Dual-Bank Bootloader on SAM E54 Microcontroller (MCU) Using MPLAB Harmony v3

AN3508

Introduction

The bootloader is a piece of code used to program or re-program the application code (firmware) to the internal Flash of the microcontroller without the need for an external programmer or debugger.

The following are key features of the dual-bank bootloader:

- It is the first program to run on Power-on-Reset (POR), and responsible to load the firmware into a specific memory location
- It can communicate to the host program to receive the firmware through communication interfaces, such as USB, Ethernet, CAN, UART, I^2C and SPI
- It is programmed into the microcontroller using the normal conventional programming methods such as an external programmer or debugger (SWD, JTAG)
- It is responsible to check whether the user is intending to update the firmware or run the existing firmware. A microcontroller can have two code images co-existing in the same memory space (bootloader and user application (firmware)).

The SAM E54 MCU provides a dual-bank support on the internal Flash memory. The dual-bank Flash enables the programming of the inactive bank with a new version of the firmware without affecting the existing application on an active bank.

The MPLAB Harmony v3 provides a bootloader framework for 32-bit microcontrollers, which can be used to upgrade the firmware on a target device without using the external programmer or debugger. This document describes the dual-bank bootloader provided by MPLAB Harmony v3. The dual-bank bootloader utilizes the dual-bank feature of the internal Flash for safer application upgrade.

Table of Contents

Introduction....1

1. Hardware and Software Requirements....3
   - 1.1. SAM E54 Xplained Pro Evaluation Kit....3
   - 1.2. MPLAB® X Integrated Development Environment (IDE) and XC Compilers....3
   - 1.3. MPLAB Harmony v3....3
   - 1.4. Python....3

2. Description....4
   - 2.1. Bootloader Framework....4
   - 2.2. Modes of Operation....5
   - 2.3. UART Bootloader Protocol....7
   - 2.4. Bootloader Trigger Methods....8
   - 2.5. Bootloader System Level Execution Flow....10

3. Configuring the Dual-Bank Bootloader....11
   - 3.1. Bootloader Linker Script....12
   - 3.2. Test Application Configurations....13
   - 3.3. Test Application Project Settings....14

4. Running the Demonstration....17
   - 4.1. Running the Bootloader Application....17
   - 4.2. Running the Test Application....19

5. References....22

6. Revision History....23

Microchip Information....24
- Trademarks....24
- Legal Notice....24
- Microchip Devices Code Protection Feature....24

1. Hardware and Software Requirements

1.1 SAM E54 Xplained Pro Evaluation Kit

The SAM E54 Xplained Pro Evaluation Kit is a development kit for evaluating the SAM E54 microcontrollers (MCUs). The SAM E54 is based on an Arm® Cortex® -M4 capable of running at 120 MHz. This pro-evaluation kit includes an on-board Embedded Debugger, which eliminates the need for external tools to program or debug the SAM E54. The evaluation kit also offers external connectors to extend the features of the board and ease the development of custom designs.

The SAM E54 Xplained Pro Evaluation Kit is available for download at Microchip Direct.

1.2 MPLAB® X Integrated Development Environment (IDE) and XC Compilers

The MPLAB X IDE is an expandable, highly-configurable software program that incorporates powerful tools to help users discover, configure, develop, debug, and qualify embedded designs for most of the Microchip's microcontrollers.

The MPLAB X IDE is available at Microchip Website. This document uses MPLAB X IDE version 6.20.

MPLAB XC Compilers are available at Microchip Website. This document uses MPLAB XC32 version 4.45.

1.3 MPLAB Harmony v3

MPLAB Harmony v3 is a fully-integrated embedded software development framework that provides flexible and interoperable software modules that allow users to dedicate their resources to create applications for 32-bit PIC® and SAM devices, rather than dealing with device details, complex protocols, and library integration challenges.

MPLAB® Code Configurator (MCC) is a free graphical programming environment that generates easy-to-understand C code for the project. It provides an intuitive interface to configure peripherals, and functions specific to the application. MCC supports our 8-bit, 16-bit, and 32-bit devices, including PIC®, AVR®, SAM microcontrollers (MCUs), and dsPIC® Digital Signal Controllers (DSCs).

The MCC is available as a plugin that integrates with the MPLAB X IDE and has a separate Java™ executable for stand-alone use with other development environments.

The examples used in this document use the following repositories, which can be downloaded from GitHuB:

- csp: Harmony 3 Chip Support Package
- bootloader: Harmony 3 Bootloader
- bootloader_apps_uart: Harmony 3 Bootloader UART examples

The MCC Content Manager can also be used to download the repositories.

1.4 Python

This document describes using the python scripts for converting binary output to a 'C' style array containing a Hex output. Python is also used to merge the bootloader binary and the application binary.

The conversion and merging covered in this document are performed using Python v3.11.9.

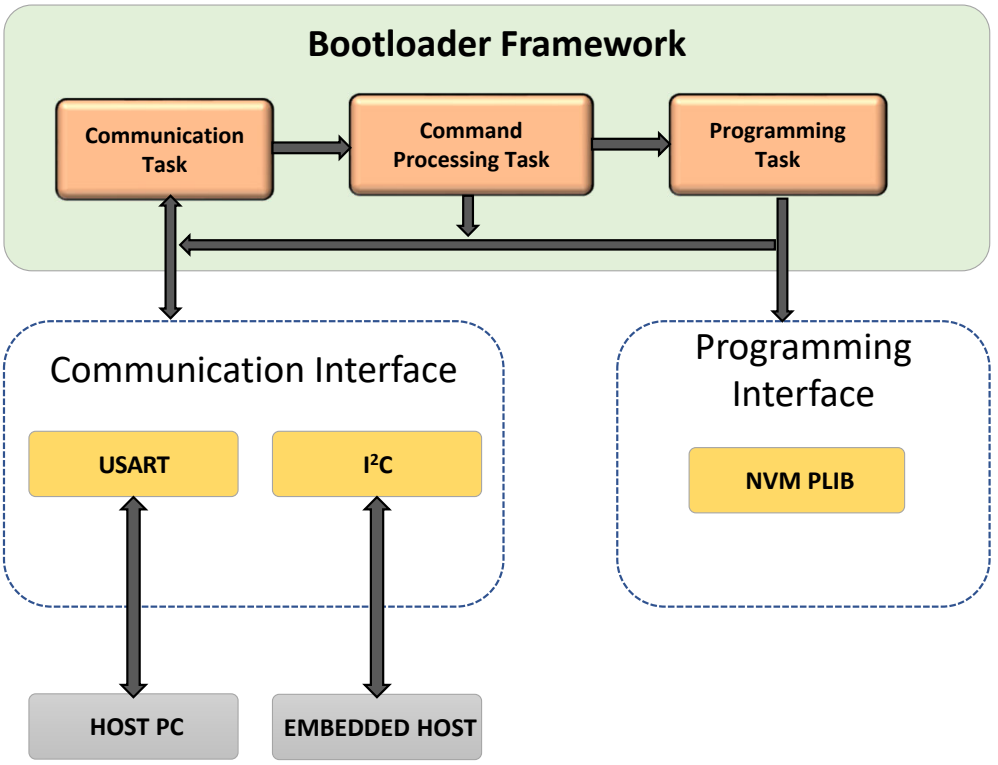
2. Description

2.1 Bootloader Framework

The MPLAB Harmony v3 bootloader framework is divided into the following sub tasks:

- Communication Task
- Command Processing Task
- Programming Task

Figure 2-1. Harmony v3 Bootloader Framework



Communication Task

This task is responsible for receiving data from the host PC or embedded host through the selected communication interface in polling mode. It validates the incoming packet from the host with the expected header information before passing it to the command processing task.

Command Processing Task

This task processes the commands received from communication task and acts upon it, providing the response back to the host PC accordingly. If the command received is a program command, then it gives control to the programming task.

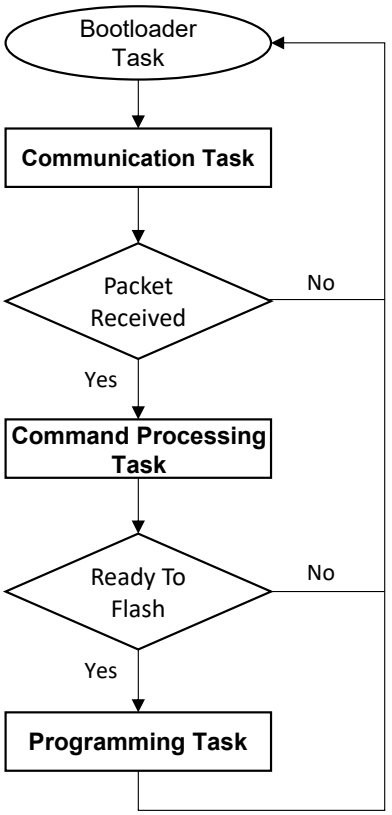
Programming Task

This task is responsible to program the internal Flash memory with a data packet received. It uses the Non-Volatile Memory (NVM) peripheral library to perform the unlock, erase, or write operations. It invokes the communication task in parallel to receive the next packet while waiting for the Flash operation to complete.

Flow Chart

The firmware upgrade execution flowchart is shown in the following figure.

Figure 2-2. Bootloader Framework Execution Flowchart



2.2 Modes of Operation

The bootloader communicates with the personal computer host application through a predefined communication protocol, for additional information refer to UART Bootloader Protocol).

The bootloader framework works in these two modes:

- Basic mode (Single-bank bootloader)
- Fail-safe Update mode (Dual-bank bootloader)

2.2.1 Basic Mode (single-bank bootloader)

The basic mode bootloader resides at the starting location of the Flash memory. It performs Flash erase, program, or verify operations on the binary sent from the host. Once the firmware upgrade and verification are completed, it jumps to the starting address of the application.

For a detailed explanation on the basic mode bootloader, refer to the documents specified in the References section.

2.2.2 Fail-Safe Update Mode (Dual-Bank Bootloader)

One of the challenges with a basic mode bootloader is the failure of the booting process. The booting process could fail during the firmware upgrade stage. The bootloader may not be able to complete the ongoing firmware upgrade due to several reasons, for example, interface disconnect, power cut, and so on. When the firmware upgrade process is aborted in between, the embedded device goes into an unstable state and may not work as expected.

A fail-safe bootloader overcomes the limitation of the basic bootloader. A fail-safe bootloader is designed on the premise that even if there is a firmware upgrade failure during the booting process, the system is still have a stable application image to run.

A fail-safe update is supported on devices which have the dual-bank Flash memory. Typically, memory in a microcontroller is organized in one or more banks. While most of the microcontrollers have single bank memory, there are some high-end microcontrollers that have dual bank. The dual-bank Flash memory enables the user to program one bank without affecting the application code of the other bank.

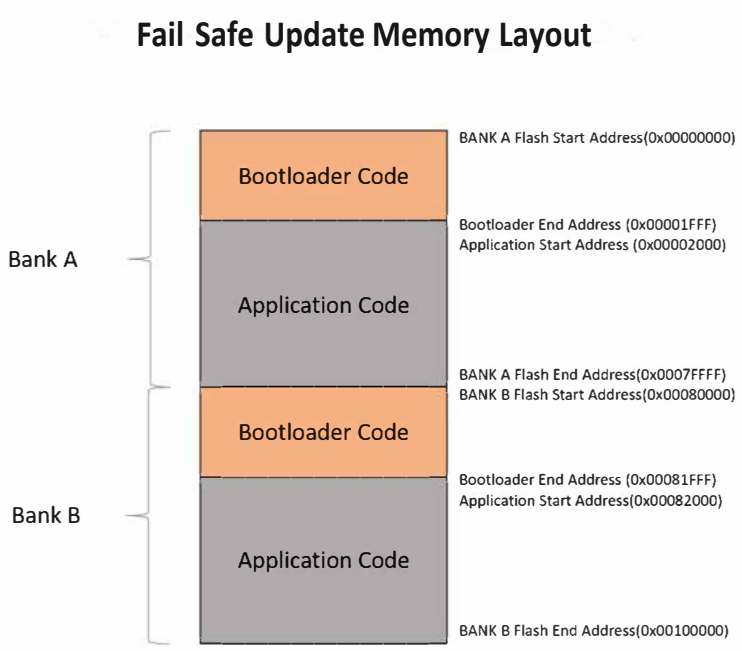
The boot failure situation is addressed by a dual-bank bootloader (Fail-safe update mode). With a dual-bank bootloader, whenever the device is running in one memory bank, the user can upgrade the firmware with the new features into the other bank and swap the firmware once the upgrade completes. If the upgrade process fails, the working copy of the firmware which is running in the first memory bank will help the device to work normally.

In a dual-bank bootloader the memory is distinguished into two banks. Each bank holds the bootloader code residing at the beginning location of the respective bank, and the firmware (application code) follows as shown in the following figure.

When booting from one bank, another bank is used as an upgrade buffer to accept the new firmware. After the new firmware is received and verified, the boot banks are switched. Therefore, there can be two workable firmware versions in the memory. The bootloader can perform a Flash operation in either of the banks based on the address sent by the host application. It performs a bank swap and resets the system to run the application programmed in the opposite bank after the verification is completed.

The following figure shows the memory layout of dual-bank bootloader:

Figure 2-3. Dual-Bank Bootloader Memory



The SAM E54 Flash memory is configured to two banks: Bank A and Bank B. At the start of both the banks, the bootloader is situated and then followed by an application image as shown in the figure above.

By default, Bank A is mapped to the address 0x00000000 and Bank B is mapped to the address 0x00080000. The bank mapped to the address 0x00000000 is referred to as the active bank (by default Bank A), whereas the other bank mapped to the address 0x00080000 is referred to as the inactive bank.

Note: The bank mapped at the address 0x00000000 is called as an active bank as the Cortex-M CPU architecture is designed to run the starting instruction from the address 0x00000000. Therefore, the code that needs to be run at reset needs to be mapped at 0x00000000.

The bootloader in an active bank can receive the following upgrade requests:

- Upgrade the active bank application image at address 0x00002000
- Upgrade the inactive bank application image at address 0x00082000
- Upgrade the inactive bank merged image (bootloader + application) at address 0x00080000

Upgrade Active Bank Application Image

The bootloader receives the application image from the host. Once the bootloader performs a successful upgrade, it notifies the host application. The host then sends a reset command to run the upgraded application, and examples are given below:

- This request to the bootloader is usually made when the device is loaded with the bootloader at the factory but not the application. This application upgrade request is made on the field when the user chooses this option for the first application image upgrade request to the device.
- This option could be used to upgrade certain metadata into the Flash memory. The metadata being added or upgraded is just a small part of the application image and therefore it does not require an upgrade to the whole memory region of the application to update the metadata.

Upgrade Inactive Bank Application or Merged Image

The bootloader, which is running from the active bank, receives the application or merged image from the host. Once the bootloader performs the successful upgrade, it notifies the host application. The host then sends the bank swap and system reset (BKSWRST) command. The BKSWRST performs the following actions:

- Swaps the memory banks to make the inactive bank active, and the active bank as inactive. BANK A is made inactive, while BANK B is made active.
- Issues a reset command to run the upgraded application

The information of which bank is mapped to the Flash address 0x00000000 is self-contained in special fuse bits in the Flash memory. These fuse bits can be erased or programmed individually. When the bootloader receives the BKSWRST command from the host, it sets the BKSWRST bit in the Flash (NVM) control register. When the BKSWRST is set, the Flash (NVM) controller swaps the banks and sets the Fuse bit (STATUS.AFIRST) based on the last status of the Fuse bit (STATUS.AFIRST) as given below:

- STATUS.AFIRST = 0; Start address of the Bank B is mapped to 0x00000000
- STATUS.AFIRST = 1; Start address of the Bank A is mapped to 0x00000000

On reset, the Flash (NVM) controller checks the status of the fuse bit (STATUS.AFIRST) and jumps to the active memory bank to run the code.

2.3 UART Bootloader Protocol

The bootloader firmware communicates with the personal computer host application by using a predefined communication protocol to exchange data between the Harmony v3 bootloader framework and the host.

The UART bootloader protocol comprises of a Guard, Data size, Command, and Data bytes as shown in the following figure.

Figure 2-4. Bootloader Protocol

The protocol details are as follows:

- **GUARD**
  - The Guard is a constant value: **0x5048434D**
  - This value provides protection against the spurious commands
  - Bootloader always checks for the Guard value at the start of packet reception, and proceeds further accordingly
- **Data Size**
  - This field indicates the number of data bytes to be received
  - This value varies for different commands
- **Command**
  - Indicates the command to be processed. Each command width is one Byte
  - The following commands are supported:
    - Unlock (0xA0)
    - Data (0xA1)
    - Verify (0xA2)
    - Reset (0xA3)
    - Bank Swap and reset (0xA4)
- **Data**
  - Contains the actual data to be processed based on the command
  - Length of the data to be received is indicated by a Data Size field
  - Bootloader receives the data in size of words (4 bytes)
  - All data words must be sent in a little-endian (LSB first) format

Response Codes

The bootloader will send a single character response code in response to each command. The sequential commands can only be sent after the response code is received for a previous command, or after a 100 ms timeout without a response.

The valid response codes are as follows:

- OK (0x50) – Command was received and processed successfully
- Error (0x51) – There were errors during the processing of the command
- Invalid (0x52) – Invalid command is received
- CRC OK (0x53) – CRC verification was successful
- CRC Fail (0x54) – CRC verification failed

2.4 Bootloader Trigger Methods

The bootloader can be invoked using these methods:

- The bootloader runs on every system reset, or when there is no valid firmware in the device. The bootloader continuously waits in a loop to receive the firmware from the host to upgrade.

The firmware is considered valid if the first word at the application start address is Bank A (0x00002000), and Bank B (0x00082000) is not 0xFFFFFFFF. Normally this word contains an initial stack pointer value, therefore it will never be 0xFFFFFFFF unless the device is erased. On a system reset, the bootloader checks whether a trigger to upgrade the firmware is present. If there is no valid trigger for the firmware upgrade, it tries to run the existing firmware. If there is no valid firmware, it jumps to a loop waiting to receive the firmware from the host.

- The bootloader provides a function `bootloader_Trigger()` which allows the user to upgrade the existing application. The `bootloader_Trigger()` function checks a switch press event or a pattern in the SRAM to know if there is a request to upgrade the existing application. The code for the `bootloader_Trigger()` function is shown below. This trigger function is called from the bootloader system initialization function.

```
#define BTL_TRIGGER_PATTERN  0x5048434D

static uint32_t *ramStart = (uint32_t *)BTL_TRIGGER_RAM_START;

bool bootloader_Trigger(void)
{
    uint32_t i;

    // Cheap delay. This should give at leat 1 ms delay.
    for (i = 0; i < 2000; i++)
    {
        asm("nop");
    }

    /* Check for Bootloader Trigger Pattern in first 16 Bytes of RAM to enter Bootloader.
*/
    if (BTL_TRIGGER_PATTERN == ramStart[0] && BTL_TRIGGER_PATTERN == ramStart[1] &&
    BTL_TRIGGER_PATTERN == ramStart[2] && BTL_TRIGGER_PATTERN == ramStart[3])
    {
        ramStart[0] = 0;
        return true;
    }

    /* Check for Switch press to enter Bootloader */
    if (SWITCH_Get() == 0)
    {
        return true;
    }

    return false;
}
```

The following methods can be used to upgrade the firmware while the application is running:

- **External Trigger:** While the application is running, the user presses the external system reset switch and a user switch simultaneously. The device resets and starts running the bootloader. Since the user switch is pressed, the `bootloader_Trigger()` function detects the switch press using the `SWITCH_Get()` function and returns true, indicating a firmware upgrade is requested. The bootloader takes care of receiving the data from the host and upgrades the device.
- **Software application Trigger:** If an application does not have an option for an external trigger or application requirement to upgrade the firmware based on specific commands, it can use a software trigger method to run the bootloader for the firmware upgrade.
- The application implements the `invoke_bootloader()` function. While the application is running, it intends to upgrade by itself. The application will call the `invoke_bootloader()` function and fill a dedicated area in the SRAM with a known pattern (0x5048434D) and issue a software reset. This prefilled SRAM pattern is compared in the `bootloader_trigger()` function, and if there is a match, the `bootloader_trigger()` function returns true indicating a firmware upgrade is requested. The bootloader takes care of receiving the data from the host and upgrades the device.

The following code can be used by the application to request the bootloader execution:

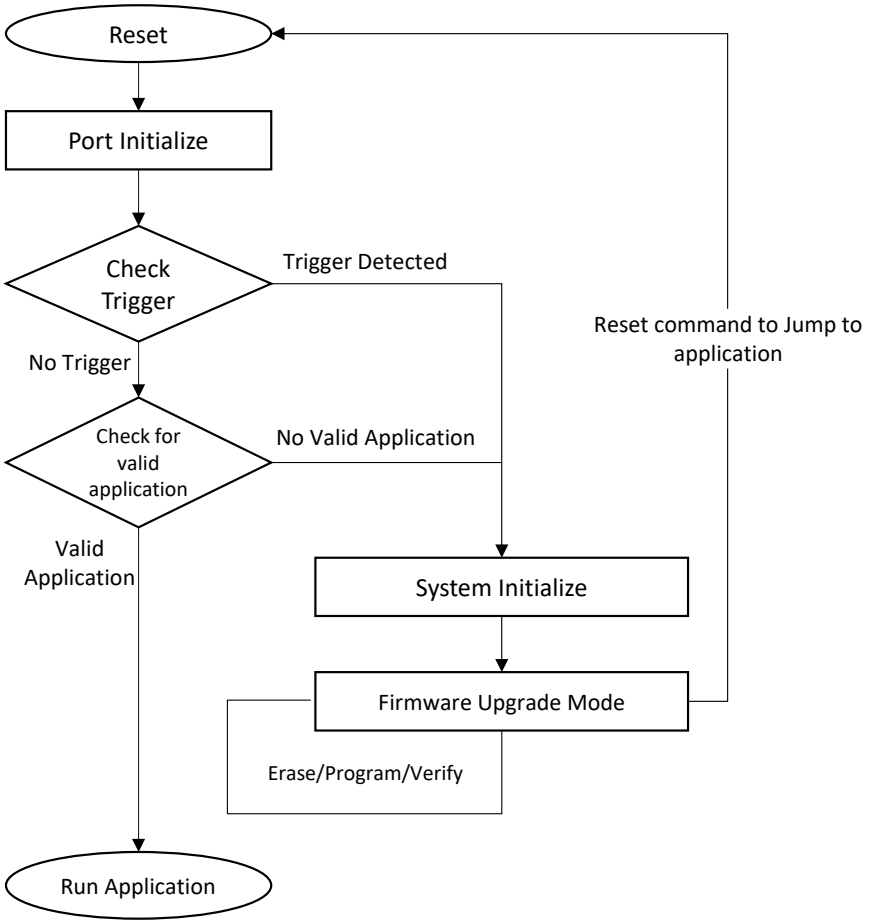
```
void invoke_bootloader(void)
{
    uint32_t *sram = (uint32_t *)BTL_TRIGGER_RAM_START;

    sram[0] = 0x5048434D;
    sram[1] = 0x5048434D;
    sram[2] = 0x5048434D;
    sram[3] = 0x5048434D;

    NVIC_SystemReset();
}
```

2.5 Bootloader System Level Execution Flow

Figure 2-5. Bootloader System Level Execution Flowchart



- Upon device reset, the bootloader initializes the system and port, then starts executing
- If no valid trigger is received from the user to upgrade the firmware, the bootloader starts executing the user application if the user application is already present
- If a trigger is valid, the bootloader initializes the system, upgrades the firmware and issues a reset BKSWRST command to run the upgraded application

3. Configuring the Dual-Bank Bootloader

The dual-bank bootloader is referred to as the UART fail-safe bootloader in MPLAB Harmony v3, and it comprises of these applications:

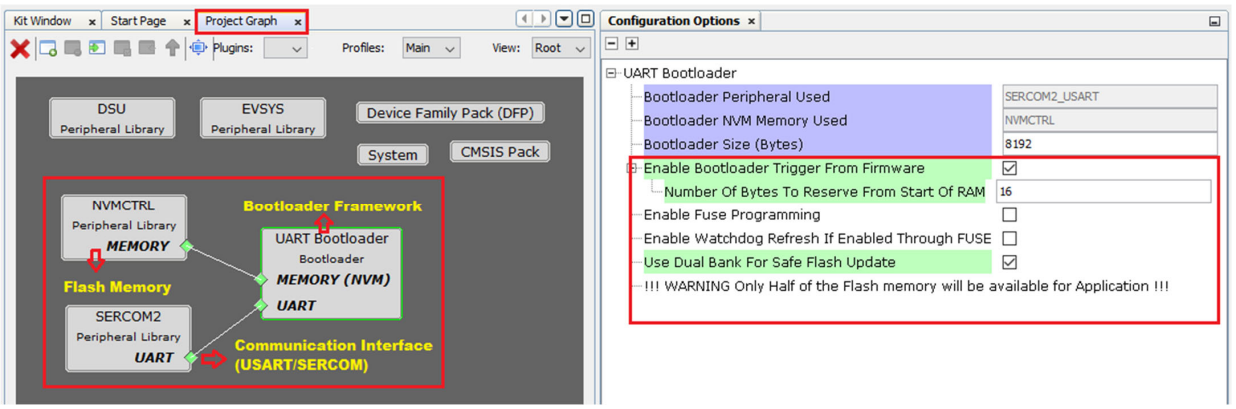
- **Bootloader:** `uart_fail_safe_bootloader_sam_e54_xpro` is the bootloader code, which performs upgrading of the firmware.
- **Test application:** `uart_fail_safe_bootloader_test_app_sam_e54_xpro` is the user application code.

Note: Projects are available in the `bootloader_apps_uart` MPLAB Harmony v3 repository, users can download it from the following path: *<Harmony framework>\bootloader_apps_uart\apps\uart_fail_safe_bootloader*.

Configuring the bootloader library in MPLAB Harmony v3

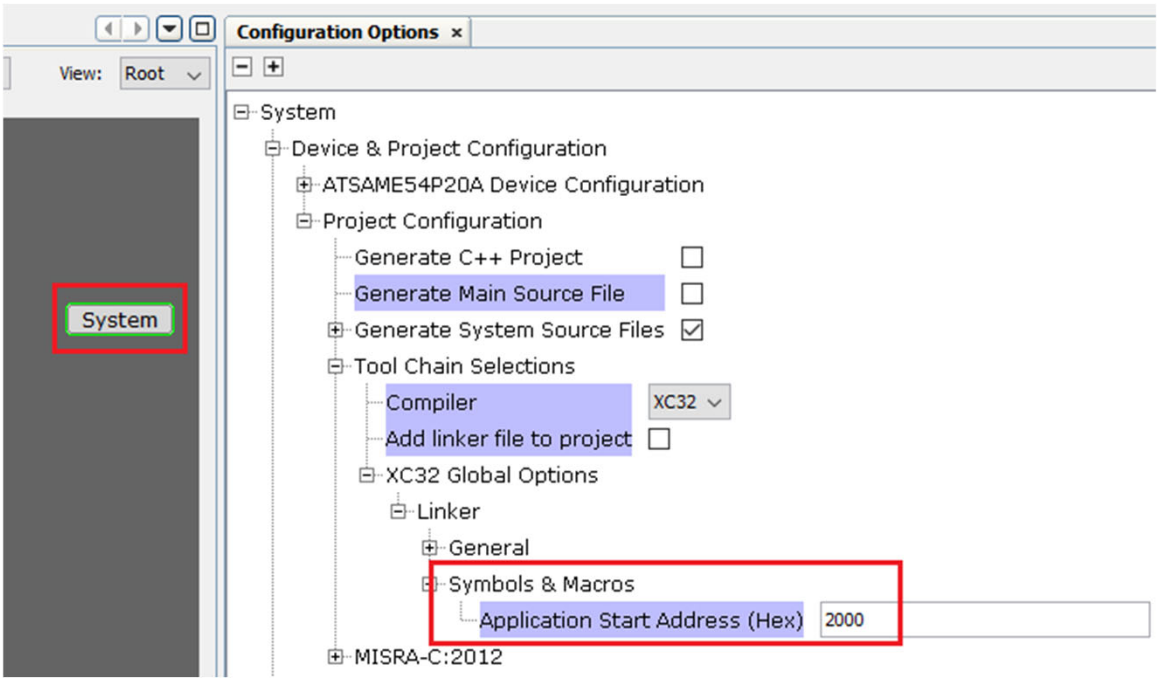
- **Use Dual-Bank for Safe Flash Update:**
  - Can be used to configure the bootloader to use the dual banks of the device to upload the application.
- **Bootloader Peripheral Used:**
  - Specifies the communication peripheral used by the bootloader to receive the application in this case, it is the serial communication (SERCOM) or USART.
- **Bootloader Memory Used:**
  - Specifies the memory peripheral used by the bootloader to perform Flash operations.
- **Bootloader Size (Bytes):**
  - Specifies the maximum size of Flash required by the bootloader.
  - This size is calculated based on the bootloader type and memory used.
  - This size will vary from device to device and must always be aligned with the device erase unit size.
- **Enable Bootloader Trigger from Firmware:**
  - This option can be used to force trigger the bootloader from the application firmware after a soft reset.
- **Number of Bytes to Reserve from Start of RAM:**
  - This option adds the provided offset to the RAM start address in the bootloader linker script.
  - The application firmware can store a pattern in the reserved bytes region of the RAM start for the bootloader to check at reset in the `bootloader_Trigger()` function.

Figure 3-1. Bootloader Configuration



- **Application Start Address (Hex):**
  - Start address of the application is programmed by the bootloader.
  - The application start address is auto filled by the MCC when the user configures the bootloader size as shown in the previous figure. This value will be equal to the bootloader size (size of bootloader = 8K (0x00002000)).
  - This value will be used by the bootloader to jump to the application at device reset.

Figure 3-2. Application Start Address Configuration



3.1 Bootloader Linker Script

The bootloader library uses a custom linker (btl.ld) script generated through the MCC. The MCC generates the specified bootloader size, ROM (Read-only memory) and RAM (Random-access memory) address as highlighted in the following figure.

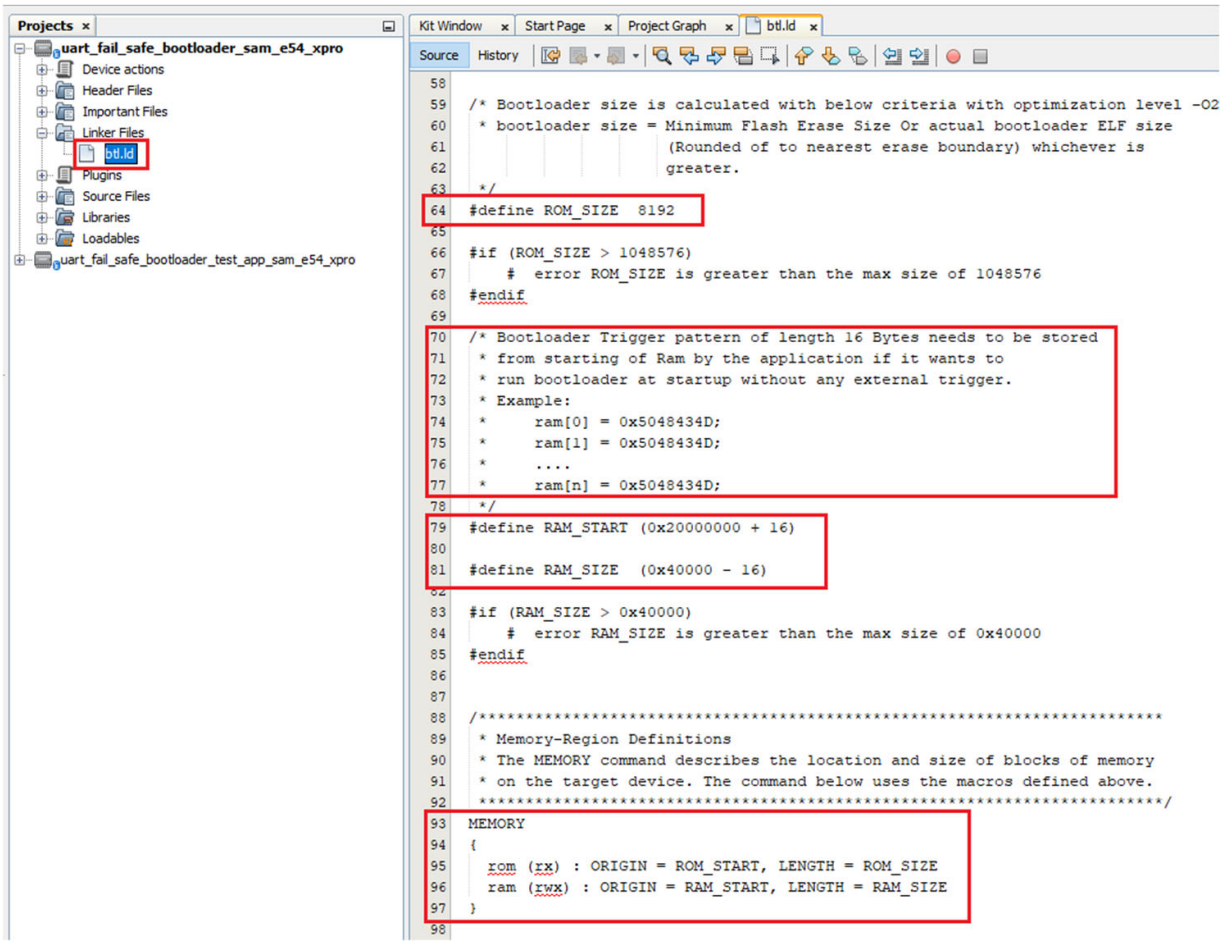
The values populated in the linker script are based on the bootloader component of the MCC configurations (bootloader configuration).

Configure the Linker script for the bootloader to run from the RAM to achieve the simultaneous Flash memory write and reception of the next block of data.

The bootloader request pattern must be stored in 16 Bytes of RAM on start by the application if it wants to run the bootloader at startup without any external trigger as shown in the following figure.

The bootloader size for the SAM E54 will be rounded off to the nearest erase unit size (8192 Bytes), even though the size of the bootloader is 1672 Bytes in -O1 optimization. This helps for the addition of additional features on the bootloader, and to avoid application overlap with the bootloader.

Figure 3-3. Bootloader Linker Script



Note: Users need to ensure that the memory region of the user application does not overlap with the memory region reserved for the bootloader.

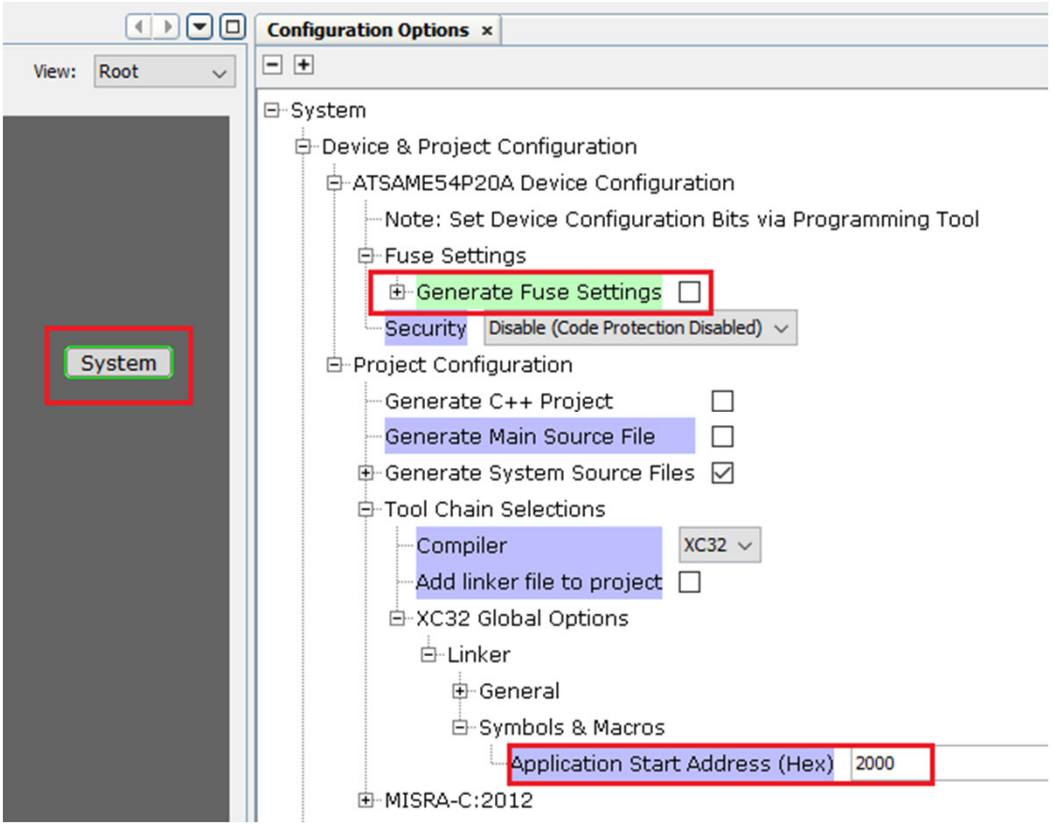
3.2 Test Application Configurations

- **Disable Generate Fuse Settings:**
  Generally, fuse configuration settings are programmed through the programming tool. In the reference application discussed in this document, the fuse settings are disabled due to the application being programmed through the bootloader.

  Note: The fuse settings are not programmable through firmware. Enabling the fuse settings increases the size of the binary when generated through the Hex file.

- **Application Start Address (Hex):**
  - Start address of the application.
  - The application start address value must be equal to or greater than the Flash base address + bootloader size.
  - An application start address value will be used by the bootloader to jump to the application at device reset. It must match the value provided to the bootloader code during generation as shown in Application Start Address Configuration.
  - The application start address will be used to generate the MPLAB XC32 compiler settings to place the code at the intended address as shown in the following figure and Test Application Project Settings.

Figure 3-4. Test Application Configuration



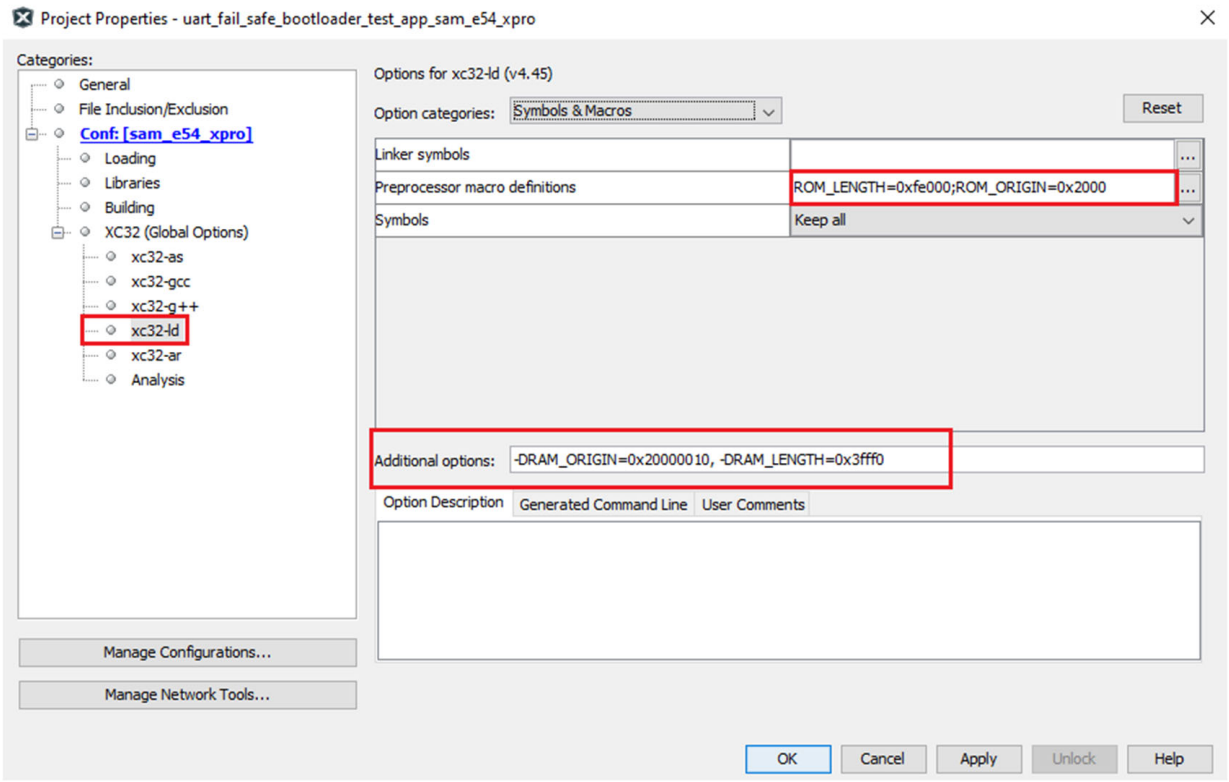
3.3 Test Application Project Settings

- **Preprocessor Macro Definitions:**
  - ROM-ORIGIN and ROM_LENGTH are the XC32 linker variables which will be overridden with values provided here.
  - Application start address value is auto populated in the linker script with the value of the application start address provided in the MCC (bootloader linker script) after regeneration.
- **Additional Options:**
  - RAM_ORIGIN and RAM_LENGTH values must be provided by reserving 16 bytes of start of RAM to trigger the bootloader from the firmware.

- This is optional and can be ignored if not required to soft trigger the bootloader.

```
Custom linker options: -DRAM_ORIGIN=0x20000010, -DRAM_LENGTH=0x3fff0
```

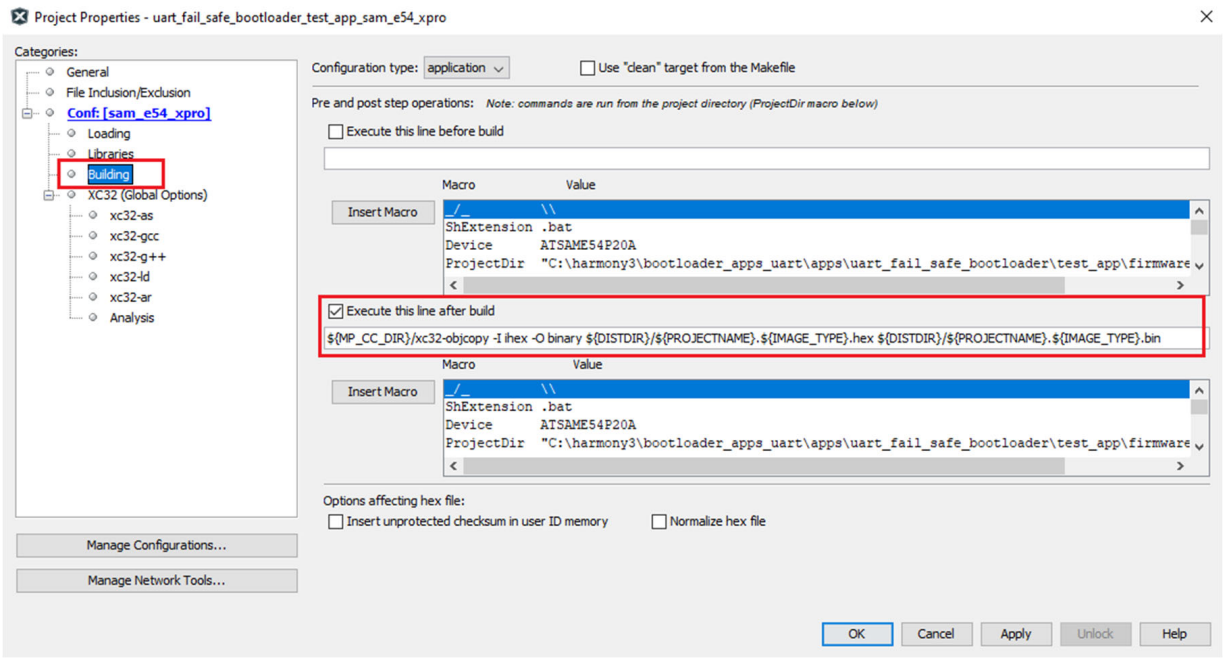
Figure 3-5. Test Application Project Settings



- **Execute the line after Build:**
  - This option can be used to automatically generate the binary file from the Hex file after the build is complete.

```
Custom linker options: "${MP_CC_DIR}/xc32-objcopy -I ihex -O binary ${DISTDIR}/$
{PROJECTNAME}.${IMAGE_TYPE}.hex ${DISTDIR}/${PROJECTNAME}.${IMAGE_TYPE}.bin"
```

Figure 3-6. Test Application Binary Generation Settings



4. Running the Demonstration

4.1 Running the Bootloader Application

Follow these steps to run the bootloader application:

1. Connect a micro USB cable to the DEBUG port of the SAM E54 Xplained Pro board.
2. Build and program the UART fail safe bootloader (dual-bank bootloader) using the MPLAB X IDE.
3. Launch the MCC for the UART fail safe bootloader application:
   a. Disable the Fuse Settings as shown in the Test Application Configurations section.
   b. Regenerate the code.
   c. Enable and execute this line after build option in the MPLAB X IDE Project Properties as shown in the Test Application Configurations section Figure 3-5.
4. Build the bootloader application (uart_fail_safe_bootloader_sam_e54_xpro) again using the MPLAB X IDE.
   – This is required to generate the binary file for the bootloader application.
5. Build the Test application (uart_fail_safe_bootloader_test_app_sam_e54_xpro) using the MPLAB X IDE but do not program.
6. Run `btl_app_merge_bin.py` from the command prompt to merge the generated bootloader binary and the application binary. The following output must be displayed on the command prompt.

```
Command: python <python script> -o <Offset> -b bootloader image -a application image

<Offset>: Application start address (E.g. - 0x00002000)
<python script>: btl_app_merge_bin.py

Example: python <harmony3_path>\bootloader\tools\btl_app_merge_bin.py -o 0x00002000 -b
<harmony3_path>\bootloader\apps\uart_fail_safe_bootloader\bootloader\firmware\sam_e54_xpro.
X\dist\sam_e54_xpro\production
\sam_e54_xpro.X.production.bin -a
<harmony3_path>\bootloader\apps\uart_fail_safe_bootloader\test_app\firmware\sam_e54_xpro.X\
dist\sam_e54_xpro\production
\sam_e54_xpro.X.production.bin
```

Figure 4-1. Bootloader and Application Binary Merger Output

```
C:\Users\[redacted]>python D:\bootloader\tools\btl_app_merge_bin.py -o 0x00002000 -b C:\harmony3\bootloader_apps_uart\app
s\uart_fail_safe_bootloader\bootloader\firmware\sam_e54_xpro.X\dist\sam_e54_xpro\production\sam_e54_xpro.X.production
.bin -a C:\harmony3\bootloader_apps_uart\apps\uart_fail_safe_bootloader\test_app\firmware\sam_e54_xpro.X\dist\sam_e54
_xpro\production\sam_e54_xpro.X.production.bin

##### Merged Bootloader and Application binaries to  btl_app_merged.bin #####
```

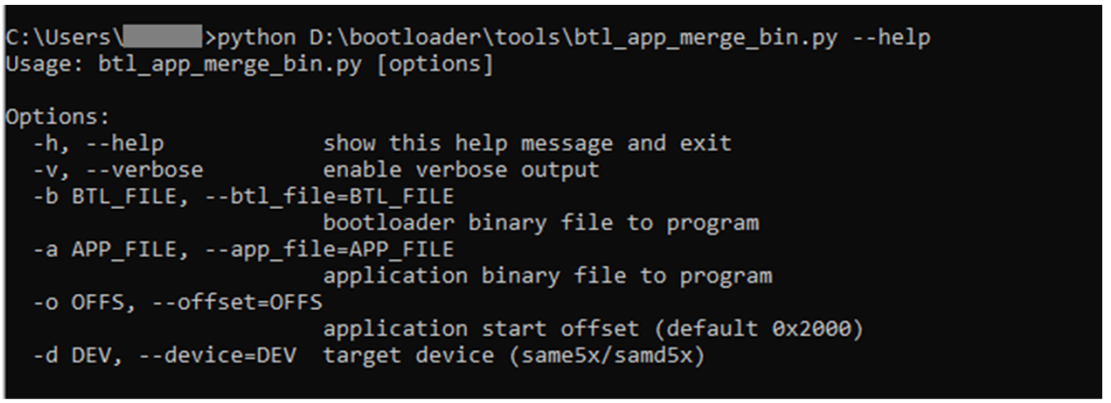
```
Note: Running the help command provides a brief overview of options available as shown
below.

Command: python <python script> --help

<python script>: btl_app_merge_bin.py

Example: python <harmony3_path>\bootloader\tools\btl_app_merge_bin.py --help
```

Figure 4-2. Application Binary Merge Help Window



7. Run `btl_host.py` from the command prompt to program the merged binary to the opposite panel. The merged binary `btl_app_merged.bin` will be generated in the path from where `btl_app_merge_bin.py` was executed.

```
Command: python <python script> -v -s -i <COM PORT> -d <Device Name> -a <Address> -f
<bootloader_application_merged_image>

<python script>: btl_host.py
<COM PORT>: Serial communication port
<Device Name>: SAME54
<Address>: Application start address (Bank A: 0x00002000 / Bank B: 0x00080000)

Example: python <harmony3_path>\bootloader\tools\btl_host.py -v -s -i COM18 -d same5x -a
0x00080000 -f btl_app_merged.bin
```

```
Note: Running the help command provides a brief overview of options available as shown
below.

Command: python <python script> --help

<python script>: btl_host.py

Command: python <harmony3_path>\bootloader\tools\btl_host.py --help
```

Figure 4-3. Application Bootloader Host Help Window

```
C:\Users\[redacted]>python D:\bootloader\tools\btl_host.py --help
Usage: btl_host.py [options]

Options:
  -h, --help            show this help message and exit
  -v, --verbose         enable verbose output
  -r BAUD, --baud=BAUD  UART baudrate
  -u PARITY, --parity=PARITY
                        UART Parity (none/even/odd)
  -t, --tune            auto-tune UART baudrate
  -i PATH, --interface=PATH
                        communication interface
  -f FILE, --file=FILE  binary file to program
  -c DEVCFGFILE, --devcfgfile=DEVCFGFILE
                        device configuration text file
  -a ADDR, --address=ADDR
                        destination address
  -p SectSize, --sectorSize=SectSize
                        Device Sector Size in Bytes
  -b, --boot            enable write to the bootloader area
  -s, --swap            swap banks after programming
  -d DEV, --device=DEV  target device (samc2x/samd1x/samd2x/samd5x/samda1/same
                        7x/same5x/samg5x/saml2x/samha1/sama5/sama7/sam9x6/sam9
                        x7/pic32cz/pic32mk/pic32mx/pic32mz/pic32mzw/pic32cm/pi
                        c32mm/wbz451/pic32cxbz2/wbz45x)
```

8. The following figure shows the example output of the firmware programming.

Figure 4-4. Firmware Upgrade Output

```
C:\Users\[redacted]>python D:\bootloader\tools\btl_host.py -v -s -i COM30 -d same5x -a 0x00080000 -f btl_app_merged.bin

Reading Bootloader Version

Bootloader version : v3.7

Unlocking

Programming: |||||||||||||||||||||||||||||||||||||||||||||||||| 100.0% Complete

Verification

... success

Swapping Bank And Rebooting

Reboot Done
```

Notes:

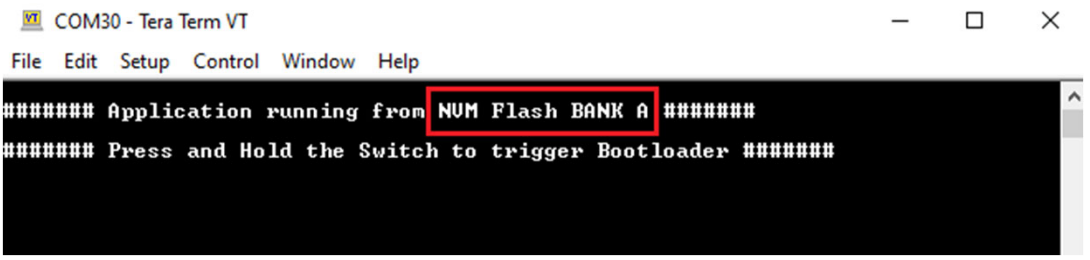
1. After the successful programming of the application, Bank B is mapped to the 0x00000000 address, and Bank A is mapped to the 0x00080000 address.
2. The starting address for the firmware always remains the same, due to the banks being swapped after each firmware upgrade.

4.2 Running the Test Application

1. Perform the Running the Bootloader Application steps for the UART fail-safe bootloader application if not done already.
2. If the above step is successful, then the LED0 on the SAM E54 Xplained Pro board must start blinking.
3. Open the Terminal application (for example, Tera Term) on the computer.

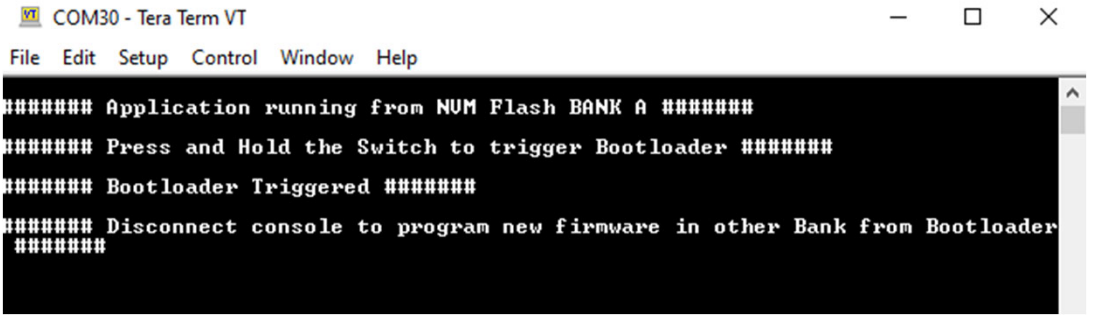
4. Configure the Serial Port settings as follows:
   - **Baud:** 115200
   - **Data:** 8 Bits
   - **Parity:** None
   - **Stop:** 1 Bit
   - **Flow Control:** None
5. Reset or power cycle the device.
6. The LED must start blinking, and the following output will display on the console:
   - The NVM Flash Bank Can be BANK A or BANK B based on where the program is running.

Figure 4-5. Application Running on BANK A



7. Press and hold Switch SW0 to trigger the bootloader to program the firmware in the other Bank and the following output will be displayed on the console.

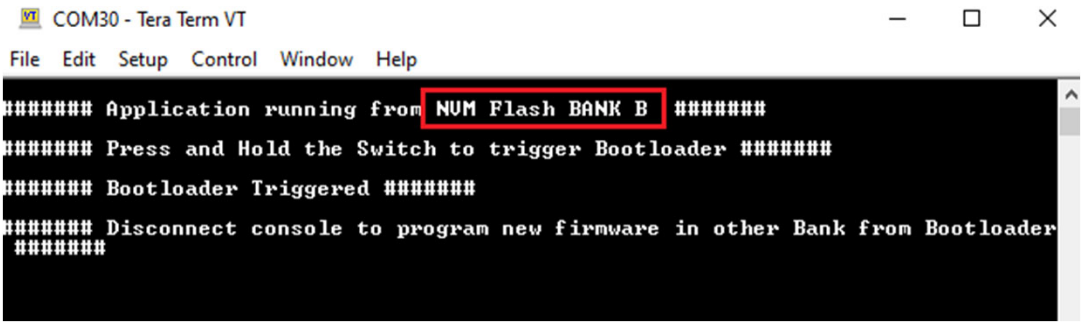
Figure 4-6. Application Triggered to Enter Bootloader



Repeat Steps 6 to 8 provided in the section Running the Bootloader Application to switch to the Banks.

- This step is used to verify that the bootloader is running after triggering the bootloader from the *Test Application*, and to program the new firmware in opposite Bank.
- Observe the change in Bank displayed in the Test Application console as compared to the first run as shown in the following figure:

Figure 4-7. Application Running on BANK B



5. References

- For a detailed explanation on the bootloader, refer to *<Harmony path>\bootloader\docs\index.html*
- MPLAB Harmony GitHub:
  github.com/Microchip-MPLAB-Harmony
- How to Setup MPLAB Harmony v3 Software Development Framework:
  ww1.microchip.com/downloads/aemDocuments/documents/MCU32/ProductDocuments/SupportingCollateral/How_to_Setup_MPLAB_Harmony_v3_Software_Development_Framework_DS90003232C.pdf
- Getting Started with MPLAB Harmony v3 Peripheral Libraries on SAM D5x/E5x MCUs:
  developerhelp.microchip.com/xwiki/bin/view/software-tools/harmony/archive/same54-getting-started-training-module/
- MPLAB Harmony v3 landing page:
  www.microchip.com/en-us/tools-resources/configure/mplab-harmony
- Clock System Configuration and Usage on SAM E5x (Cortex M4) Devices:
  ww1.microchip.com/downloads/aemDocuments/documents/MCU32/ProductDocuments/SupportingCollateral/Clock-System-Configuration-and-Usage-on-SAM-E5x-Devices-DS90003226.pdf
- MPLAB® Harmony 3 Bootloader Module:
  https://github.com/Microchip-MPLAB-Harmony/bootloader
- MPLAB® Harmony Bootloader Application Examples for UART:
  github.com/Microchip-MPLAB-Harmony/bootloader_apps_uart
- For additional information about 32-bit Microcontroller Collaterals and Solutions, refer to:
  DS70005534: 32-bit Microcontroller Collateral and Solutions Reference Guide
- For other relevant information, refer to the Microchip web site.
  www.microchip.com/
- SAM E54 Xplained Pro Evaluation Kit product page:
  www.microchip.com/en-us/development-tool/ATSAME54-XPRO
- SAM E54 Xplained Pro User's Guide:
  ww1.microchip.com/downloads/aemDocuments/documents/OTH/ProductDocuments/UserGuides/70005321A.pdf

6. Revision History

Revision C - 11/2024

The following updates were performed to the content in this revision:

- All references to the MHC have been updated throughout the document to the MCC to reflect a Tools change
- Updated Tools Version number in MPLAB®X Integrated Development Environment (IDE) and XC Compilers
- Updated Tool naming and version numbers in MPLAB Harmony v3
- Updated the version numbering in Python
- Updated obsolete code in Bootloader Trigger Methods
- Updated or Replaced images in the following sections:
  - Configuring the Dual-Bank Bootloader
  - Bootloader Linker Script
  - Test Application Configurations
  - Test Application Project Settings
  - Running the Bootloader Application
  - Running the Test Application
- Replaced all links in References with updated links

Revision B - 09/2022

Numerous editorial updates were performed throughout this document.

The following updates were performed to the content in this revision:

- Added a new reference and link to the UART bootloader in MPLAB Harmony v3
- Updated the links for the UART bootloader in Configuring the Dual Bank Bootloader
- Added a new reference for the UART bootloader in References

Revision A - 06/2020

This is the initial release of this document.

Microchip Information

Trademarks

The “Microchip” name and logo, the “M” logo, and other names, logos, and brands are registered and unregistered trademarks of Microchip Technology Incorporated or its affiliates and/or subsidiaries in the United States and/or other countries (“Microchip Trademarks”). Information regarding Microchip Trademarks can be found at https://www.microchip.com/en-us/about/legal-information/microchip-trademarks.

ISBN: 979-8-3371-0180-4

Legal Notice

This publication and the information herein may be used only with Microchip products, including to design, test, and integrate Microchip products with your application. Use of this information in any other manner violates these terms. Information regarding device applications is provided only for your convenience and may be superseded by updates. It is your responsibility to ensure that your application meets with your specifications. Contact your local Microchip sales office for additional support or, obtain additional support at www.microchip.com/en-us/support/design-help/client-support-services.

THIS INFORMATION IS PROVIDED BY MICROCHIP “AS IS”. MICROCHIP MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND WHETHER EXPRESS OR IMPLIED, WRITTEN OR ORAL, STATUTORY OR OTHERWISE, RELATED TO THE INFORMATION INCLUDING BUT NOT LIMITED TO ANY IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE, OR WARRANTIES RELATED TO ITS CONDITION, QUALITY, OR PERFORMANCE.

IN NO EVENT WILL MICROCHIP BE LIABLE FOR ANY INDIRECT, SPECIAL, PUNITIVE, INCIDENTAL, OR CONSEQUENTIAL LOSS, DAMAGE, COST, OR EXPENSE OF ANY KIND WHATSOEVER RELATED TO THE INFORMATION OR ITS USE, HOWEVER CAUSED, EVEN IF MICROCHIP HAS BEEN ADVISED OF THE POSSIBILITY OR THE DAMAGES ARE FORESEEABLE. TO THE FULLEST EXTENT ALLOWED BY LAW, MICROCHIP’S TOTAL LIABILITY ON ALL CLAIMS IN ANY WAY RELATED TO THE INFORMATION OR ITS USE WILL NOT EXCEED THE AMOUNT OF FEES, IF ANY, THAT YOU HAVE PAID DIRECTLY TO MICROCHIP FOR THE INFORMATION.

Use of Microchip devices in life support and/or safety applications is entirely at the buyer’s risk, and the buyer agrees to defend, indemnify and hold harmless Microchip from any and all damages, claims, suits, or expenses resulting from such use. No licenses are conveyed, implicitly or otherwise, under any Microchip intellectual property rights unless otherwise stated.

Microchip Devices Code Protection Feature

Note the following details of the code protection feature on Microchip products:

- Microchip products meet the specifications contained in their particular Microchip Data Sheet.
- Microchip believes that its family of products is secure when used in the intended manner, within operating specifications, and under normal conditions.
- Microchip values and aggressively protects its intellectual property rights. Attempts to breach the code protection features of Microchip products are strictly prohibited and may violate the Digital Millennium Copyright Act.
- Neither Microchip nor any other semiconductor manufacturer can guarantee the security of its code. Code protection does not mean that we are guaranteeing the product is “unbreakable”. Code protection is constantly evolving. Microchip is committed to continuously improving the code protection features of our products.