USB Mass Storage Device Host Bootloader for K22 and K64 MCUs

by: Marco Aurelio P. Coelho
Field Applications Engineer
Siletec Eletronica

Contents:

1. Bootloader Overview

Bootloader is a system that allows the user to program the application firmware into the MCU Flash memory through several interfaces: UART, SPI, IIC, Ethernet, CAN, USB, etc.

In the bootloader presented here, the MCU will act as a USB Mass Storage Host that receives the application firmware binary file from a USB stick, in blocks of 1024 bytes, stores it in a RAM buffer, parses the data, and finally, writes it into the Flash memory area destined to the application.

Obviously, the MCU must have a USB host interface. NXP offers several MCU's that have such interfaces. I developed two versions of this bootloader: one for FRDM-K22F (MCU MK22FN512VLH12 mounted on) and one for FRDM-K64F (MCU MK64FN1M0VLL12 mounted on).

The figures given below shows the memory maps of the MK22FN512VLH12 and MK64FN1M0VLL12 bootloader systems.

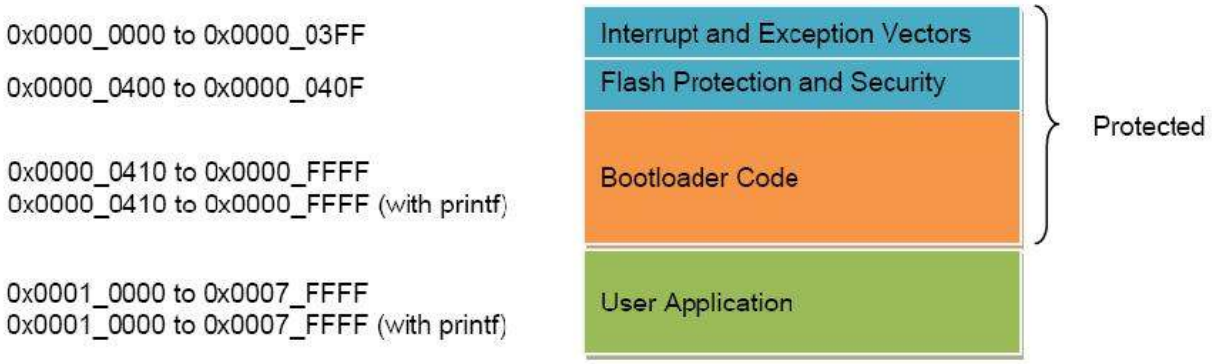


Figure 1. MK22FN512VLH12 bootloader memory map

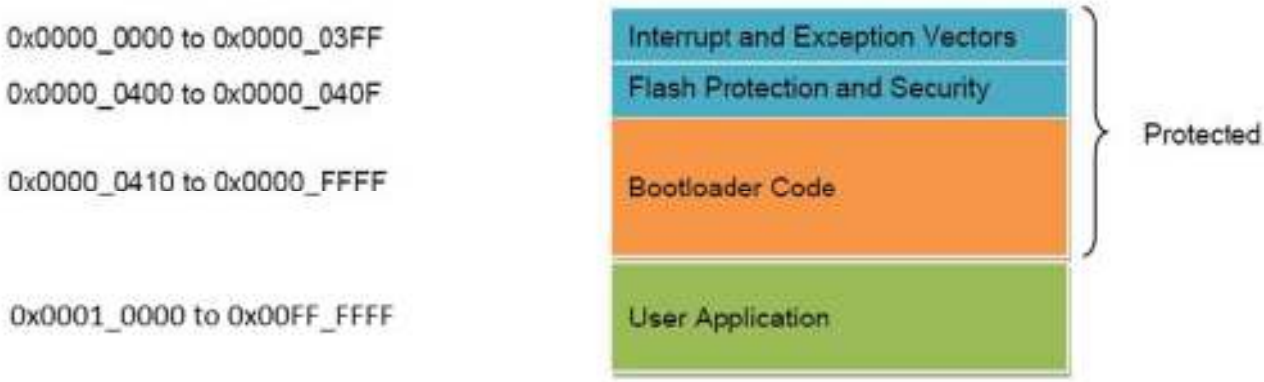


Figure 2. MK64FN1M0VLL12 bootloader memory map

The default interrupt and exception vectors are put into the starting address of the flash area and are used by the bootloader, which must not be altered. The user application interrupt and exception vectors must be put and redirected to the application flash area.

The bootloader erases and programs the Application Flash, which is the free Flash memory after Flash memory area reserved to bootloader. The bootloader Flash area has to be protected and the starting address of the Application Flash memory must be block aligned.

For MK22FN512VLH12, the bootloader occupies 64,972 bytes of Flash. Since the Flash protectable block size is 16 KB, the flash memory region of 0x0000 to 0xFFFF (65,536 bytes or 4 blocks of 16384 bytes) must be protected to prevent damage to the bootloader. After protection, the bootloader occupies 65,536 bytes. The rest of the flash memory from 0x10000 to 0x7FFFF (458,752 bytes) is available for user application.

For MK64FN1M0VLL12, the bootloader occupies 62,532 bytes of Flash, and, in the same way, 65,536 bytes (2 blocks of 32KB), after the block protection. Flash area from 0x10000 to 0xFFFFF is available for user application.

If all printf messages are removed from the code, the consumption of memory is drastically reduced and leds, for example, can be used to show the status of the bootloader, instead of messages.

The user application can use the whole RAM memory regardless of how much RAM the bootloader uses.

Note: Both demo codes have HUB class disabled to save Flash memory and thus fit into 64KB. If you enable HUB class for your application, the code will surpass 64KB and you must consider losing another block for proper bootloader code protection: one more 16KB block, in case of MK22FN512; and one more 32KB block, in case of MK64FN1M0

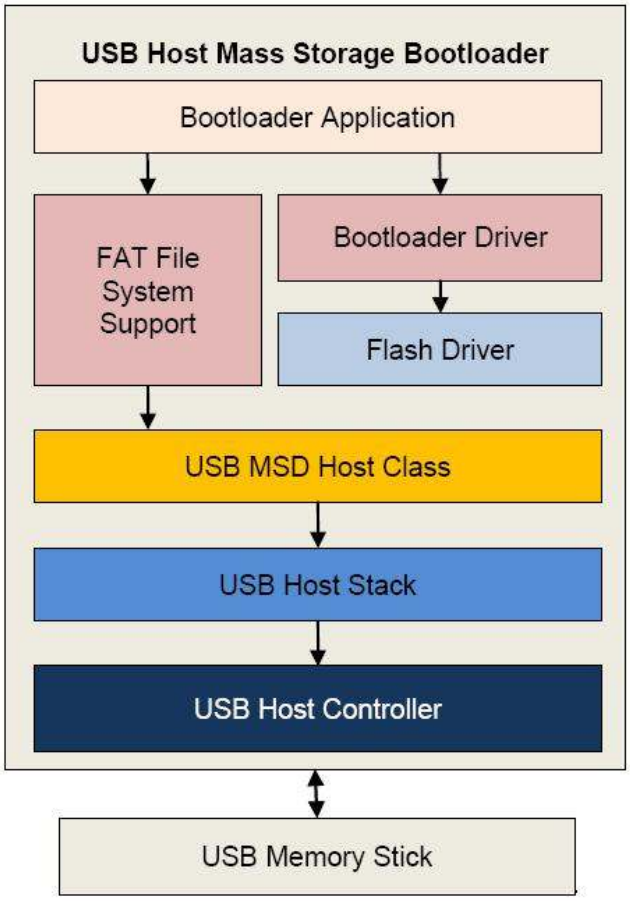
2. Bootloader Architecture

The bootloader includes a bootloader application, a file allocation table (FAT) file system-supporting module, a bootloader driver, a flash driver, a USB MSD host class, a USB host stack, and a USB host controller.

All the firmware was developed in MCUXpresso with SDK v. 2.2 drivers and FreeRTOS. Most of code was taken and reused from SDK's "usb_host_msd_fatfs_freertos" demo code and the application was adapted from AN4368 USB MSD Host bootloader code for TWR-K70, also posted on Community (https://community.nxp.com/docs/DOC-102616). So make sure that you have downloaded and installed MK22FN512VLH12 and MK64FN1M0VLL12 SDKs into your workspace in MCUXpresso. Here is a link of a video explaining how to download and install SDK 2.2 package: https://community.nxp.com/videos/7489.

Note: Before installation, make sure that you have selected MCUXpresso as your IDE and FreeRTOS among the middleware options, in SDK Builder.

The following figure shows the architecture of the bootloader system:



- The bootloader controls the loading process. It uses the SDK USB host stack and the USB host controller to communicate with the USB memory stick, which holds the application's raw binary file, through the USB MSD protocols.
- Through the SDK FAT-32 file system, it reads the image file from the USB memory stick.
- Then, the bootloader erases application Flash area and programs the contents of the image file to it, through the SDK Flash driver.

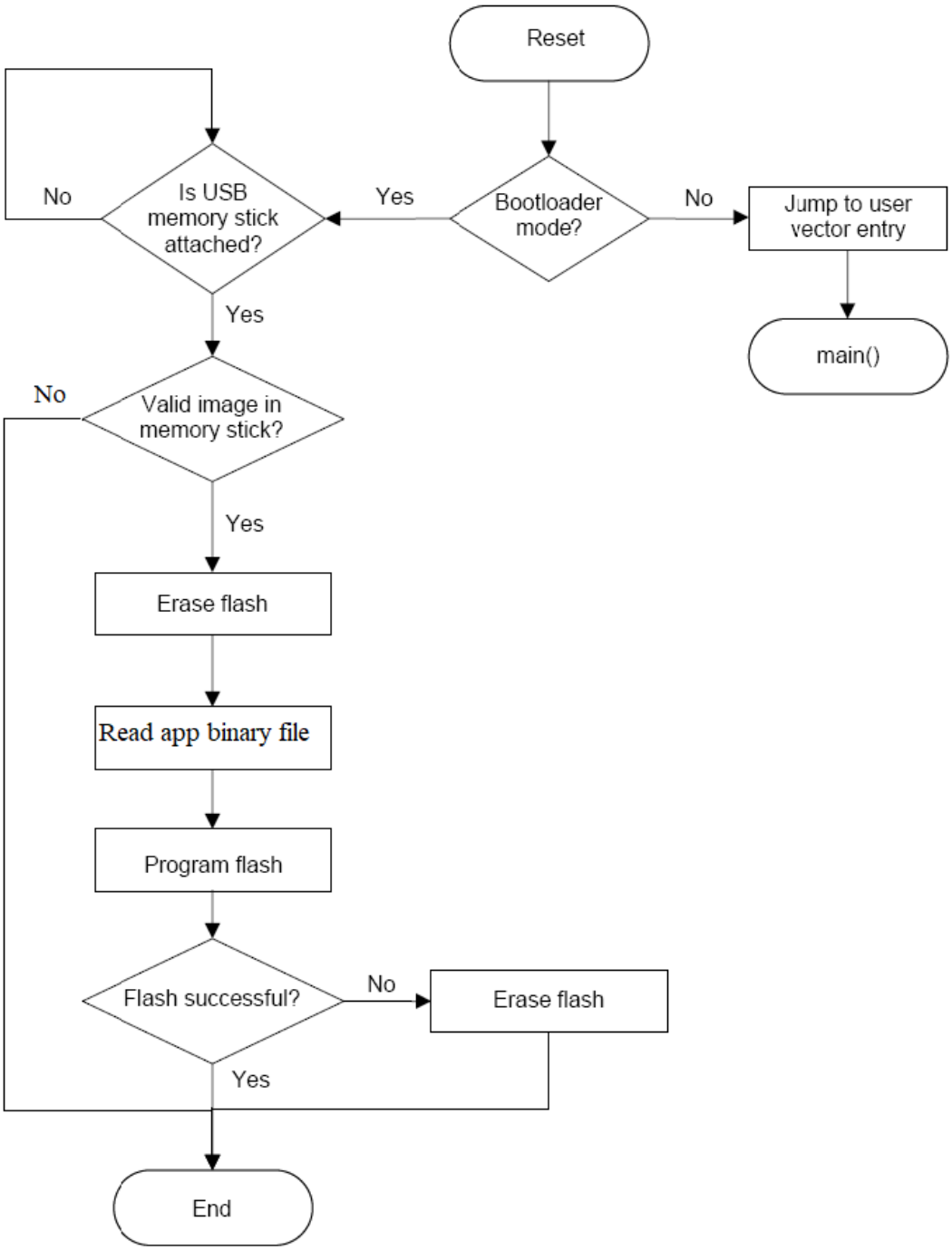
Bootloader software flow:

The Bootloader system is comprised by a bootloader for user program upgrade, and a user application performing the main function of a product. After reset and initialization, the system determines if starts either the user application program or the bootloader. If there is no valid user application program, the device will automatically run the bootloader. If there is a valid application, the device will run the bootloader program on pressing a specific key, otherwise, it will run the user application. The following table shows the bootloader examples designed for different development boards along with the specified keys used in the examples.

Development board	Specified key
FRDM-K22F	PTB17 (SW3)
FRDM-K64F	PTC6 (SW2)

Table 1. Specified keys for entering bootloader mode

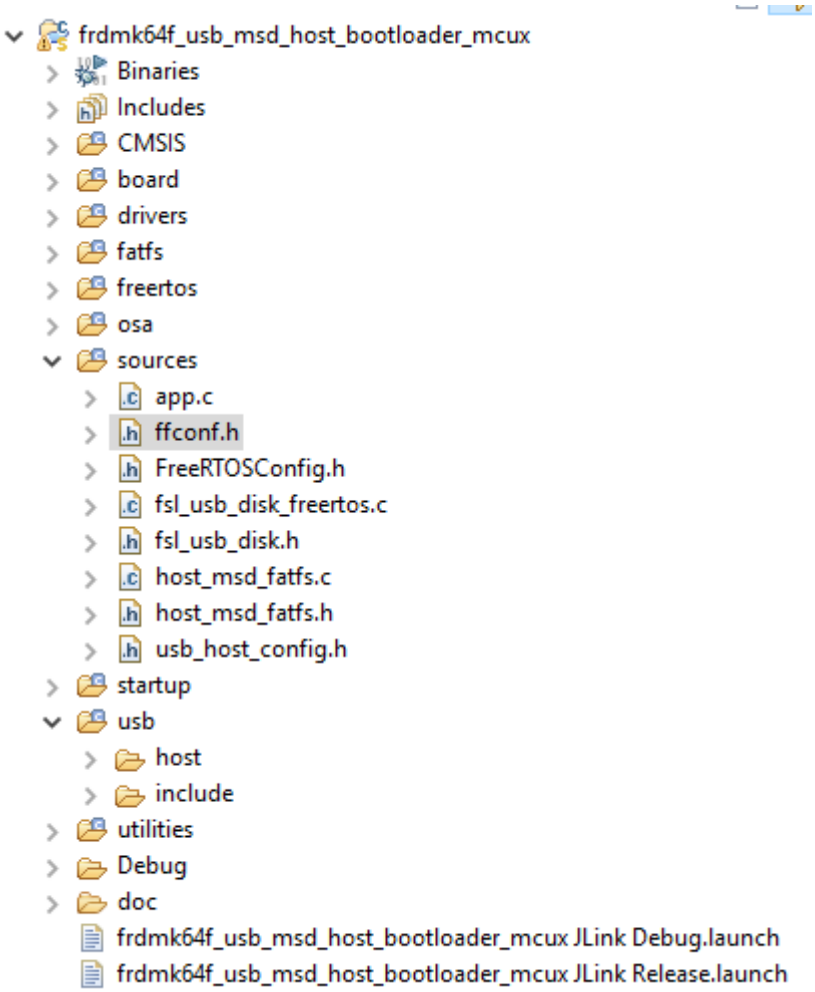
Once the system has entered the bootloader mode, it keeps on checking whether a USB memory stick is attached or not. If a USB memory stick is attached, it will search for the “image.bin” file. If a valid raw binary file exists, it programs it to the application region. The following is the flow chart of the bootloader:



Bootloader software flow

3. Bootloader file structure

The following figure shows the file structure of the “frdm-k64f_usb_msd_host_bootloader_mcux” demo code:



Bootloader file structure

Let’s highlight some of those folders and files:

- ***board***: Contains clock system, pin mux and UART SDK initialization code.
- ***drivers***: Contains all the SDK peripheral (Flash, GPIO, UART, clock, etc) drivers with initialization, configuration and handling functions, used in the project.
- ***fatfs***: Contains the FAT-32 File System functions for file system mounting and file opening, reading and closing.
- ***freertos***: Contains FreeRTOS libraries and files
- ***sources***: Contains application source code:

- ***app.c:*** Contains main loop, where the MCU and USB Host initialization functions are called, determines if it enters the bootloader mode, processes USB Host events, creates USB_HostTask and USB_HostApplicationTask and starts FreeRTOS:
    - ***USB_HostTask:*** Calls USB_HostKhciTaskFunction, which handles KHCI controller messages
    - ***USB_HostApplicationTask*****:** Calls USB_HostMsdTask function, which handles USB Host MSD events and runs bootloader application code, as the USB memory stick is detected
- ***ffconf.h:*** FAT-32 File System configuration
- ***FreeRTOSConfig.h:*** FreeRTOS configuration
- ***fsl_usb_disk_freertos.c:*** Contains FATFs Lower Layer API
- ***host_msd_fatfs.c:*** Contains the USB_HostKhciTaskFunction and USB_HostMsdTask functions, described above, as well as the bootloader application code itself
- ***usb_host_config.h:*** USB Host user configuration, including macros to include or exclude features as USBCLASS_INC_MASS (to support Mass Storage Device) and USBCLASS_INC_HUB (to support Hub)

- **usb (folder):** Contains initialization, settings and definitions of USB, USB Host Controller and USB Host MSD

4. Preparing user applications developed in MCUXpresso for the bootloader system

For normal applications using Kinetis MCUs, the interrupt vector table is located at the beginning of the Flash area and the application code can be put in any of the remaining Flash areas. In a bootloader system, the interrupt vector table and the bootloader program are put into the beginning of Flash, and the user application is placed at the remaining Flash areas. The application's linker file must be modified to put the application interrupt vector table and the application code into the remaining Flash area.

This section describes how normal applications developed in MCUXpresso can be modified for the bootloader system. To see how to prepare user applications developed in Kinetis Design Studio for the bootloader system, please skip to Section 5.

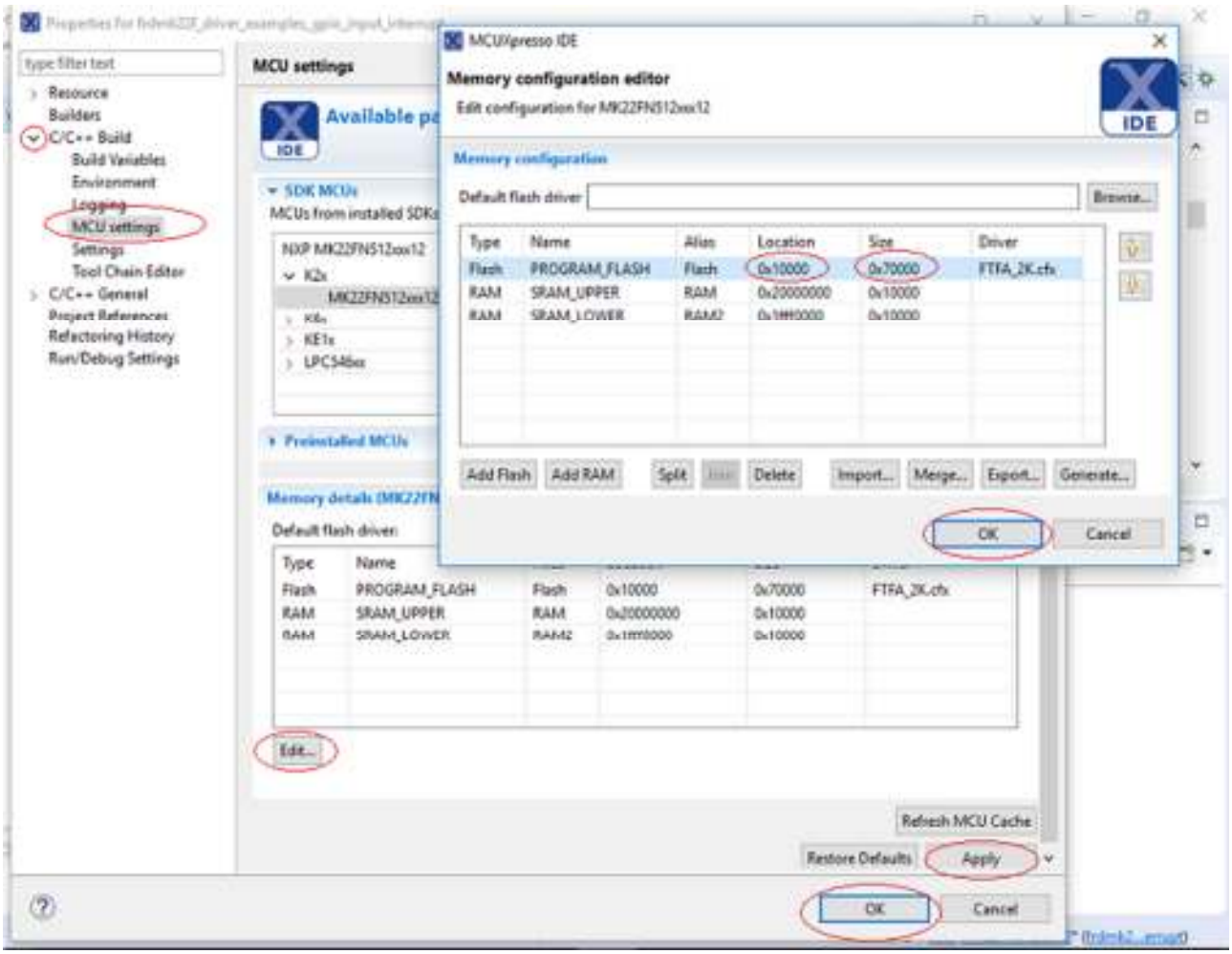
4.1. Modify linker files for Kinetis in MCUXpresso

The linker file defines what region of Flash memory (including interrupt vector table and Flash Configuration Field) the firmware will be placed at. Let's take a look, for example, at the memory definitions in a typical application with MK22FN512VLH12:

```
MEMORY
{
  /* Define each memory region */
  PROGRAM_FLASH (rx): ORIGIN = 0x0, LENGTH = 0x80000 /* 512K bytes (alias Flash) */
  SRAM_UPPER (rwx): ORIGIN = 0x20000000, LENGTH = 0x30000 /* 192K bytes (alias RAM) */
  SRAM_LOWER (rwx): ORIGIN = 0x1fff0000, LENGTH = 0x10000 /* 64K bytes (alias RAM2) */
  FLEX_RAM (rwx): ORIGIN = 0x14000000, LENGTH = 0x1000 /* 4K bytes (alias RAM3) */
}
```

To work with the bootloader system for FRDM-K22F or FRDM-K64F, the user application must be placed at the Flash memory area starting from the address 0x10000. The linker file can be modified, in MCUXpresso, by using its Memory Configuration Editor:

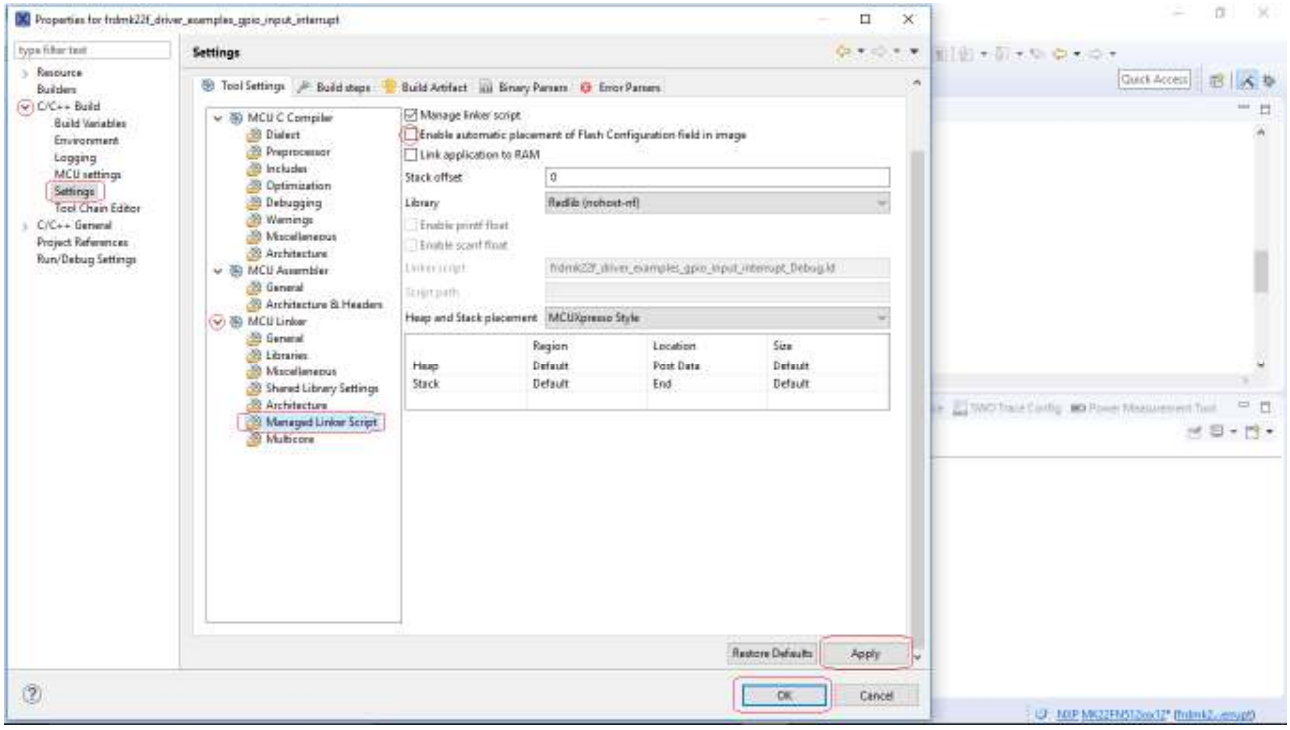
- In Project Explorer, right-click on the application project and select “Properties”, expand “C/C++ Build” and select “MCU Settings”. Then, click on “Edit” and modify PROGRAM_FLASH to start at 0x10000 and subtract 0x10000 from its size. Click on OK, to confirm the settings, click on “Apply” and “OK” to close the window:



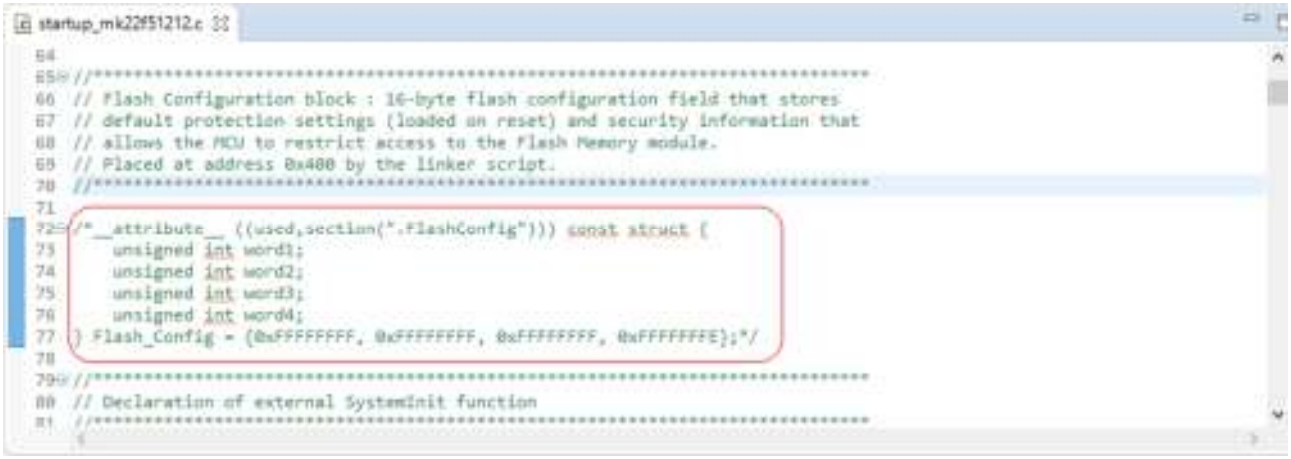
4.2. Disable allocation of Flash Configuration Field in image

Since Flash Configuration Field (0x400 to 0x40F) was already allocated and configured in Flash bootloader area, we should disable its allocation in application's Flash area:

- Right-click on application project in "Project Explorer"
- Select "Properties"
- Expand "C/C++ Build" and select "Settings"
- Click on "Tool Settings"
- Expand "MCU Linker" and select "Managed Linker Script"
- Uncheck the option "Enable automatic placement of Flash Configuration field in image"
- Click on "Apply" and "OK" buttons



- Comment or simply remove the Flash Configuration Field initialization in startup_*****.c file, which is located at startup folder:



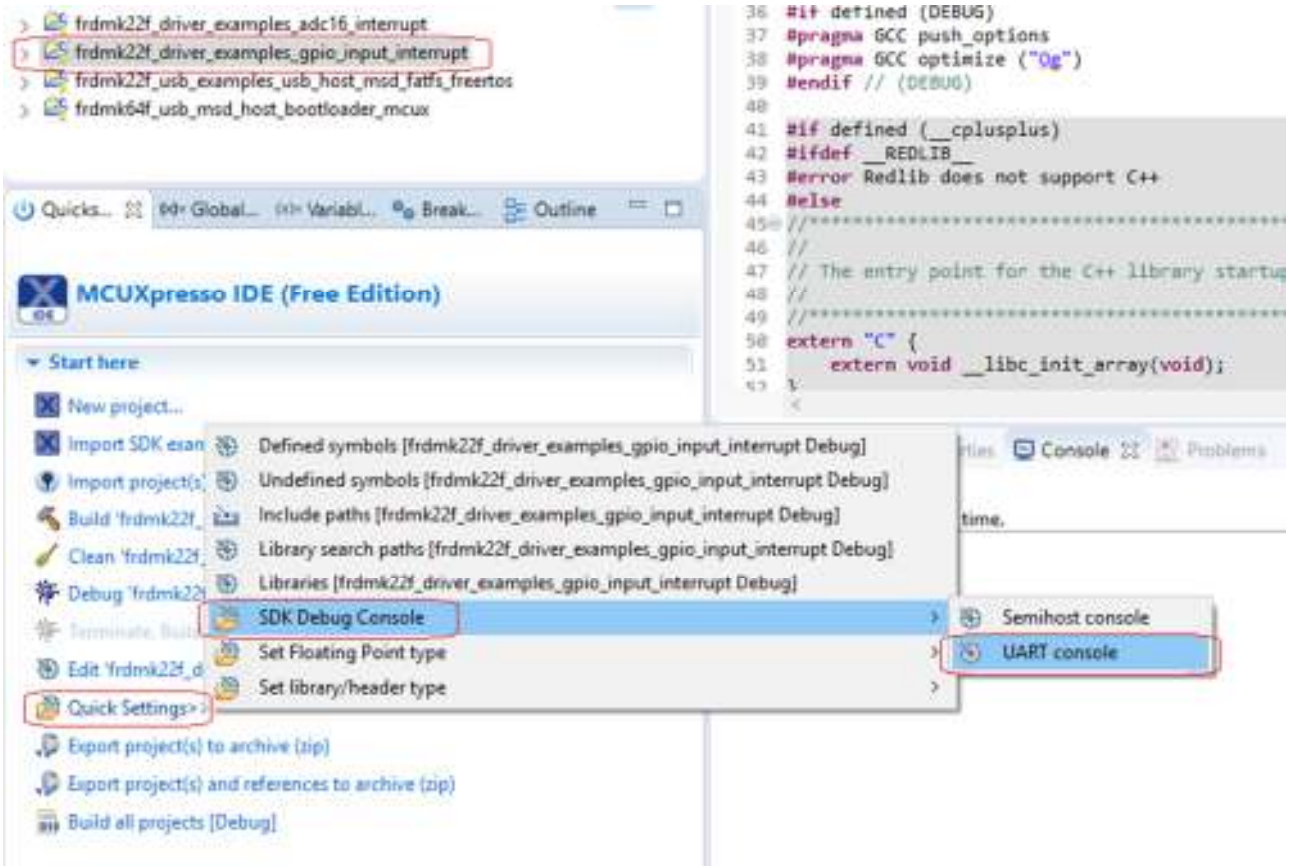
4.3. Redirect Kinetis interrupt and exception vectors in MCUXpresso

To redirect interrupt and exception vectors for Kinetis in MCUXpresso, you need to load SCB->VTOR register with the new application's Flash starting address in main loop like that:

```
int main(void)
{
    /* Define the init structure for the input switch pin */
    gpio_pin_config_t sw_config = {
        kGPIO_DigitalInput, 0,
    };

    /* Define the init structure for the output LED pin */
    gpio_pin_config_t led_config = {
        kGPIO_DigitalOutput, 0,
    };

    BOARD_InitPins();
    BOARD_BootClockRUN();
    BOARD_InitDebugConsole();

    SCB->VTOR = (uint32_t)0x10000;

    /* Print a note to terminal. */
    PRINTF("\r\n GPIO Driver example\r\n");
    PRINTF("\r\n Press %s to turn on/off a LED \r\n", BOARD_SW_NAME);

    /* Init input switch GPIO. */
    PORT_SetPinInterruptConfig(BOARD_SW_PORT, BOARD_SW_GPIO_PIN, kPORT_InterruptFallingEdge);
    EnableIRQ(BOARD_SW_IRQ);
    GPIO_PinInit(BOARD_SW_GPIO, BOARD_SW_GPIO_PIN, &sw_config);
```

4.4. Make sure that the console selected for the application is UART

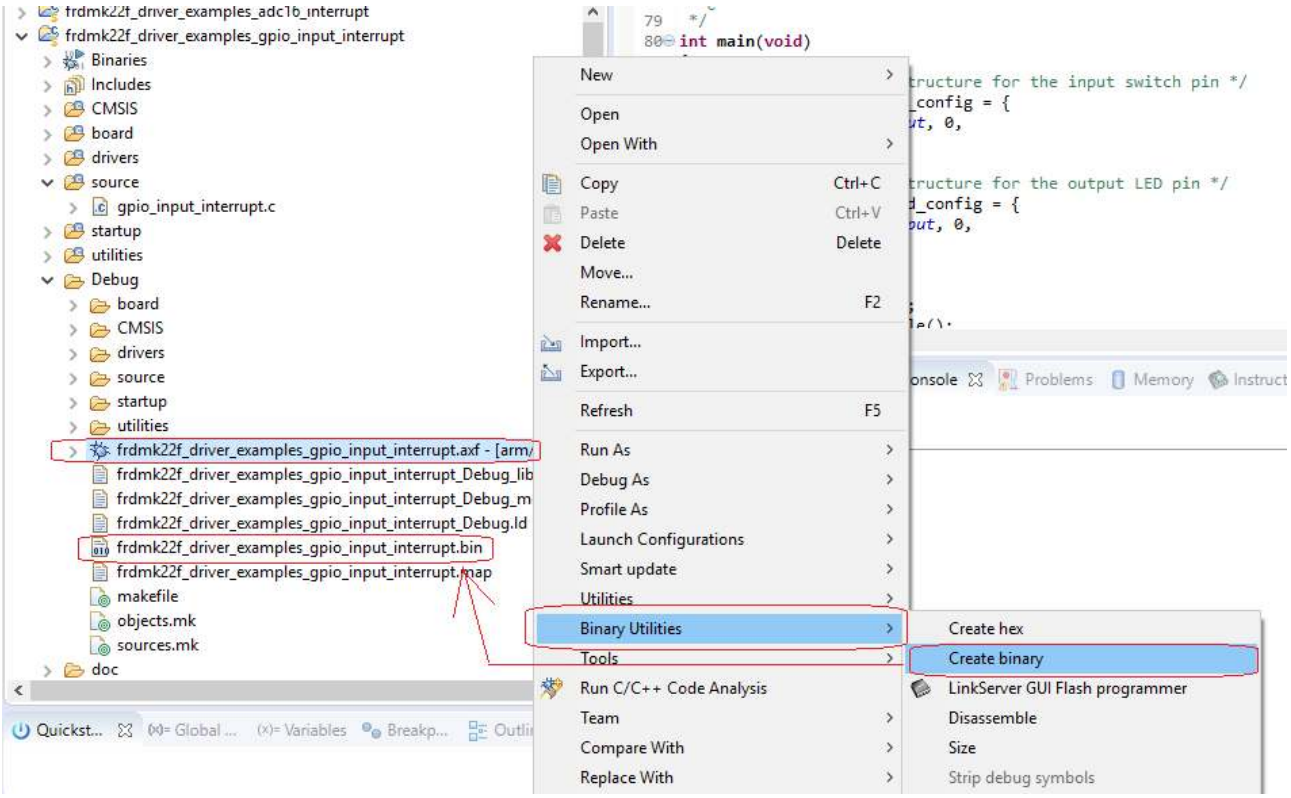
Select the application project in Project Explorer and, in Quick Start Panel, select UART as your Console, by clicking on “Quick Settings” / “SDK Debug Console” / “UART Console”, as shown in the figure bellow:



After the changes performed, click on hammer icon to build the project:

4.5. Generate the binary file in MCUXpresso

After building the project, MCUXpresso generates only a file with .axf extension in Debug Folder, but not with .bin extension. In order to generate a raw binary file, you must right-click on the .axf file, then select “Binary Utilities” and “Create binary”:



Copy and store the binary file into a USB memory stick and rename it to “image.bin”.

5. Preparing user applications developed in Kinetis Design Studio for the bootloader system

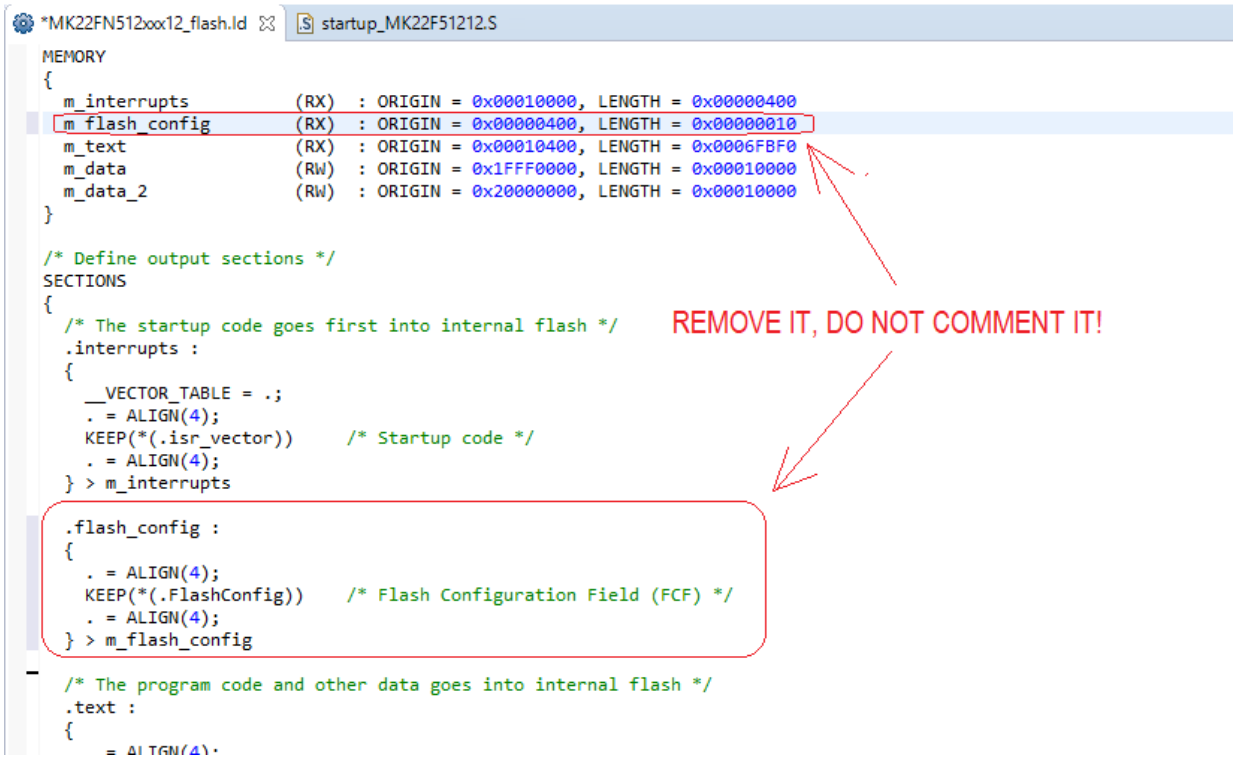
This section describes how normal applications developed in Kinetis Design Studio can be modified for the bootloader system.

5.1. Modify linker files in Kinetis Design Studio

The linker file defines what region of Flash memory the firmware will be placed at. Let’s take a look, for example, at the memory definitions in a typical application with MK22FN512VLH12:

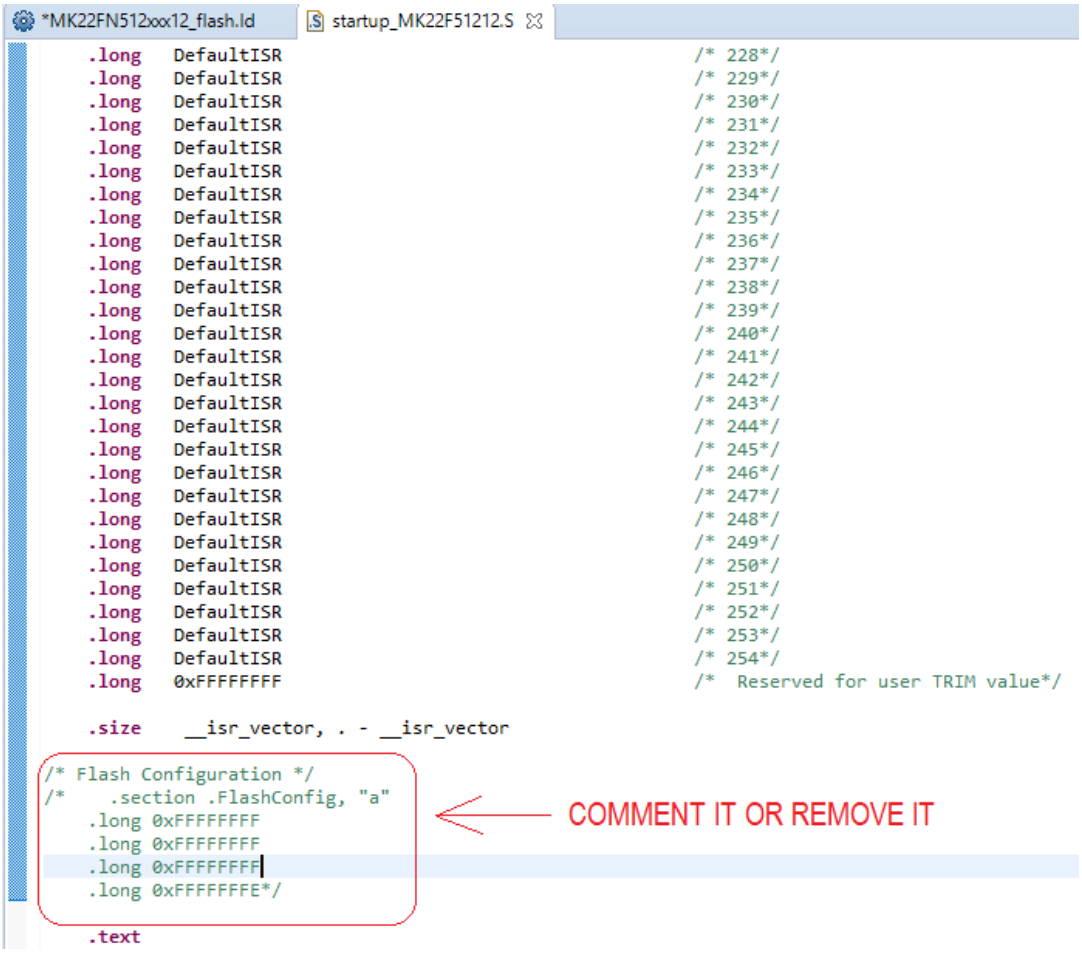
```
MEMORY

{

  m_interrupts      (RX)  : ORIGIN = 0x00000000, LENGTH = 0x00000400

  m_flash_config    (RX)  : ORIGIN = 0x00000400, LENGTH = 0x00000010

  m_text            (RX)  : ORIGIN = 0x00000410, LENGTH = 0x0007FBF0

  m_data            (RW)  : ORIGIN = 0x1FFF0000, LENGTH = 0x00010000

  m_data_2          (RW)  : ORIGIN = 0x20000000, LENGTH = 0x00010000

}
```

To work with the bootloader system for MK22FN512VLH12 or MK64FN1M0VLL12, the interrupt vector table must be located at the flash memory area starting from the address 0x10000 and user application code must start at 0x10400. The sizes must be adjusted accordingly. The “m_flash_config” memory area must be removed, since it was already defined and allocated in bootloader area, from 0x400 to 0x40F. In addition, the flash_config section must also be removed (not commented):



5.2. Remove Flash Configuration Field initialization

Remove or comment Flash Configuration Field initialization in startup****.S file, located in startup folder:



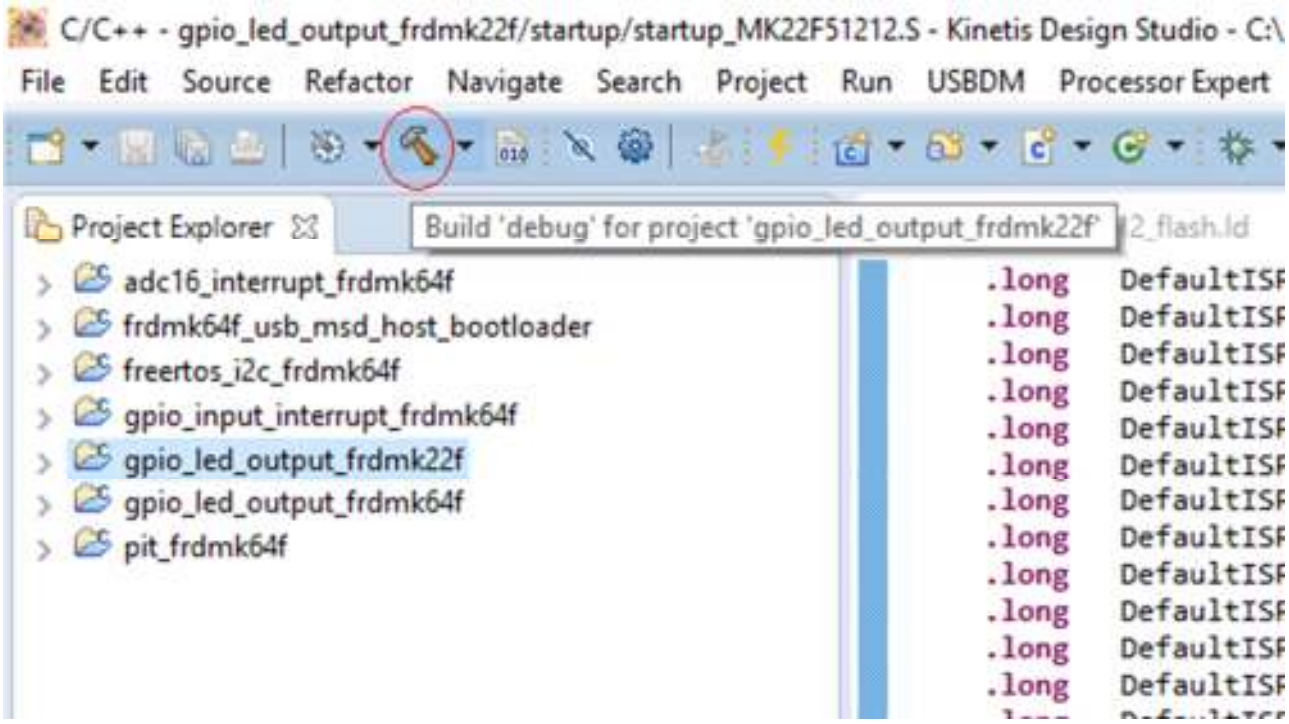
5.3. Redirect interrupt and exception vectors in Kinetis Design Studio

To redirect interrupt and exception vectors for Kinetis in KDS, you need to load SCB->VTOR register with the new application's Flash starting address in main loop as follows:

*adc16_interrupt.c

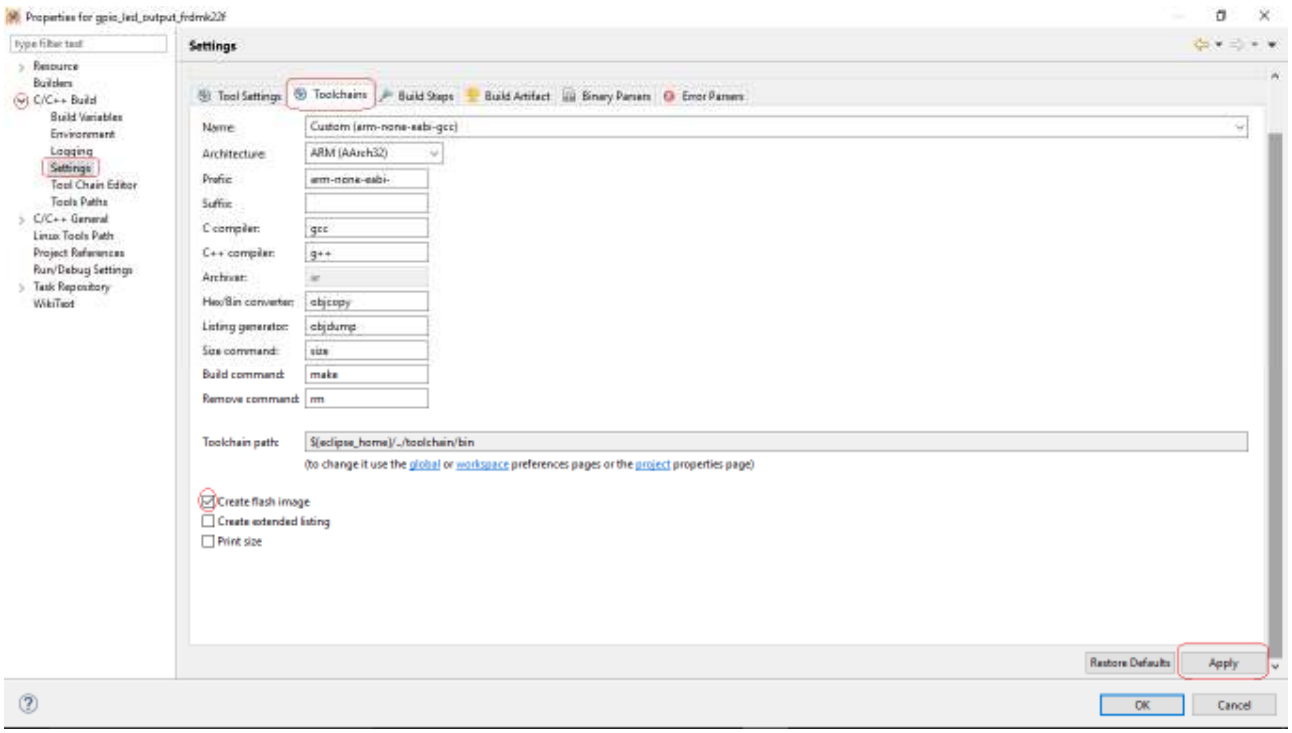
```
volatile uint32_t g_Adc16InterruptCounter;

/*******************************************************************************
 * Code
 ******************************************************************************/


void DEMO_ADC16_IRQ_HANDLER_FUNC(void)
{
    g_Adc16ConversionDoneFlag = true;
    /* Read conversion result to clear the conversion completed flag. */
    g_Adc16ConversionValue = ADC16_GetChannelConversionValue(DEMO_ADC16_BASE, DEMO_ADC16_CHANNEL_GROUP);
    g_Adc16InterruptCounter++;
}

/*!
 * @brief Main function
 */
int main(void)
{
    adc16_config_t adc16ConfigStruct;
    adc16_channel_config_t adc16ChannelConfigStruct;

    BOARD_InitPins();
    BOARD_BootClockRUN();
    BOARD_InitDebugConsole();
    EnableIRQ(DEMO_ADC16_IRQn);

    SCB->VTOR = (uint32_t)0x10000;

    PRINTF("\r\nADC16 interrupt Example.\r\n");

    /*
     * adc16ConfigStruct.referenceVoltageSource = kADC16_ReferenceVoltageSourceVref;
```

After the changes performed, click on hammer icon to build the project:

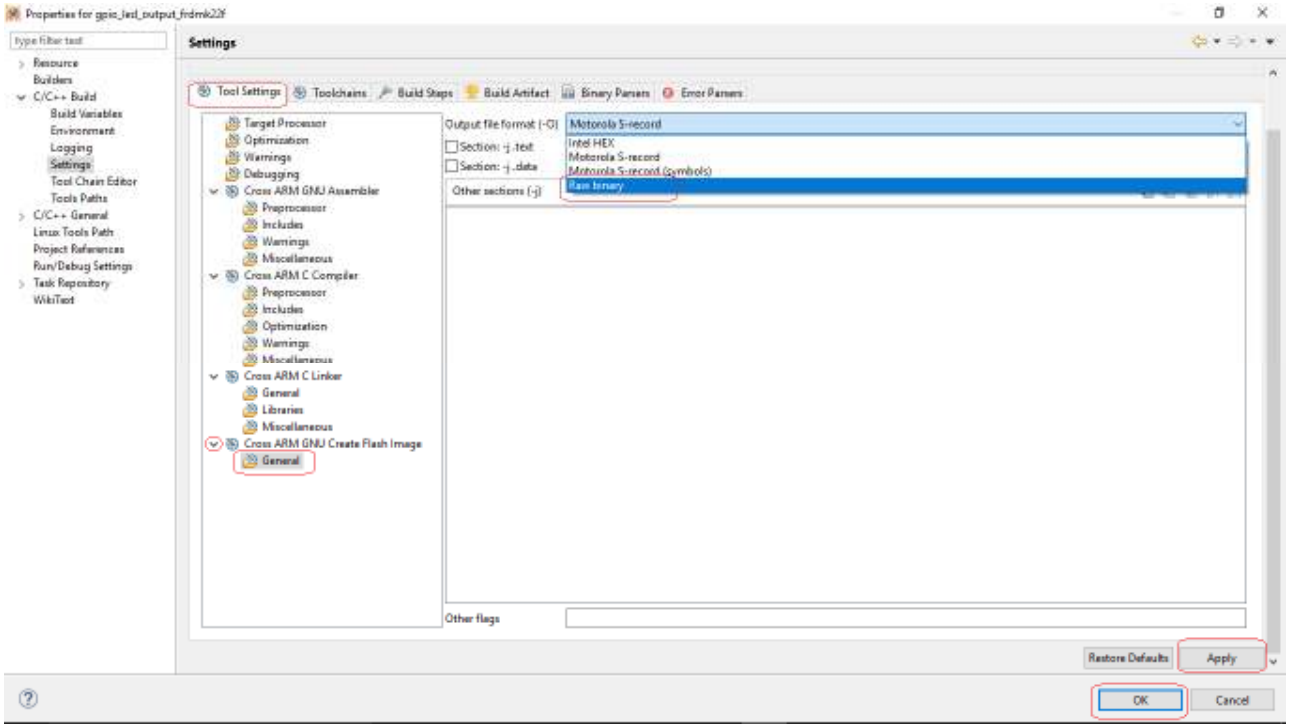


5.4. Generate the binary file in Kinetis Design Studio

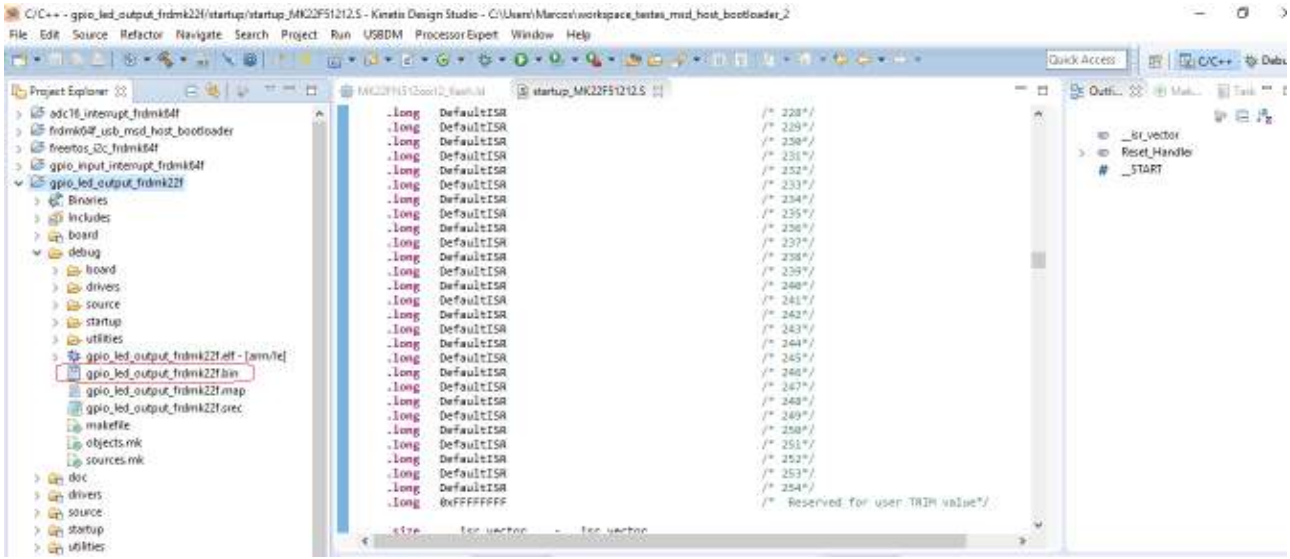
- In Project Explorer, right-click on the application project
- Select “Properties”
- Expand “C/C++ Build” and select “Settings”
- Click on “Toolchains” tab, check “Create Flash image” option and click on “Apply” button



- Click on Tool Settings tab
- Expand “Cross ARM GNU Create Flash Image” and select “General”
- On “Output File Format (-O)” selection field, select “Raw binary”
- Click on “Apply” and then on “OK” button



- After rebuilding the project, the raw binary file comes up in debug folder:



Copy this file to a USB memory stick and rename it as “image.bin”

6. Using the bootloader

This section describes how to run and use the bootloader both on FRDM-K22F and FRDM-K64F kits.

The following steps are described in this section:

- Preparing software and hardware
- Programming the bootloader
- Running the bootloader

6.1. Preparing software and hardware

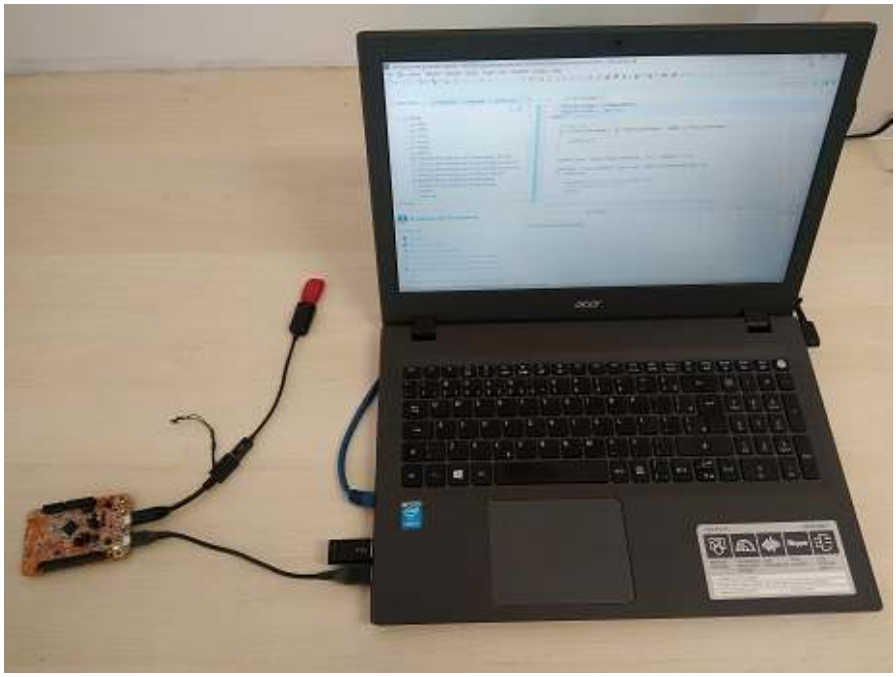
Software required:

• MCUXpresso

• Hyper Terminal, Tera Term or any other Serial Terminal application

Hardware required:

- A personal computer
- A FRDM-K64F or FRDM-K22F development kit
- A USB memory stick
- A micro-AB to A USB cable
- A micro-AB to A adapter (for USB memory stick)



Hardware Setup:

For FRDM-K22F:

- Connect jumper J22
- Connect an USB cable from the PC to the USB Open SDA board port (J5)
- Connect a micro-AB to A adapter to the MCU USB port (J16)

For FRDM-K64F:

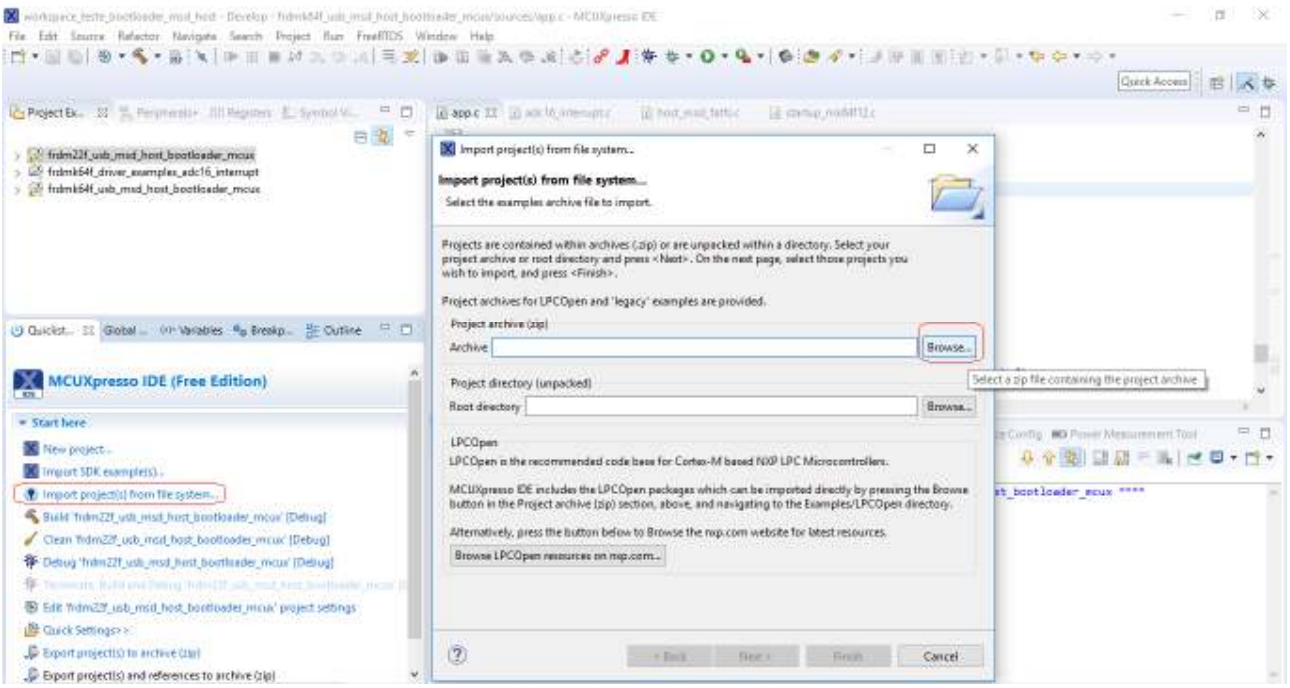
- Connect jumper J21
- Connect an USB cable from the PC to the USB Open SDA board port (J26)
- Connect a micro-AB to A adapter to the MCU USB port (J22)

6.2. Programming the bootloader

The following steps show how the bootloader can be programmed to the MCU:

1. Import Project:

- In Quick Start Panel, click on “Import project(s) from file system…”
- Click on “Browse” button



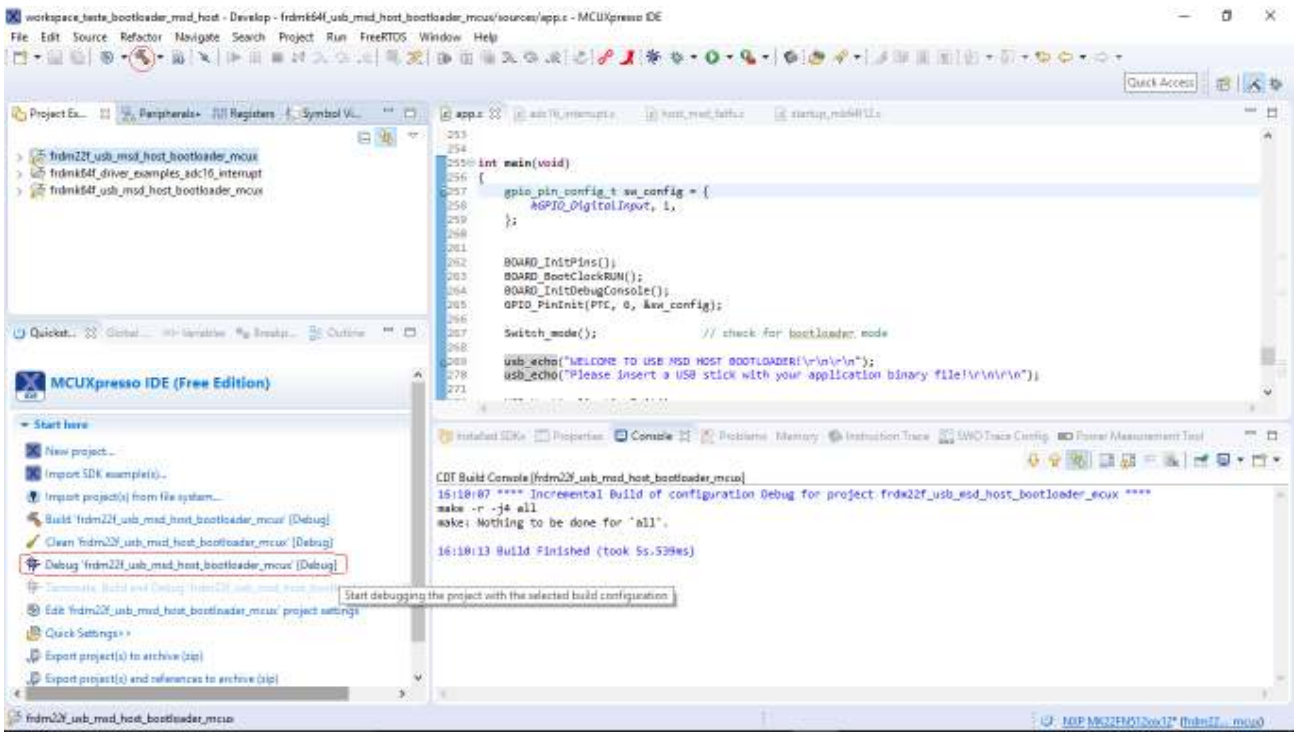
- Select the zipped bootloader project folder, click on “Next” and then on “Finish” button

2. Select UART as Debug Console:

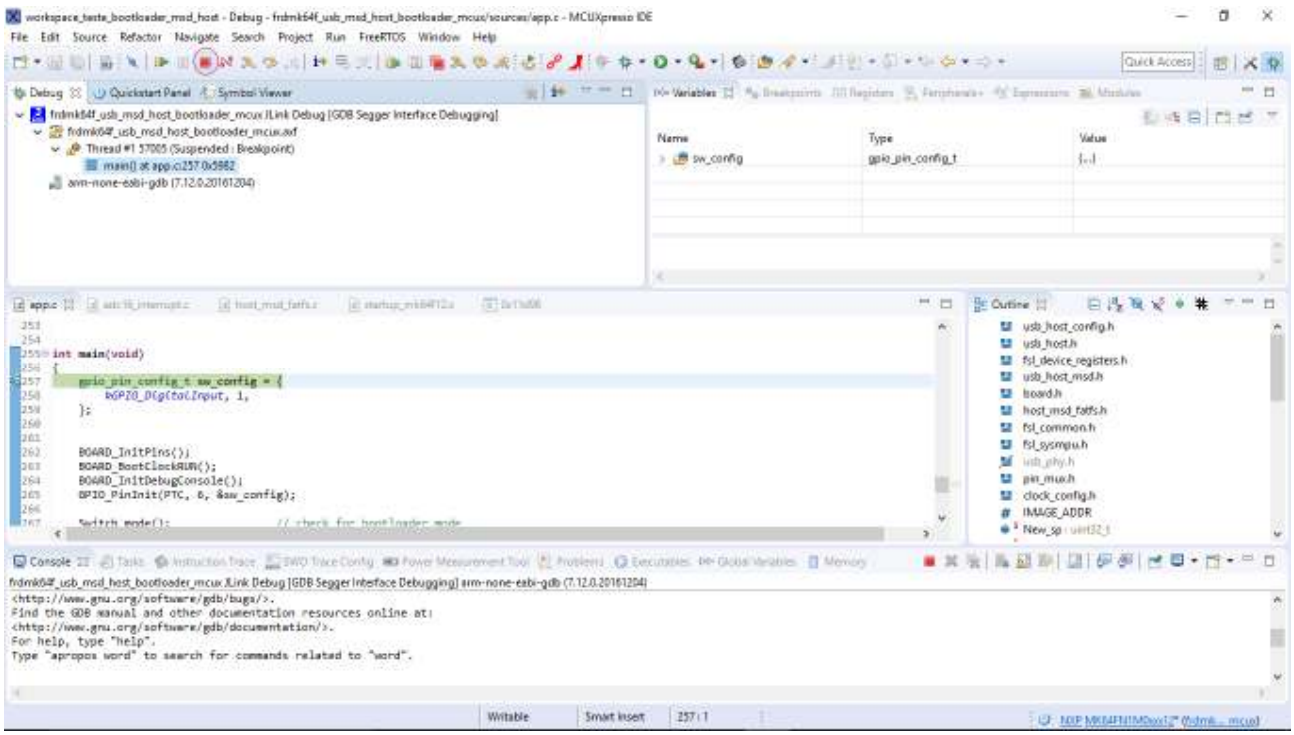
- Repeat the steps described in item “4.4 Make sure that the console selected for the application is UART” for bootloader project

3. Programming the bootloader:

- Make sure the bootloader project is selected in Project Explorer
- Click on “hammer” icon to build the bootloader project
- In Quick Start Panel, click on “Debug *bootloader project*” [Debug]



- Terminate the debugging session and close MCUXpresso



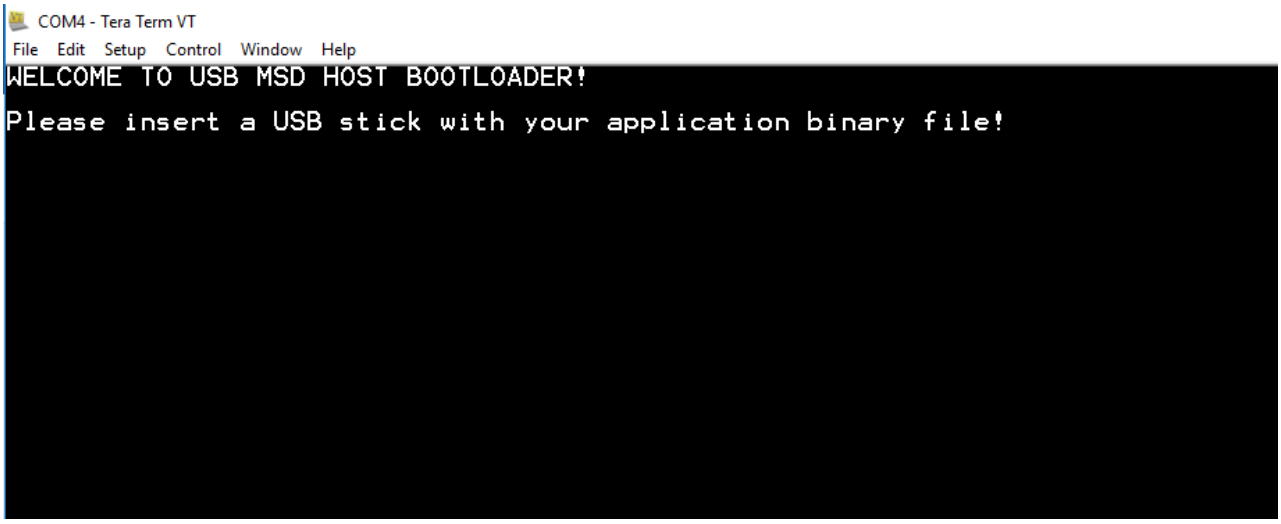
6.3. Running the bootloader

1. Disconnect and reconnect Freedom USB cable from Open SDA port to PC.

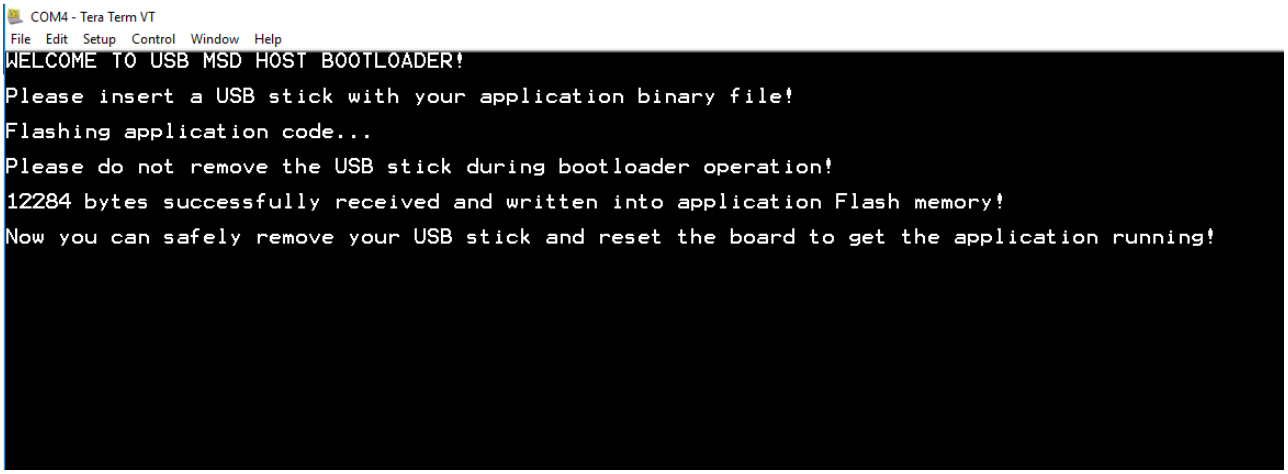
2. Open a Serial Terminal application, like Tera Term or Hyper Terminal, select Open SDA Virtual COM port and the following setup:

- Baud rate: 115200
- Data length: 8 bit
- Parity: none
- 1 Stop bit
- No flow control

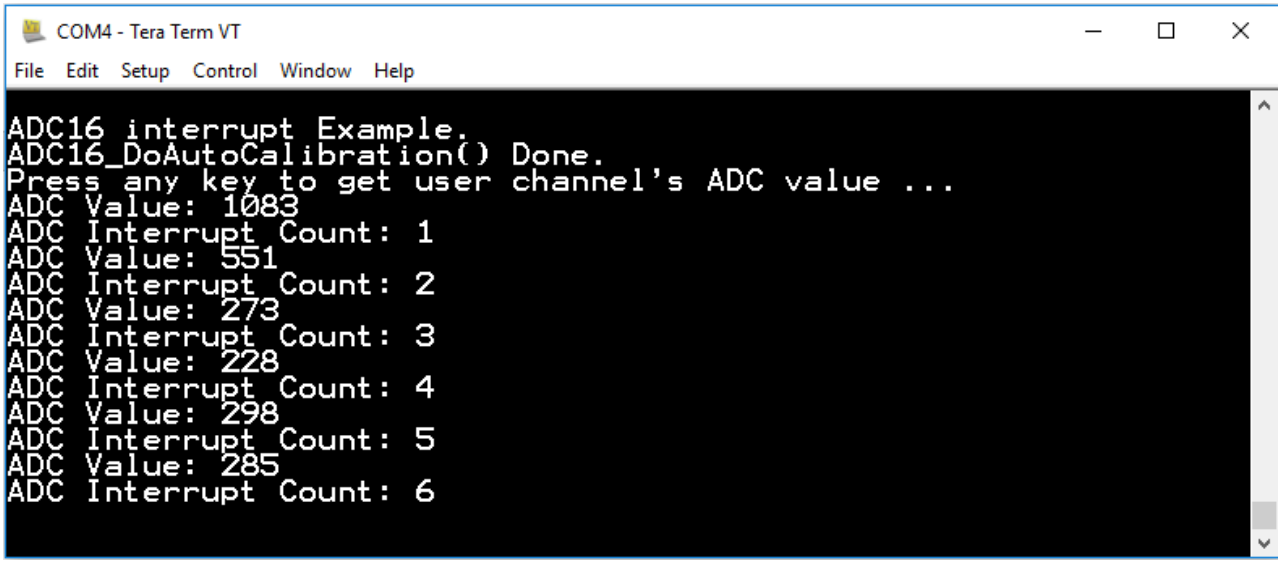
3. Hold SW2 button, in case of FRDM-K64F kit, or SW3 button, in case of FRDM-K22F kit, and press Reset button. The following messages will be displayed on the screen:



4. Insert an USB stick containing your application binary file into the MCU USB connector (J22 on FRDM-K64F and J16 on FRDM-K22F). To see how to prepare your application for the bootloader system, please refer to sections 4 (for applications developed in MCUXpresso) and 5 (for applications developed in KDS). If everything is running fine, you should see the messages like those displaying on your screen:



5. After pressing Reset button, you should see your application running



7. Error messages

There are many messages that signal failures on task creation, FAT-Fs and USB initialization and transactions. Here we will board only some error messages related to the main portion of the bootloader application, considering that the USB MSD device has been successfully attached and identified:

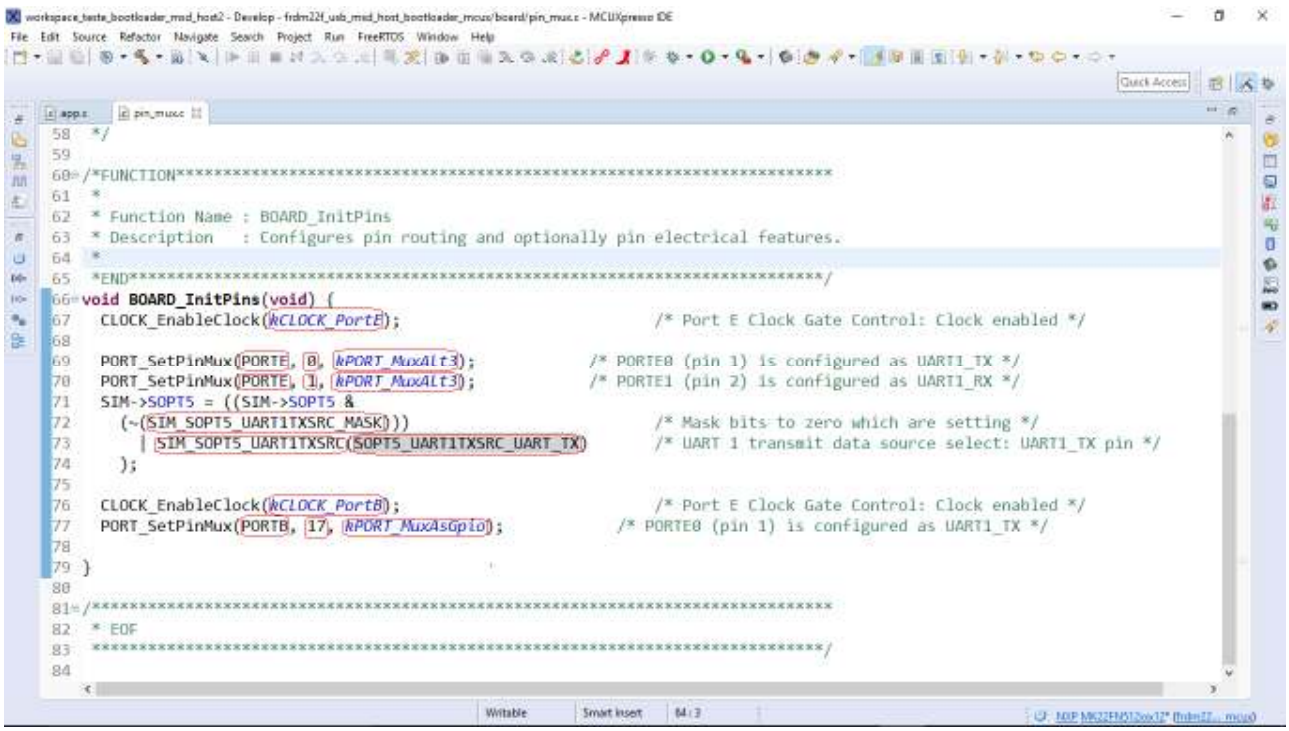
1. **"Fail to mount File System":** Failed to mount the FAT-32 file system to handle the file in the USB MSD device
2. **"Fail to open file":** The file could not be created or opened
3. **"Fail to initialize Flash memory driver":** Something went wrong with Flash driver initialization
4. **"Fail to erase application Flash memory area":** Could not erase Flash memory
5. **"Fail to read data block n…"**: Failed to read a 1024 bytes data block containing application code from the USB memory stick
6. **"Fail to write data block into application Flash memory":** Failed to write a 1024 bytes data block containing application code into Flash

8. Modifying bootloader code

This section boards the following things in the bootloader that may be changed by the user and how to configure them accordingly, taking FRDM-K22F demo code as reference:

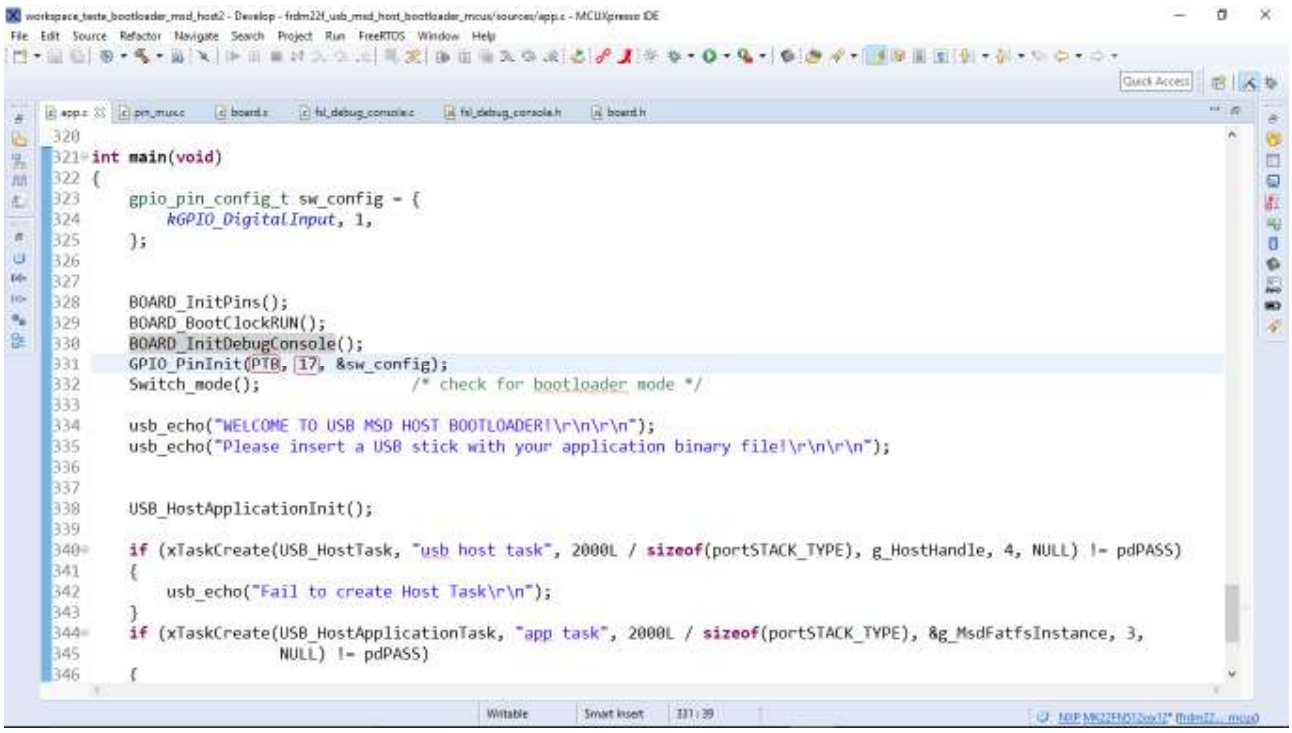
8.1. Changing the button and UART pins:

Change the parameters passed to “CLOCK_EnableClock”, “PORT_SetPinMux” and the masks (corresponding to UART transmit data source selection) loaded into SIM->SOPT5 register, in “BOARD_InitPins”, in “pin_mux.c” file:



```
58  */
59
60 /*FUNCTION**********************************************************************
61  *
62  * Function Name : BOARD_InitPins
63  * Description   : Configures pin routing and optionally pin electrical features.
64  *
65  *END**********************************************************************/
66 void BOARD_InitPins(void) {
67   CLOCK_EnableClock(kCLOCK_PortE);                    /* Port E Clock Gate Control: Clock enabled */
68
69   PORT_SetPinMux(PORTE, 0, kPORT_MuxAlt3);          /* PORTE0 (pin 1) is configured as UART1_TX */
70   PORT_SetPinMux(PORTE, 1, kPORT_MuxAlt3);          /* PORTE1 (pin 2) is configured as UART1_RX */
71   SIM->SOPT5 = ((SIM->SOPT5 &
72     (~(SIM_SOPT5_UART1TXSRC_MASK)))                 /* Mask bits to zero which are setting */
73       | SIM_SOPT5_UART1TXSRC(SOPT5_UART1TXSRC_UART_TX)  /* UART 1 transmit data source select: UART1_TX pin */
74     );
75
76   CLOCK_EnableClock(kCLOCK_PortB);                    /* Port E Clock Gate Control: Clock enabled */
77   PORT_SetPinMux(PORTB, 17, kPORT_MuxAsGpio);       /* PORTE0 (pin 1) is configured as UART1_TX */
78
79 }
80
81 /*******************************************************************************
82  * EOF
83  ******************************************************************************/
84
```

Change the parameters passed to “GPIO_PinInit” function, in “app.c” file:

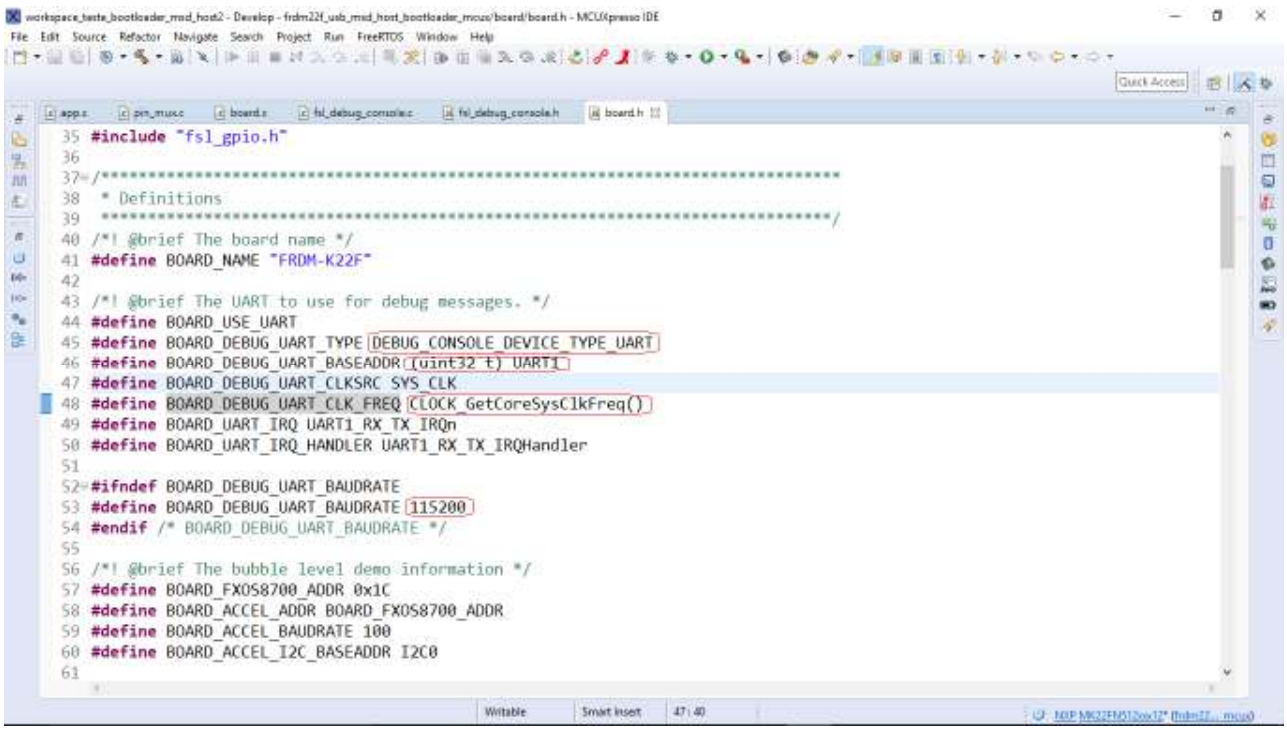


```
320
321 int main(void)
322 {
323     gpio_pin_config_t sw_config = {
324         kGPIO_DigitalInput, 1,
325     };
326
327
328     BOARD_InitPins();
329     BOARD_BootClockRUN();
330     BOARD_InitDebugConsole();
331     GPIO_PinInit(PTB, 17, &sw_config);
332     Switch_mode();              /* check for bootloader mode */
333
334     usb_echo("WELCOME TO USB MSD HOST BOOTLOADER!\r\n\r\n");
335     usb_echo("Please insert a USB stick with your application binary file!\r\n\r\n");
336
337
338     USB_HostApplicationInit();
339
340     if (xTaskCreate(USB_HostTask, "usb host task", 2000L / sizeof(portSTACK_TYPE), g_HostHandle, 4, NULL) != pdPASS)
341     {
342         usb_echo("Fail to create Host Task\r\n");
343     }
344     if (xTaskCreate(USB_HostApplicationTask, "app task", 2000L / sizeof(portSTACK_TYPE), &g_MsdFatfsInstance, 3,
345                     NULL) != pdPASS)
346     {
```

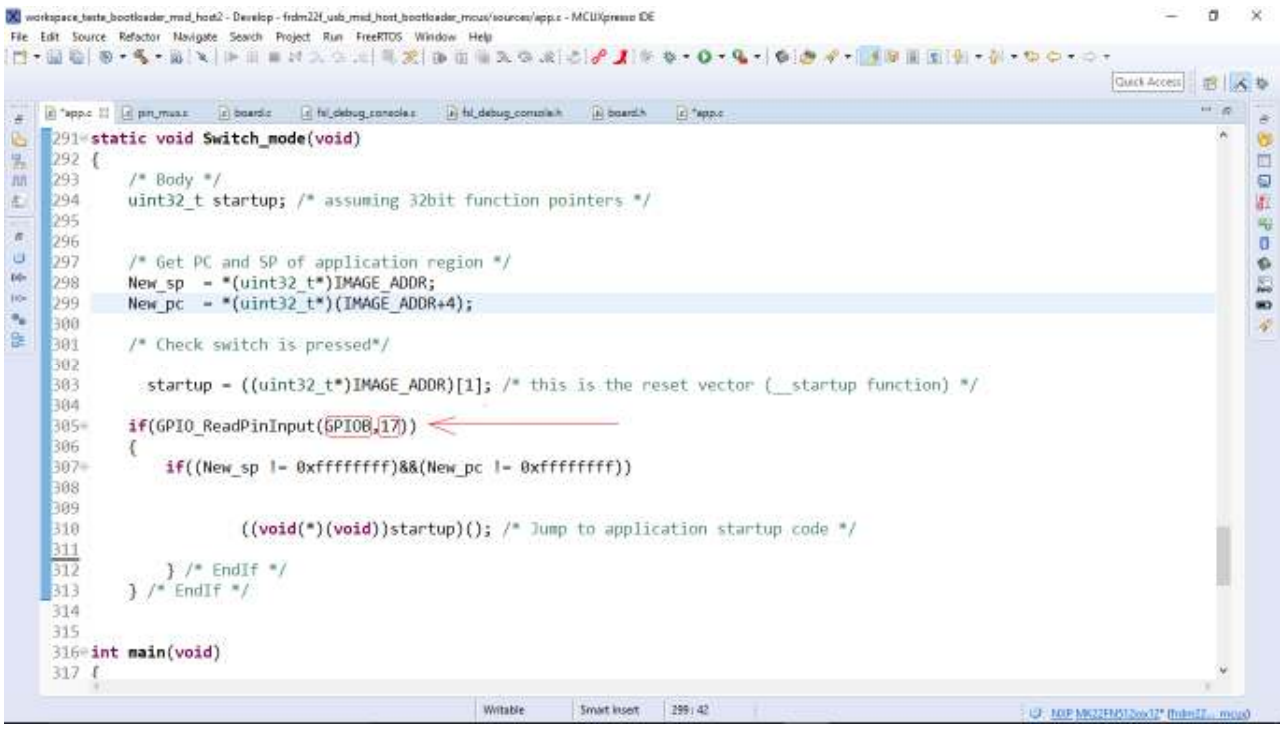
Change parameters passed to “BOARD_InitDebugConsole” function that define the address of the peripheral used for sending messages, the baud rate, the peripheral used (UART, LPSCI, LPSPI or USB-CDC) and the frequency of the peripheral source clock:

```
/* Initialize debug console. */
void BOARD_InitDebugConsole(void)
{
    uint32_t uartClkSrcFreq = BOARD_DEBUG_UART_CLK_FREQ;

DbgConsole_Init(BOARD_DEBUG_UART_BASEADDR,
BOARD_DEBUG_UART_BAUDRATE, BOARD_DEBUG_UART_TYPE,
uartClkSrcFreq);
}
```

All parameters can be modified in “board.h” file:



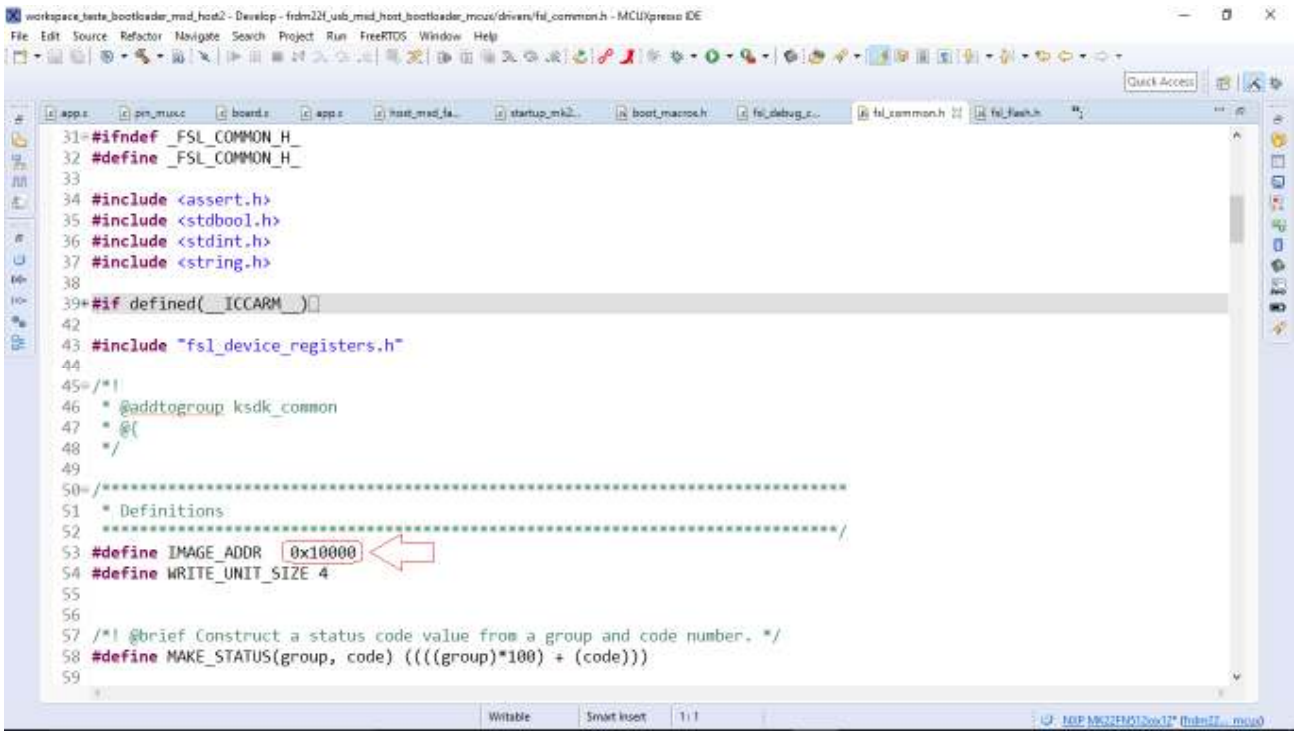
“Switch_mode” function checks if the button was pressed and if so, it runs the bootloader. Otherwise, it jumps to the application. You should change the parameters in “GPIO_ReadPinInput” function call, in case you are connecting other pin to the button:



```
static void Switch_mode(void)
{
    /* Body */
    uint32_t startup; /* assuming 32bit function pointers */


    /* Get PC and SP of application region */
    New_sp  = *(uint32_t*)IMAGE_ADDR;
    New_pc  = *(uint32_t*)(IMAGE_ADDR+4);

    /* Check switch is pressed*/

      startup = ((uint32_t*)IMAGE_ADDR)[1]; /* this is the reset vector (__startup function) */

    if(GPIO_ReadPinInput(GPIO8,17))
    {
        if((New_sp != 0xffffffff)&&(New_pc != 0xffffffff))


                ((void(*)(void))startup)(); /* Jump to application startup code */

        } /* EndIf */
    } /* EndIf */


int main(void)
{
```

8.2. Changing application's starting address and bootloader Flash protection:

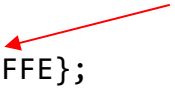
As commented in Section 1, you may want to add more features to bootloader like enabling Hub support, for example, and then you will have to change "IMAGE_ADDR" macro, which is the Flash address where the application code will be programmed at by the bootloader. This macro is defined in "fsl_common.h" file and it is used by "Switch_mode" and Flash erase and write functions. Important to mention that IMAGE_ADDR has to be block aligned, if you want to keep the bootloader area protected:



```
#ifndef _FSL_COMMON_H_
#define _FSL_COMMON_H_

#include <assert.h>
#include <stdbool.h>
#include <stdint.h>
#include <string.h>

#if defined(__ICCARM__)

#include "fsl_device_registers.h"

/*!
 * @addtogroup ksdk_common
 * @{
 */

/*******************************************************************************
 * Definitions
 ******************************************************************************/
#define IMAGE_ADDR  0x10000
#define WRITE_UNIT_SIZE 4


/*! @brief Construct a status code value from a group and code number. */
#define MAKE_STATUS(group, code) ((((group)*100) + (code)))
```

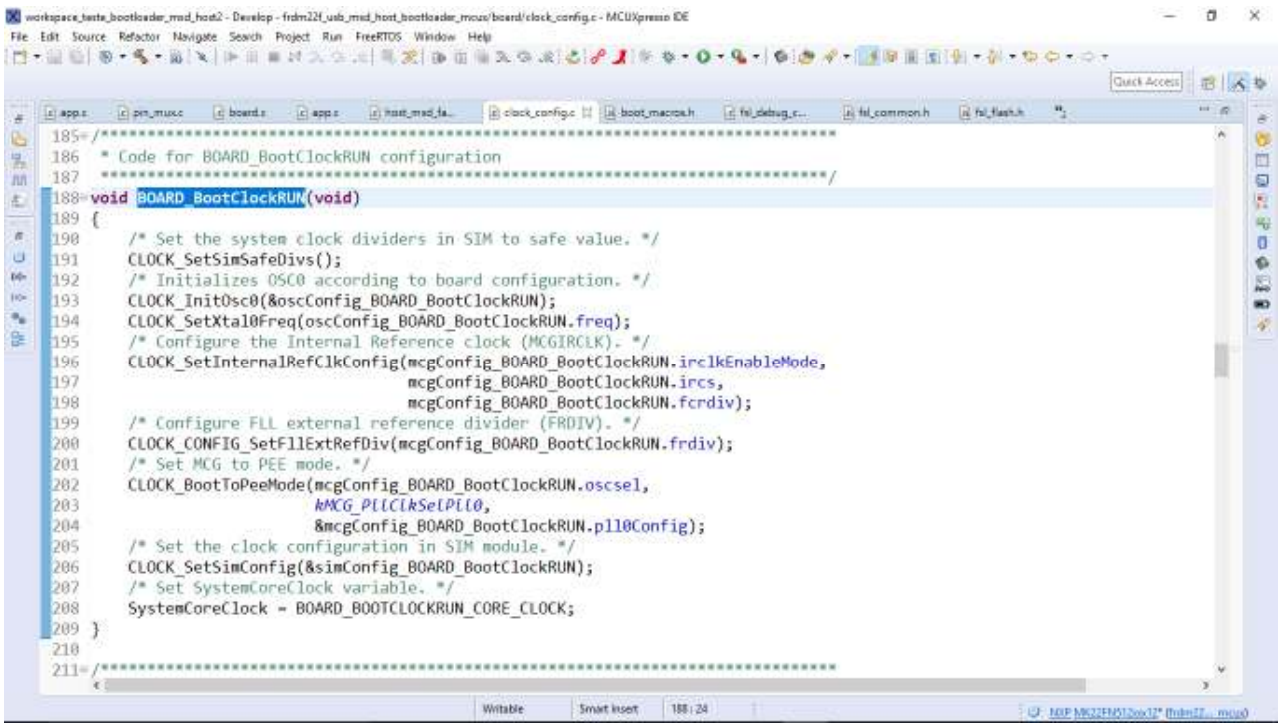
In Flash Configuration Field, more specifically from 0x408 to 0x40B, you have the Flash protection settings. 0x40B is the most significant byte and 0x408 is the least significant byte for NVPROT, whose contents are copied to FPROT register during startup. If you add a feature like Hub support to bootloader and the code surpass 65536 bytes, you must protect one more block. The least significant bit in NVPROT represents the lowest block in Flash memory. When you want to protect a certain block, you just clear its corresponding bit in FCF. Let's take a look at what FCF settings look in our FRDM_K22F bootloader demo code. FCF settings are accessible in startup_mk22f51212.c file (or startup_mk64f12.c, in the case of FRDM_K64F bootloader demo), present in startup folder:

```
__attribute__ ((used,section(".FlashConfig"))) const struct {
    unsigned int word1;
    unsigned int word2;
    unsigned int word3;
    unsigned int word4;
} Flash_Config = {0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFF0, 0xFFFEFFFE};
```



8.3. Changing clock configuration:

The last but not the least important point we need to board is the clock configuration. In the demos, the clock is set to PEE mode, meeting the hardware setup of the Freedom boards. In our demos, "BOARD_BootClockRUN" function is called in main initialization code, it is in "clock_config.c" file and it is responsible for initializing the clock system. So, you must change that code and clock initialization parameters, as well, to meet your system requirements. Important to mention that Flash erase and write functions can only be performed in Run mode, that is, at core frequencies of up to 80MHz and USB clock must be set to 48MHz. Please refer to "MCUXpresso SDK API Reference Manual", present in "docs" folder, in SDK installation directory to see how to use the clock drivers to initialize the clock system, as desired.



9. Conclusion

Although a USB MSD Host bootloader consumes more Flash memory than other ways of communication like UART, SPI, I2C or even USB MSD Device (the MCU itself is a USB mass storage device), the kind of bootloader system presented here doesn't need to be connected to a personal computer, since the firmware could be modified in order to show the status messages on a LCD or simply turn on/off or change colors of leds, which makes the application updating possible anywhere. Besides, USB sticks are inexpensive portable memories, widely available.

10. Referred documents

- K64 Sub-Family Reference Manual
- K22 Sub-Family Reference Manual
- FRDM-K64F Freedom Module User's Guide
- Freedom Board for Kinetis K22F Hardware (FRDM-K22F) User's Guide
- Getting Started with MCUXpresso SDK
- MCUXpresso SDK API Reference Manual
- AN4368 - USB Mass Storage Device Host Bootloader