIdle;
    EraseSectors(FlashMemFWUInfoAddr, FlashMemFWUInfoSize);
    result = FLASH_Program(&s_flashDriver, FlashMemFWUInfoAddr, (uint8_t *)&fwu_info, sizeof(fwu_info));
  }
  else
  {
    DontCheckEOLCount = 4 + fwu_info.fwuHeader.blockSize; /* BlockNumber + BlockSize */
    if(fwu_info.fwuStatus == kFWUStatusInitiated)
    {
      fwu_info.fwuStatus = kFWUStatusTransfer;
      EraseSectors(FlashMemFWUInfoAddr, FlashMemFWUInfoSize);
      result = FLASH_Program(&s_flashDriver, FlashMemFWUInfoAddr, (uint8_t *)&fwu_info, sizeof(fwu_info));
    }
  }
}
```

Figure 11. Firmware block transfer C code

kFWUStatusActivationInitiated – firmware image activation has been initiated. Image activation is triggered by writing a unique flag code 0xACAC in the iRTC RAM locations. MCU is reset at this moment so that the boot loader program can restart.

iRTC module in the kinetis MCU is powered by VBAT supply and its RAM area is not impacted by MCU warm reset. Any update to iRTC RAM area is write protected by MCU and can be unlocked to write and can be locked again. This RAM area has been used to store a flag code 0xACAC to communicate activation request between application and boot loader.

kFWUStatusActivationSuccess – firmware image activation was successfully done. Actual image activation process is done by the boot loader which once finding the unique flag code 0xACAC post-reset execution, copies the second flash bank resident new firmware to the application partition in the first flash bank. At this moment another MCU reset done by boot loader (after clearing the iRTC RAM flag) enables the boot loader to jump to application entry point which was being replaced with the new firmware application.

kFWUStatusActivationFailed – firmware image activation failed. In case the activation process fails due to any reason, this state is indicated.

WARNING

It is possible that application partition is damaged when the kFWUStatusActivationFailed state is arrived. But the boot loader may still assist upgrading with the new firmware. Such fail-safe or alternative update process is beyond the scope of this document.

Figure 12 Explains the states of the communication process –

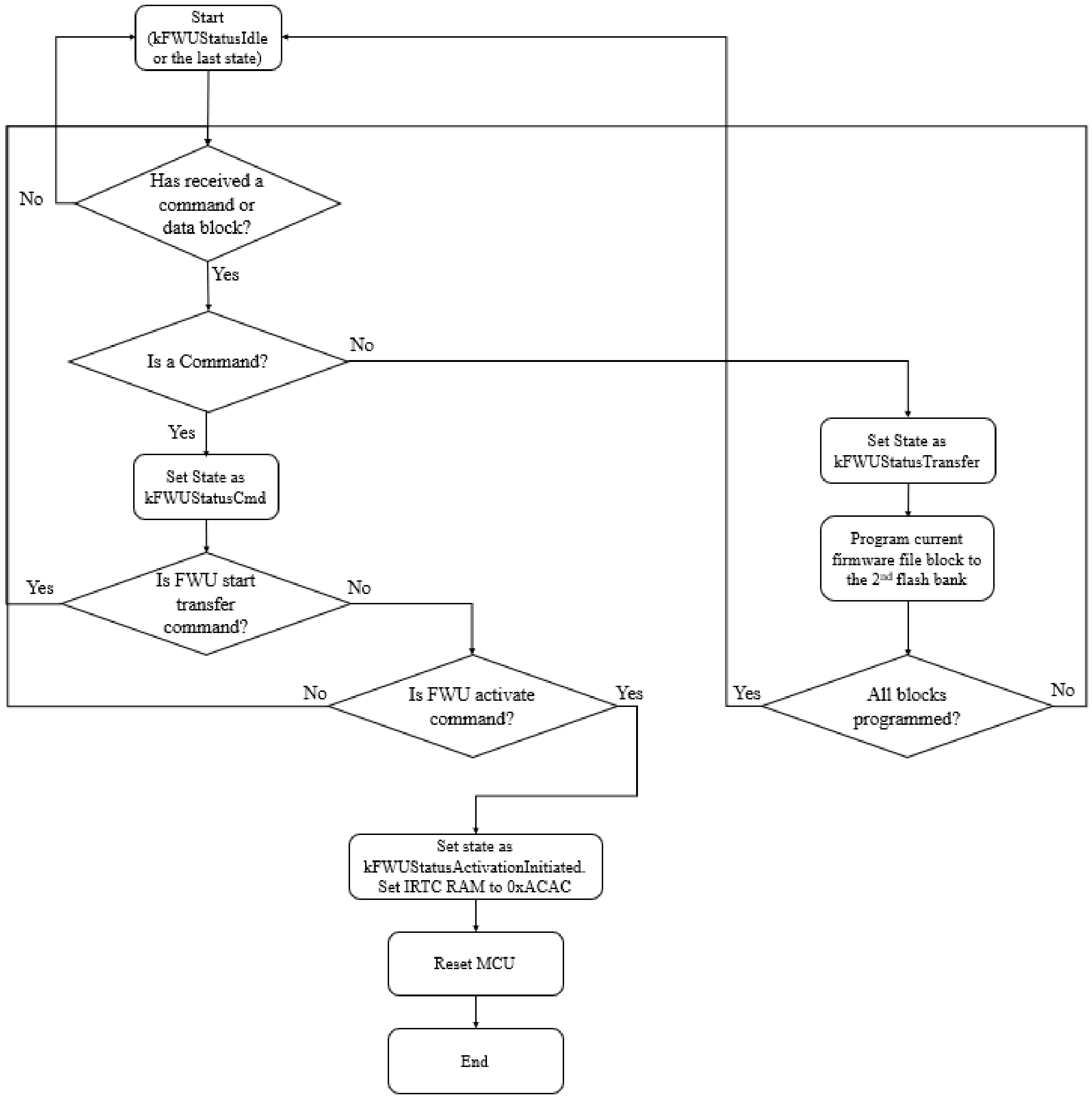

Start (kFWUStatusIdle or the last state)
Has received a command or data block?
No
Yes
Is a Command?
No
Yes
Set State as kFWUStatusCmd
Set State as kFWUStatusTransfer
Program current firmware file block to the 2nd flash bank
Is FWU start transfer command?
Yes
No
Is FWU activate command?
No
Yes
All blocks programmed?
Yes
No
Set state as kFWUStatusActivationInitiated. Set IRTC RAM to 0xACAC
Reset MCU
End

Figure 12. Commutation flow control

5.1.2.4 Firmware save and activation process

Firmware saving and activation process has been described in this document and can be visualized as shown in Figure 13 –

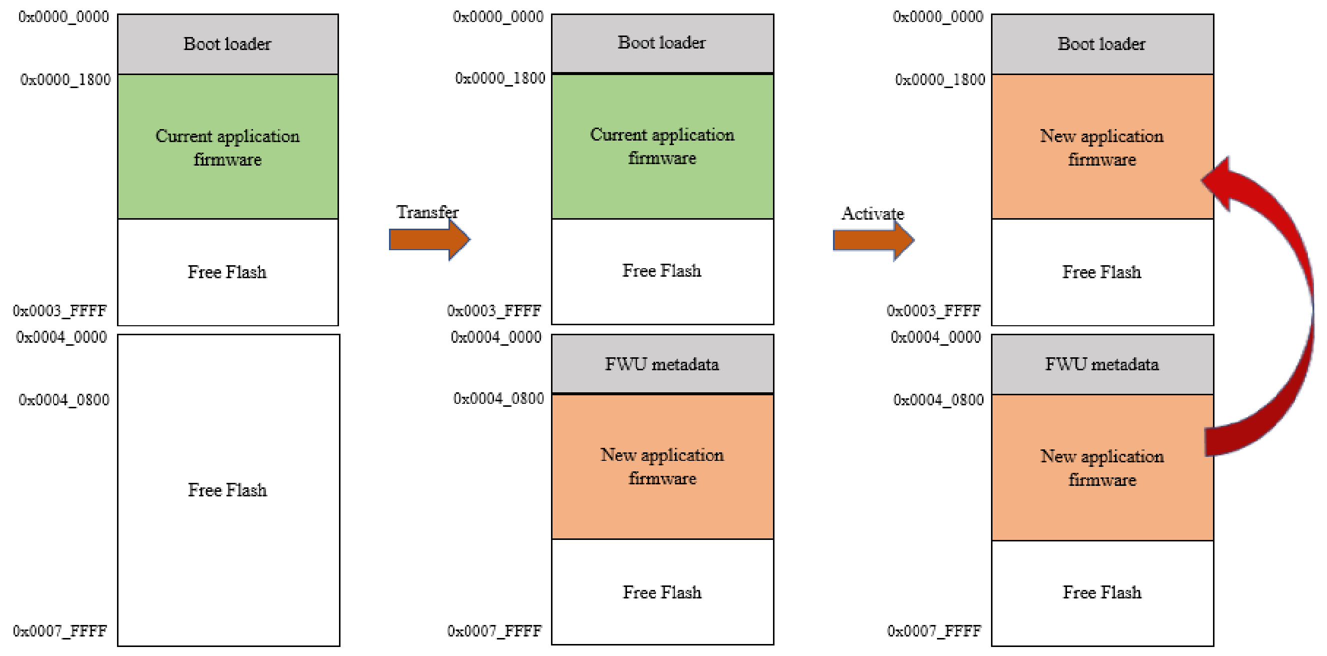


Figure 13. Firmware saving and activation process

5.2 Application setup

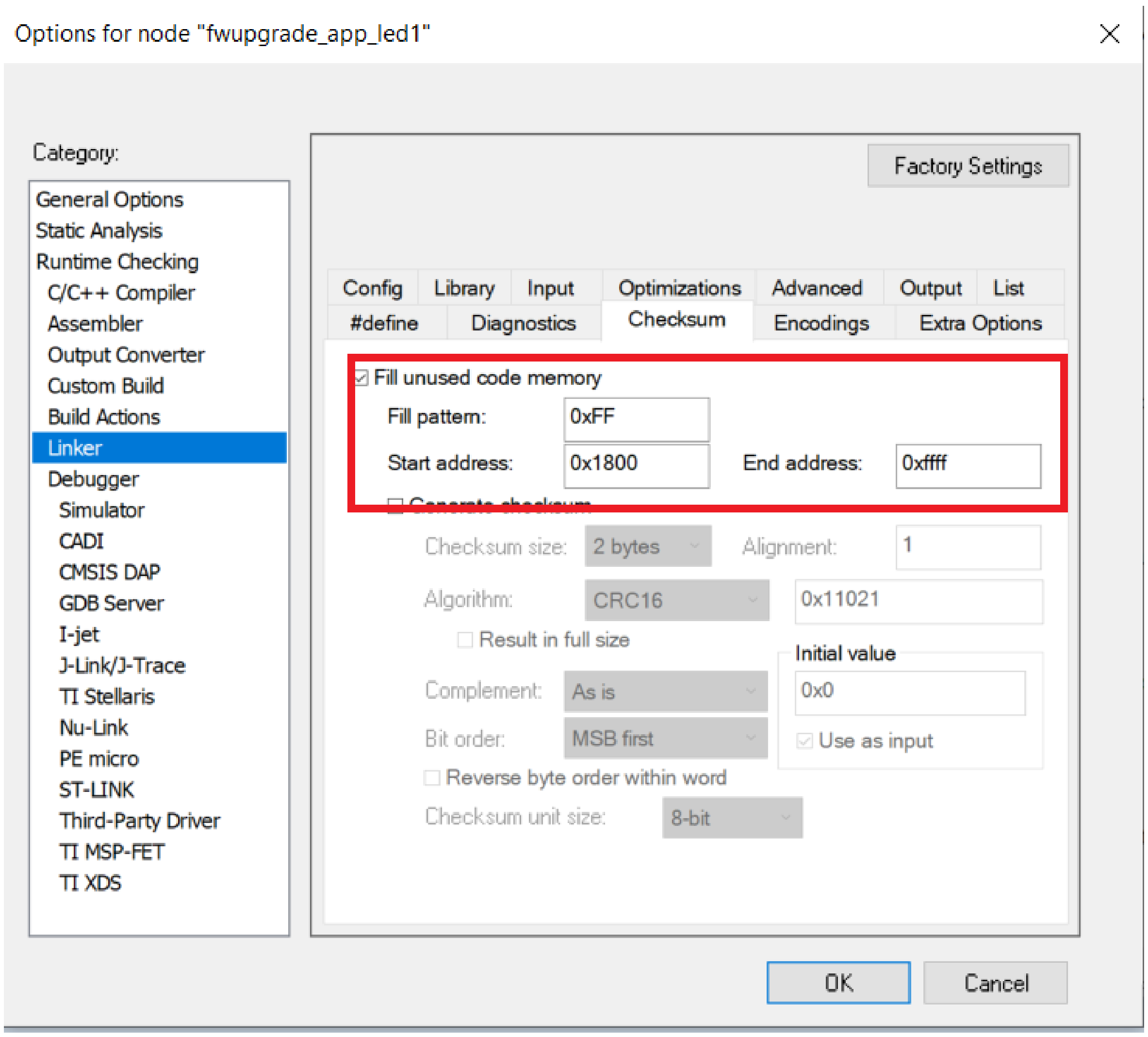
Application setup is done in 2 steps – first prepare the firmware binary to be upgraded and second, setup the TWR-KM35Z7M to exercise the upgrade process. To facilitate the transfer, and activate the new firmware, a personal computer with Windows OS is used.

5.2.1 Preparation of firmware binary

New application firmware is created by aligning the generated binary size to an integral multiple of block size, for example, 256 bytes. In the current case, 2 application binaries are prepared with application binary file sizes of 231 blocks x 256 bytes = 58 Kbytes. Although the actual application sizes are lesser than this, application binary file size is increased and adjusted by using linker option of the IAR IDE as shown below.

NOTE

The image size need not to be multiple of 256 bytes in real use cases but has been done here for simplification only.



Options for node "fwupgrade_app_led1"

Category: Linker

Checksum

☑ Fill unused code memory

Fill pattern: 0xFF

Start address: 0x1800

End address: 0xffff

☐ Generate checksum

Checksum size: 2 bytes

Alignment: 1

Algorithm: CRC16

0x11021

☐ Result in full size

Initial value: 0x0

Complement: As is

Bit order: MSB first

☑ Use as input

☐ Reverse byte order within word

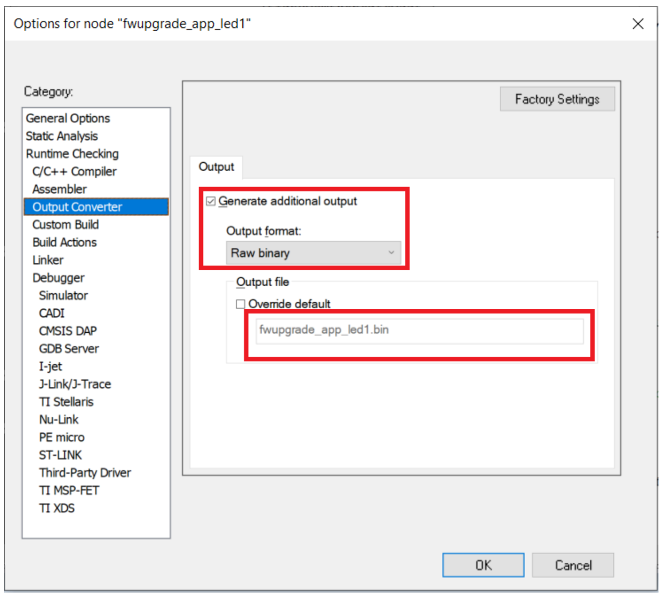
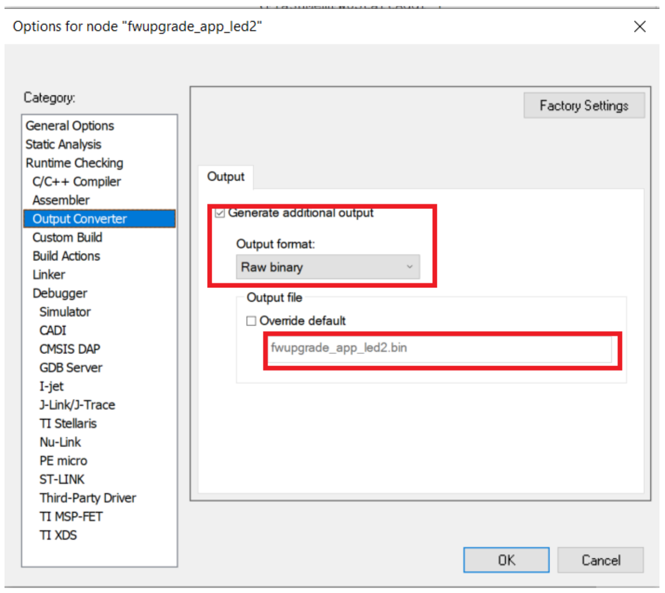
Checksum unit size: 8-bit

Figure 14. IAR project linker file option update

Both **fwupgrade_app_led1** and **fwupgrade_app_led2** are adjusted to create image binary of size 0xFFFF – 0x1800 = 58 Kbytes.

To generate a binary file of the application firmware, **output converter** in IAR tool project options can be utilized, as shown below –

Table 1. IAR project option to create .bin file

Options for node "fwupgrade_app_led1"

Category: General Options, Static Analysis, Runtime Checking, C/C++ Compiler, Assembler, Output Converter, Custom Build, Build Actions, Linker, Debugger, Simulator, CADI, CMSIS DAP, GDB Server, I-jet, J-Link/J-Trace, TI Stellaris, Nu-Link, PE micro, ST-LINK, Third-Party Driver, TI MSP-FET, TI XDS

Output

☑ Generate additional output

Output format: Raw binary

Output file

☐ Override default

fwupgrade_app_led1.bin

Options for node "fwupgrade_app_led2"

Category: General Options, Static Analysis, Runtime Checking, C/C++ Compiler, Assembler, Output Converter, Custom Build, Build Actions, Linker, Debugger, Simulator, CADI, CMSIS DAP, GDB Server, I-jet, J-Link/J-Trace, TI Stellaris, Nu-Link, PE micro, ST-LINK, Third-Party Driver, TI MSP-FET, TI XDS

Output

☑ Generate additional output

Output format: Raw binary

Output file

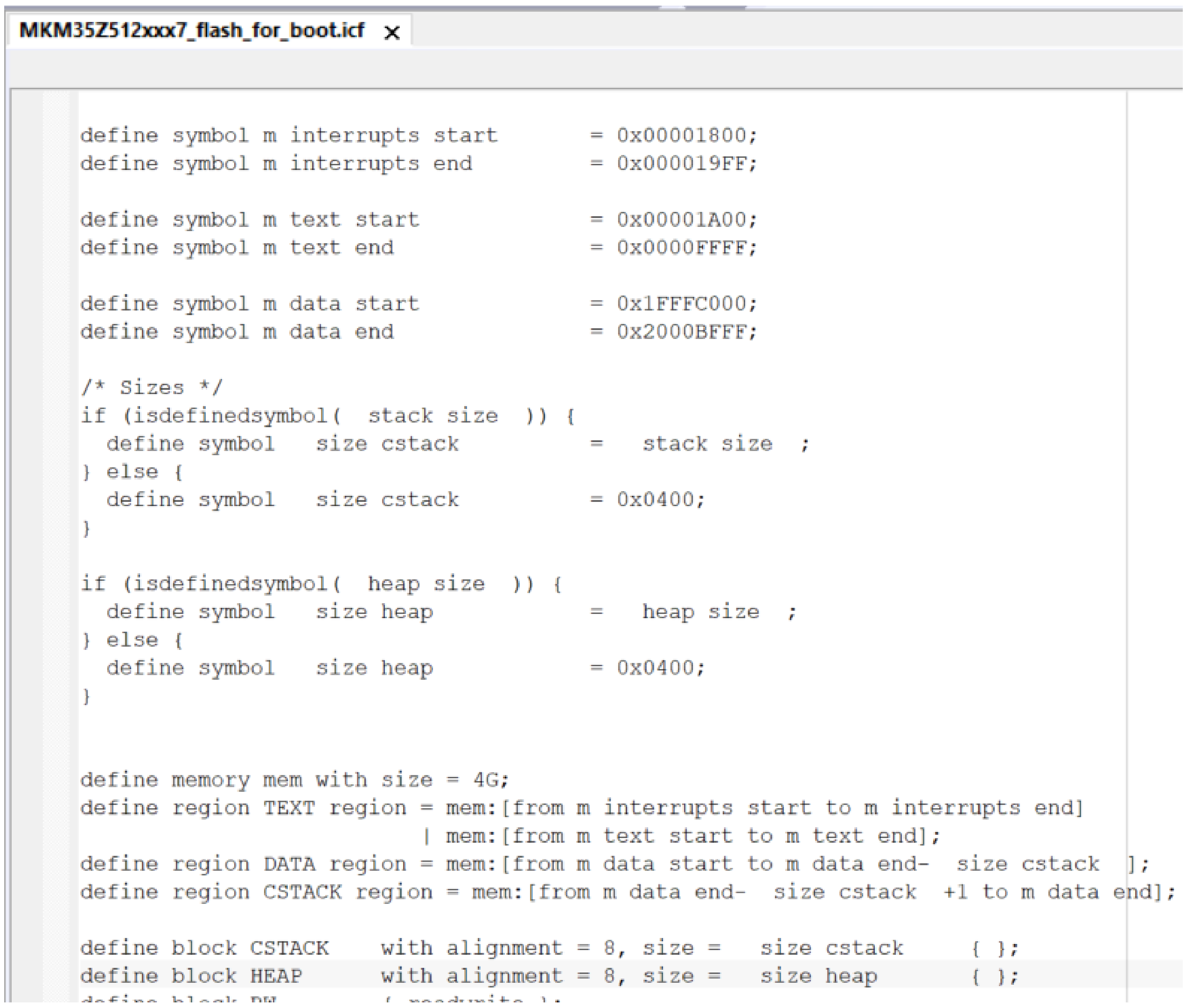
☐ Override default

fwupgrade_app_led2.bin

5.2.1.1 Application LED program linker file setup

Application **fwupgrade_app_led1** and **fwupgrade_app_led2** linker files limited the program code and initialized data sections between 0x1800 to 0xFFFF = 58 Kbytes only. This leaves the free space between 0x0001_0000 to 0x0003_FFFF, which can be claimed when the application size is increased to include more functionalities. Generated binary file size can be adjusted as mentioned in this document for block size aligned binary file size again.

Below is the linker file of the application, the starting address is at 0x0000_01800 as the partition starts at this address.



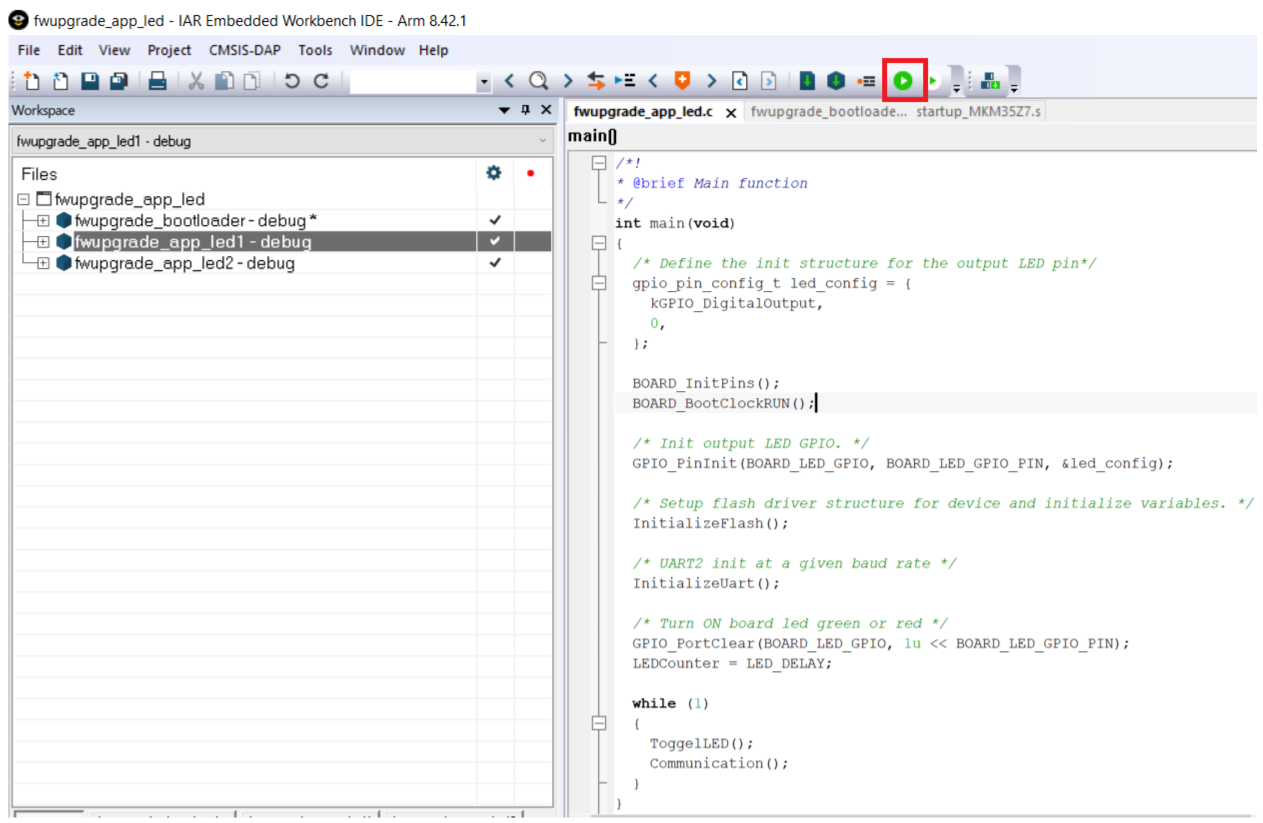
MKM35Z512xxx7_flash_for_boot.icf

```
define symbol m interrupts start        = 0x00001800;
define symbol m interrupts end          = 0x000019FF;

define symbol m text start              = 0x00001A00;
define symbol m text end                = 0x0000FFFF;

define symbol m data start              = 0x1FFFC000;
define symbol m data end                = 0x2000BFFF;

/* 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;
}


define memory mem with size = 4G;
define region TEXT region = mem:[from m interrupts start to m interrupts end]
                          | mem:[from m text start to m text end];
define region DATA region = mem:[from m data start to m data end-  size cstack  ];
define region CSTACK region = mem:[from m data end-  size cstack  +1 to m data end];

define block CSTACK    with alignment = 8, size =   size cstack     { };
define block HEAP      with alignment = 8, size =   size heap       { };
```

Figure 15. IAR project linker file for application

5.2.2 Execution of Firmware upgrade process

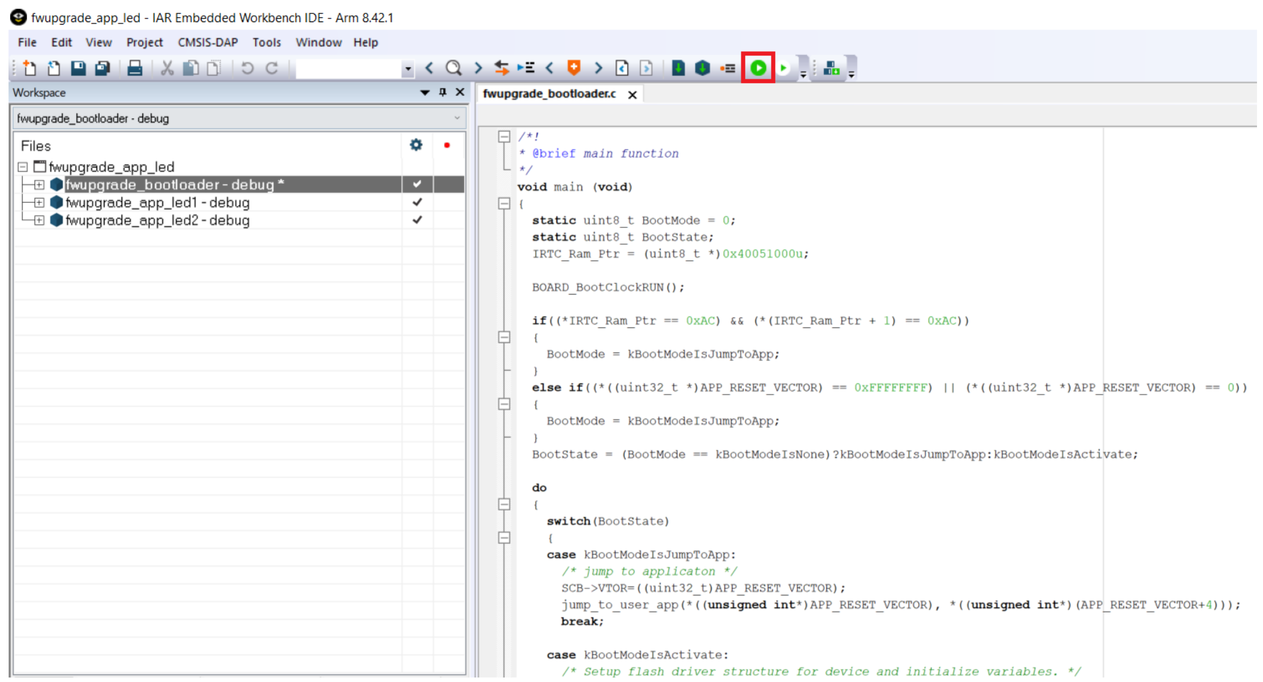
1. Build and download **fwupgrade_app_led1** to the TWR-KM35Z7M -


fwupgrade_app_led - IAR Embedded Workbench IDE - Arm 8.42.1
File Edit View Project CMSIS-DAP Tools Window Help
Workspace
fwupgrade_app_led1 - debug
Files
fwupgrade_app_led
fwupgrade_bootloader - debug *
fwupgrade_app_led1 - debug
fwupgrade_app_led2 - debug
fwupgrade_app_led.c
fwupgrade_bootloade...
startup_MKM35Z7.s
main()

```
/*!
 * @brief Main function
 */
int main(void)
{
  /* Define the init structure for the output LED pin*/
  gpio_pin_config_t led_config = {
    kGPIO_DigitalOutput,
    0,
  };

  BOARD_InitPins();
  BOARD_BootClockRUN();

  /* Init output LED GPIO. */
  GPIO_PinInit(BOARD_LED_GPIO, BOARD_LED_GPIO_PIN, &led_config);

  /* Setup flash driver structure for device and initialize variables. */
  InitializeFlash();

  /* UART2 init at a given baud rate */
  InitializeUart();

  /* Turn ON board led green or red */
  GPIO_PortClear(BOARD_LED_GPIO, 1u << BOARD_LED_GPIO_PIN);
  LEDCounter = LED_DELAY;

  while (1)
  {
    ToggleLED();
    Communication();
  }
}
```

Figure 16. IAR Embedded work bench build and download option for application

2. Build and download **fwupgrade_bootloader** program to the tower board –


fwupgrade_app_led - IAR Embedded Workbench IDE - Arm 8.42.1
File Edit View Project CMSIS-DAP Tools Window Help
Workspace
fwupgrade_bootloader - debug
Files
fwupgrade_app_led
fwupgrade_bootloader - debug *
fwupgrade_app_led1 - debug
fwupgrade_app_led2 - debug
fwupgrade_bootloader.c

```
/*!
 * @brief main function
 */
void main (void)
{
  static uint8_t BootMode = 0;
  static uint8_t BootState;
  IRTC_Ram_Ptr = (uint8_t *)0x40051000u;

  BOARD_BootClockRUN();

  if((*IRTC_Ram_Ptr == 0xAC) && (*(IRTC_Ram_Ptr + 1) == 0xAC))
  {
    BootMode = kBootModeIsJumpToApp;
  }
  else if((*((uint32_t *)APP_RESET_VECTOR) == 0xFFFFFFFF) || (*((uint32_t *)APP_RESET_VECTOR) == 0))
  {
    BootMode = kBootModeIsJumpToApp;
  }
  BootState = (BootMode == kBootModeIsNone)?kBootModeIsJumpToApp:kBootModeIsActivate;

  do
  {
    switch(BootState)
    {
    case kBootModeIsJumpToApp:
      /* jump to applicaton */
      SCB->VTOR=((uint32_t)APP_RESET_VECTOR);
      jump_to_user_app(*((unsigned int*)APP_RESET_VECTOR), *((unsigned int*)(APP_RESET_VECTOR+4)));
      break;

    case kBootModeIsActivate:
      /* Setup flash driver structure for device and initialize variables. */
```

Figure 17. IAR Embedded work bench build and download for boot loader

3. Reset TWR-KM35Z357M by pressing SRESET button of the board. The board’s green LED starts blinking.

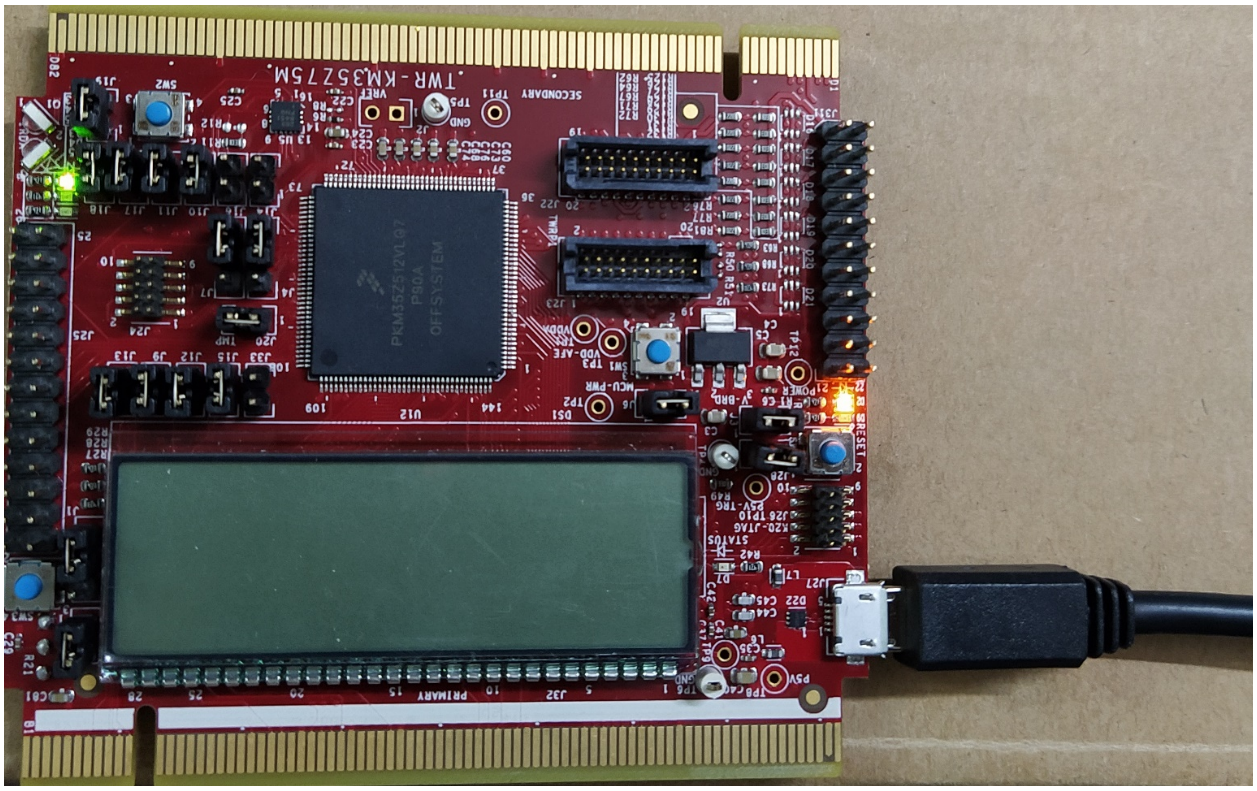
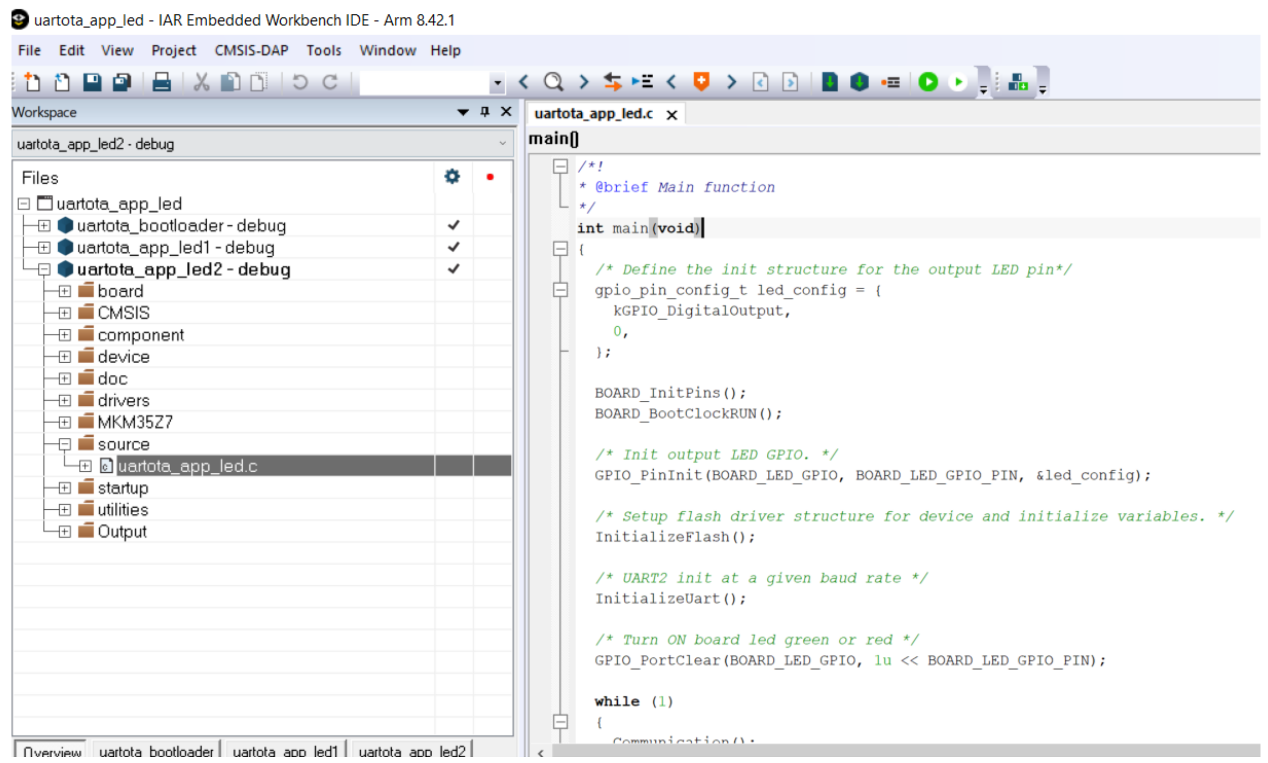


Figure 18. Blinking green LED program execution

4. Build **fwupgrade_app_led2** project and copy its generated **fwupgrade_app_led2.bin** file to the same subfolder where your PC host utility is available. In our case, a PC host utility called **FWUpgradeClient.exe** was created to facilitate the .bin file transfer.



```c
/*!
 * @brief Main function
 */
int main(void)
{
  /* Define the init structure for the output LED pin*/
  gpio_pin_config_t led_config = {
    kGPIO_DigitalOutput,
    0,
  };

  BOARD_InitPins();
  BOARD_BootClockRUN();

  /* Init output LED GPIO. */
  GPIO_PinInit(BOARD_LED_GPIO, BOARD_LED_GPIO_PIN, &led_config);

  /* Setup flash driver structure for device and initialize variables. */
  InitializeFlash();

  /* UART2 init at a given baud rate */
  InitializeUart();

  /* Turn ON board led green or red */
  GPIO_PortClear(BOARD_LED_GPIO, 1u << BOARD_LED_GPIO_PIN);

  while (1)
  {
```

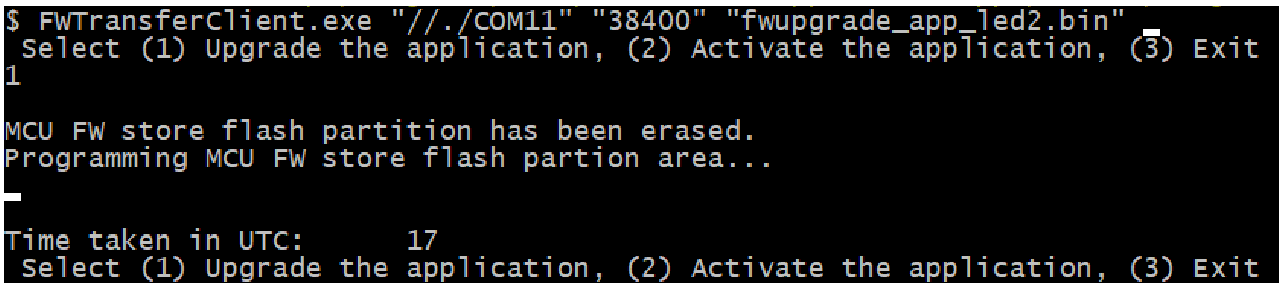
Figure 19. Preparation of blinking red LED program binary

NOTE

Do not download fwupgrade_app_led2 program as the TWR-KM35Z7M is already programmed with fwupgrade_app_led1, the green LED program.

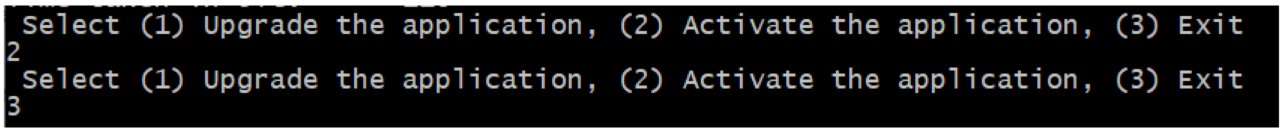
5. The generated **fwupgrade_app_led2.bin** file can be located in the build type specific that is, either \debug or \release sub folder.

   Start transferring fwupgrade_app_led2.bin with the PC host utility as shown below –



```
$ FWTransferClient.exe "//./COM11" "38400" "fwupgrade_app_led2.bin"
 Select (1) Upgrade the application, (2) Activate the application, (3) Exit
1

MCU FW store flash partition has been erased.
Programming MCU FW store flash partion area...


Time taken in UTC:      17
 Select (1) Upgrade the application, (2) Activate the application, (3) Exit
```

   After the file transfer is over, activate the image copy by as shown below:



```
 Select (1) Upgrade the application, (2) Activate the application, (3) Exit
2
 Select (1) Upgrade the application, (2) Activate the application, (3) Exit
3
```

   At this point, the red LED blinks and green LED is turned off, as shown in Figure 20.

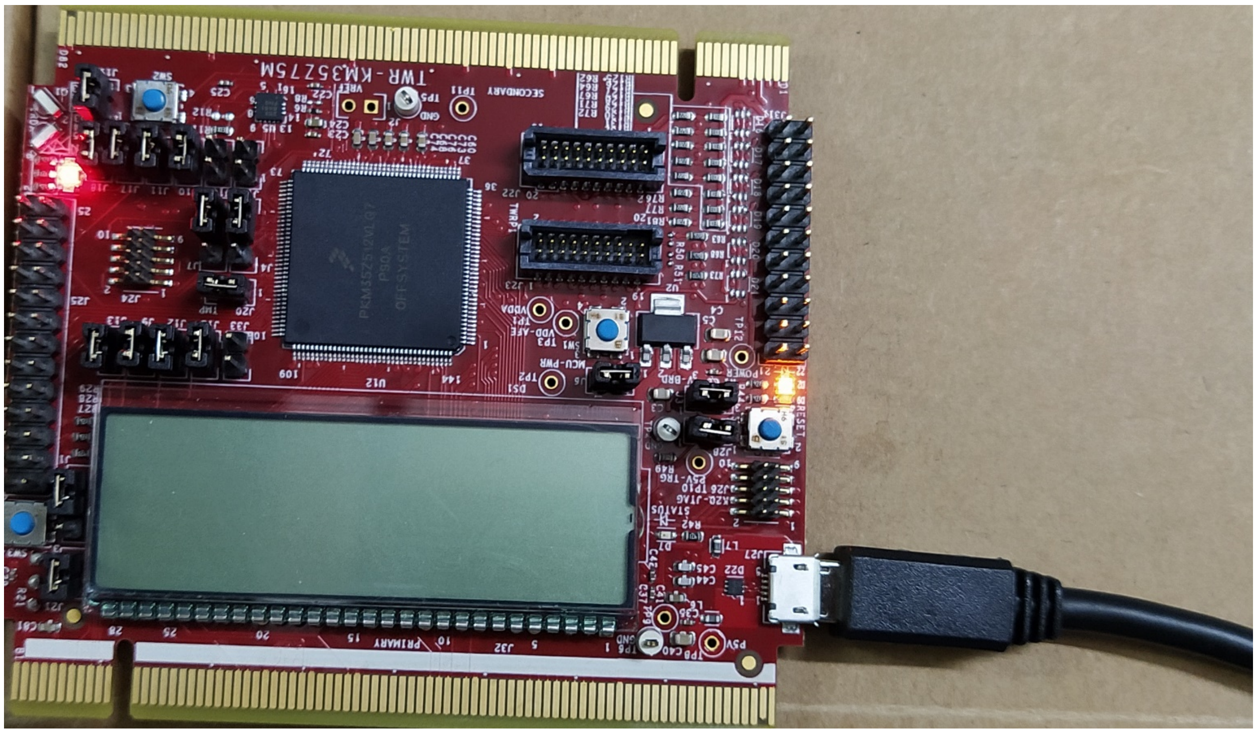
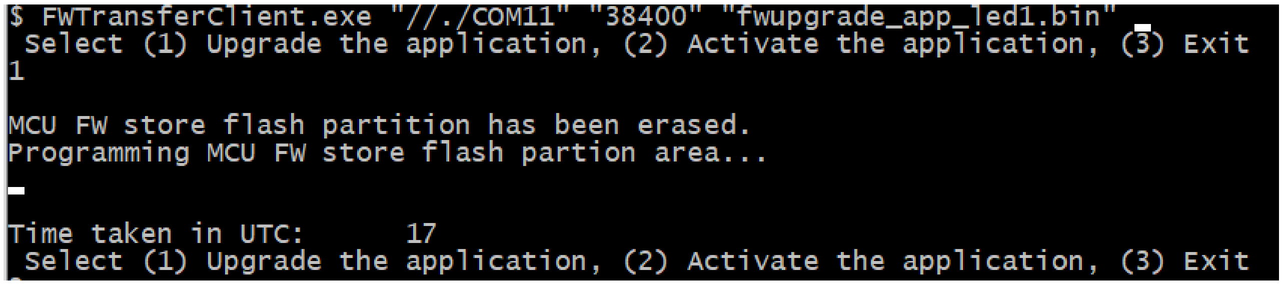


Figure 20. Blinking red LED program execution after firmware update activation

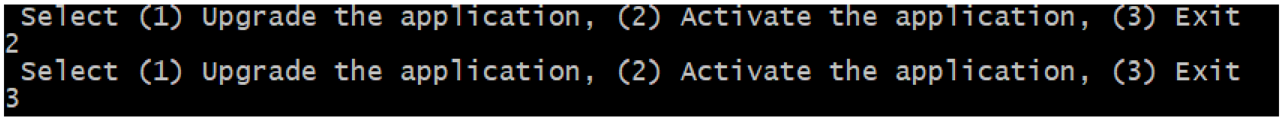
At this moment the red LED application program built by fwupgrade_app_led2 is running in the TWR-KM35z7M board.

New firmware activation is successful.

Similarly, **fwupgrade_app_led1.bin**, the green LED program can also be located in \debug or \release subfolders and the firmware update process can be done. The process is a repeat of the above and find below step by step –



```
$ FWTransferClient.exe "//./COM11" "38400" "fwupgrade_app_led1.bin"
 Select (1) Upgrade the application, (2) Activate the application, (3) Exit
1

MCU FW store flash partition has been erased.
Programming MCU FW store flash partion area...

Time taken in UTC:       17
 Select (1) Upgrade the application, (2) Activate the application, (3) Exit
```



```
 Select (1) Upgrade the application, (2) Activate the application, (3) Exit
2
 Select (1) Upgrade the application, (2) Activate the application, (3) Exit
3
```

6 Summary

This application note describes a process to upgrade application firmware in Kinetis KM35Z512 MCU. KM35Z512 has 512 KB of total flash memory which is divided in two 256 KB program flash banks. RWW feature is available which means, program read can happen in one bank while erase/write or programming can be done in the second bank simultaneously. New firmware is received from a PC host machine using PC-to-MCU serial port communication and stored in the second flash bank of KM35Z512. The application running in the first bank of the MCU does this image transfer and storing job. The implementation also takes help of a bootloader to activate the stored image by copying the new firmware image from the second bank to the first bank, thereby replacing only the application present in the first bank application area.

The implementation is based on TWR-KM35Z7M tower board at default configuration as described in the quick-start-guide of this tower board. A PC with Windows 10 OS is used to build the program code using IAR embedded workbench version 8.42. Both the application and bootloader are downloaded to the tower board, using the OpenSDA debug probe supported on tower board. This prepares the tower board to be ready for the firmware transfer process.

New application transfer to the tower board is facilitated by the PC by transferring new firmware binary in fixed size blocks, through the serial port connection (OpenSDA USB). Once the whole firmware binary is transferred, PC host resident program can trigger an activation command through the serial port to the tower board. The current running application along with the bootloader makes sure to copy and update the first bank application program with the new program which was stored in the second flash bank. From this point, the old application firmware is no longer available in the MCU and completely being replaced with the new firmware.

7 References

1. Quick Start Guide TWR-KM35Z37M

8 Revision history

Revision number	Date	Substantive changes
0	04/2020	Initial release

Date of release: 25 April 2020
Document identifier: AN12829