# SAMD21 ARM Cortex-M0+ Microcontroller & Harmony v3

## SAM2002ADV v1.01


A Leading Provider of Smart, Connected and Secure Embedded Control Solutions


**CAE Taiwan Team**
**Microchip Technology Inc.**

October 2025

# SAM series course history and brief

- **SAM1001 – MCU32 Peripheral Library (PLIB)**
  - (2017~2018)
    - Peripheral Library (PLIB) on APP045v2.0 (SAMD21 Arm Cortex-M0+)
    - Advance Software Framework (ASF) v3.x + GNU C compiler + Microchip Studio 7
  - (2019~2022)
    - Peripheral Library (PLIB) on APP045v2.0/v4.x (SAMD21 Arm Cortex-M0+)
    - MPLAB Harmony Framework v3.x + XC32 Compiler + MPLAB X IDE
  - (2023~2024)
    - Peripheral Library (PLIB) on APP045v4.x/v5.x (SAMD21 Arm Cortex-M0+)
    - MCC Harmony + XC32 Compiler + MPLAB X IDE
- **SAM2001ADV – MCU32 Advanced Peripheral Library (PLIB)**
  - (2021~2022)
    - Peripheral Library (PLIB) Advanced modules on APP045v4.x (SAMD21 Arm Cortex-M0+)
    - MPLAB Harmony Framework v3.x + XC32 Compiler + MPLAB X IDE
  - (2023~2024)
    - Peripheral Library (PLIB) Advanced modules on APP045v4.x/v5.x (SAMD21 Arm Cortex-M0+)
    - MCC Harmony + XC32 Compiler + MPLAB X IDE
- **SAM2002 – MCU32 System Service and Driver**
  - (2019~2022)
    - System Service and Driver on APP045v4.x (SAMD21 Arm Cortex-M0+)
    - MPLAB Harmony Framework v3.x + XC32 Compiler + MPLAB X IDE
  - (2023~2024)
    - System Service and Driver on APP045v4.x/v5.x (SAMD21 Arm Cortex-M0+)
    - MCC Harmony + XC32 Compiler + MPLAB X IDE
- **SAM2002ADV – MCU32 Touch and USB Device**
  - (2021~2022)
    - QTouch Library and USB Device Driver on APP045v4.x (SAMD21 Arm Cortex-M0+)
    - MPLAB Harmony Framework v3.x + XC32 Compiler + MPLAB X IDE
  - (2023~2024)
    - QTouch Library and USB Device Driver on APP045v4.x (SAMD21 Arm Cortex-M0+)
    - MCC Harmony + XC32 Compiler + MPLAB X IDE

# Background Knowledge and Preparation

- **SAM2002ADV is for advanced users, then we will not introduce :**
  - MPALB X IDE and Harmony Configuration. (Please study from SAM2001 course)
  - How to download(prepare) latest Harmony Framework. (Please study from SAM2001 course)
  - How to create a new project. (Please study from SAM2001 course)
  - PLIB (Peripheral Library) implementation. (Please study from SAM2001 and SAM2001ADV course)
  - System Service and Driver implementation. (Please study from SAM2002 course)

- **SAM2002ADV will use below tool versions for Labs.**
  - MPLAB® X IDE v6.15 https://www.microchip.com/mplab/mplab-x-ide
  - MPLAB® XC32 v4.35 https://www.microchip.com/mplab/compilers
  - MPLAB® Code Configurator v5.4.1 (Install in MPLAB X IDE Plug-in)
  - SAMD21 DFP v3.6.144 (Device Family Pack download from X IDE \Tools\Packs)
  - CMSIS v5.8.0 (Common Microcontroller Software Interface Standard)
  - MCC Core v5.6.1 (Install with MPLAB X IDE v6.15)
  - MPLAB® MCC Harmony Framework (From X IDE MCC Harmony Content Manager)
    - Required Content (Must install)
      - csp 3.18.4
      - harmony-services 1.3.2
      - CMSIS_5 5.9.0
      - quick_docs 1.6.0
    - Optional Content (Manual select in Content Manager)
      - touch v3.15.0
      - usb v3.12.0
      - dev_packs v3.18.1
      - core v3.13.3

# Advance Course

- Introduction to QTouch® Peripheral Touch Controller (PTC)
- Harmony Touch Configurator
- APP045 EVB QTouch® Design and Pattern
  - Lab1 Touch Buttons
  - Lab2 Touch Slider
  - Lab3 Data Visualizer for Touch
- USB Fundamentals
- USB Device Library (USB Device Architecture)
- Harmony USB Device - Device Layer Configurator
  - Lab4 USB Device - Device Layer Initialize
- USB Device CDC Class
  - Lab5 USB Device CDC Class - Basic TX and RX
- USB Device HID Class
  - Lab6 USB Device HID Class - Basic TX and RX

# Course Coverage

## Course will not include :

- The principle of Capacitive Touch.
- Pattern design for Capacitive Touch.
- The principle of our PTC/CVD.
- PTC tuning parameters.

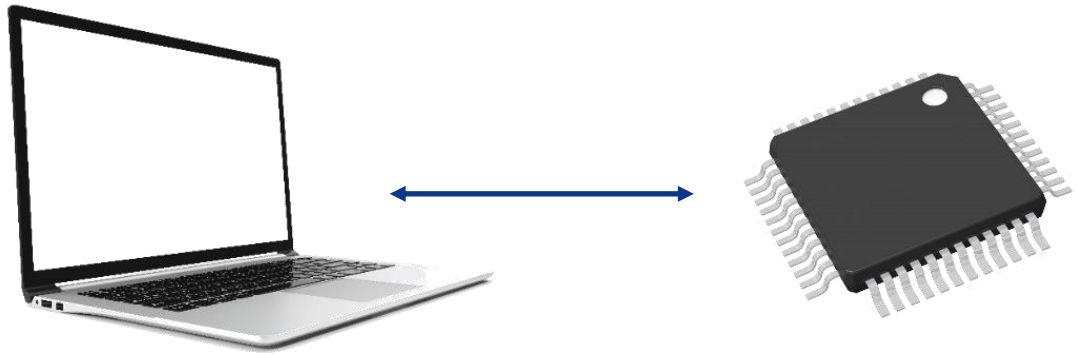
## Course will include :

- Usage of MCC Harmony Touch library.
- Introduction to the MCC Harmony Touch Configurator.
- Touch visualization with MPLAB Data Visualizer.

# Bootloader Fundamentals

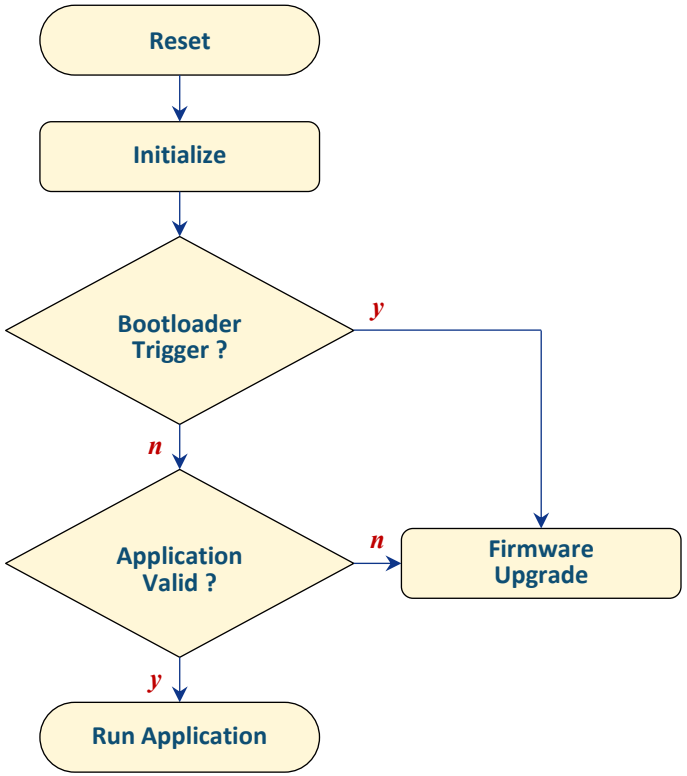
# What’s a Bootloader ?

- **Bootloader is a small application that starts the operation of the device, that can be used to upgrade firmware on a target device without the need for an external programmer/debugger.**
- **Why firmware update ?**
  - Add/remove functionality
  - Update interface
  - Parameter update
  - Any upgradable function in firmware



# Control Flow at Reset

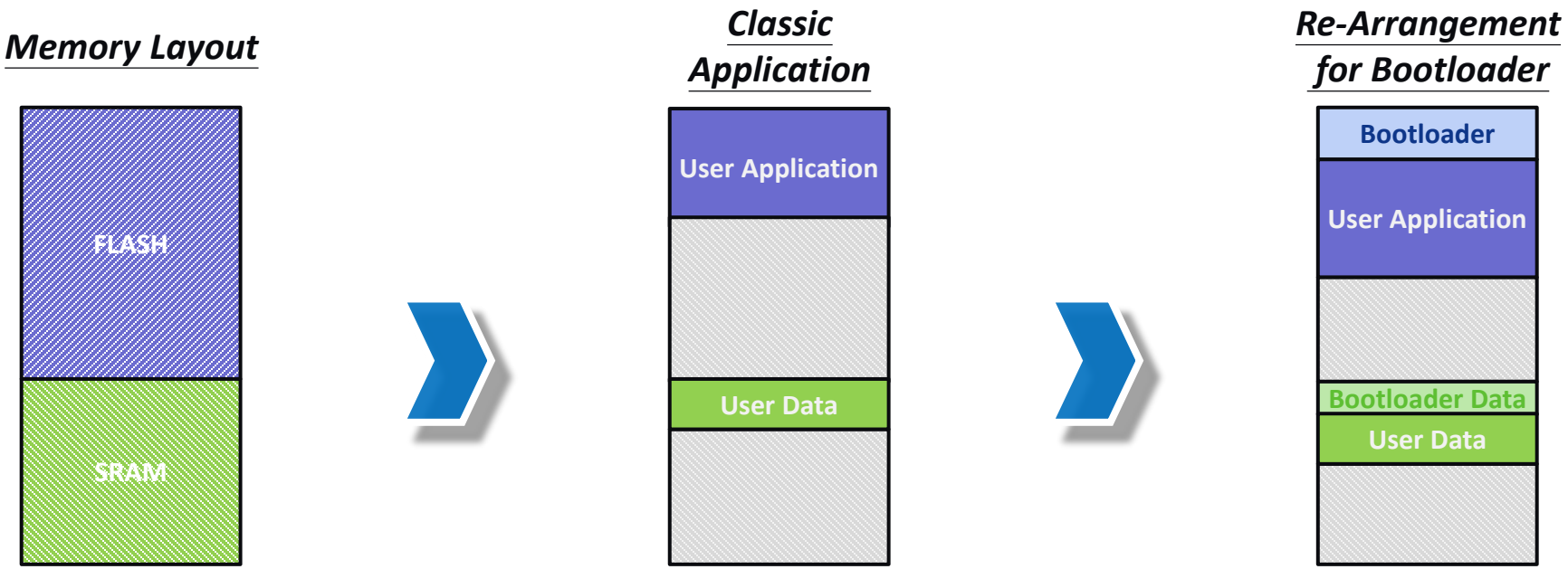
- The flowchart below outlines the startup handoff between the Bootloader and the Application.
- After reset, the Bootloader takes control first and evaluates predefined trigger conditions to decide whether to enter firmware-update mode or transfer control to the Application.
- The Bootloader and Application occupy separate Flash and SRAM regions and do not interfere with each other. Each can maintain an independent VT (Vector Table)
- The device Configuration bits/Fuses are the only shared resource.



# Memory Reallocation

# Memory Reallocation for Bootloader

- In a typical system without a Bootloader, the Application owns the entire memory space. When a Bootloader is present, portions of Flash and SRAM must be reserved for both the Bootloader and the Application.
- Use two independent linker scripts(*.*ld*) to map non-overlapping memory regions and prevent conflicts.



# Linker Script Design

- **The example below uses two independent linker scripts (*.*ld*),**
  **one for the Bootloader and one for the Application to define their memory allocations.**
- **In the linker scripts,**
  **you can see the Flash(*rom (LRX)*) and SRAM(*ram (WX!R)*)**
  **reserved separately for the Bootloader and the Application.**

***Bootloader***

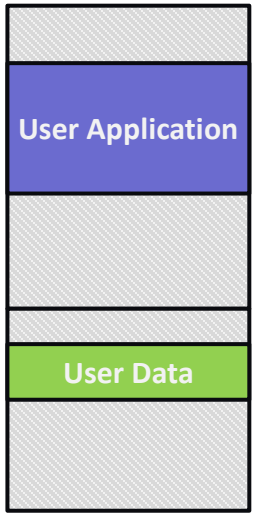

***Linker Script for Bootloader***

```
MEMORY
{
  rom (LRX) : ORIGIN = 0, LENGTH = ROM_LENGTH_BOOT
  ram (WX!R) : ORIGIN = 0, LENGTH = RAM_LENGTH_BOOT
  config_00804000 : ORIGIN = 0x00804000, LENGTH = 0x4
  config_00804004 : ORIGIN = 0x00804004, LENGTH = 0x4
}
```

***Linker Script for Application***

```
MEMORY
{
  rom (LRX) : ORIGIN = ROM_ORIGIN_APP, LENGTH = ROM_LENGTH_APP
  ram (WX!R) : ORIGIN = RAM_ORIGIN_APP, LENGTH = RAM_LENGTH_APP
  config_00804000 : ORIGIN = 0x00804000, LENGTH = 0x4
  config_00804004 : ORIGIN = 0x00804004, LENGTH = 0x4
}
```

***Application***




MICROCHIP

# Bootloader Trigger

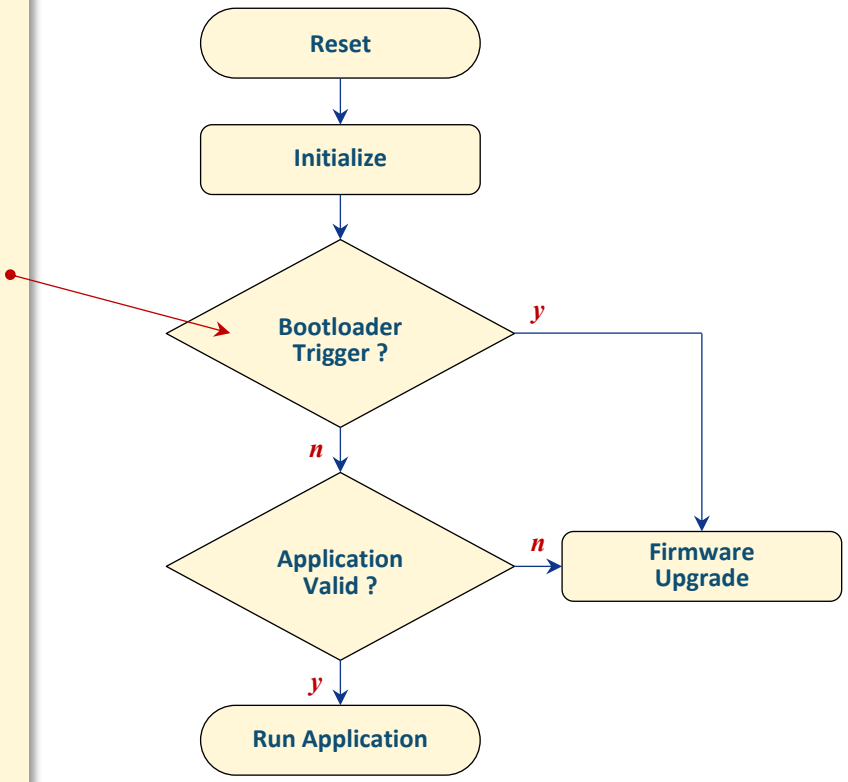
# Bootloader Trigger Mechanisms

- Bootloader should be triggered through specific mechanisms, e.g.,
  - Hardware I/O
    Check a button or GPIO state at startup to decide whether to enter the bootloader.
  - Firmware State
    Validate firmware version or markers; if they mismatch or
    the Application looks invalid, Application Reset vector is $0xFFFFFFFF$.
  - Application-invoked
    The application writes a magic value to a reserved SRAM location,
    then issues a software reset.
    *In MCC Harmony Bootloader, use "Number of Bytes to Reserve from Start of RAM"*
    *(and optional offset) to reserve this area in the linker script.*
  - Others
    Host handshake within a timeout,
    double-tap reset, debug probe detected, pin-strap/DIP switch, etc.

# Hardware I/O and Magic Number Trigger

- **The example below**
  **that triggers the bootloader via a magic number or a button status.**

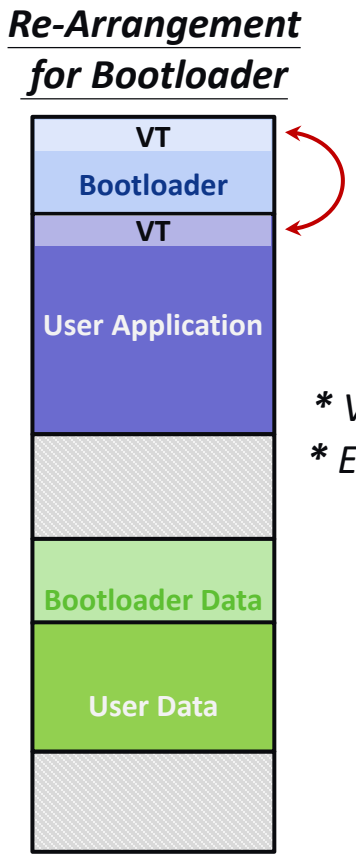
```
#define BTL_TRIGGER_PATTERN (0x5048434DUL)
static uint32_t *ramStart = (uint32_t *)BTL_TRIGGER_RAM_START;

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 Magic Number */
    if (BTL_TRIGGER_PATTERN == ramStart[0] &&
        BTL_TRIGGER_PATTERN == ramStart[1] &&
        BTL_TRIGGER_PATTERN == ramStart[2] &&
        BTL_TRIGGER_PATTERN == ramStart[3])
    {
        ramStart[0] = 0;
        DCACHE_CLEAN_BY_ADDR(ramStart, 4);
        return true;
    }

    /* Check for Button */
    if (!BT1_Get())
        return true;

    return false;
}
```



# Handoff Control

# VTs Allocation

- **Beyond Flash and SRAM placement, planning how to allocate and remap the VT (Vector Table) is critical.**
- **Decide how to allocate separate VTs for the Bootloader and the Application, and how to switch between them during handoff.**
- **At handoff to the Application, Reallocate the VT to the Application's vector table; when returning to the Bootloader, Reallocate it to the Bootloader's vector table.**
  ***(Cortex-M: VTOR; PIC32/MIPS: EBASE).***




***VT Reallocated When Switching Bootloader and Application***

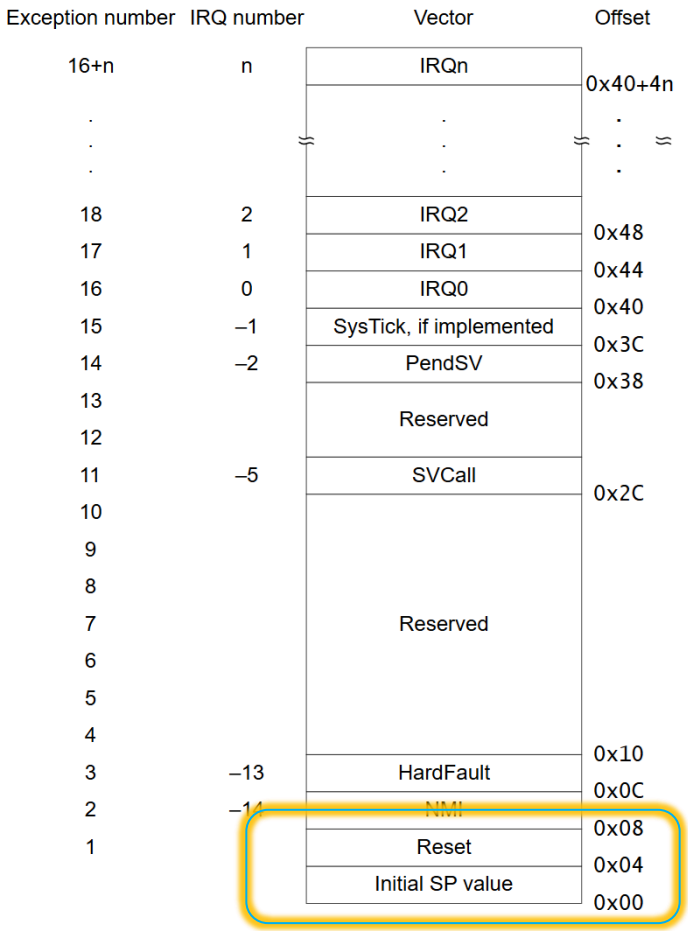
** Vector Table Offset Register (Cortex M)*

** Exception Base Address Register (MIPS)*

MICROCHIP

# Handoff Control to Application *(Cortex M as example)*

- **The figure shows the order of the vectors in the vector table.**
  **It contains the reset value of the stack pointer and the start addresses,**
  **also called exception vectors, for all exception handlers.**
- **Required settings for a handoff,**
  - **Stack Pointer**
    Load Application's SP Value to MSP Register.
  - **VTOR** ***(Optional)***
    Set SCB->VTOR to the application's vector table base.
  - **Reset Vector**
    Branch to the address in the application's Reset vector (*Reset_Handler*).




*https://developer.arm.com/documentation/dui0662/b/The-Cortex-M0--Processor/Exception-model/Vector-table*

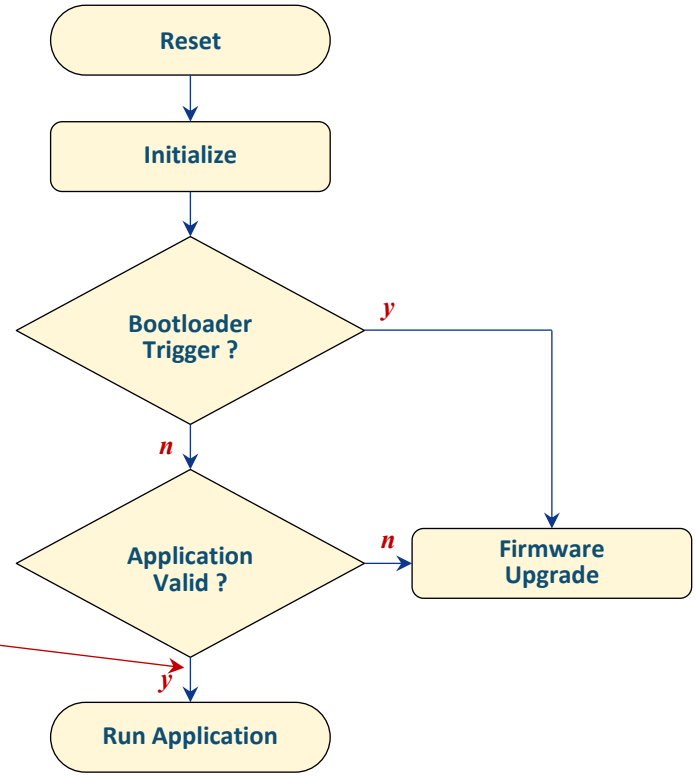
MICROCHIP

# Handoff Control to Application *cont.* *(Cortex M as example)*

- **The following code example shows the mechanism used in the bootloader to transfer control to the user application.**
- **Sets the Main Stack Pointer (MSP) to the top of the user application's stack, and finally jumps to the user Application's reset handler to begin execution.**

***In MCC Harmony Bootloader,***
***use "Application Start Address (Hex)" to set start address.***

```
void run_Application(uint32_t address)
{
    uint32_t msp = *(uint32_t *)(address); // Application Start
    uint32_t reset_vector = *(uint32_t *)(address + 4U);

    if (msp == 0xffffffffU)
        return;

    SYS_DeInitialize(NULL);

    __set_MSP(msp); // Re-initial Stack Pointer (Point to Application Stack)
    asm("bx %0"::"r" (reset_vector)); // Jumper to Application
}
```



# Robust Handoff Control to Application *(Cortex M as example)*

- **VTOR setting is optional,**
  **the Application's startup code typically setting SCB->VTOR to its own vector table anyway.**

  ** startup_xc32.c*

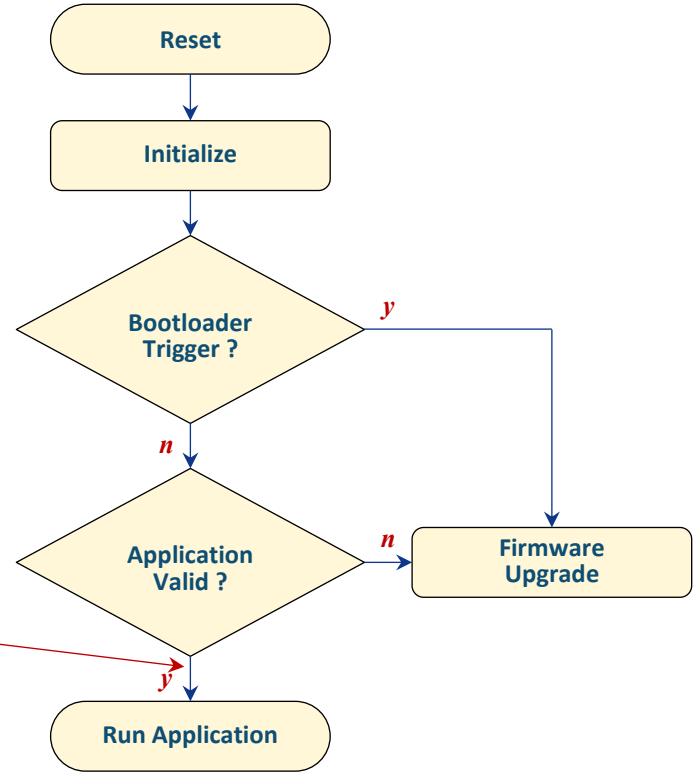
```
#  ifdef SCB_VTOR_TBLOFF_Msk
   /*  Set the vector-table base address in FLASH */
   pSrc = (uint32_t *) & __svectors;
   SCB->VTOR = ((uint32_t) pSrc & SCB_VTOR_TBLOFF_Msk);
#  endif /* SCB_VTOR_TBLOFF_Msk */
```

- **For a more robust handoff,**
  **set VTOR in advance, disable interrupts,**
  **clear any interrupt pending flags, and stop SysTick**
  **before branching to Application reset vector.**

# Robust Handoff Procedure

- **The following code example shows the mechanism used in the bootloader to transfer control to the user application with robust handoff procedure.**

```
void run_Application(uint32_t address)
{
    uint32_t msp = *(uint32_t *)(address); // Application Start
    uint32_t reset_vector = *(uint32_t *)(address + 4U);

    if (msp == 0xffffffffU)
        return;

    SYS_DeInitialize(NULL);

    SysTick->CTRL = 0;

    __disable_irq();
    NVIC->ICER[0] = 0xffffffff;
    NVIC->ICPR[0] = 0xffffffff;

    SCB->VTOR = address;
    __DSB();
    __ISB();

    __set_MSP(msp); // Re-initial Stack Pointer (Point to Application Stack)
    asm("bx %0"::"r" (reset_vector)); // Jumper to Application
}
```



# Configuration Bits & Fuses

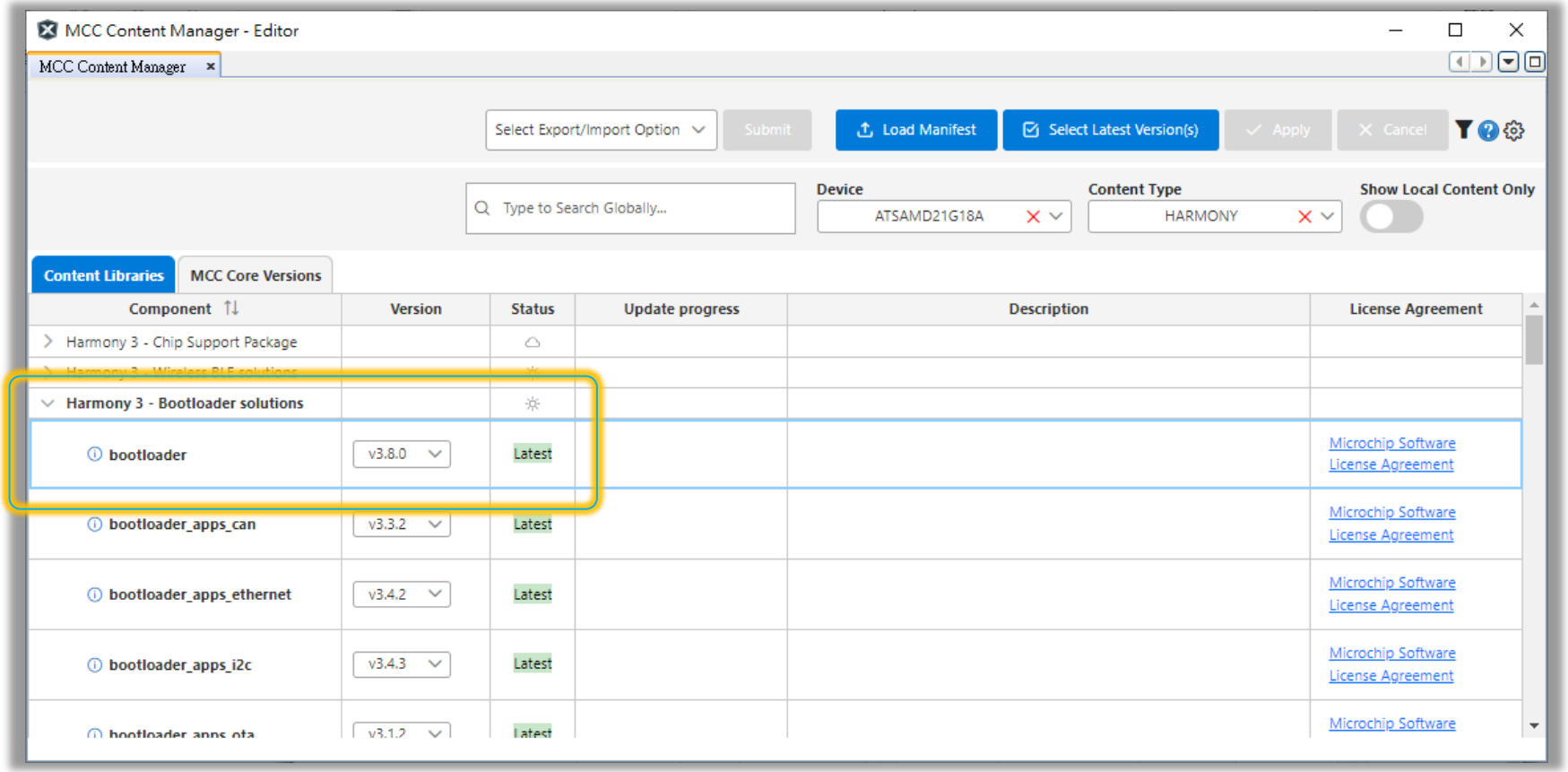
# Configuration Bits & Fuses Definition

- In A Bootloader-based design,
  the device Configuration Bits/Fuses are shared.
- The setting needs applied to both the Bootloader and the Application,
  choose a suitable setting that work for both is very important.
- In the current design, the device Configuration Bits/Fuses are defined and programmed by the Bootloader.
  The Application project bypasses these settings.
- In advanced designs, you can dynamically change (On Fly Change) the device Configuration Bits/Fuses,
  so the Bootloader and the Application can run with independent settings.
  However, this approach carries substantial risk
  (e.g., bricking the device, losing debug access, or misconfiguring clock/BOD/WDT),
  so we will not cover it in this course.”

# Bootloader Libraries Maintain at Microchip Harmony 3

# Bootloader Libraries at MH3

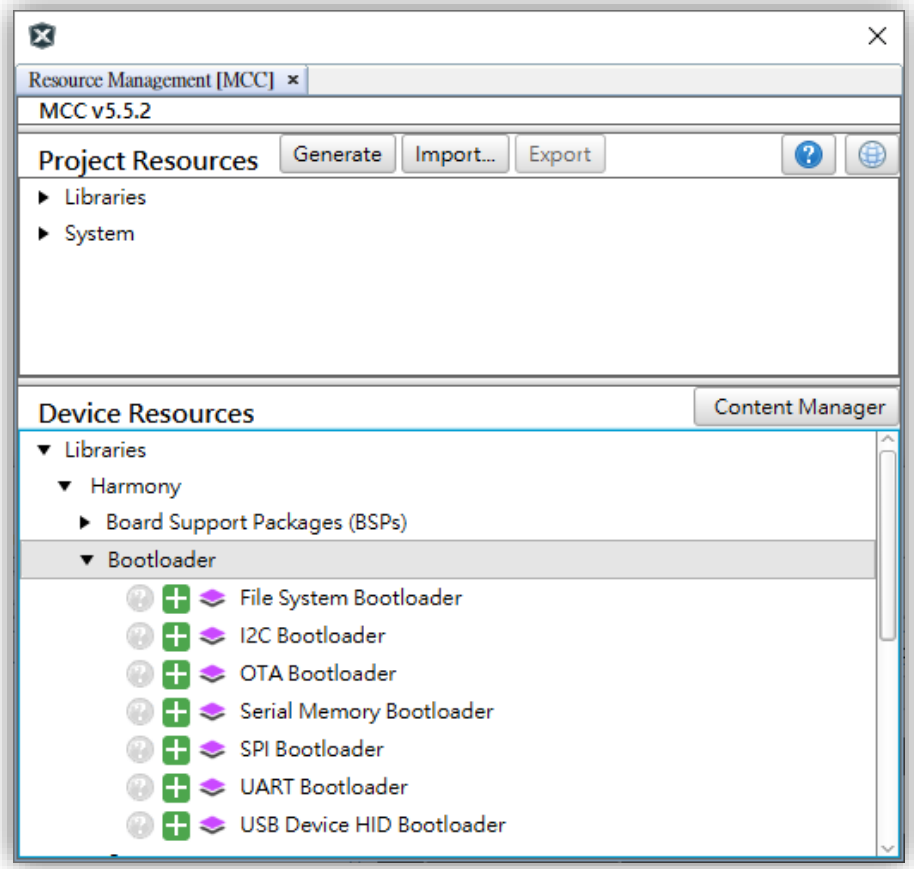
- **Bootloader Libraries must be preparation, before develop the Bootloader application.**
- **Select suitable version of “Harmony 3 - Bootloader solutions” at MCC Content Manager & Apply it.**



# Bootloader Loader Library Modes

# MCC Harmony Bootloader Loader Library

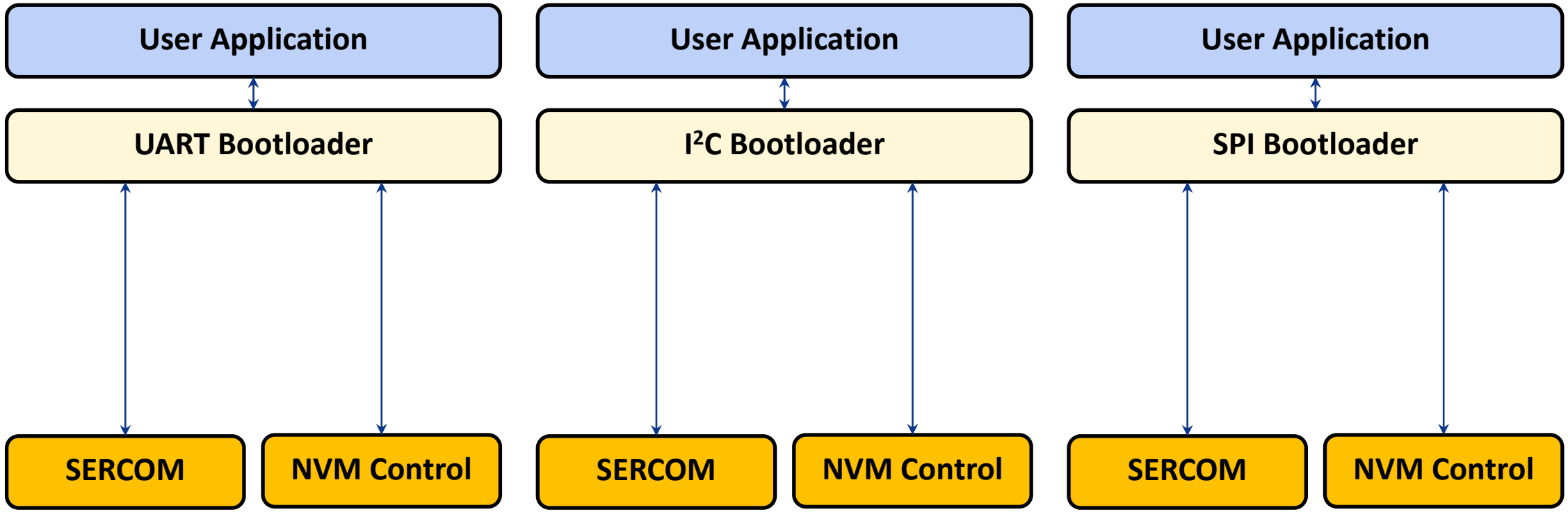
- The MCC Harmony Bootloader library provide multiple firmware updates paths mode.
  for example,
  UART, USB HID, I²C, SPI, CAN, and
  Ethernet (UDP), as well as file systems,
  serial memories,
  and over-the-air (OTA).
- The Figure shows for SAMD21G18A,
  (e.g., UART, USB HID, I²C).
- Actual support varies by device family
  and by the hardware present
  on the target board,
  such as an SD-card slot or
  an Ethernet/Wi-Fi module.




MICROCHIP

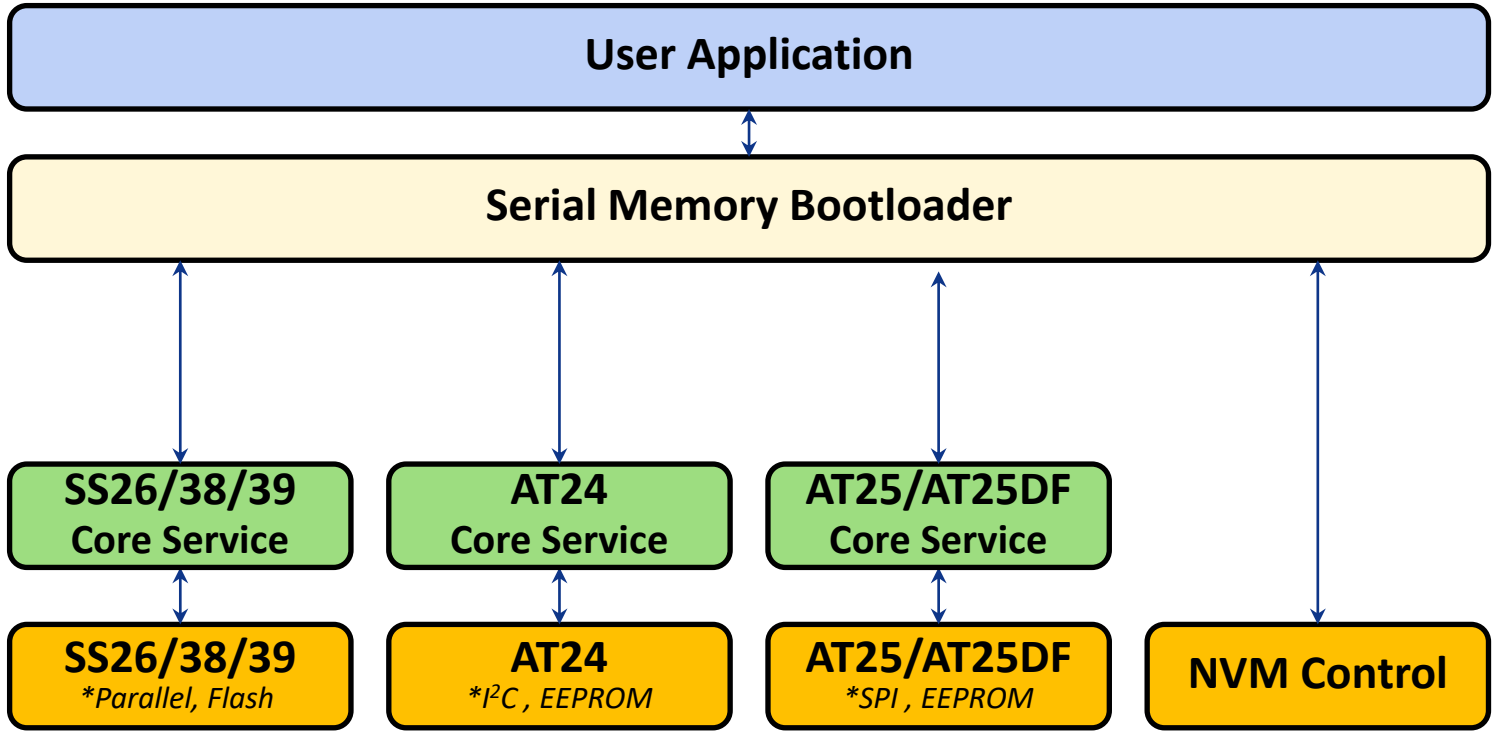
# Bootloader Loader Library

## UART, I²C & SPI Bootloader for connection and load firmware with Host



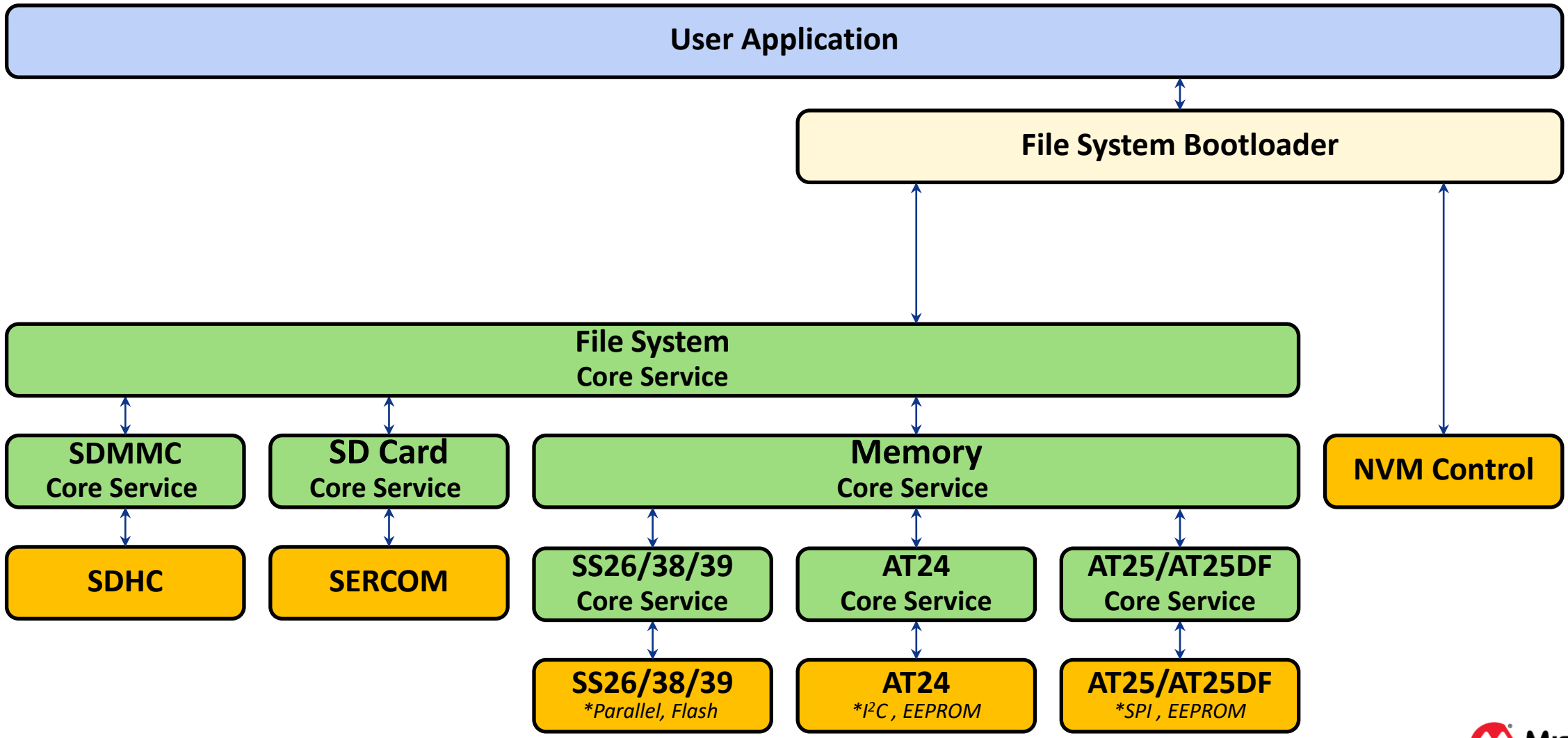
# Bootloader Loader Library

## Serial Memory Bootloader for load firmware from external memory



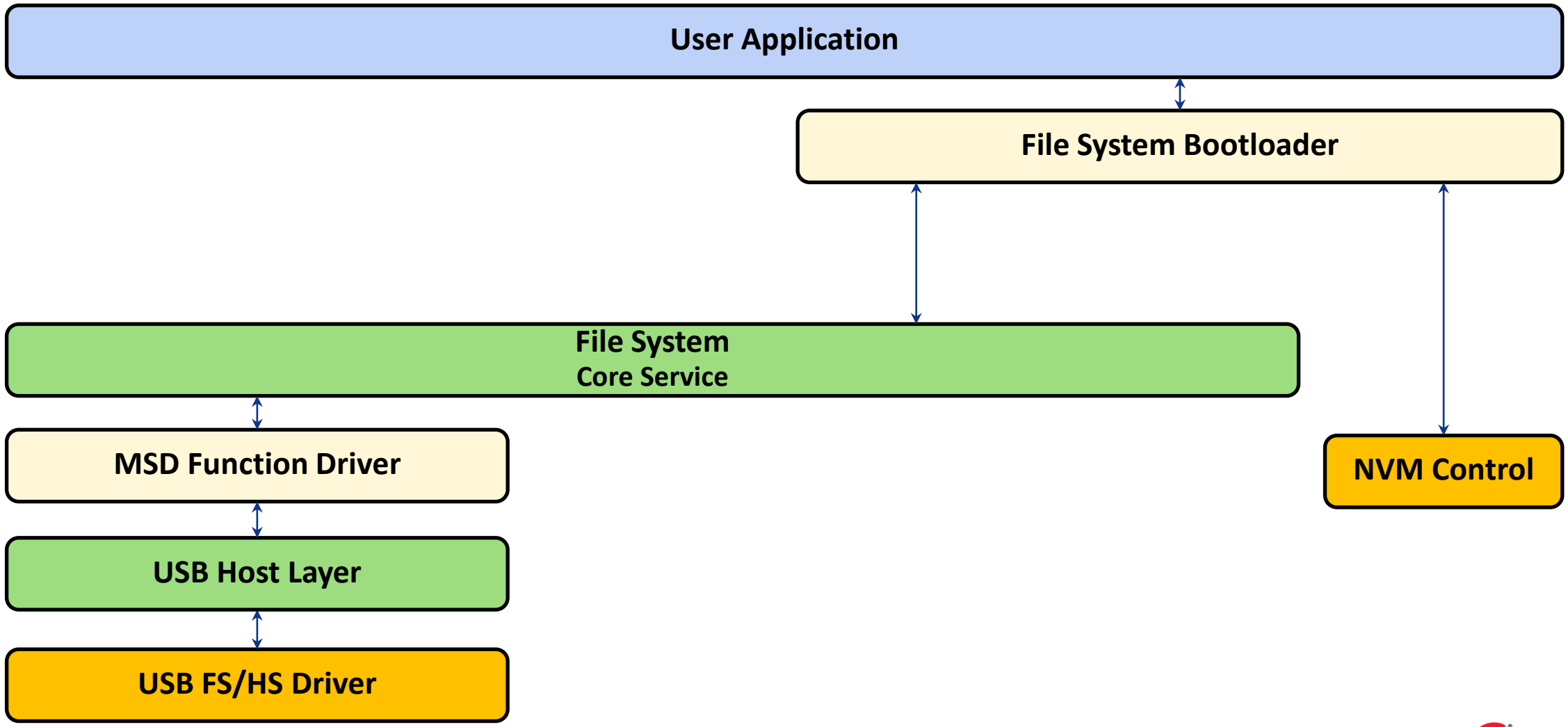
# Bootloader Loader Library

## File System Bootloader for load firmware storage



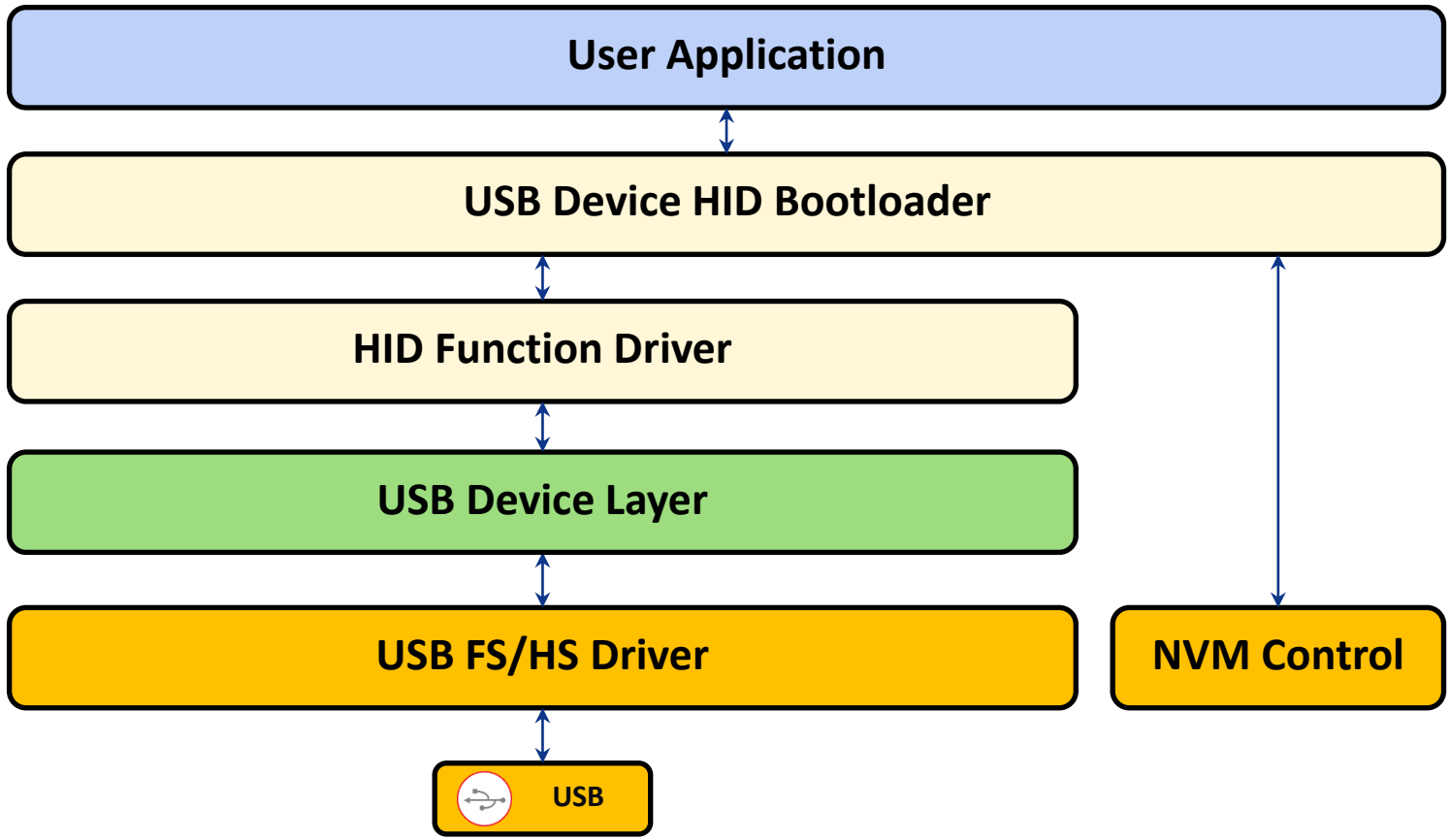
# Bootloader Loader Library

## USB Host MSD Bootloader for load firmware via USB storage



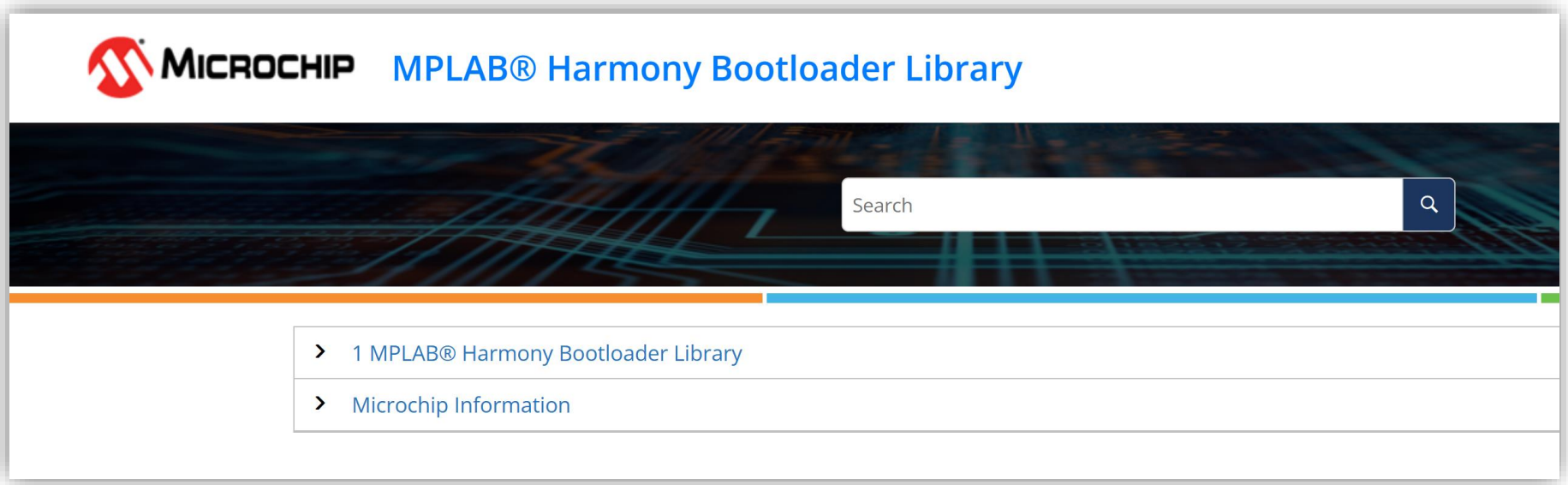
# Bootloader Loader Library

## USB Device HID Bootloader for connection and load firmware via USB Host



# Bootloader Loader Library

- **Harmony offers a wider range of bootloader modes, such as including OTA, CAN, and Ethernet (UDP), etc.**
- **For details, please refer to the example projects of MCC Harmony and documentation provided by MCC Harmony.**
  **https://microchip-mplab-harmony.github.io/bootloader/index.html**




MICROCHIP

# UART Bootloader

# UART Bootloader

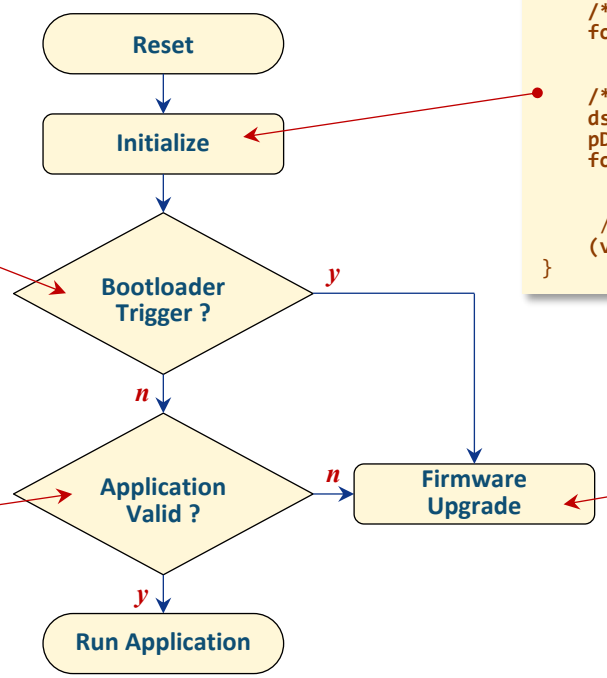
- Supported on Cortex-M based MCUs, MIPS based MCUs and MPUs
  e.g., ATSAMD21G18A (APP045 EVB), PIC32CX1025SG41128 (APP055 EVB), etc.
- MCC Harmony UART PLIB to communicate resulting in smaller bootloader size.
- Takes Binary File as input
- Uses command line host script (*btl_host.py*) to receive binary (**.BIN*) from Host PC.
- Supports Fail Safe update for the devices which have a Dual Bank flash memory.

# UART Bootloader

- **Below flowchart routine shows how to switch between the UART Bootloader and the Application at startup.**

```
#define BTL_TRIGGER_PATTERN (0x5048434DUL)
static uint32_t *ramStart = (uint32_t *)BTL_TRIGGER_RAM_START;

bool bootloader_Trigger(void)
{
    uint32_t i;

    …

    /* Check for Magic Number */
    if (BTL_TRIGGER_PATTERN == ramStart[0] &&
        BTL_TRIGGER_PATTERN == ramStart[1] &&
        BTL_TRIGGER_PATTERN == ramStart[2] &&
        BTL_TRIGGER_PATTERN == ramStart[3])
    {
        ramStart[0] = 0;
        DCACHE_CLEAN_BY_ADDR(ramStart, 4);
        return true;
    }

    /* Check for Button */
    if (!BT1_Get())
        return true;

    return false;
}
```

```
void __attribute__((noinline, section(".romfunc.Reset_Handler"))) Reset_Handler(void)
{
    register uint32_t count;
    uint32_t *pSrc, *pDst;
    uintptr_t src, dst;

    src = (uintptr_t)&_etext;
    pSrc = (uint32_t *)src;      /* flash functions start after .text */
    dst = (uintptr_t)&_sdata;
    pDst = (uint32_t *)dst;      /* boundaries of .data area to init */

    /* Init .data */
    for (count = 0U; count < (((uint32_t)&_edata - (uint32_t)dst) / 4U); count++)
        pDst[count] = pSrc[count];

    /* Init .bss */
    dst = (uintptr_t)&_sbss;
    pDst = (uint32_t *)dst;
    for (count = 0U; count < (((uint32_t)&_ebss - (uint32_t)dst) / 4U); count++)
        pDst[count] = 0U;

     /* Branch to application's main function */
    (void)main();
}
```



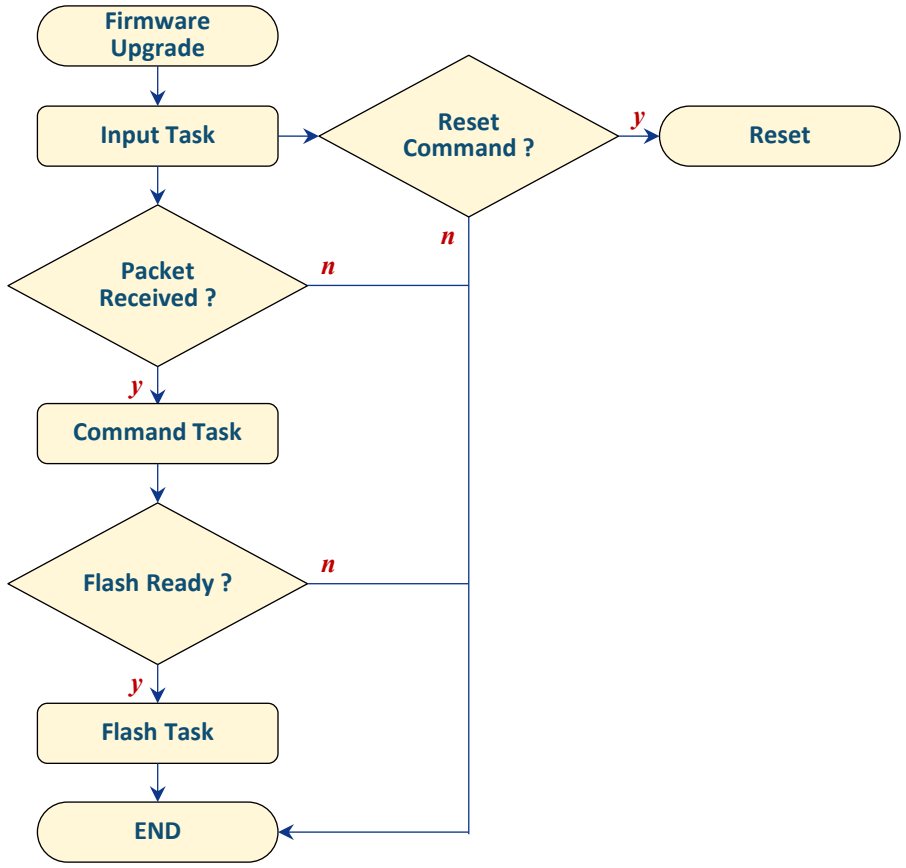

```
void run_Application(uint32_t address)
{
    uint32_t msp = *(uint32_t *)(address); // Application Start
    uint32_t reset_vector = *(uint32_t *)(address + 4U);

    if (msp == 0xffffffffU)
        return;

    …

    __set_MSP(msp); // Re-initial Stack Pointer
    asm("bx %0"::"r" (reset_vector)); // Jumper to Application
}
```

```
void bootloader_UART_Tasks(void)
{
    do
    {
        input_task();

        if (flash_data_ready)
            flash_task();

        else if (packet_received)
            command_task();


    } while (uartBLActive);
}
```

MICROCHIP

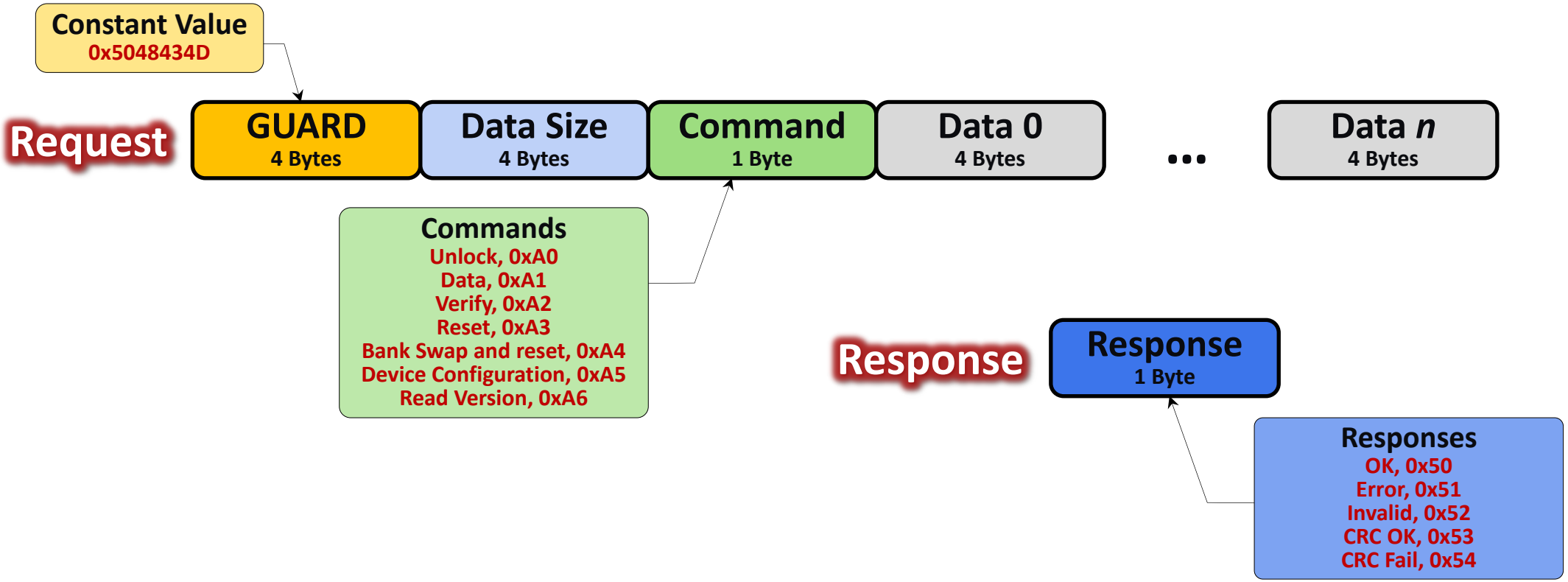
# UART Bootloader

- Below flowchart routine shows how the firmware upgrade works.
- Receives packets, parses commands, and executes them; upon receiving a reset command, the system reboots.



# UART Bootloader Protocol

- **The UART bootloader protocol as shown in below figure is common for all the supported command.**



# Binary Generation

- The UART bootloader uses a binary transfer format,
  so you must first convert the Intel HEX (.hex) file to a raw binary (.bin) image.
- xc32-objcopy,
  bundled with Microchip's XC32 toolchain, is used to convert and edit firmware images.
- It supports ELF, Intel HEX, S-record, and raw binary.
  You can select sections, shift addresses, fill gaps, and pad sizes to produce
  deterministic, bootloader-ready outputs.
- ideal for ELF -> BIN/HEX and HEX -> BIN in automated, reproducible build scripts.
- In an MPLAB X IDE project,
  you can use "Execute After Build" to run xc32-objcopy
  after compilation and automatically generate a binary (.bin) file.

```
${MP_CC_DIR}/xc32-objcopy -I ihex -O binary ${DISTDIR}/${PROJECTNAME}.${IMAGE_TYPE}.hex ${DISTDIR}/${PROJECTNAME}.${IMAGE_TYPE}.bin
```

# Bootloader Host Script

- The UART bootloader host (*btl_host.py*) is implemented as a Python script compatible with Python 3.x and later.
- It requires the "*pyserial*" package to communicate with the device over UART. Install it with:

```
pip3 install pyserial
```

- Run *btl_host.py* from the command line to perform a firmware update. e.g.

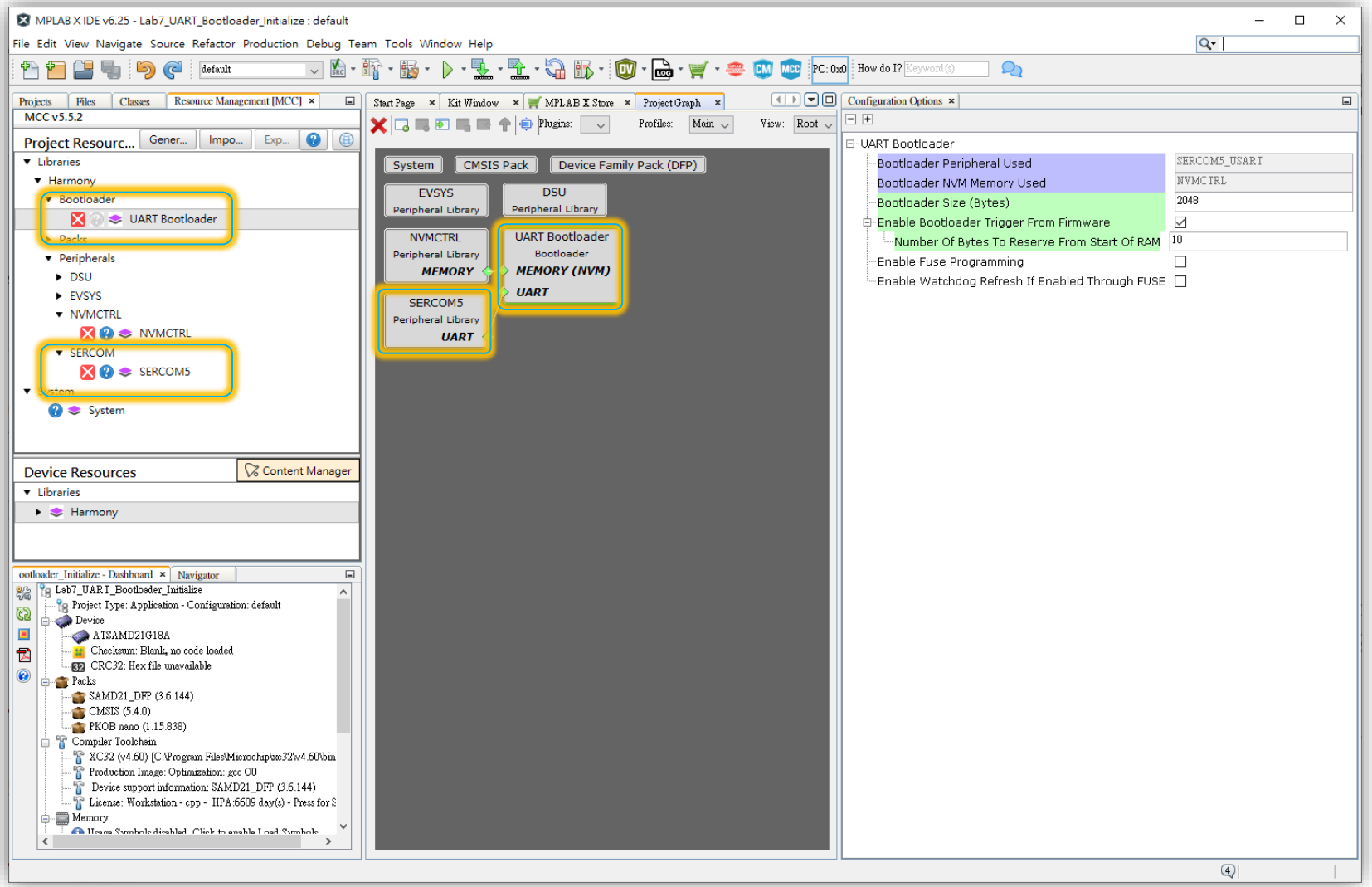
```
btl_host.py -v -i COM10 -d samd2x -a 0x800 -f .\Application.bin
```

- Below is the syntax to show help menu for the script

```
btl_host.py --help
```

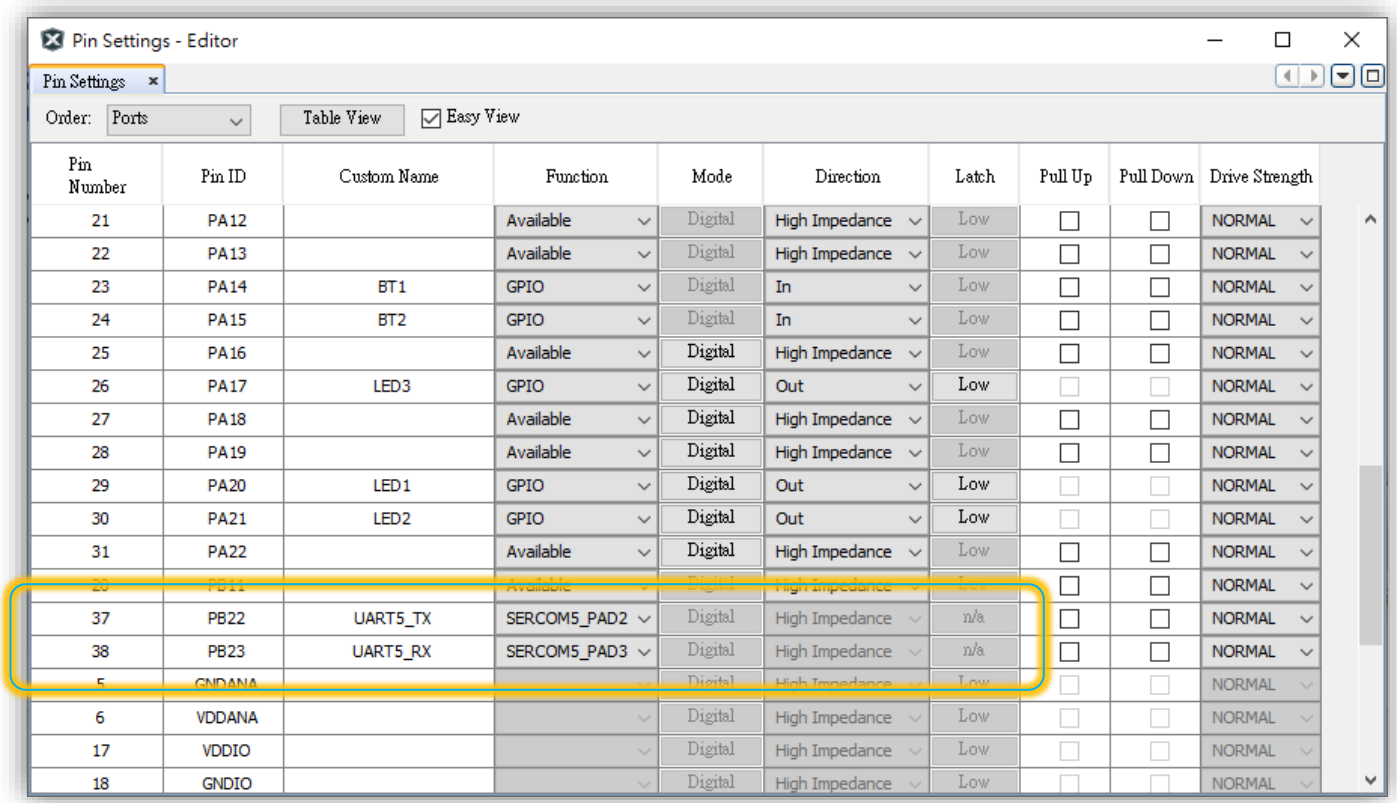
# Harmony Bootloader - UART Bootloader Configurator

## Add UART Bootloader and SERCOM in MHC



# Harmony Bootloader - UART Bootloader Configurator

## Configure SERCOM PADs at Pin Table



# UART Bootloader API

**API functions in bootloader_common.c**

| | |
|---|---|
| **bootloader_Tasks( )** | **Starts bootloader execution.** |
| **bootloader_Trigger( )** | **Checks if Bootloader has to be executed at startup.** |
| **bootloader_GetVersion( )** | **Returns the current bootloader version.** |
| **run_Application( )** | **Runs the programmed application at startup.** |
| **bootloader_ProgramFlashBankSelect( )** | **Selects Appropriate Program Flash Bank after reset.** |

# 🛠 Lab7 UART Bootloader Initialize

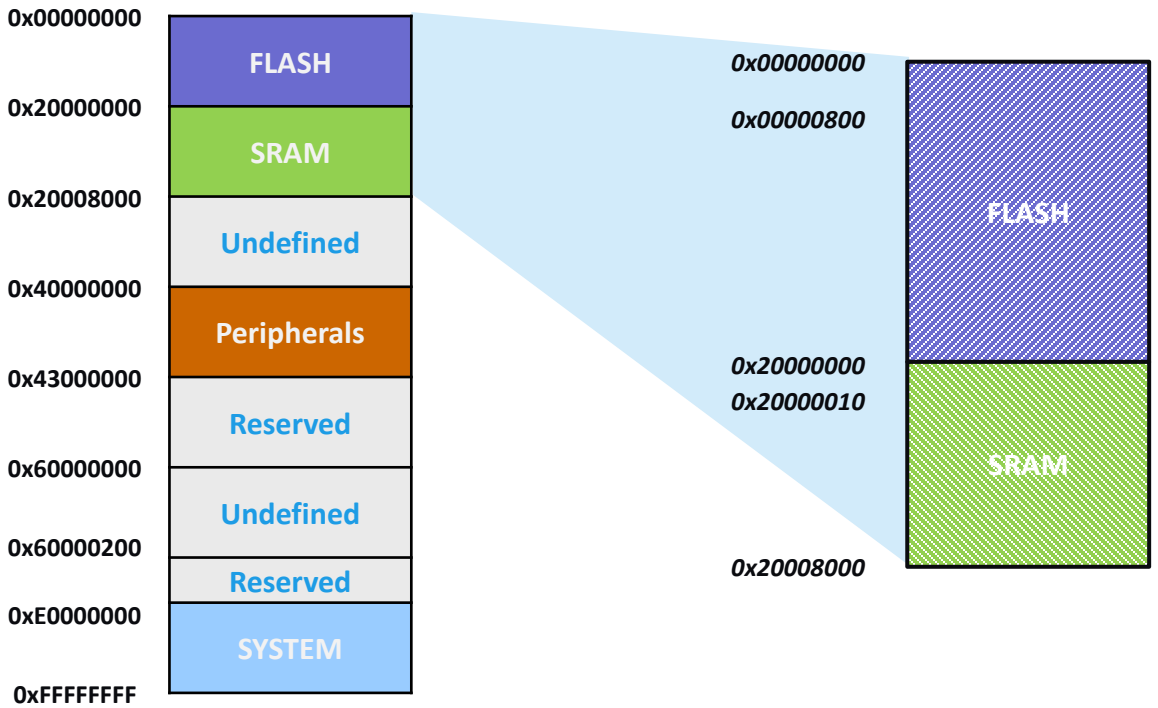
# 🛠 Lab7 UART Bootloader Initialize

- At this lab, we try to getting started with UART Bootloader.
- Add UART Bootloader in MHC and assign parameters and application start address.
- Implement suitable bootloader trigger method use to trigger Bootloader.
- Implement an LEDs status to indicate Bootloader enter or not.
- Try to use python script to upload “DemoApplication” firmware to APP045 EVB via UART based on UART Bootloader.

- **Let’s go !**

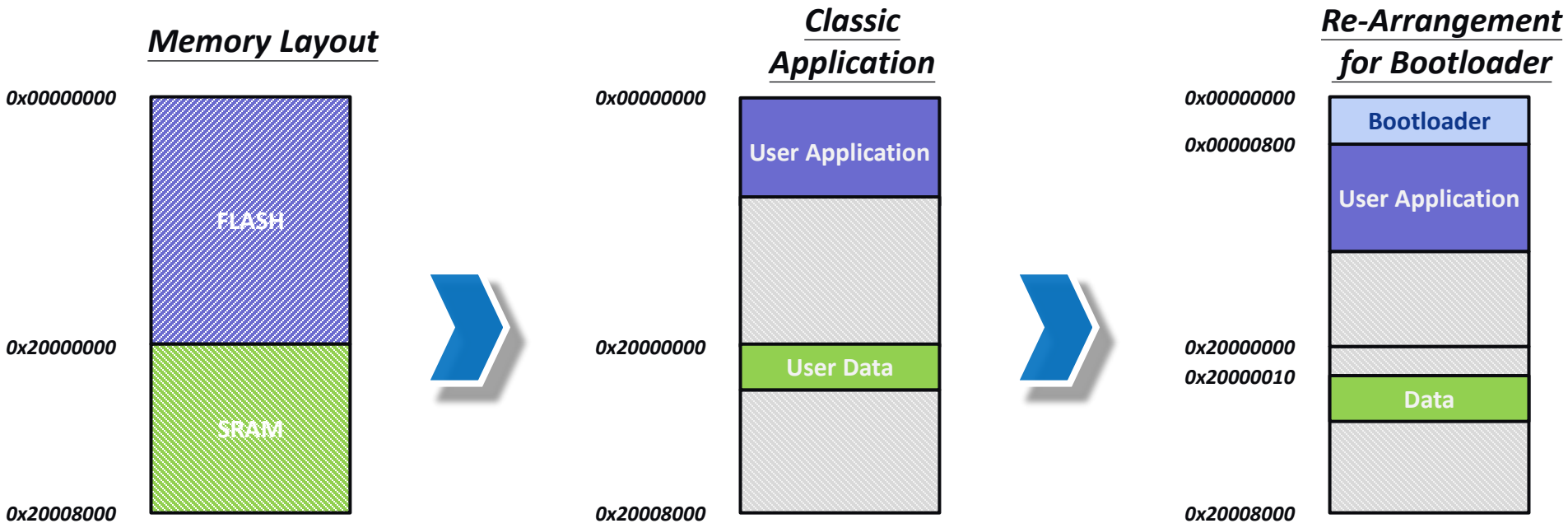
# Memory Layout for ATSAMD21

- **This figure shows the memory layout of the ATSAMD21x18A, which features 256 KB of Flash and 32 KB of SRAM within a 4 GB addressable space.**
  *Flash, 0x00000000 - 0x0003FFFF, 256KB*
  *SRAM, 0x20000000 - 0x20007FFFF, 32KB*



# Memory Re-Arrangement for Bootloader *ATSAMD21 as an Example*

- In a typical application, the entire memory space is dedicated to the application itself.
- When a bootloader is present, part of the Flash and SRAM needs to be allocated for the bootloader.
- In UART Bootloader Application, Bootloader occupies the first 0x800 bytes of flash and reserve the first 16 Bytes of SRAM for bootloader trigger by Application.

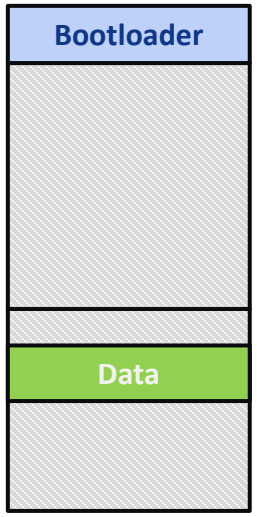



MICROCHIP

# Memory Re-Arrangement for Bootloader *cont.* *ATSAMD21 as an Example*

- **Set two independent linker scripts (**.ld*),**
  **one for the Bootloader and one for the Application to define their memory allocations.**
- **In the linker scripts,**
  **reserved separately for the Bootloader and the Application.**

***Bootloader***



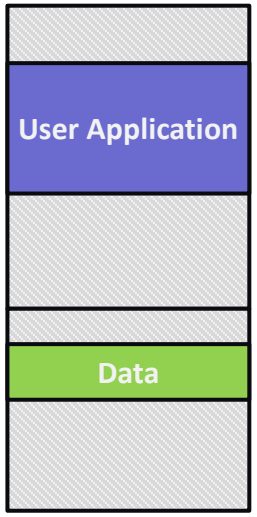

***Linker Script for Bootloader***

```
MEMORY
{
  rom (LRX) : ORIGIN = 0x00, LENGTH = 0x800
  ram (WX!R) : ORIGIN = 0x20000010, LENGTH = 0x7FF0
  config_00804000 : ORIGIN = 0x00804000, LENGTH = 0x4
  config_00804004 : ORIGIN = 0x00804004, LENGTH = 0x4
}
```

***Linker Script for Application***

```
MEMORY
{
  rom (LRX) : ORIGIN = 0x800, LENGTH = 0x3F800
  ram (WX!R) : ORIGIN = 0x20000010, LENGTH = 0x7FF0
  config_00804000 : ORIGIN = 0x00804000, LENGTH = 0x4
  config_00804004 : ORIGIN = 0x00804004, LENGTH = 0x4
}
```

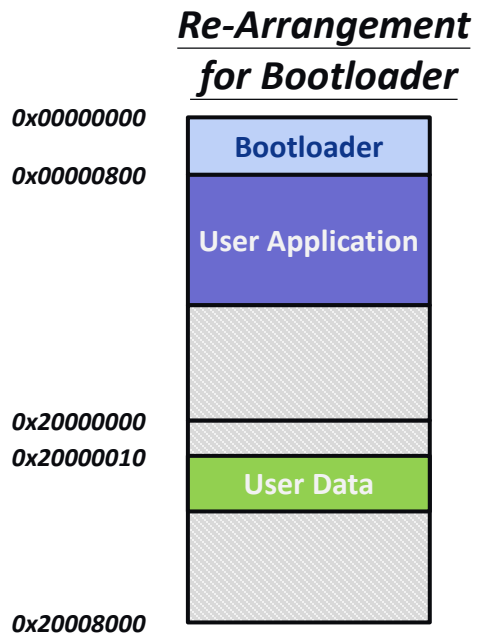
***Application***




MICROCHIP

# Link Script for ATSAMD21x18A

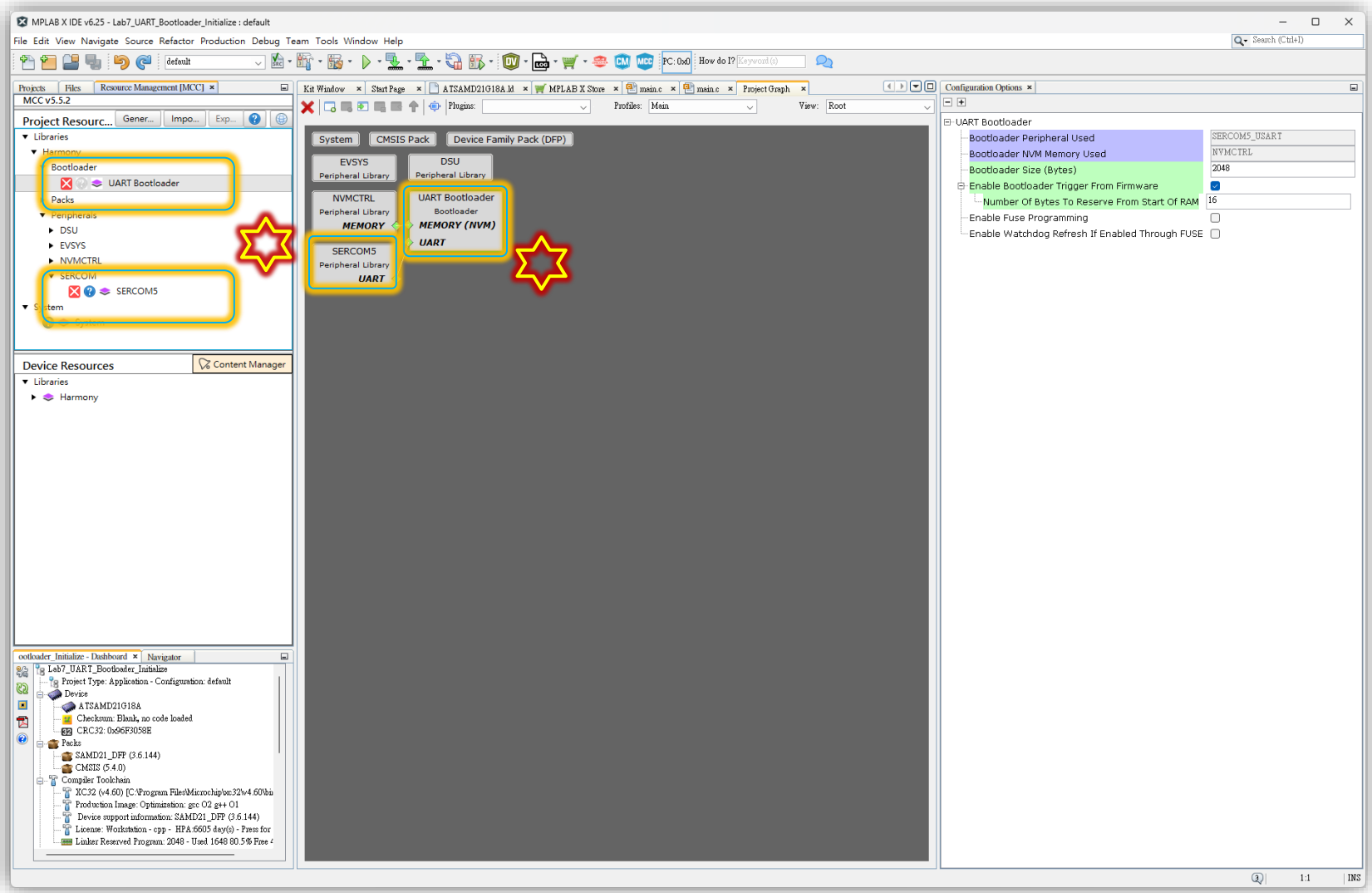
- **In MCC Harmony project, apply preprocessor definition to dynamic management linker script.**
  **The corresponding file will be generated**
  **after press “generate” button at MHC depend on your setting.**
  - **For Bootloader *(Optional)*,**
    **-DROM_ORIGIN = *0x00000000*, -DROM_LEHGTH = *0x800***
    **-DRAM_ORIGIN = *0x20000010*, -DRAM_LEHGTH = *0x7FF0***
  - **For User Application,**
    **-DROM_ORIGIN = *0x00000800*, -DROM_LEHGTH = *0x3F800***
    **-DRAM_ORIGIN = *0x20000010*, -DRAM_LEHGTH = *0x7FF0***




- **These definitions are applied in the linker script,**
  **and during the linking process,**
  **they ensure that the program code and data are in the designated memory areas.**

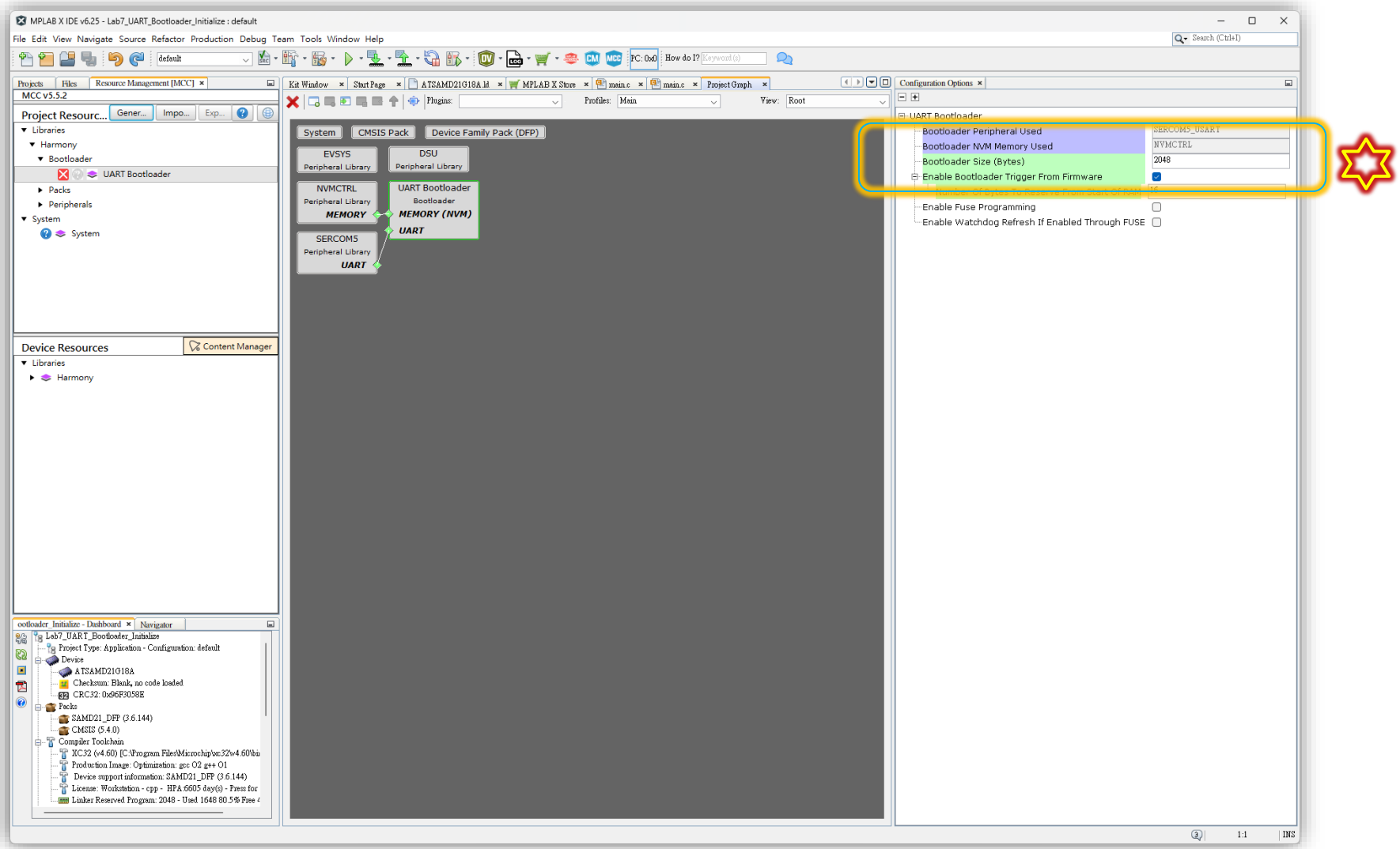
# 🛠 Lab7 UART Bootloader Initialize

- **Create a new project and add UART Bootloader and SERCOM5 in MHC.**



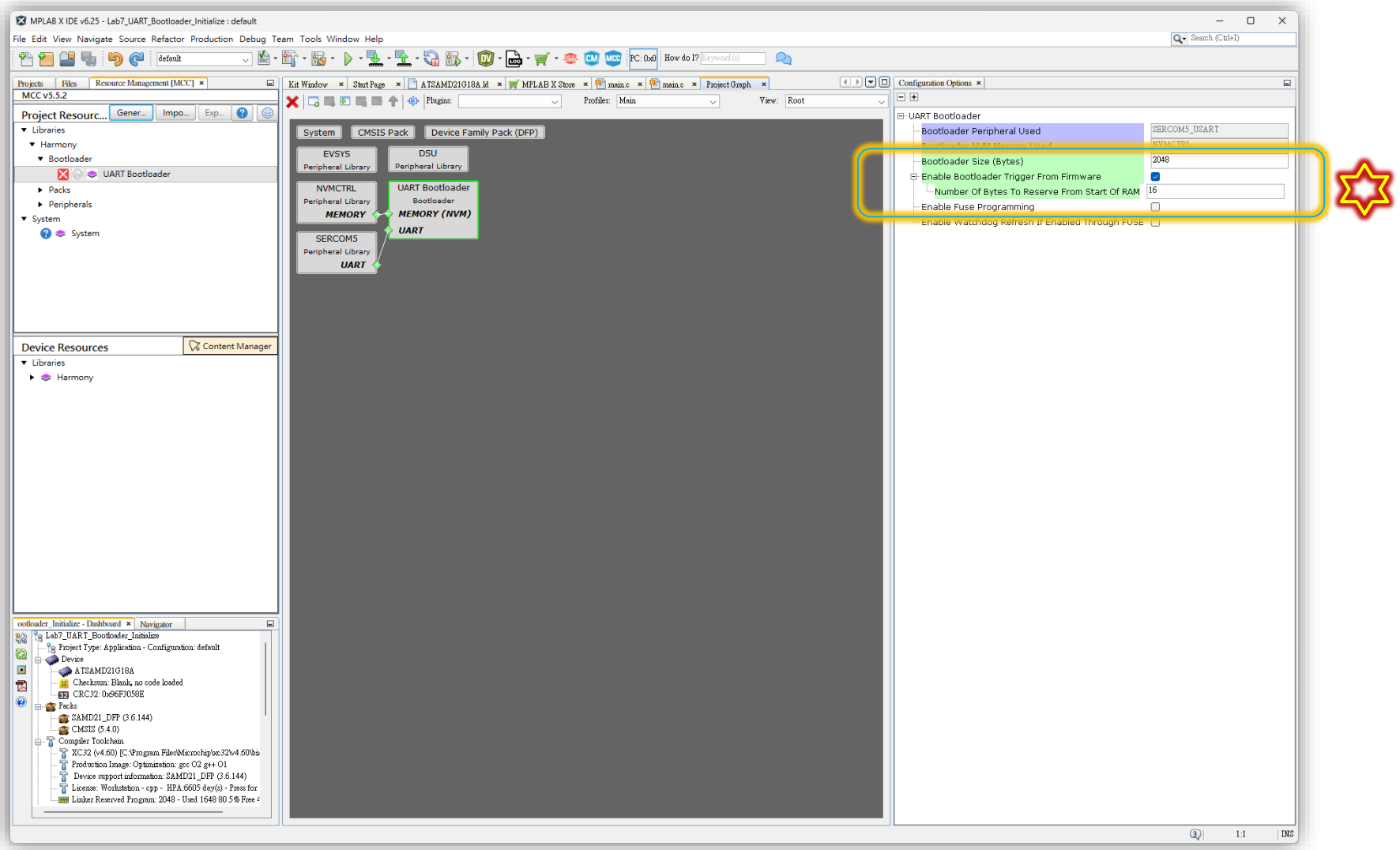
# 🛠 Lab7 UART Bootloader Initialize

- **Set “Bootloader Size (Bytes)” to 2048.**



# 🛠 Lab7 UART Bootloader Initialize

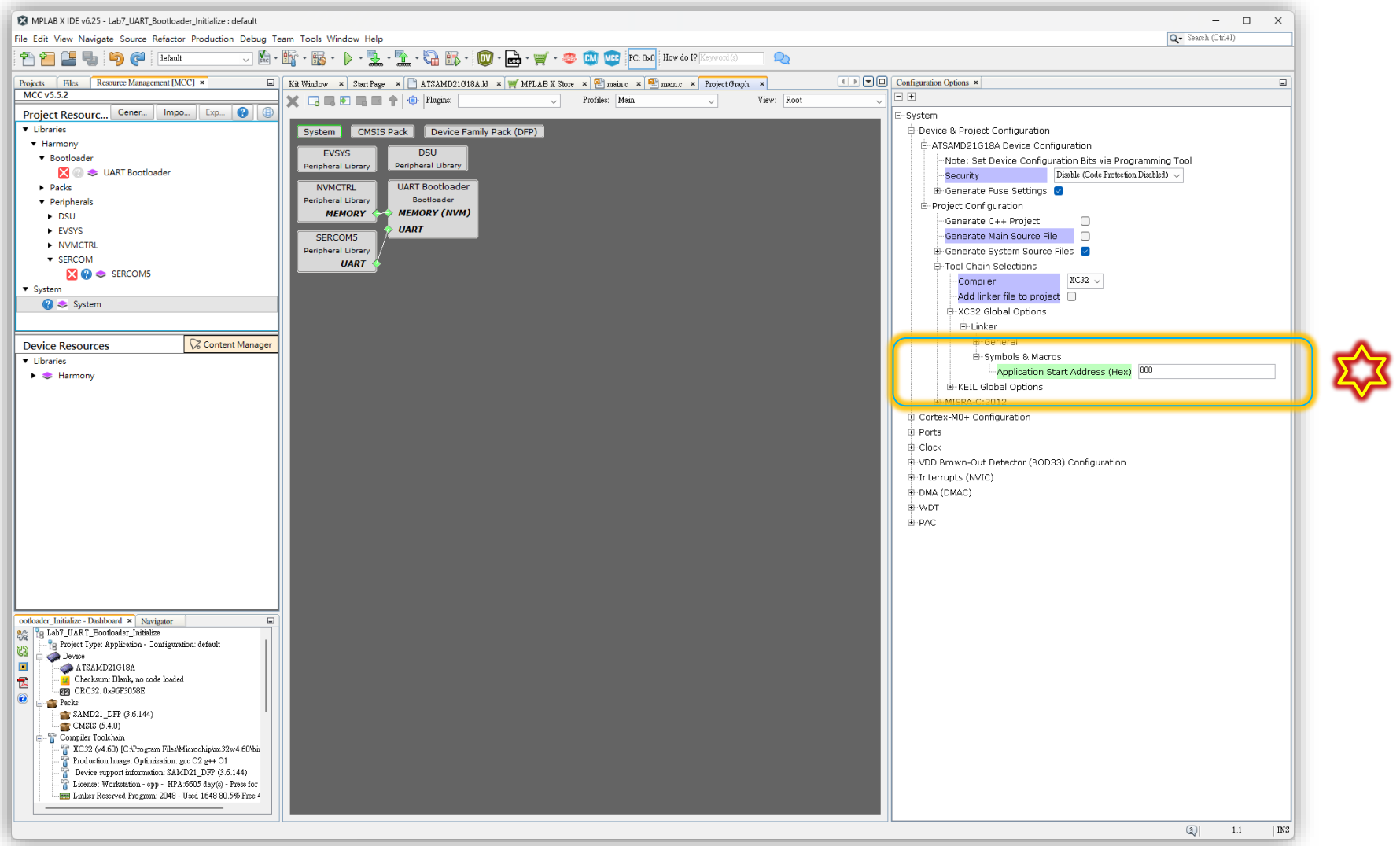
- **Check “Enable Bootloader Trigger From Firmware” and Reserve 16 bytes RAM.**




MICROCHIP

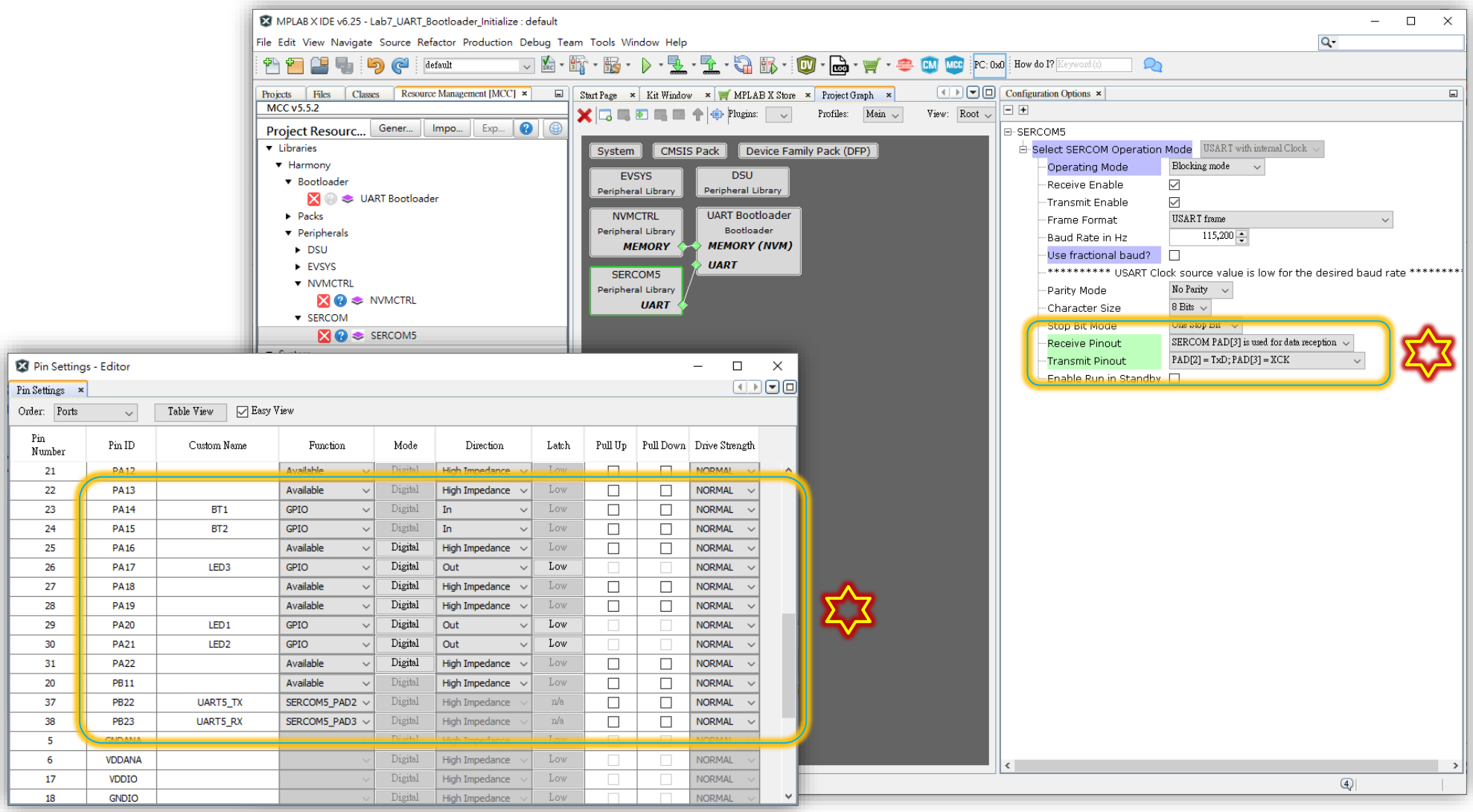
# 🛠 Lab7 UART Bootloader Initialize

- **Set Application Start Address (Hex) is 800.**



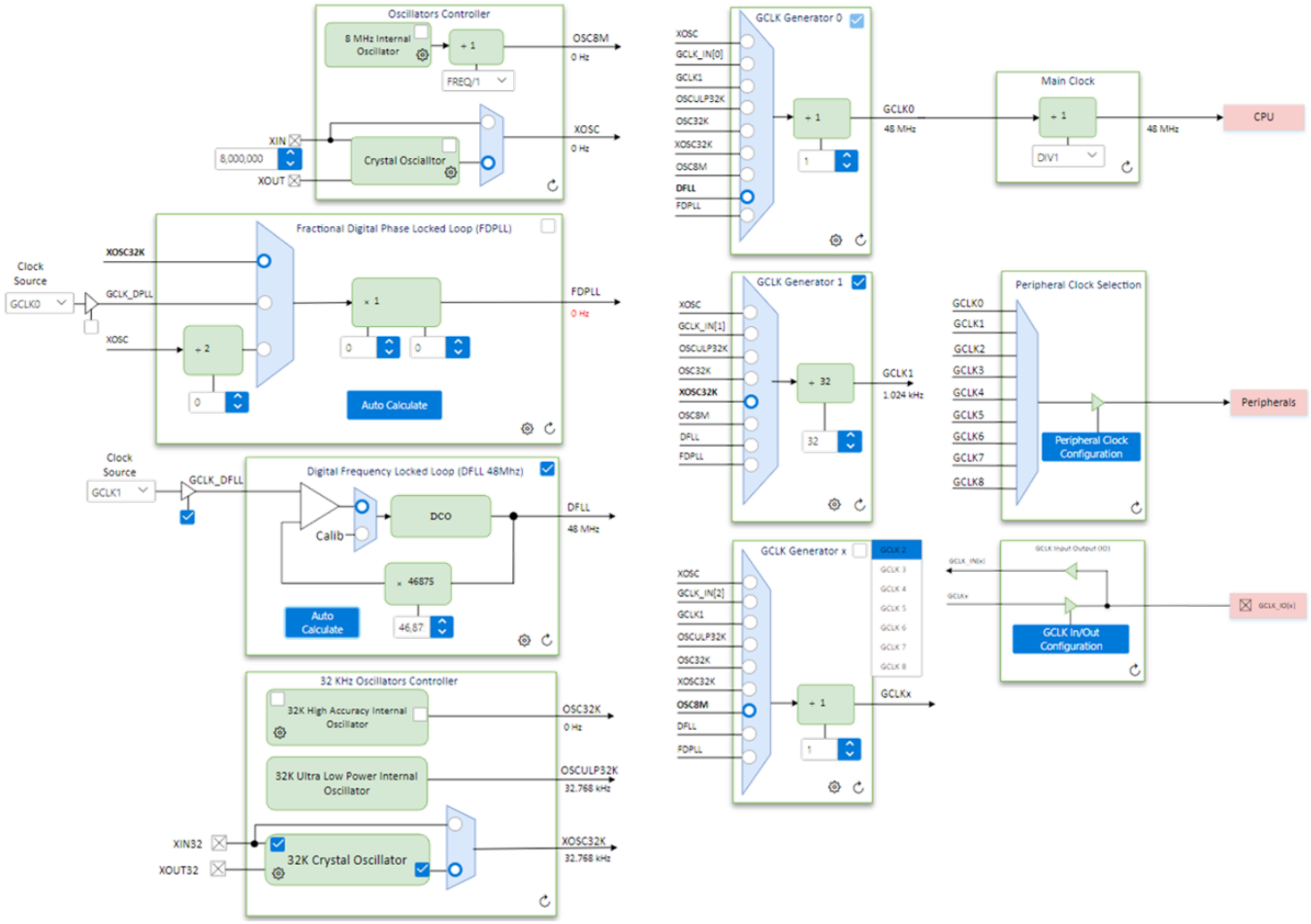
# 🛠 Lab7 UART Bootloader Initialize

- **Connect SERCOM5 with UART Bootloader and Set IO mapping.**

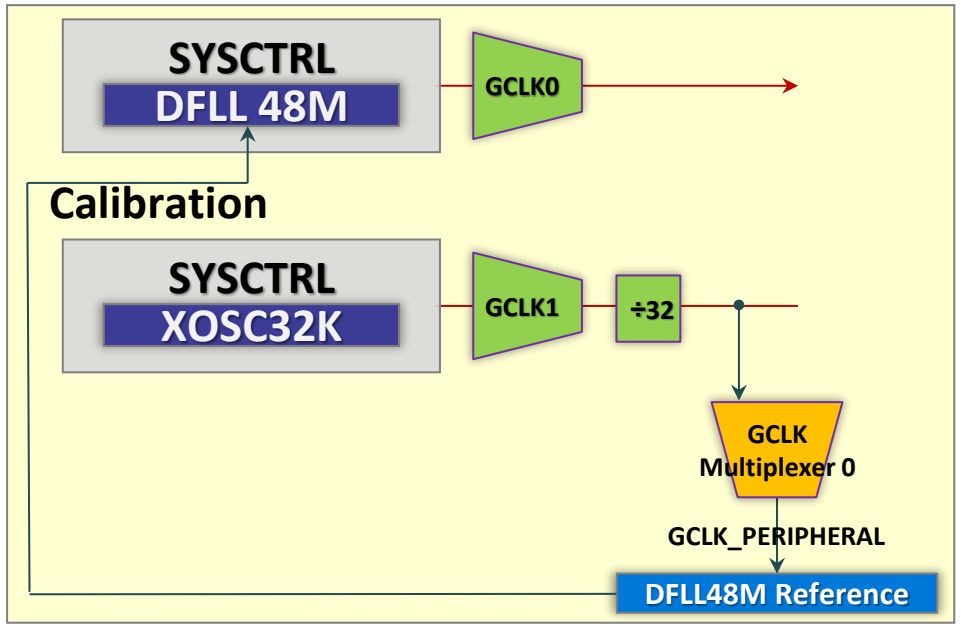


# 🛠 Lab7 UART Bootloader Initialize

- **Configure DFLL48M as Close loop as source of main clock**



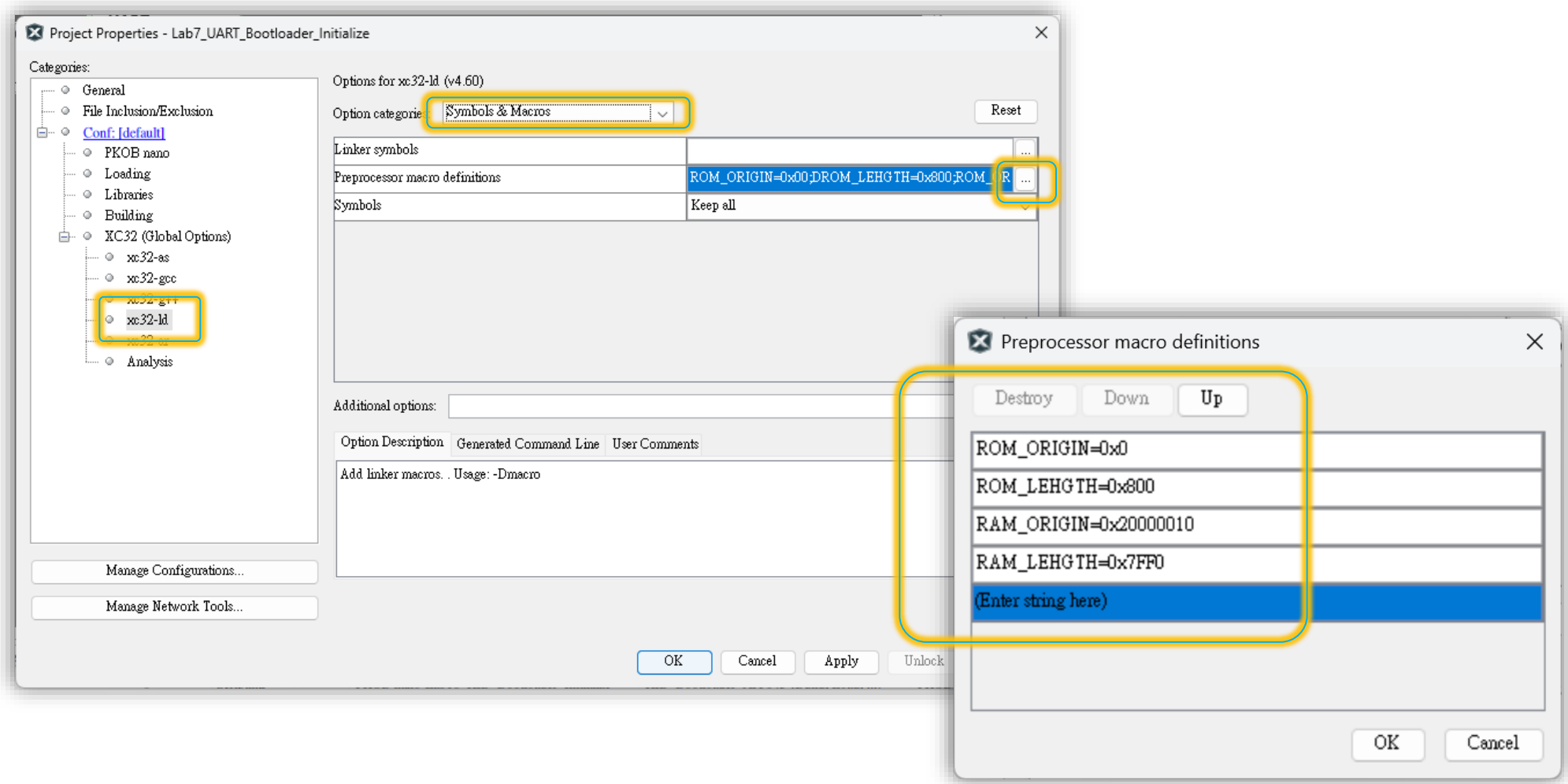

## Close Loop




MICROCHIP

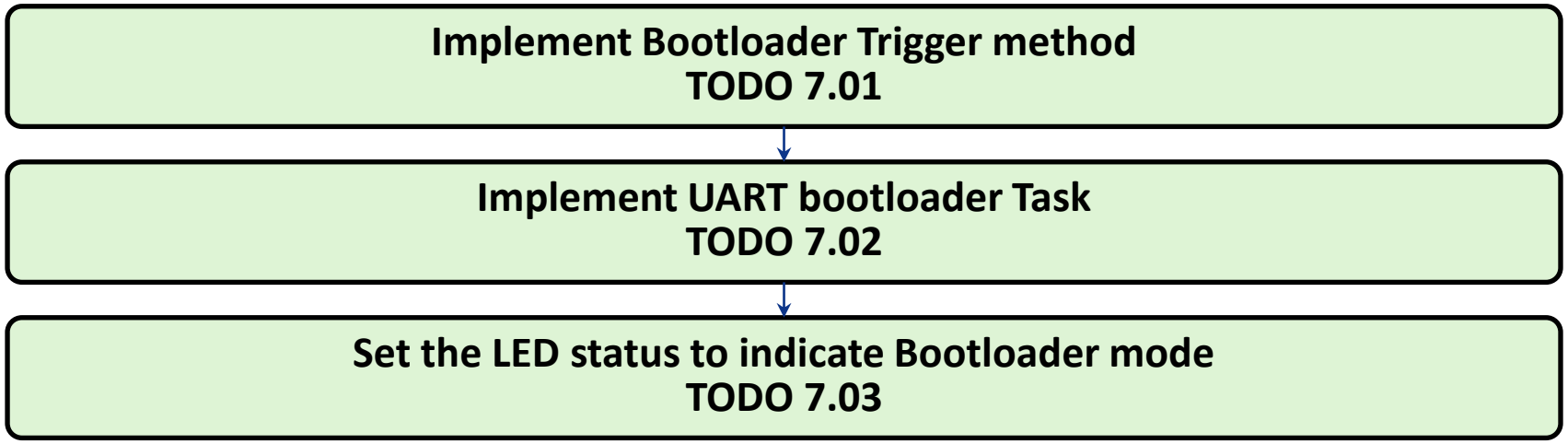
# Lab7 UART Bootloader Initialize

- **Confirm Preprocessor Definition**



# 🛠 Lab7 UART Bootloader Initialize

- **Code Implementation state**



# 🛠 Lab7 UART Bootloader Initialize

- **Implement below function at main.c**

```
// TODO 7.01
#define BTL_TRIGGER_PATTERN (0x5048434DUL)
static uint32_t *ramStart = (uint32_t *)BTL_TRIGGER_RAM_START;

bool bootloader_Trigger(void)
{
    uint32_t i;

    for (i = 0; i < 2000; i++)
        asm("nop");

    if (BT1_Get() == 0)
        return true;

    if (BTL_TRIGGER_PATTERN == ramStart[0] && BTL_TRIGGER_PATTERN == ramStart[1] &&
        BTL_TRIGGER_PATTERN == ramStart[2] && BTL_TRIGGER_PATTERN == ramStart[3])
    {
        ramStart[0] = 0;
        ramStart[1] = 0;
        ramStart[2] = 0;
        ramStart[3] = 0;
        return true;
    }

    return false;
}
```

# 🛠 Lab7 UART Bootloader Initialize

- **Implement below function at main.c**

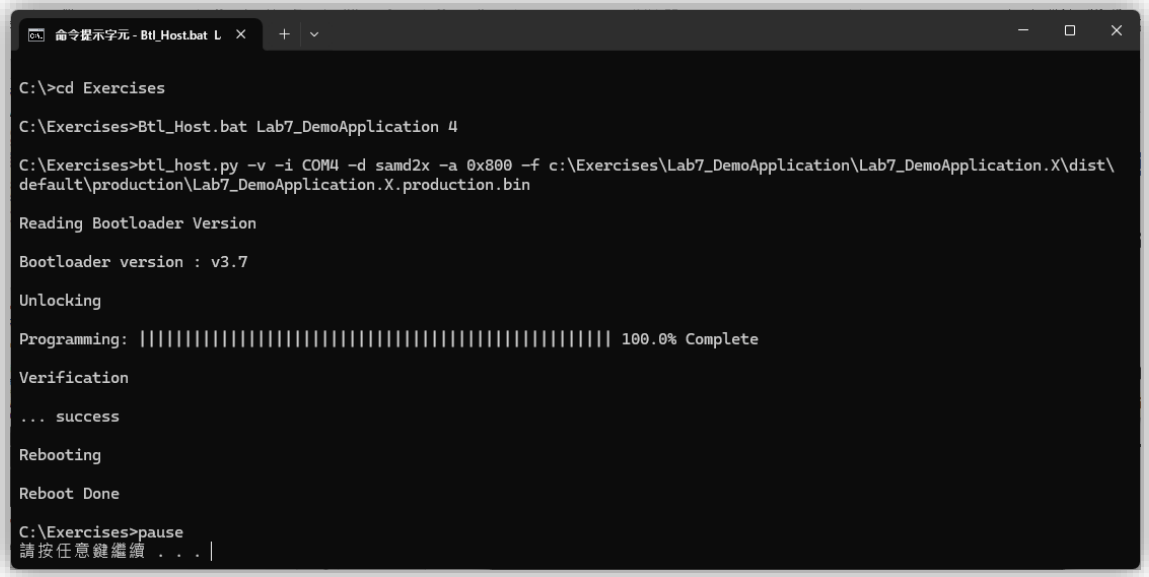
```
int main ( void )
{
    /* Initialize all modules */
    SYS_Initialize ( NULL );

    // TODO 7.03
    LED1_Set();
    LED2_Set();
    LED3_Set();

    while ( true )
    {
        /* Maintain state machines of all polled MPLAB Harmony modules. */
        SYS_Tasks ( );

        // TODO 7.02
        bootloader_UART_Tasks();
    }

    /* Execution should not come here during normal operation */

    return ( EXIT_FAILURE );
}
```

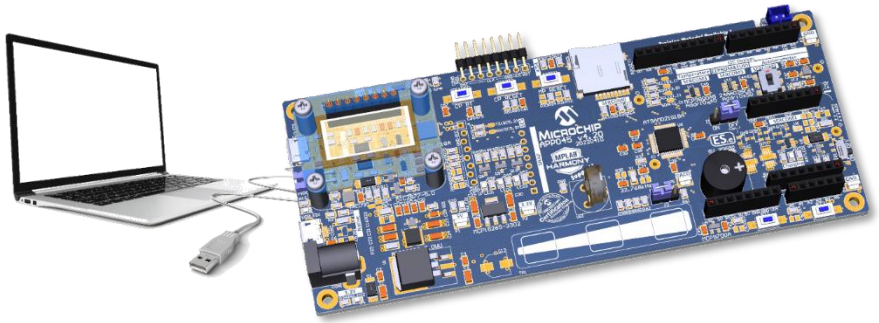
# 🛠 Lab7 UART Bootloader Initialize

- **Enter Bootloader mode automatically, because application invalid, after programming firmware.**
- **LED1 to 3 keep light to indicate Bootloader mode currently.**
- **Run *btl_host.bat* from the command line to perform a firmware update.**

```
Btl_Host.bat Lab7_DemoApplication 4
```

***Lab7_DemoApplication.bin*** **is the firmware built for use with the Bootloader, relocated to 0x800.**

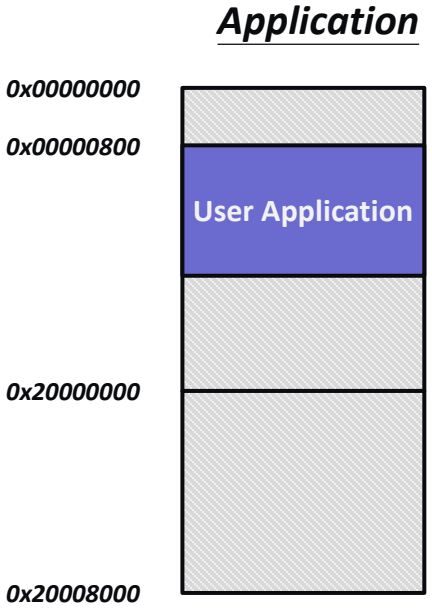


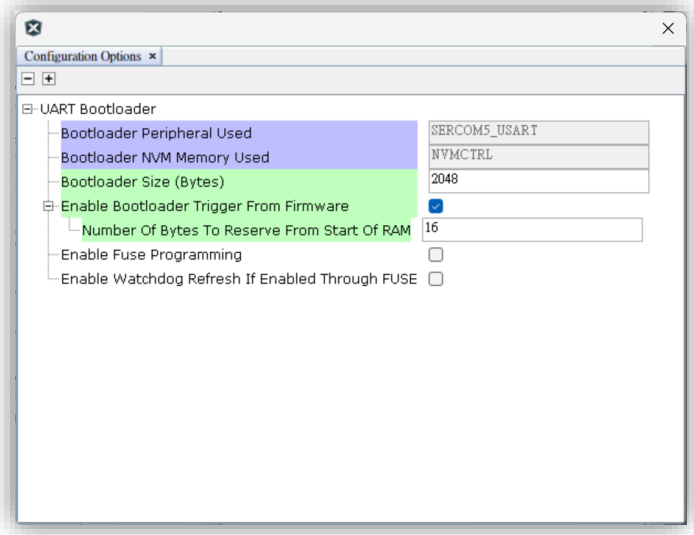



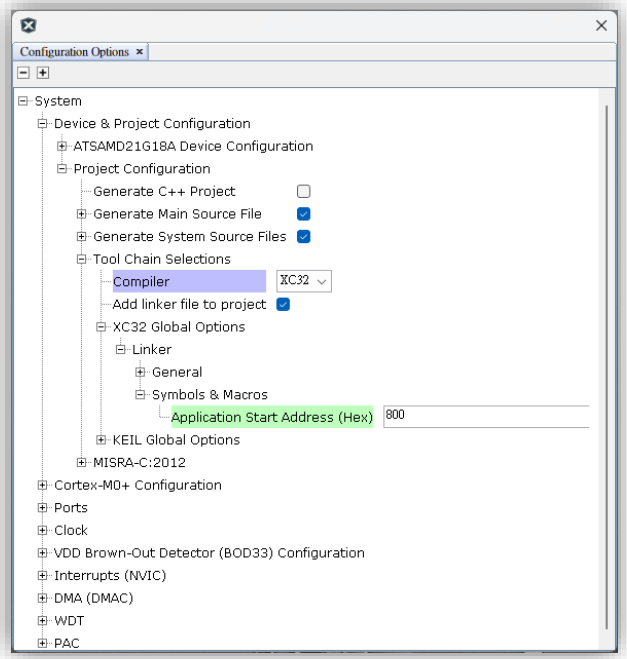
# Application Re-Arrangement

# Flash Re-Arrangement

- The UART bootloader occupies the first 0x800 bytes of flash; to preserve this region, the application must start at 0x800.
- For the Application start, you must set “Application Start Address (Hex)” under SYSTEM module in MHC let Bootloader know that how to handoff control to Application.




MICROCHIP

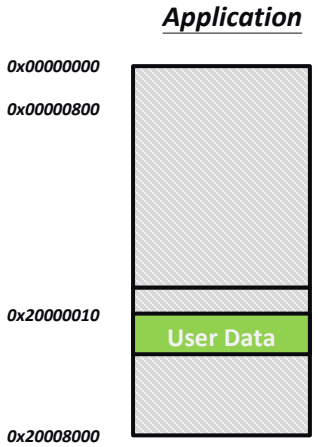
# SRAM Re-Arrangement

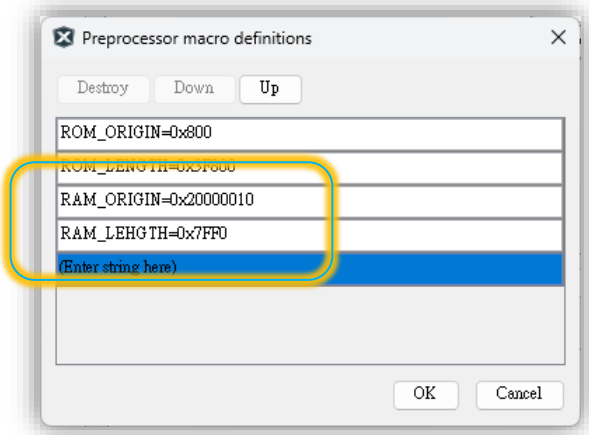
- For SRAM reserved, need manually define. Two method to do it.
  - At Linker symbols,

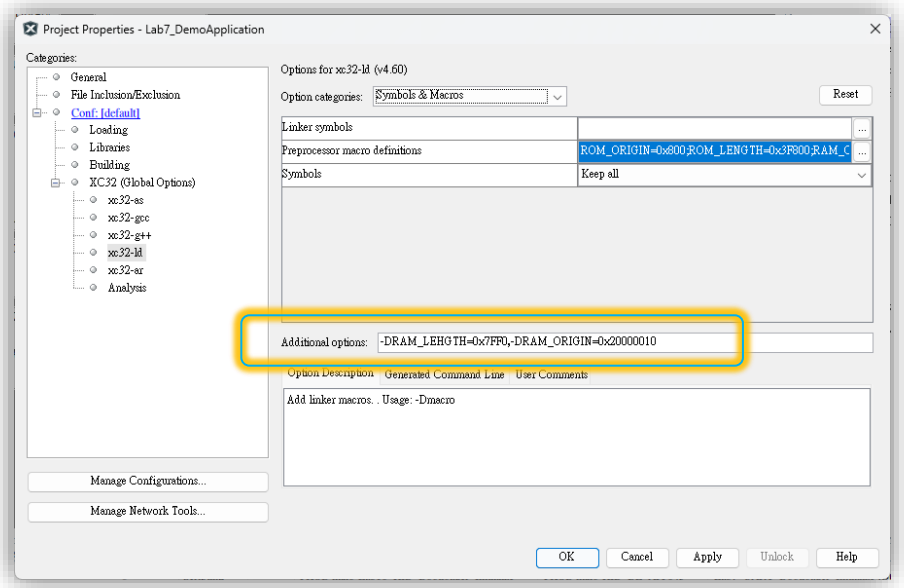
```
ROM_ORIGIN=0x800
ROM_LEHGTH=0x3F800
RAM_ORIGIN=0x20000010
RAM_LEHGTH=0x7FF0
```

  - At Additional optional,

```
-DRAM_ORIGIN=0x20000010,-DRAM_LEHGTH=0x7FF0
```




MICROCHIP

# 🛠 Lab8 Demo Application Relocation

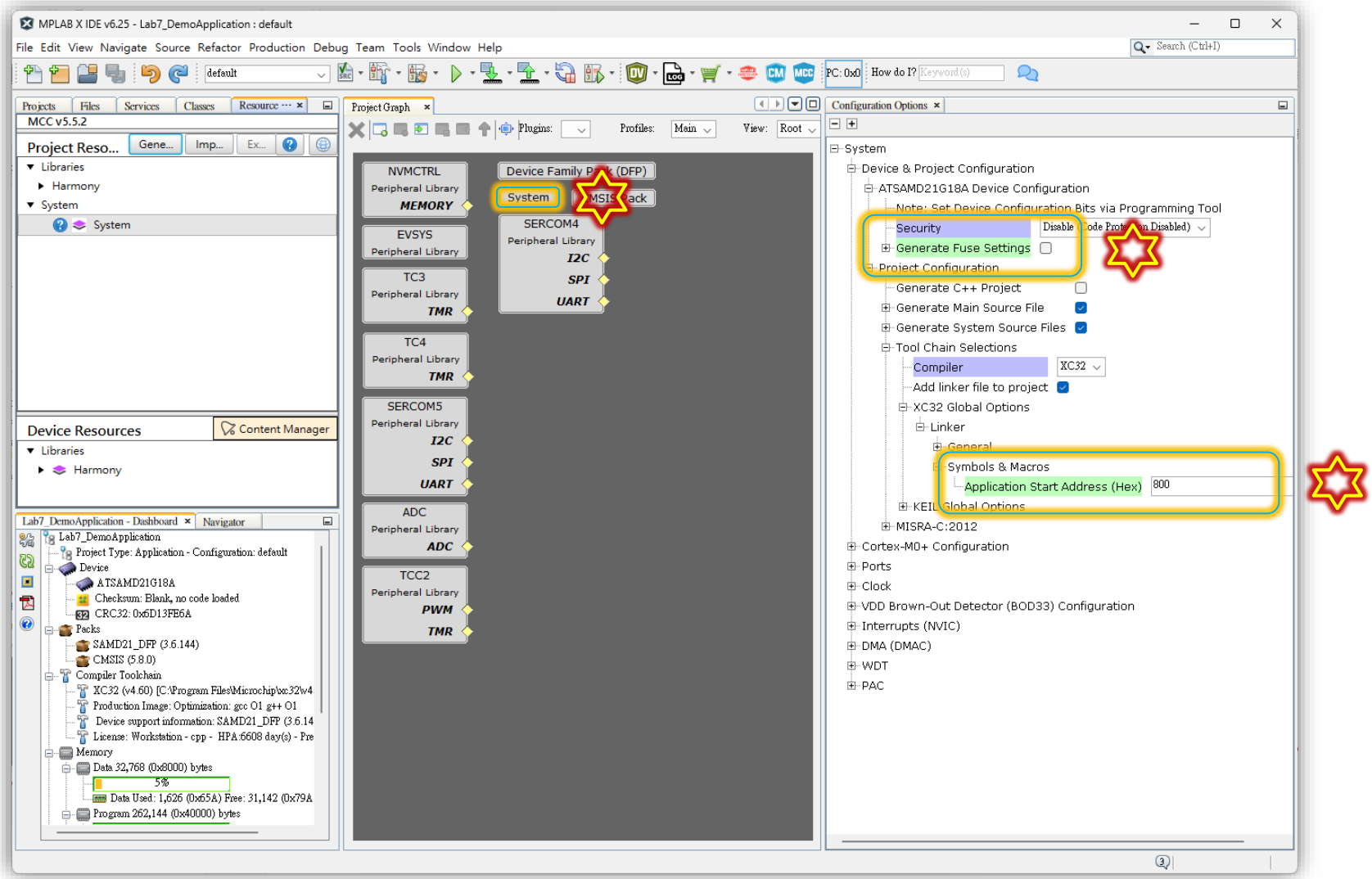
# 🛠 Lab8 Demo Application Relocation

- At this lab, we try to relocation and setting suitable parameter to fit UART Bootloader needs.
- Bypass Fuse via SYSTEM module in MHC.
- To set flash offset and  a reserved SRAM location.
- Set command line at “Execute After Build” generate a binary (.bin) file automatically after compilation.

- **Let’s go !**

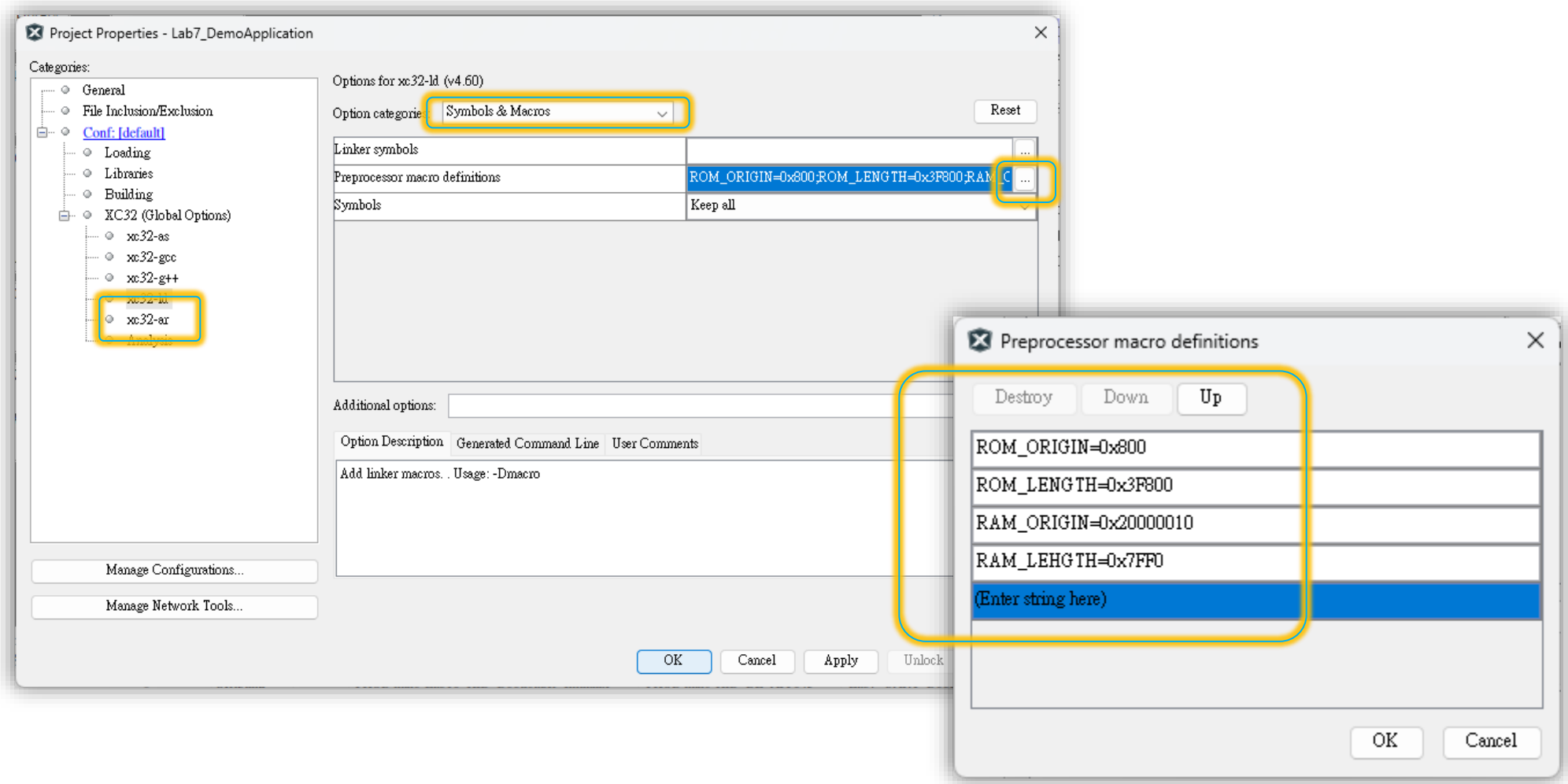
# 🛠 Lab8 Demo Application Relocation

- **Uncheck “Generate Fuse Settings” to bypass & set “Application Start Address”**



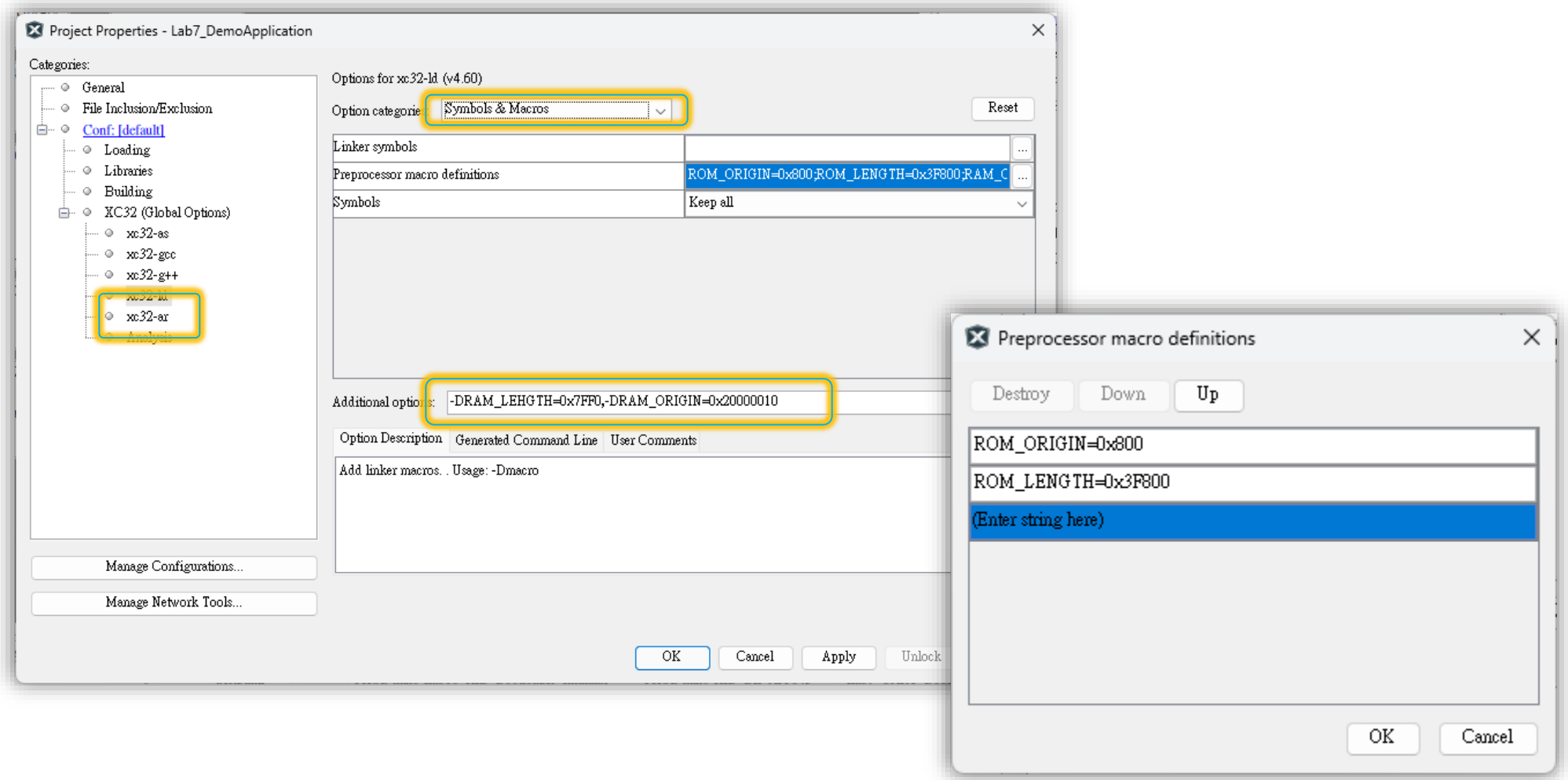
# 🛠 Lab8 Demo Application Relocation

- **Reserved SRAM location & Confirm Preprocessor Definition**



# 🛠 Lab8 Demo Application Relocation *Alternative*

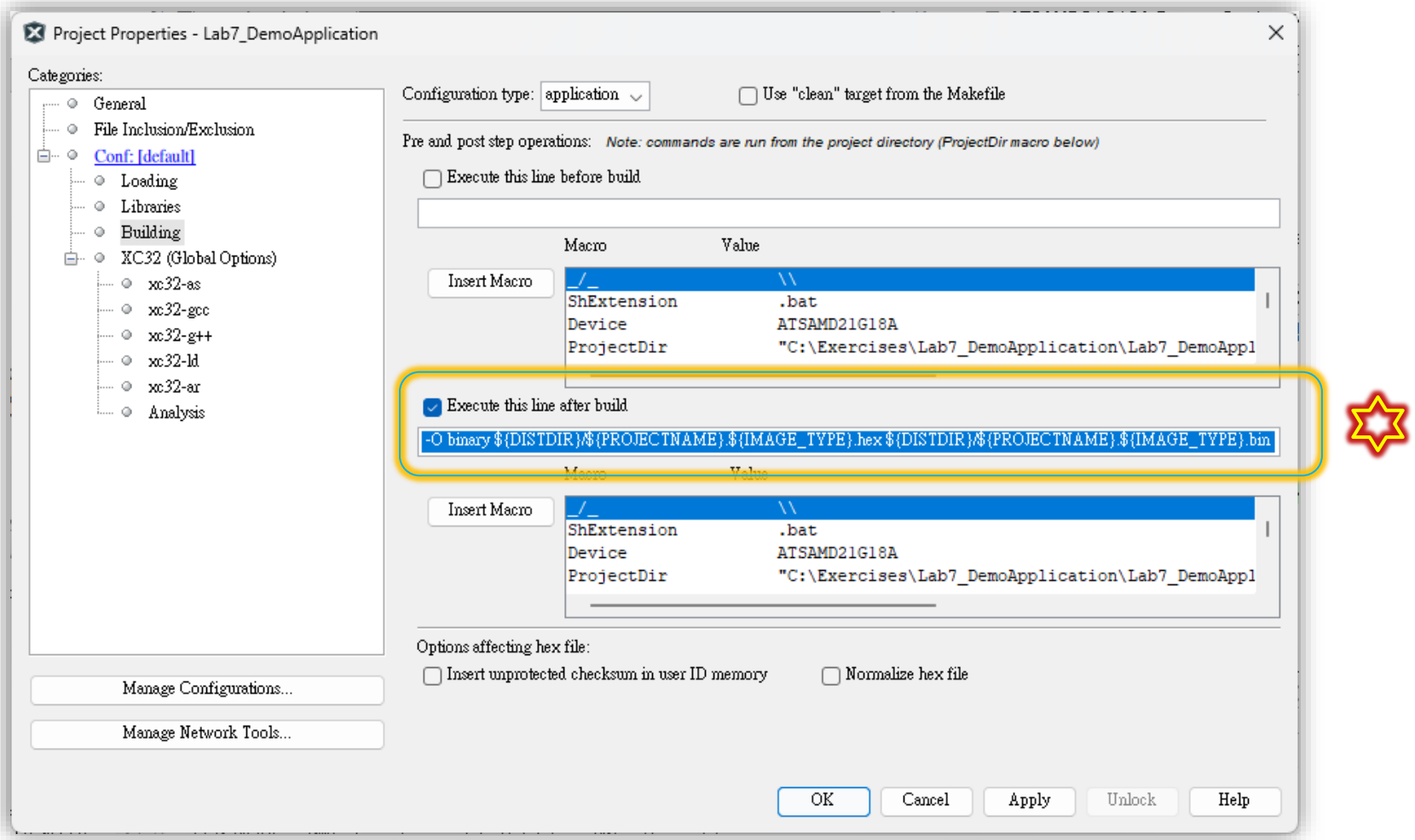
- **Reserved SRAM location & Confirm Preprocessor Definition**



# 🛠 Lab8 Demo Application Relocation

- **Check “Execute After Build” ” & set command line**

```
${MP_CC_DIR}/xc32-objcopy -I ihex -O binary ${DISTDIR}/${PROJECTNAME}.${IMAGE_TYPE}.hex ${DISTDIR}/${PROJECTNAME}.${IMAGE_TYPE}.bin
```



# 🛠 Lab8 Demo Application Relocation

- **Code Implementation state**

**Implement Bootloader Trigger method**
**TODO 8.01 - 8.02**

# 🛠 Lab8 Demo Application Relocation

- **Implement below function at main.c**

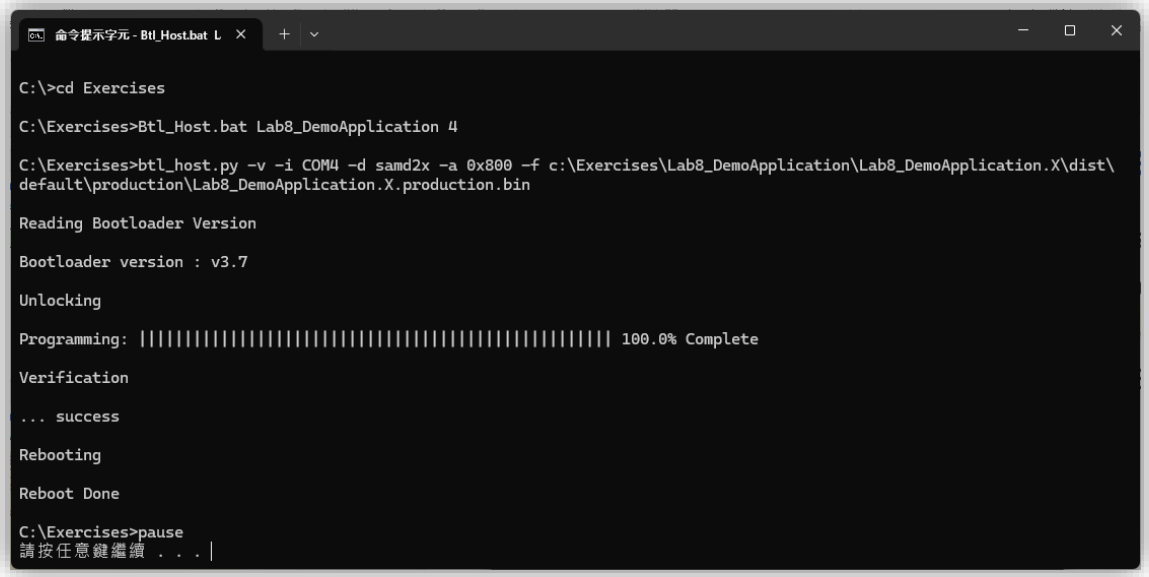
```
// TODO 8.01
#define BL_REQUEST              (0x5048434DU)
#define BL_REQUEST_LEN          (16)
#define BTL_TRIGGER_RAM_START   0x20000000U
static uint32_t *ramStart = (uint32_t *) BTL_TRIGGER_RAM_START;

int main(void)
{
    /* Initialize all modules */
    SYS_Initialize(NULL);

    while (true)
    {
        …
        // TODO 8.02
        if (!BT2_Get())
        {
            OLED_nCS_Clear();
            OLED_RS_Clear();
            memset(LCM_Blank, 0, 1024);
            LCM_DrawBitmap(0, 0, LCM_WIDTH, LCM_HEIGHT, LCM_Blank);
            sprintf((char *) USARTRTxBuffer, "Bootloader Trigger");
            LCM_DrawString(0, 0, (char *) USARTRTxBuffer);

            ramStart[0] = BL_REQUEST;
            ramStart[1] = BL_REQUEST;
            ramStart[2] = BL_REQUEST;
            ramStart[3] = BL_REQUEST;

            NVIC_SystemReset();
        }
    }
}
```

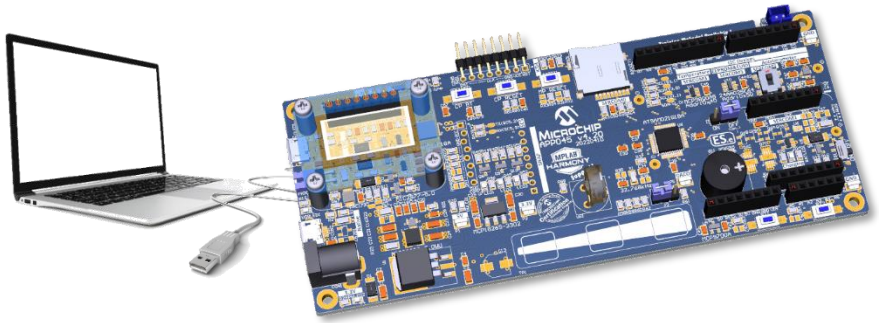
# 🛠 Lab8 Demo Application Relocation

- **Enter Bootloader mode via hold the BT1 & pressed Reset.**
- **Run *btl_host.bat* from the command line to perform a firmware update.**

```
Btl_Host.bat Lab7_DemoApplication 4
```

- **You could trigger bootloader mode via pressed BT2, after new firmware updated.**



MICROCHIP

# Firmware Combine

# 🛠 Lab9 Bootloader Application Combine

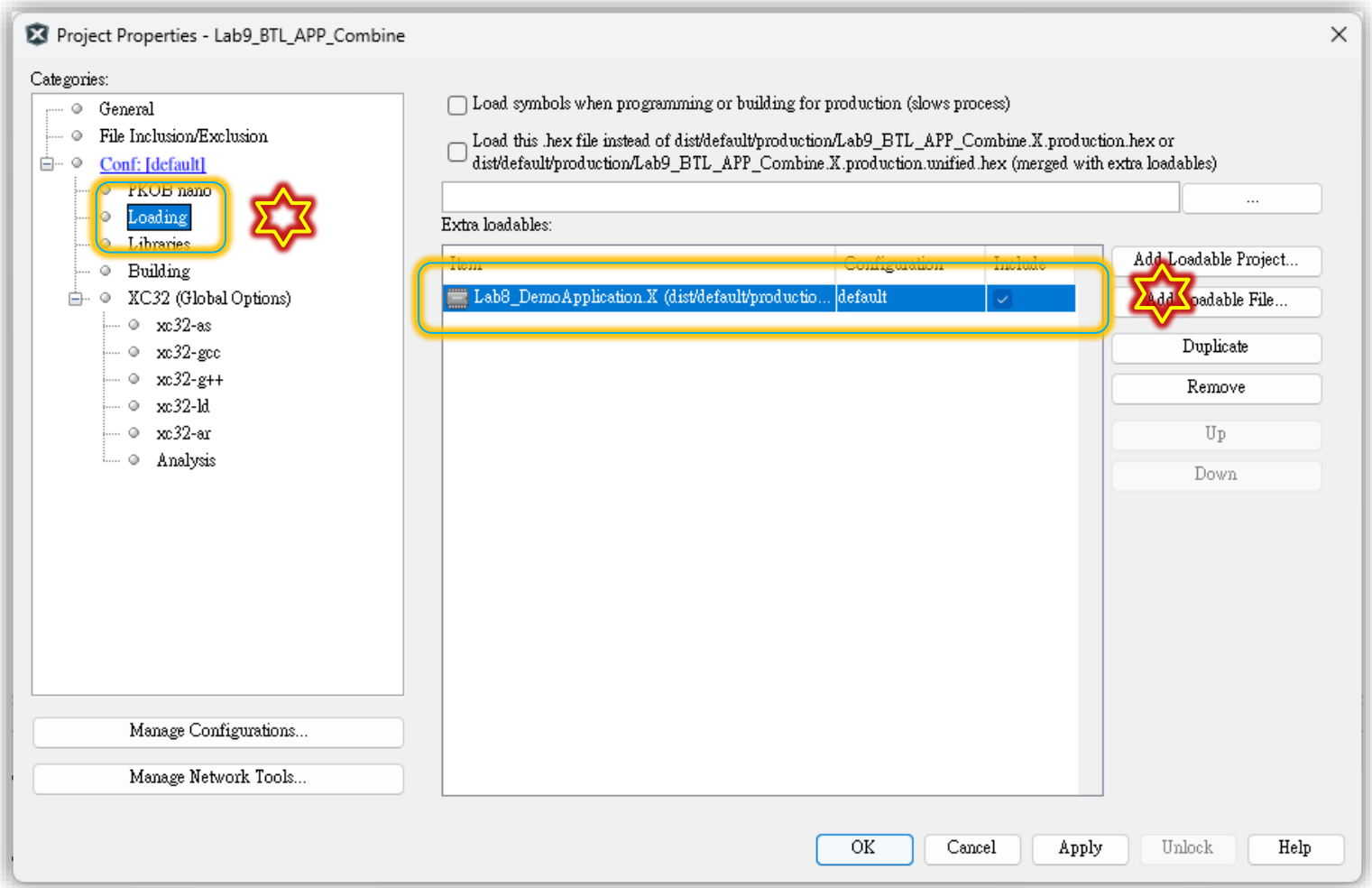
- At this lab, we try to combine two firmware at a one. It more useful for factory firmware programming.
- The new one contains both firmware Bootloader and Application.
- Open Bootloader project firstly
  and load Application project via “loading” setting in Project Properties.

- **Let’s go !**

MICROCHIP

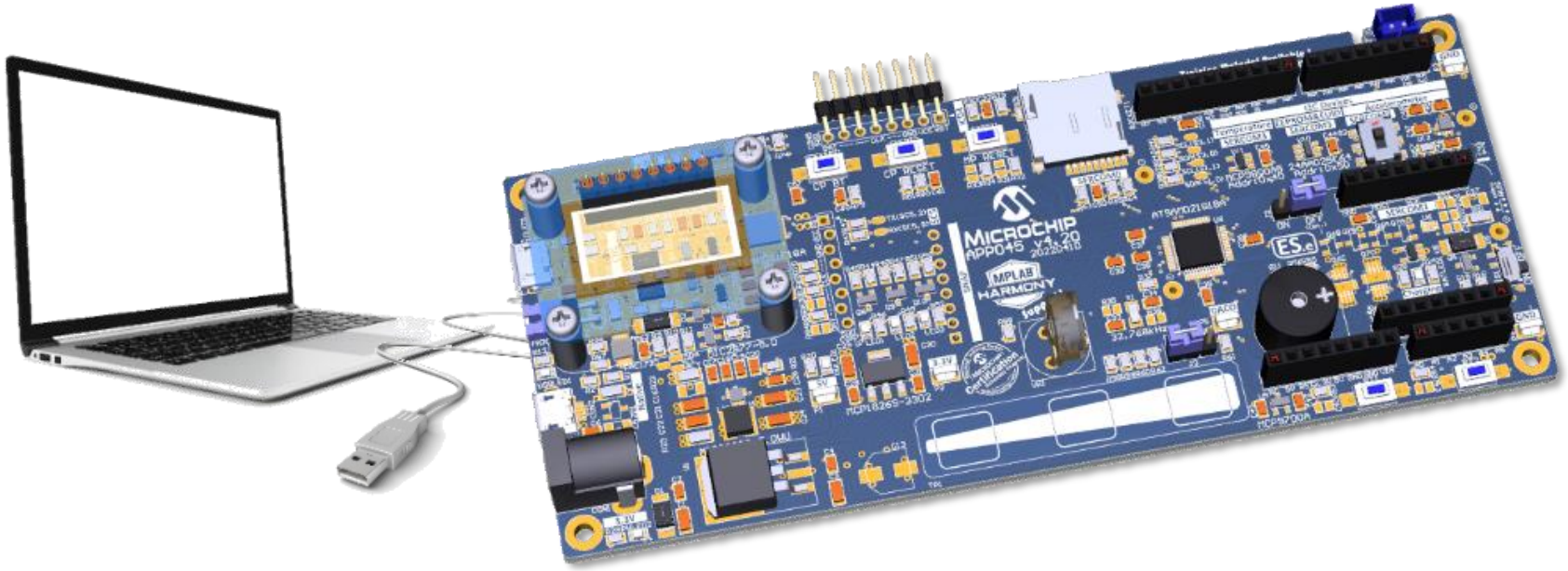
# 🛠 Lab9 Bootloader Application Combine

- load Application project via “loading” setting in Project Properties



# 🛠 Lab9 Bootloader Application Combine

- **The result same as lab8.**
- **But,**
  **Bootloader and Application programming together**
  **without via bootloader host (*btl_host.py*)**




MICROCHIP

# USB Device HID Bootloader

# USB Device HID Bootloader

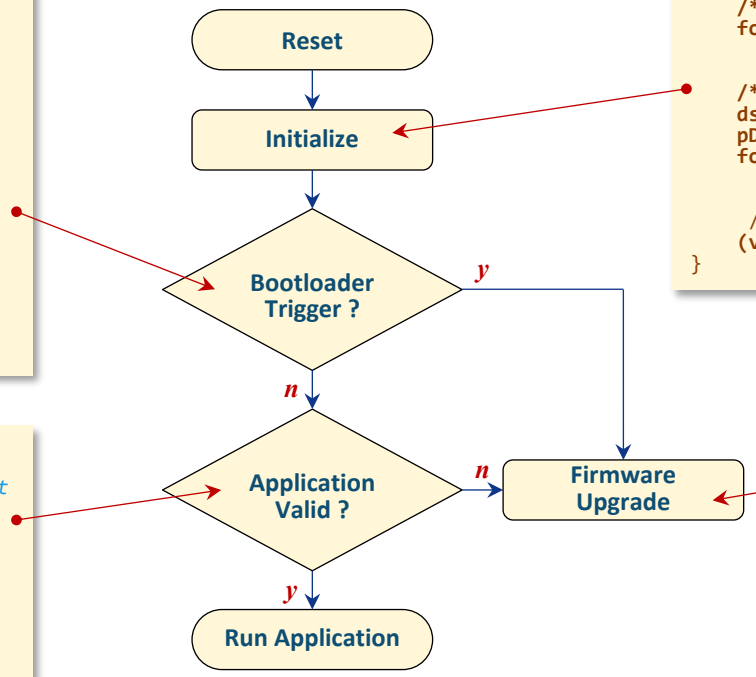
- Supported on Cortex-M based MCUs, MIPS based MCUs and MPUs
  e.g., ATSAMD21G18A (APP045 EVB), PIC32CX1025SG41128 (APP055 EVB), etc.
- Based on MCC Harmony USB Device HID driver to communicate.
- Takes Normalized Hex File as input
- Uses UBHA (Unified Bootloader Host Application)
  to receive the hex (*HEX) file from Host PC.
- Supports Live update.

# USB Device HID Bootloader

- **Below flowchart routine shows how to switch between the USB Device HID Bootloader and the Application at startup.**

```
#define BTL_TRIGGER_PATTERN (0x5048434DUL)
static uint32_t *ramStart = (uint32_t *)BTL_TRIGGER_RAM_START;

bool bootloader_Trigger(void)
{
    uint32_t i;

    …

    /* Check for Magic Number */
    if (BTL_TRIGGER_PATTERN == ramStart[0] &&
        BTL_TRIGGER_PATTERN == ramStart[1] &&
        BTL_TRIGGER_PATTERN == ramStart[2] &&
        BTL_TRIGGER_PATTERN == ramStart[3])
    {
        ramStart[0] = 0;
        DCACHE_CLEAN_BY_ADDR(ramStart, 4);
        return true;
    }

    /* Check for Button */
    if (!BT1_Get())
        return true;

    return false;
}
```

```
void run_Application(uint32_t address)
{
    uint32_t msp = *(uint32_t *)(address); // Application Start
    uint32_t reset_vector = *(uint32_t *)(address + 4U);

    if (msp == 0xffffffffU)
        return;

    …

    __set_MSP(msp); // Re-initial Stack Pointer
    asm("bx %0"::"r" (reset_vector)); // Jumper to Application
}
```



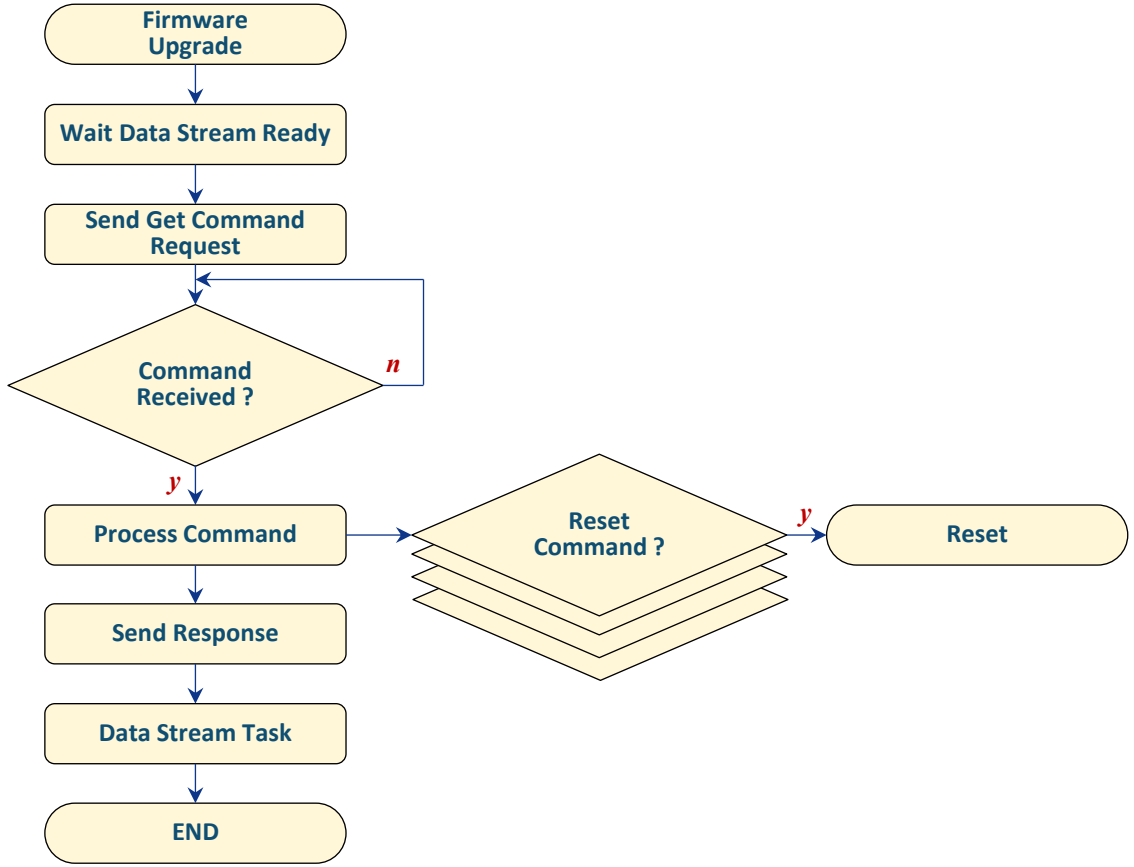

```
void __attribute__((noinline, section(".romfunc.Reset_Handler"))) Reset_Handler(void)
{
    register uint32_t count;
    uint32_t *pSrc, *pDst;
    uintptr_t src, dst;

    src = (uintptr_t)&_etext;
    pSrc = (uint32_t *)src;      /* flash functions start after .text */
    dst = (uintptr_t)&_sdata;
    pDst = (uint32_t *)dst;      /* boundaries of .data area to init */

    /* Init .data */
    for (count = 0U; count < (((uint32_t)&_edata - (uint32_t)dst) / 4U); count++)
        pDst[count] = pSrc[count];

    /* Init .bss */
    dst = (uintptr_t)&_sbss;
    pDst = (uint32_t *)dst;
    for (count = 0U; count < (((uint32_t)&_ebss - (uint32_t)dst) / 4U); count++)
        pDst[count] = 0U;

     /* Branch to application's main function */
    (void)main();
}
```

```
void bootloader_USB_DEVICE_HID_Tasks( void )
{
    switch ( )
    {
        case BOOTLOADER_OPEN_DATASTREAM: Break;
        case BOOTLOADER_GET_COMMAND: Break;
        case BOOTLOADER_PROCESS_COMMAND: Break;
        case BOOTLOADER_SEND_RESPONSE: Break;
        case BOOTLOADER_WAIT_FOR_DONE: Break;
        case BOOTLOADER_CLOSE_DATASTREAM: Break;
        case BOOTLOADER_ENTER_APPLICATION:
            bootloader_TriggerReset(); Break;
        case BOOTLOADER_ERROR: Break;
    }
    DATASTREAM_Tasks();
}
```

MICROCHIP

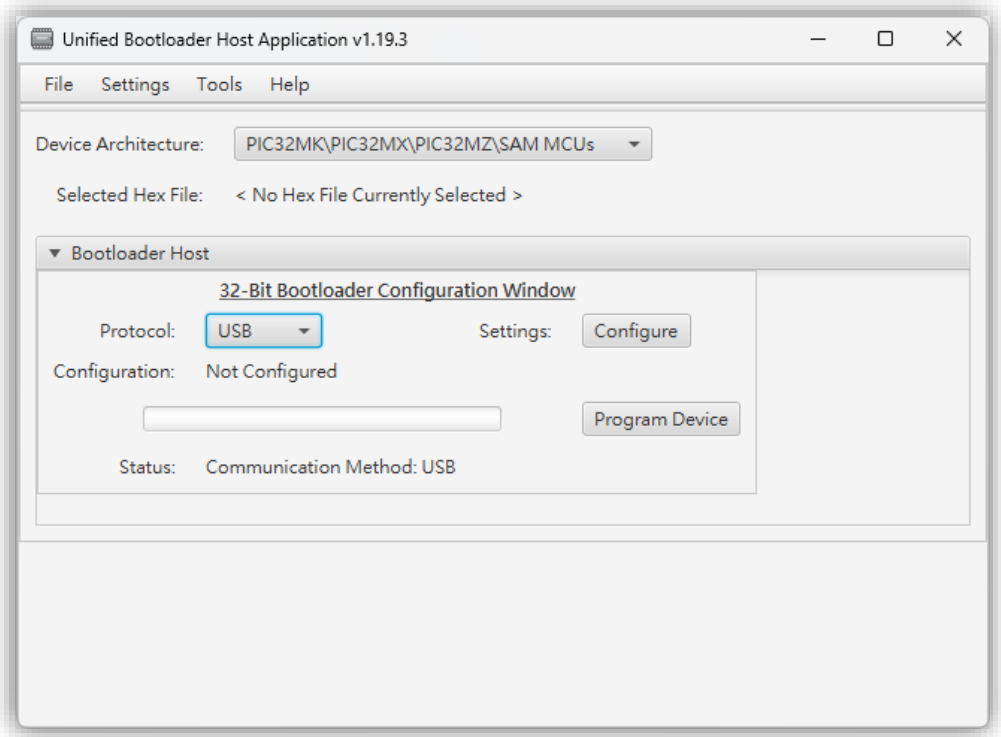
# USB Device HID Bootloader

- Below flowchart routine shows how the firmware upgrade works.
- Send command request and waiting command, parses commands, and executes them; upon receiving a reset command, the system reboots.



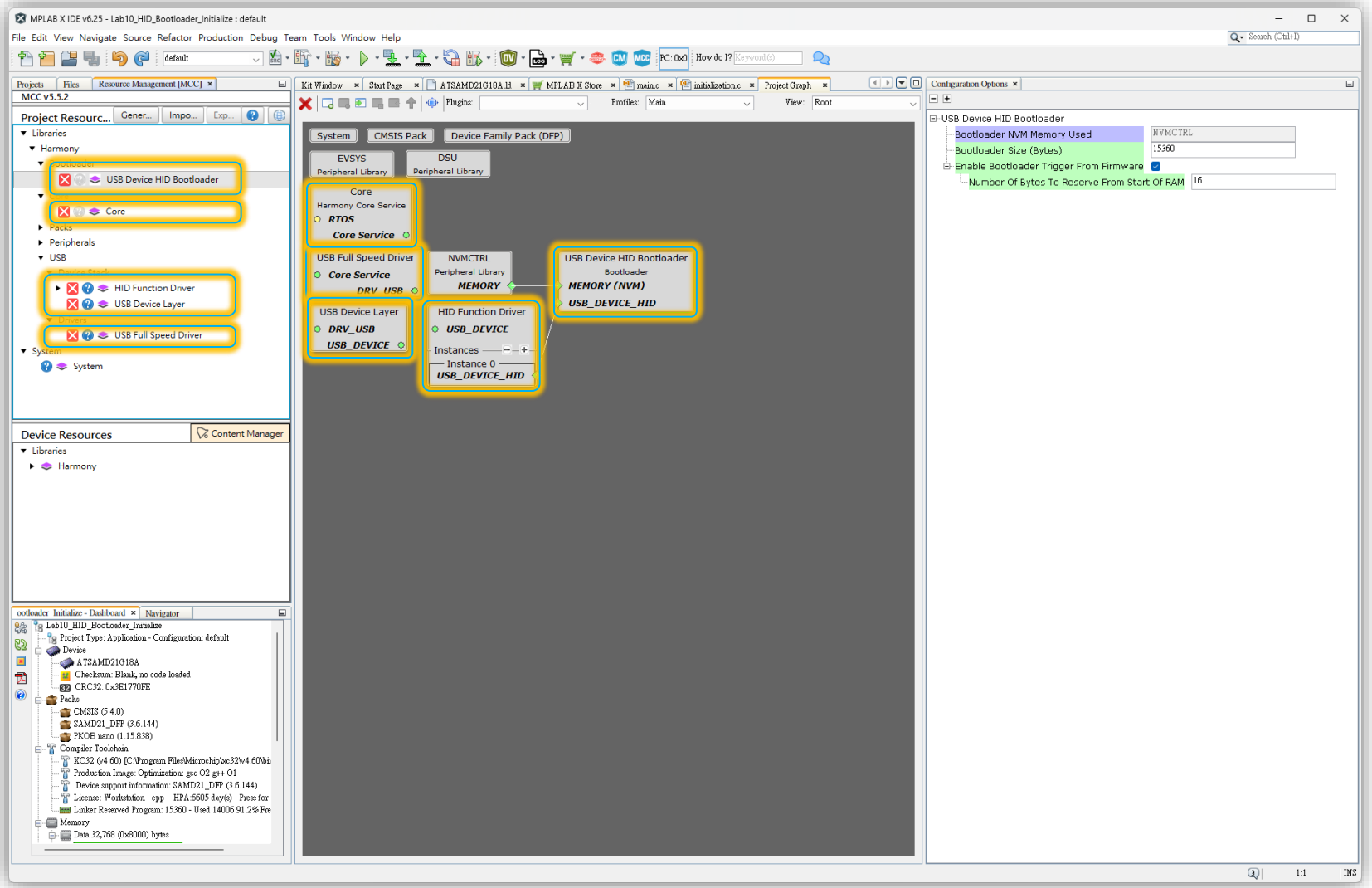
# Unified Bootloader Host Application

- **UBHA (Unified Bootloader Host Application) used to communicate with the USB Device HID Bootloader running on the device**
- **It implements the Unified bootloader protocol required to communicate from host PC**
- **It sends the Normalized Hex File of the application to be bootloader.**
- ***Important! UART Protocol is not supported in MCC Harmony using this tool.***



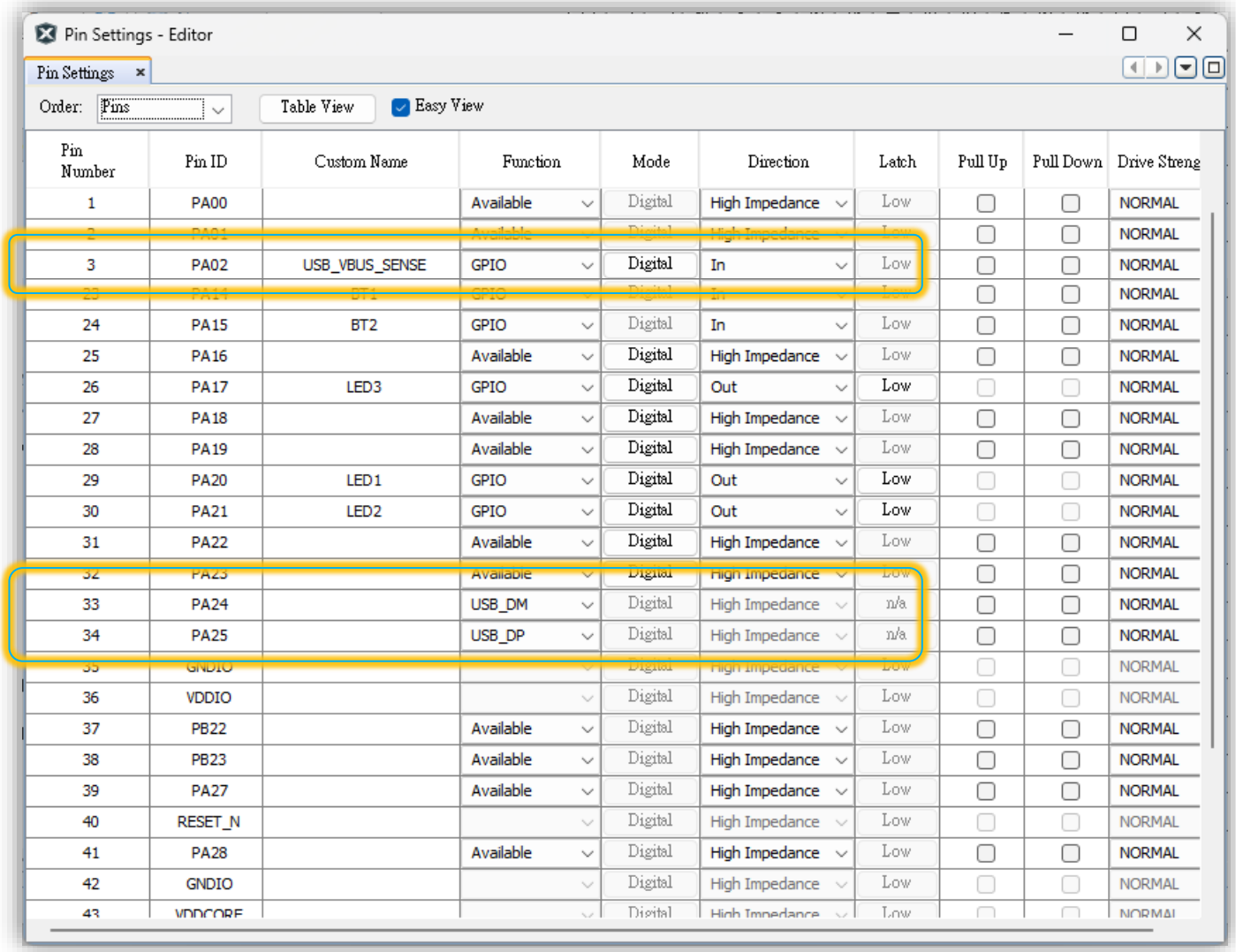
# Harmony Bootloader - USB Device HID Bootloader Configurator

## Add USB Device HID Bootloader and USB Device related in MHC



# Harmony Bootloader - USB Device HID Bootloader Configurator

## Configure USB related pins at Pin Table



# 🛠 Lab10 USB Device HID Bootloader Initialize

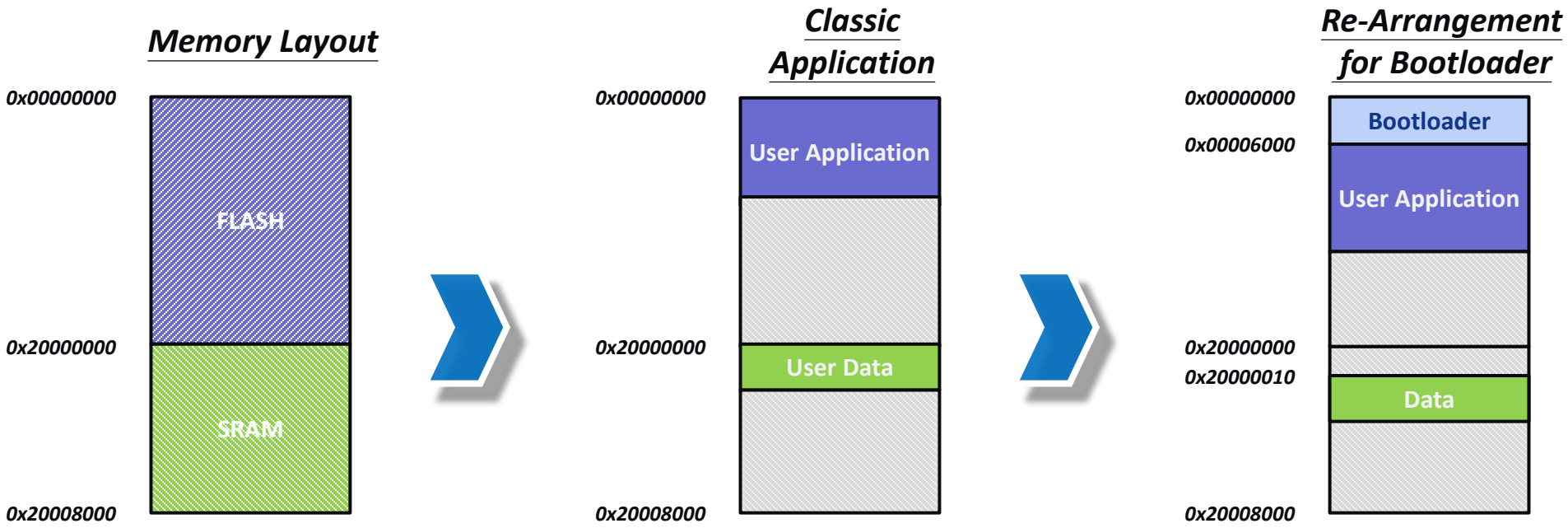
# 🛠 Lab10 USB Device HID Bootloader Initialize

- At this lab, we try to getting started with USB Device HID Bootloader.
- Add UART USB Device HID and USB Device related in MHC and assign parameters and application start address.
- Implement suitable bootloader trigger method use to trigger Bootloader.
- Implement an LEDs status to indicate Bootloader enter or not.
- Try to use UBHA to upload “DemoApplicationUSB” firmware to APP045 EVB via USB Device HID based on USB Device HID Bootloader.

- **Let’s go !**

# Memory Re-Arrangement for Bootloader *ATSAMD21 as an Example*

- **In USB Device HID Bootloader application,**
  **Bootloader occupies the first 0x6000 bytes of flash and the first 16 Bytes of SRAM.**

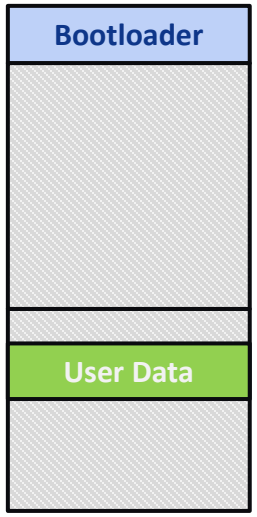



MICROCHIP

# Memory Re-Arrangement for Bootloader *cont.* *ATSAMD21 as an Example*

- **Set two independent linker scripts (**.ld*),**
  **one for the Bootloader and one for the Application to define their memory allocations.**
- **In the linker scripts,**
  **reserved separately for the Bootloader and the Application.**

***Bootloader***



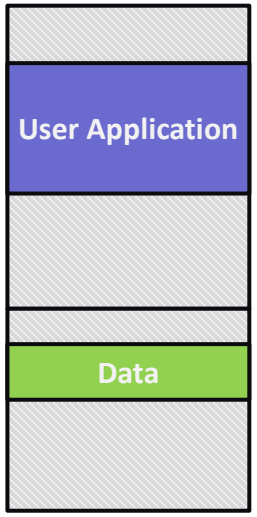

***Linker Script for Bootloader***

```
MEMORY
{
  rom (LRX) : ORIGIN = 0x0, LENGTH = 0x6000
  ram (WX!R) : ORIGIN = 0x20000010, LENGTH = 0x7FF0
  config_00804000 : ORIGIN = 0x00804000, LENGTH = 0x4
  config_00804004 : ORIGIN = 0x00804004, LENGTH = 0x4
}
```

***Linker Script for Application***

```
MEMORY
{
  rom (LRX) : ORIGIN = 0x6000, LENGTH = 0x3A000
  ram (WX!R) : ORIGIN = 0x20000010, LENGTH = 0x7FF0
  config_00804000 : ORIGIN = 0x00804000, LENGTH = 0x4
  config_00804004 : ORIGIN = 0x00804004, LENGTH = 0x4
}
```

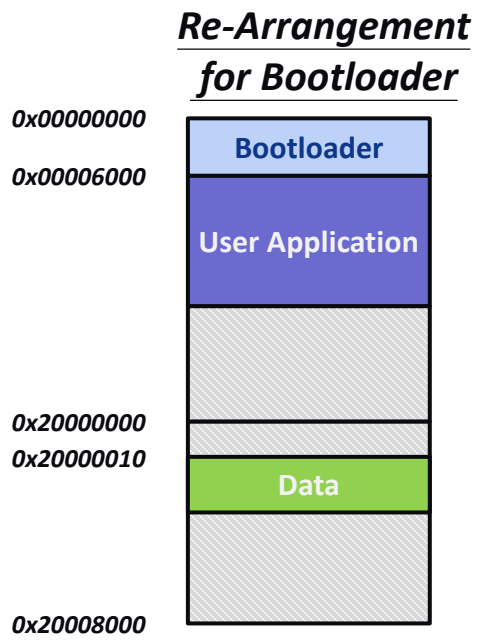
***Application***



# Link Script for ATSAMD21x18A

- In MCC Harmony project, apply preprocessor definition to dynamic management linker script.
  The corresponding file will be generated
  after press “generate” button at MHC depend on your setting.
  - For Bootloader *(Optional)*,
    -DROM_ORIGIN = *0x00000000*, -DROM_LEHGTH = *0x6000*
    -DRAM_ORIGIN = *0x20000010*, -DRAM_LEHGTH = *0x7FF0*

  For User Application,
    -DROM_ORIGIN = *0x00006000*, -DROM_LEHGTH = *0x3A000*
    -DRAM_ORIGIN = *0x20000010*, -DRAM_LEHGTH = *0x7FF0*

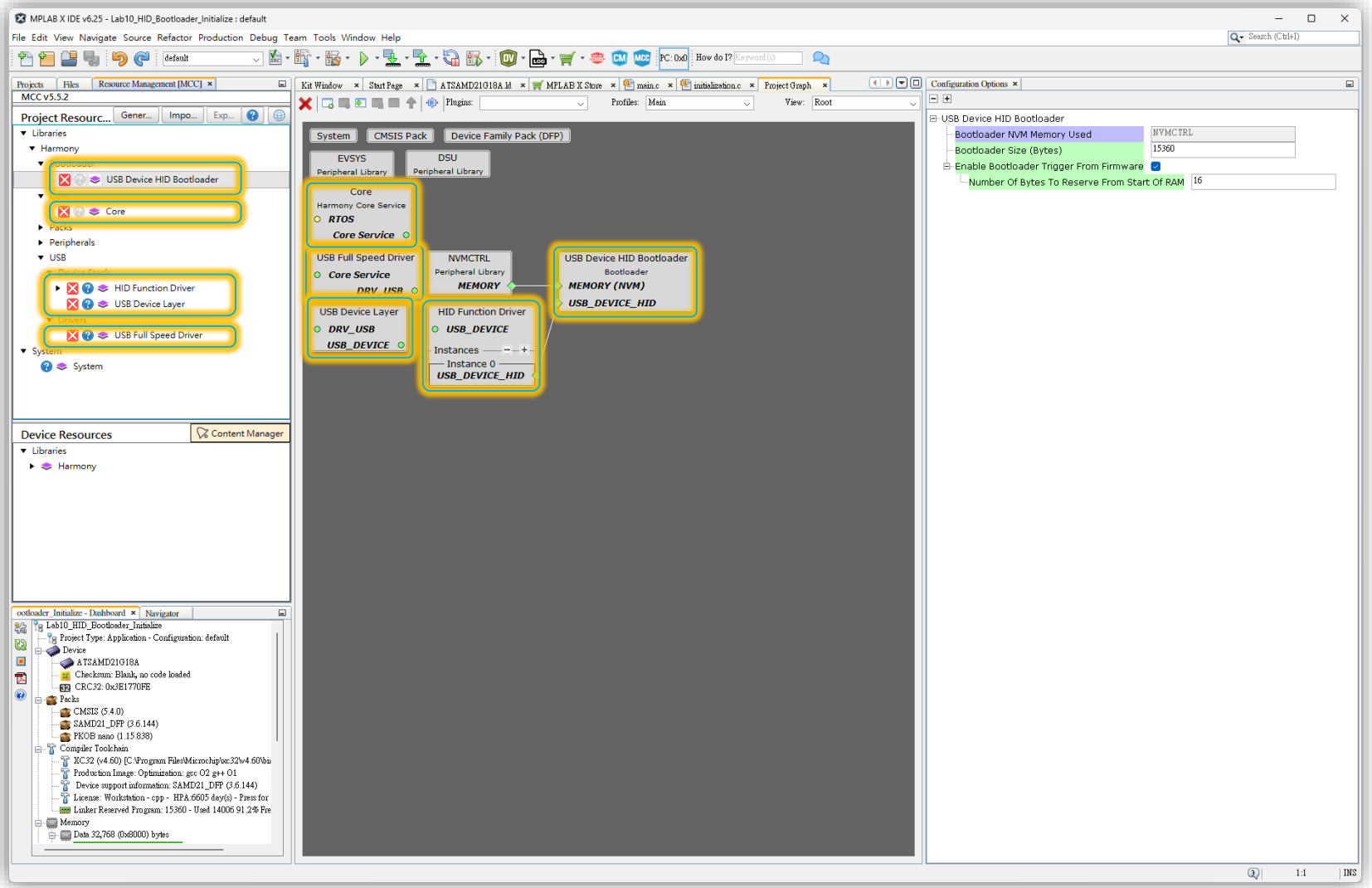



- These definitions are applied in the linker script,
  and during the linking process,
  they ensure that the program code and data are in the designated memory areas.

MICROCHIP

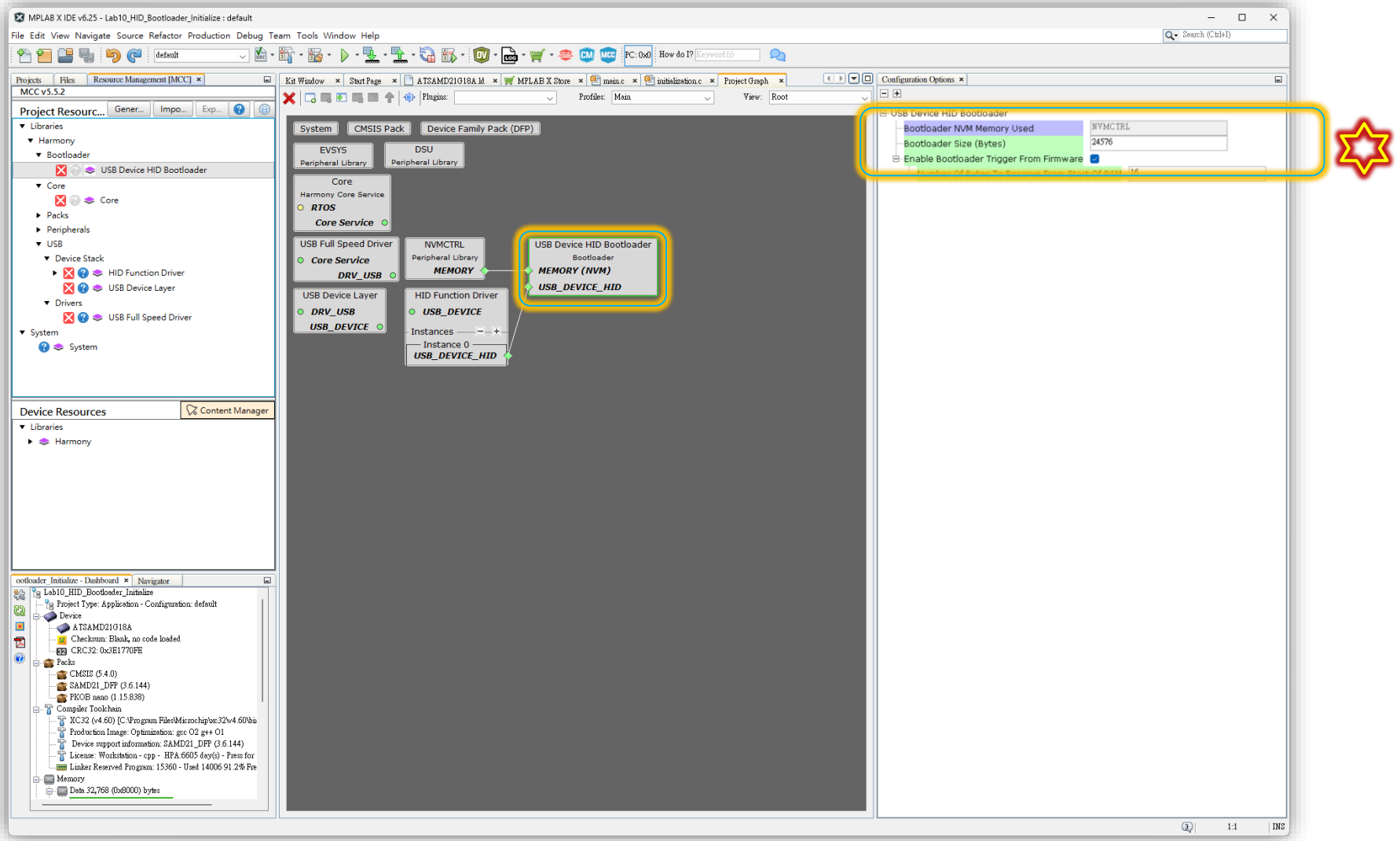
# 🛠 Lab10 USB Device HID Bootloader Initialize

- **Create a new project and USB Device HID Bootloader and USB Device related in MHC.**



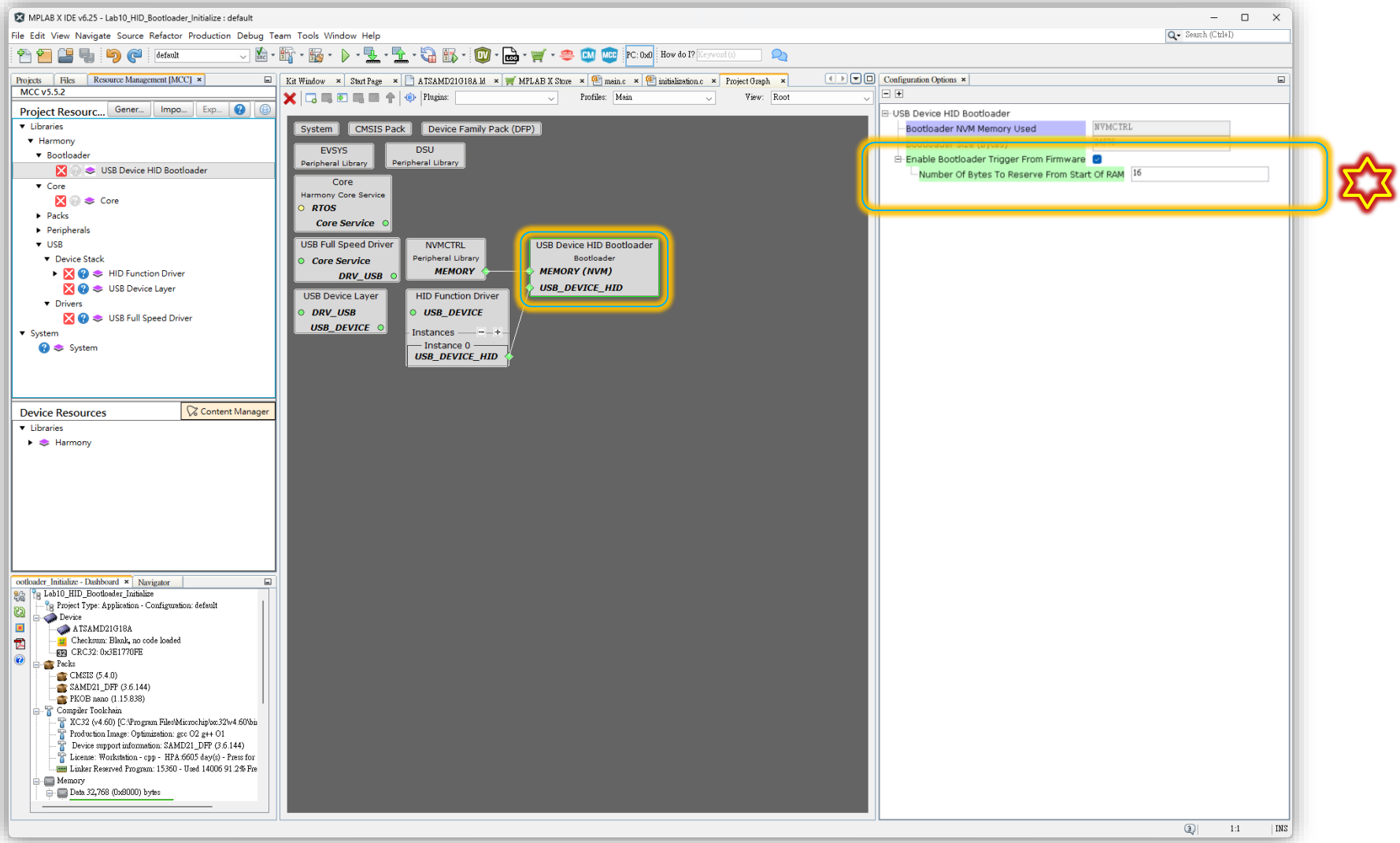
# 🛠 Lab10 USB Device HID Bootloader Initialize

- **Set “Bootloader Size (Bytes)” to 24576.**



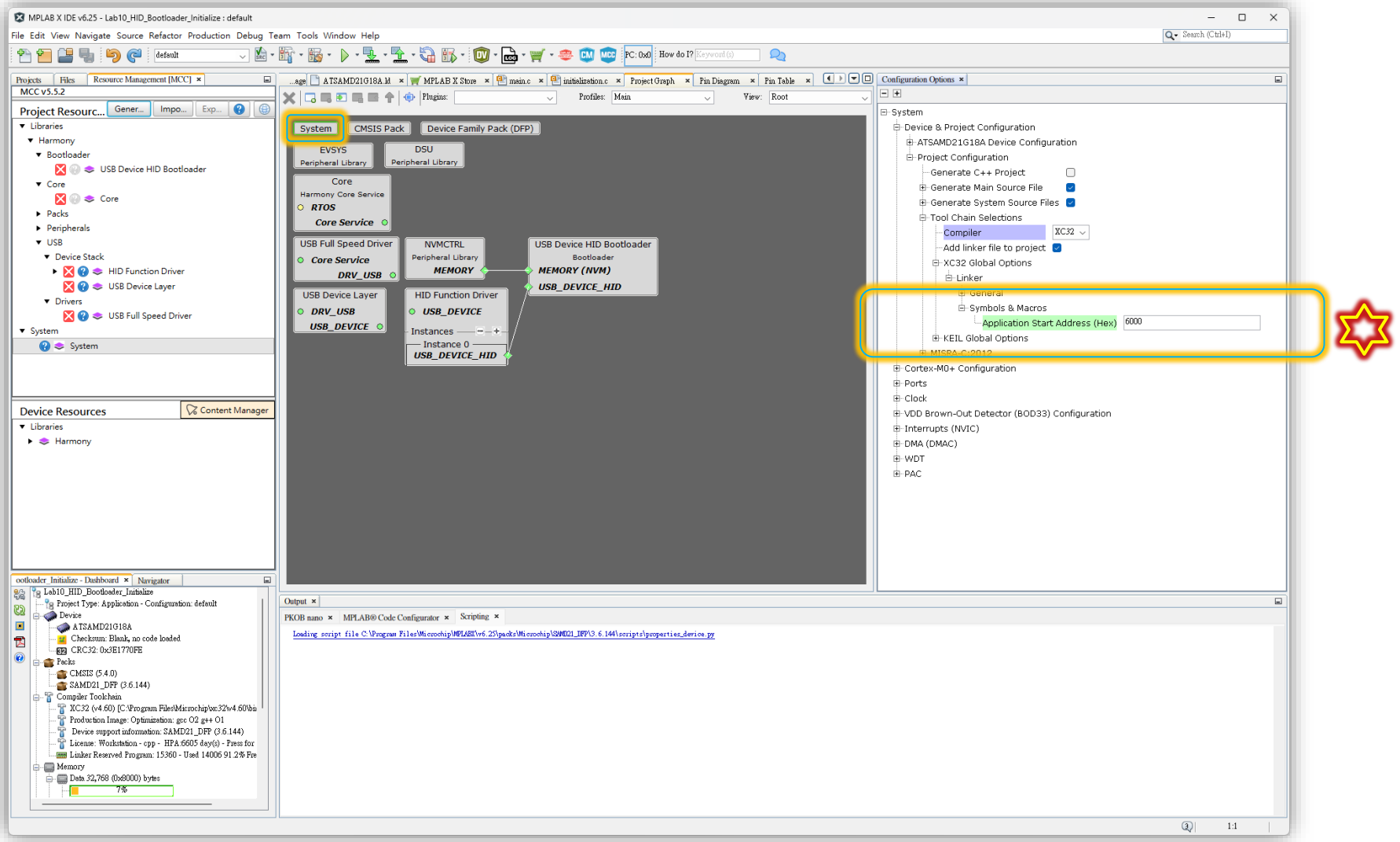
# 🛠 Lab10 USB Device HID Bootloader Initialize

- **Check “Enable Bootloader Trigger From Firmware” and Reserve 16 bytes RAM.**



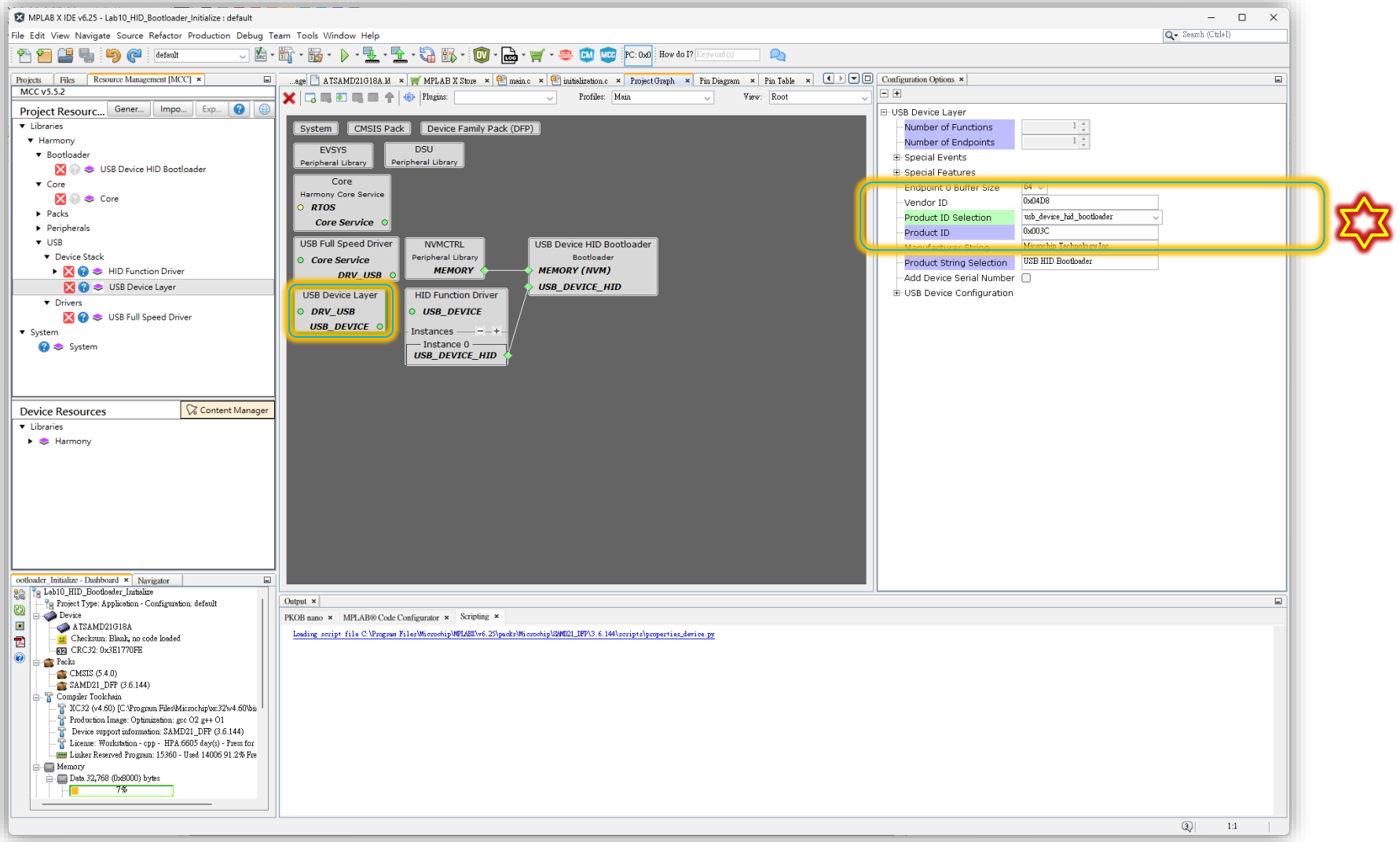
# 🛠 Lab10 USB Device HID Bootloader Initialize

- **Check Application Start Address (Hex) is 6000.**



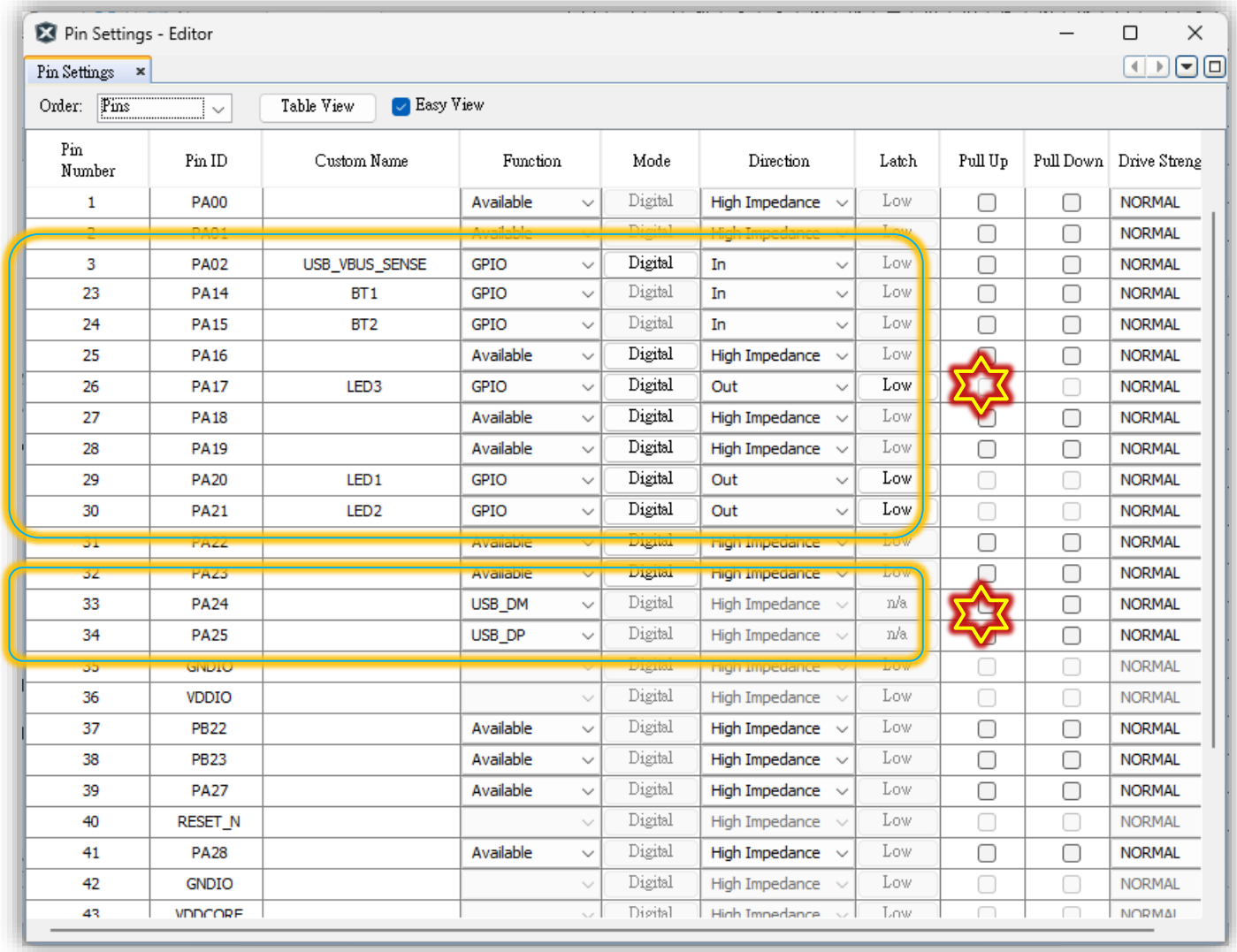
# 🛠 Lab10 USB Device HID Bootloader Initialize

- **Set VID/PID to 0x04D8, 003C (usb_device_hid_bootloader).**



# 🛠 Lab10 USB Device HID Bootloader Initialize

- **Configure USB related and Application pins at Pin Table**

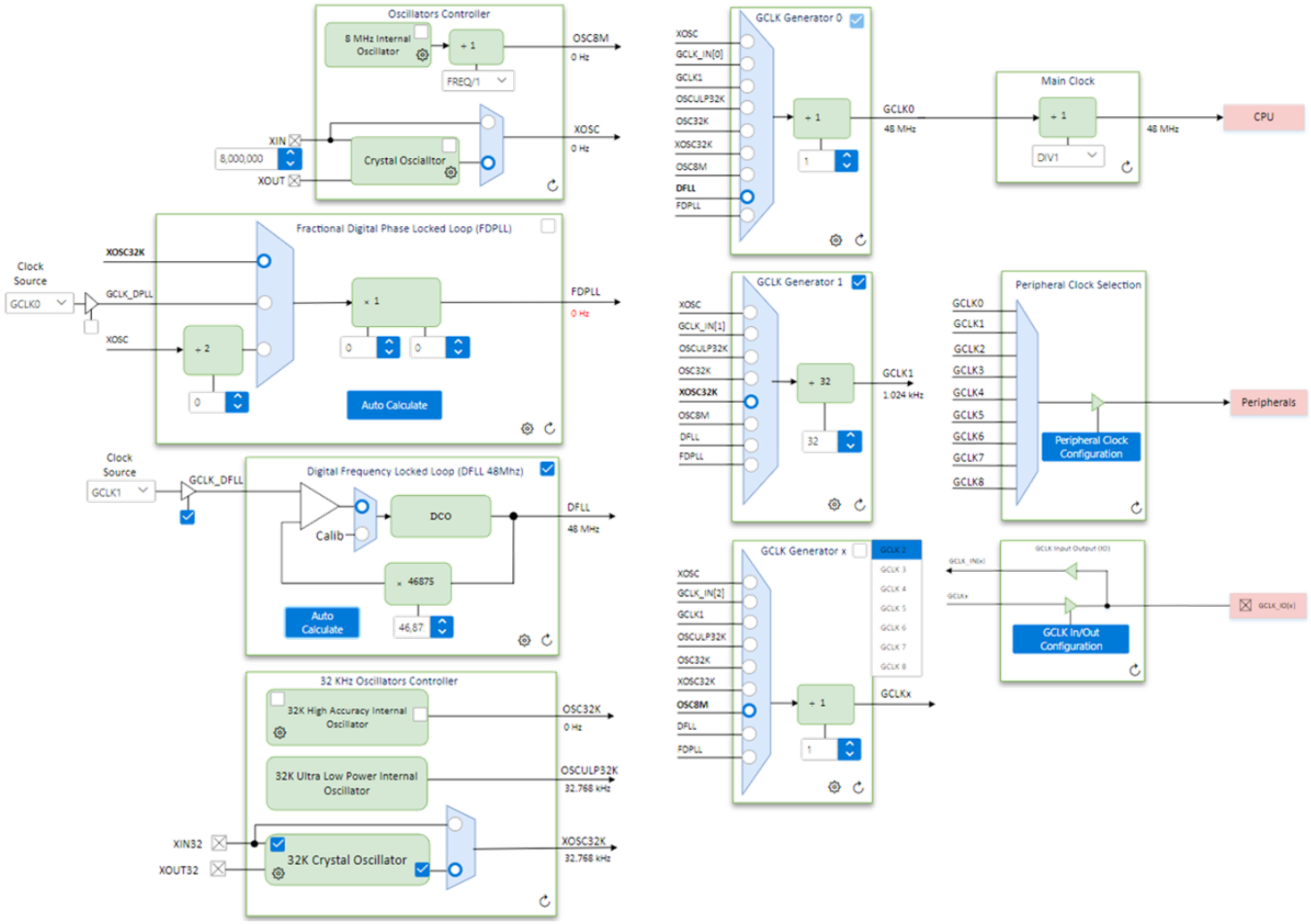



| Pin Number | Pin ID | Custom Name | Function | Mode | Direction | Latch | Pull Up | Pull Down | Drive Streng |
|---|---|---|---|---|---|---|---|---|---|
| 1 | PA00 | | Available | Digital | High Impedance | Low | ☐ | ☐ | NORMAL |
| 3 | PA02 | USB_VBUS_SENSE | GPIO | Digital | In | Low | ☐ | ☐ | NORMAL |
| 23 | PA14 | BT1 | GPIO | Digital | In | Low | ☐ | ☐ | NORMAL |
| 24 | PA15 | BT2 | GPIO | Digital | In | Low | ☐ | ☐ | NORMAL |
| 25 | PA16 | | Available | Digital | High Impedance | Low | ☐ | ☐ | NORMAL |
| 26 | PA17 | LED3 | GPIO | Digital | Out | Low | ☐ | ☐ | NORMAL |
| 27 | PA18 | | Available | Digital | High Impedance | Low | ☐ | ☐ | NORMAL |
| 28 | PA19 | | Available | Digital | High Impedance | Low | ☐ | ☐ | NORMAL |
| 29 | PA20 | LED1 | GPIO | Digital | Out | Low | ☐ | ☐ | NORMAL |
| 30 | PA21 | LED2 | GPIO | Digital | Out | Low | ☐ | ☐ | NORMAL |
| 33 | PA24 | | USB_DM | Digital | High Impedance | n/a | ☐ | ☐ | NORMAL |
| 34 | PA25 | | USB_DP | Digital | High Impedance | n/a | ☐ | ☐ | NORMAL |
| 36 | VDDIO | | | Digital | High Impedance | Low | ☐ | ☐ | NORMAL |
| 37 | PB22 | | Available | Digital | High Impedance | Low | ☐ | ☐ | NORMAL |
| 38 | PB23 | | Available | Digital | High Impedance | Low | ☐ | ☐ | NORMAL |
| 39 | PA27 | | Available | Digital | High Impedance | Low | ☐ | ☐ | NORMAL |
| 40 | RESET_N | | | Digital | High Impedance | Low | ☐ | ☐ | NORMAL |
| 41 | PA28 | | Available | Digital | High Impedance | Low | ☐ | ☐ | NORMAL |
| 42 | GNDIO | | | Digital | High Impedance | Low | ☐ | ☐ | NORMAL |

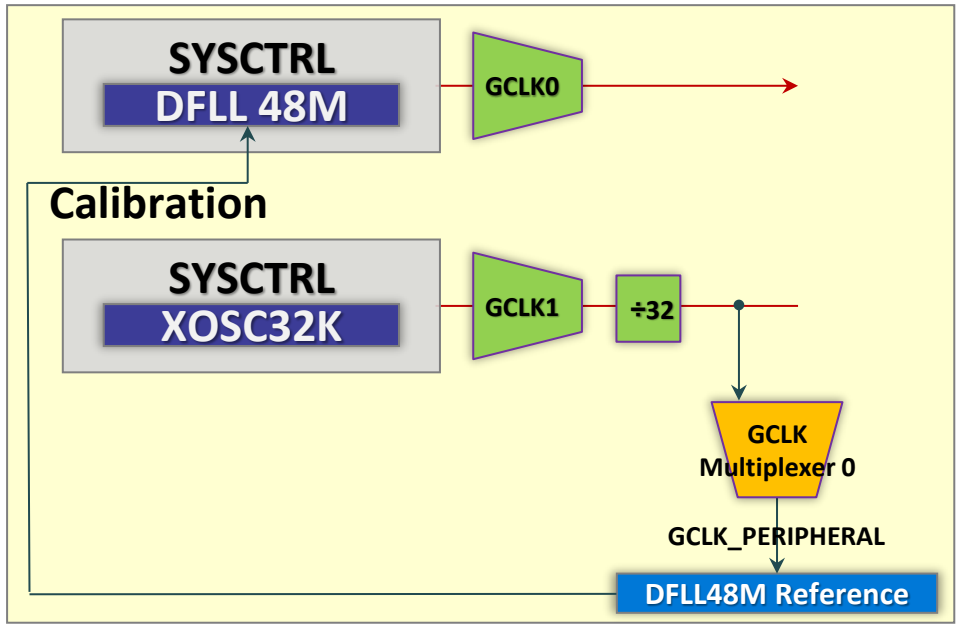
MICROCHIP

# 🛠 Lab10 USB Device HID Bootloader Initialize

- **Configure DFLL48M as Close loop as source of main clock**



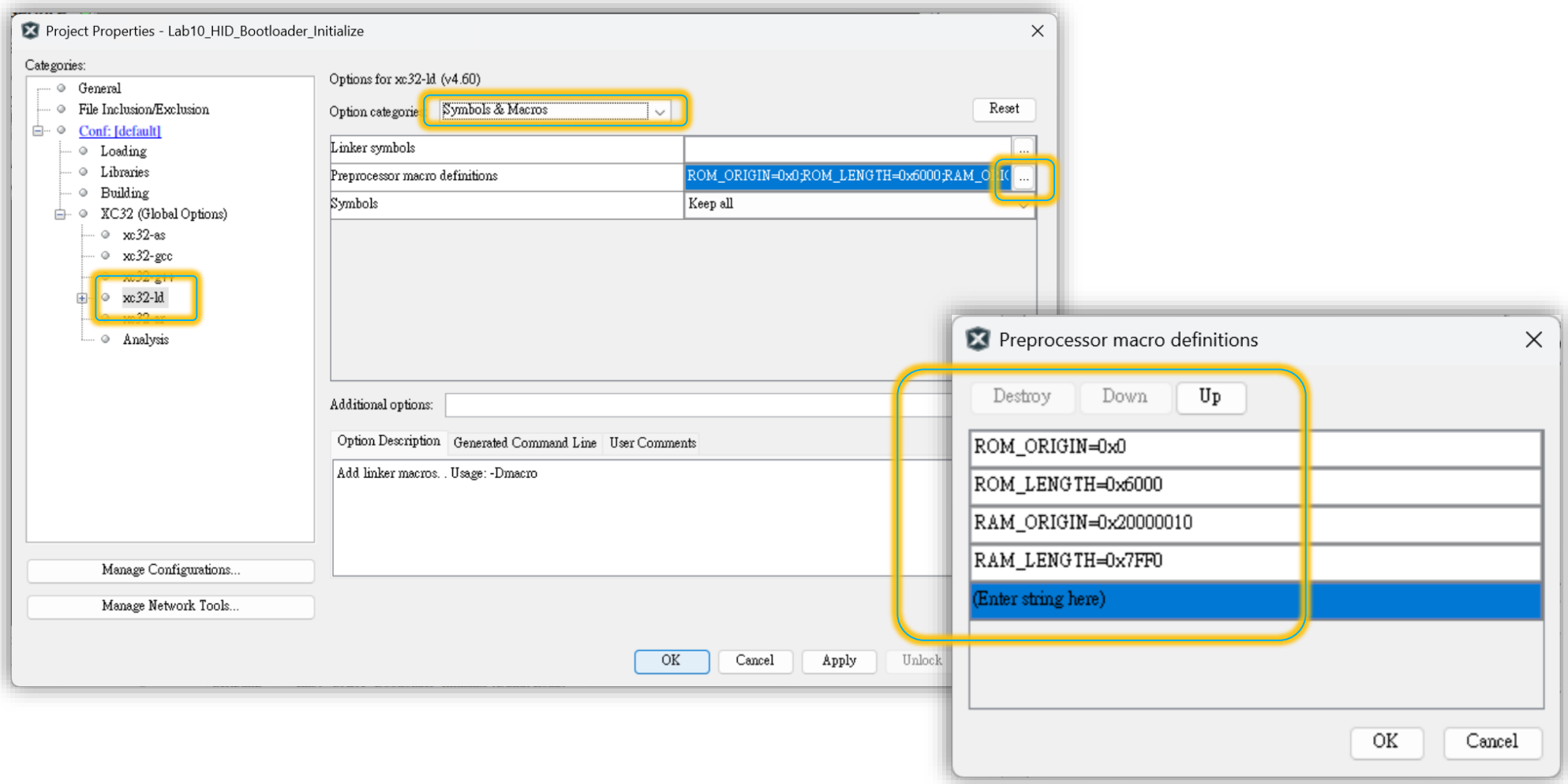

## Close Loop




MICROCHIP

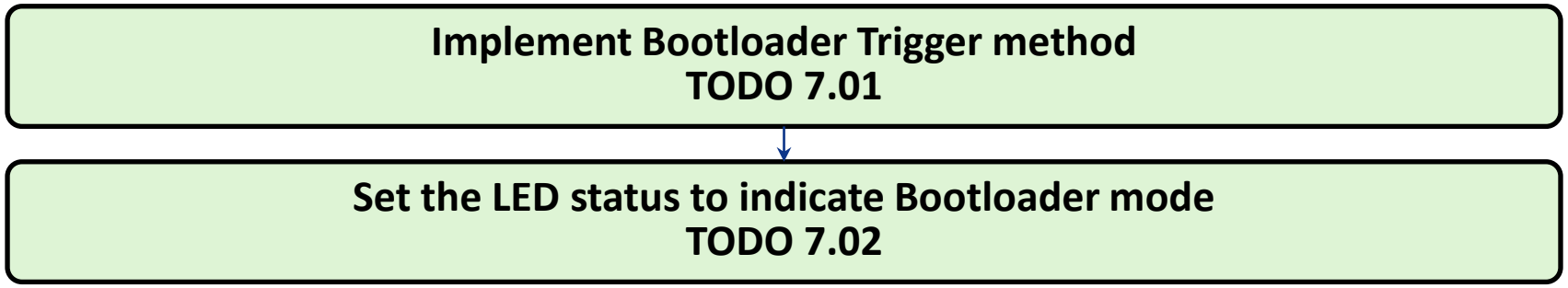
# 🛠 Lab10 USB Device HID Bootloader Initialize

- **Confirm Preprocessor Definition**



# 🛠 Lab10 USB Device HID Bootloader Initialize

- **Code Implementation state**



# 🛠 Lab10 USB Device HID Bootloader Initialize

- **Implement below function at main.c**

```
// TODO 10.01
#define BTL_TRIGGER_PATTERN (0x5048434DUL)
static uint32_t *ramStart = (uint32_t *)BTL_TRIGGER_RAM_START;

bool bootloader_Trigger(void)
{
    uint32_t i;

    for (i = 0; i < 2000; i++)
        asm("nop");

    if (BT1_Get() == 0)
        return true;

    if (BTL_TRIGGER_PATTERN == ramStart[0] && BTL_TRIGGER_PATTERN == ramStart[1] &&
        BTL_TRIGGER_PATTERN == ramStart[2] && BTL_TRIGGER_PATTERN == ramStart[3])
    {
        ramStart[0] = 0;
        ramStart[1] = 0;
        ramStart[2] = 0;
        ramStart[3] = 0;
        return true;
    }

    return false;
}
```

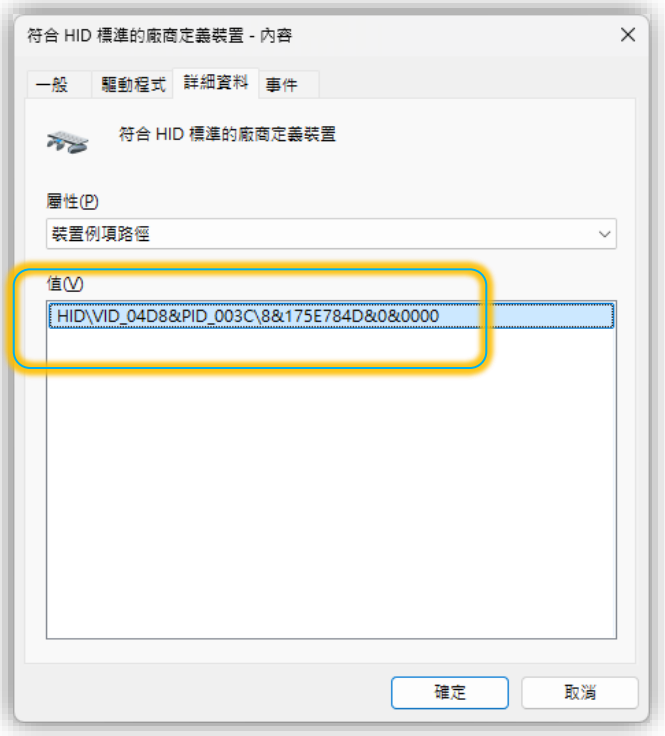
# 🛠 Lab10 USB Device HID Bootloader Initialize

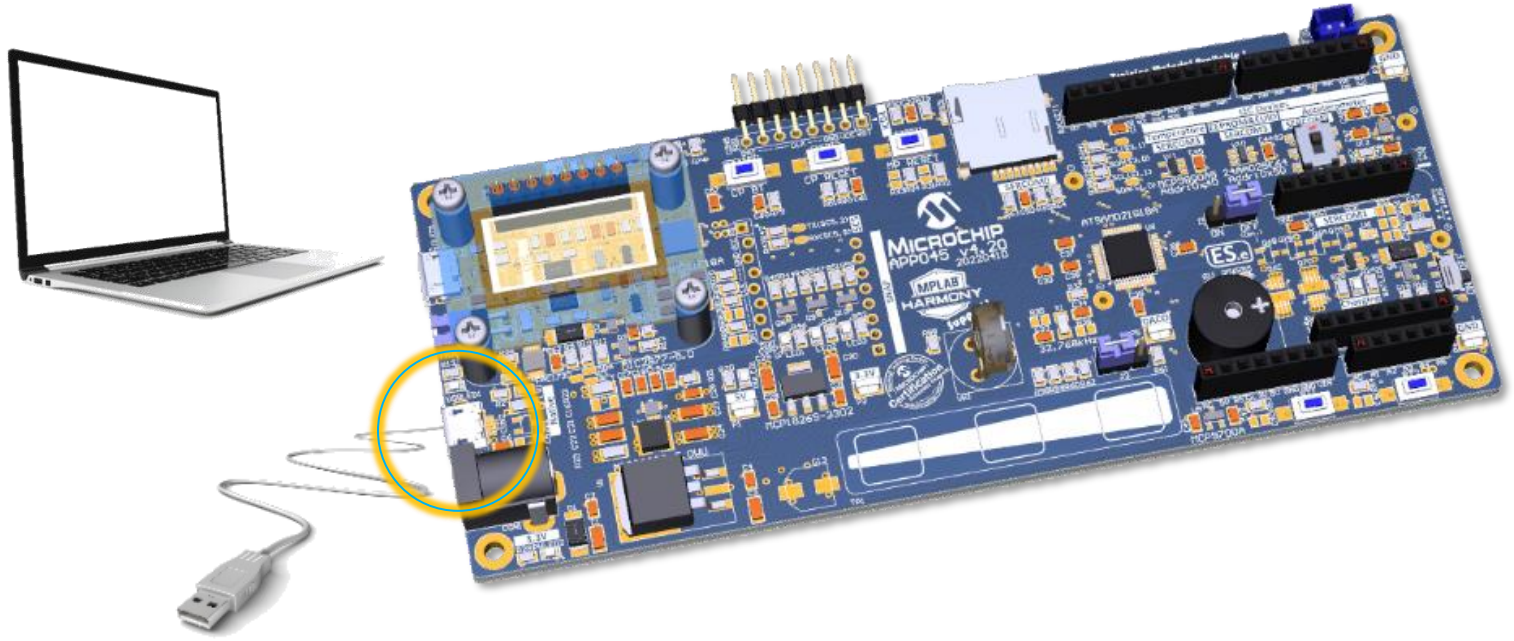
- **Implement below function at main.c**

```
int main ( void )
{
    /* Initialize all modules */
    SYS_Initialize ( NULL );

    // TODO 10.02
    LED1_Set();
    LED2_Set();
    LED3_Set();

    while ( true )
    {
        /* Maintain state machines of all polled MPLAB Harmony modules. */
        SYS_Tasks ( );

    }

    /* Execution should not come here during normal operation */

    return ( EXIT_FAILURE );
}
```

# 🛠 Lab10 USB Device HID Bootloader Initialize

- **Re-connect USB cable to CON2 after programming USB Device HID Bootloader.**
- **Check HID Bootloader is work, you will see the device at device manager.**

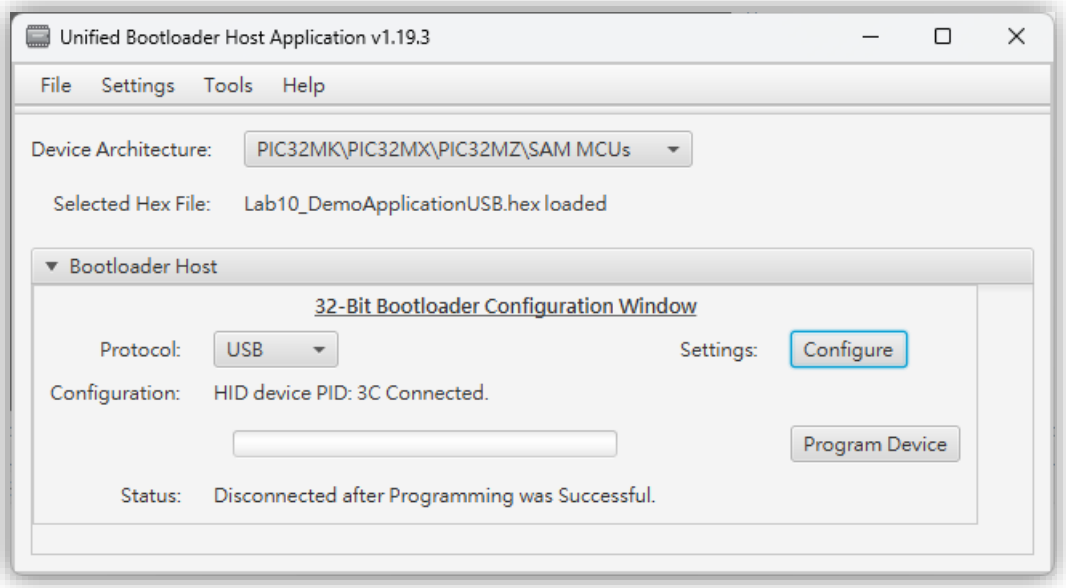


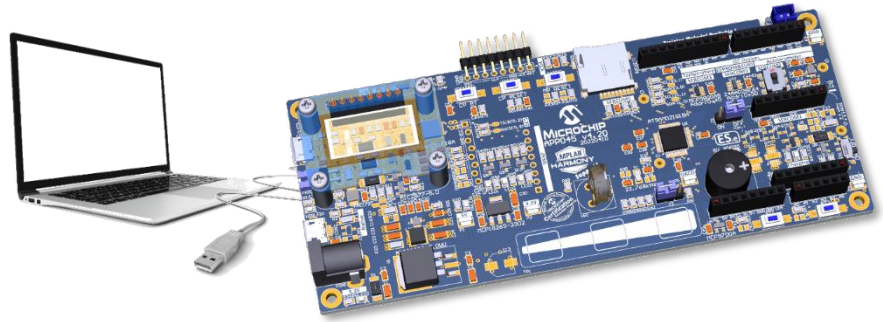




MICROCHIP

# 🛠 Lab10 USB Device HID Bootloader Initialize

- **Enter Bootloader mode automatically, because application invalid, after programming firmware.**
- **LED1 to 3 keep light to indicate Bootloader mode currently.**
- **Run *UBHA Host* and set protocol is USB, Setting PID is 0x3C, perform a firmware update after load application firmware (*Lab10_DemoApplicationUSB.hex*) *Lab10_DemoApplicationUSB.hex* is the firmware built for use with the Bootloader, relocated to 0x6000.**



MICROCHIP

# Microchip Trademarks

**Overview**

Microchip Technology Incorporated's products are marketed and sold under one or more of the following trademarks belonging to Microchip or one of Microchip's subsidiaries. Questions regarding use of these trademarks should be addressed to the Microchip's Legal Department via email: legal.department@microchip.com.

**The following are registered trademarks of Microchip in the U.S.A. and other countries:**

The Microchip name and logo, the Microchip logo, Adaptec, AnyRate, AVR, AVR logo, AVR Freaks, BesTime, BitCloud, chipKIT, chipKIT logo, CryptoMemory, CryptoRF, dsPIC, FlashFlex, flexPWR, HELDO, IGLOO, JukeBlox, KeeLoq, Kleer, LANCheck, LinkMD, maXStylus, maXTouch, MediaLB, megaAVR, Microsemi, Microsemi logo, MOST, MOST logo, MPLAB, OptoLyzer, PackeTime, PIC, picoPower, PICSTART, PIC32 logo, PolarFire, Prochip Designer, QTouch, SAM-BA, SenGenuity, SpyNIC, SST, SST Logo, SuperFlash, Symmetricom, SyncServer, Tachyon, TimeSource, tinyAVR, UNI/O, Vectron, and XMEGA

**The following are registered trademarks of Microchip in the U.S.A:**

AgileSwitch, APT, ClockWorks, The Embedded Control Solutions Company, EtherSynch, Flashtec, Hyper Speed Control, HyperLight Load, IntelliMOS, Libero, motorBench, mTouch, Powermite 3, Precision Edge, ProASIC, ProASIC Plus, ProASIC Plus logo, Quiet-Wire, SmartFusion, SyncWorld, Temux, TimeCesium, TimeHub, TimePictra, TimeProvider, TrueTime, WinPath, and ZL

**The following are registered trademarks of Microchip in other countries:** BeaconThings, GestIC, Silicon Storage Technology

**The following are trademarks of Microchip in the U.S.A. and other countries:**

Adjacent Key Suppression, AKS, Analog-for-the-Digital Age, Any Capacitor, AnyIn, AnyOut, Augmented Switching, BlueSky, BodyCom, CodeGuard, CryptoAuthentication, CryptoAutomotive, CryptoCompanion, CryptoController, dsPICDEM, dsPICDEM.net, Dynamic Average Matching, DAM, ECAN, Espresso T1S, EtherGREEN, IdealBridge, In-Circuit Serial Programming, ICSP, INICnet, Intelligent Paralleling, Inter-Chip Connectivity, JitterBlocker, maxCrypto, maxView, memBrain, Mindi, MiWi, MPASM, MPF, MPLAB Certified logo, MPLIB, MPLINK, MultiTRAK, NetDetach, Omniscient Code Generation, PICDEM, PICDEM.net, PICkit, PICtail, PowerSmart, PureSilicon, QMatrix, REAL ICE, Ripple Blocker, RTAX, RTG4, SAM-ICE, Serial Quad I/O, simpleMAP, SimpliPHY, SmartBuffer, SMART-I.S., storClad, SQI, SuperSwitcher, SuperSwitcher II, Switchtec, SynchroPHY, Total Endurance, TSHARC, USBCheck, VariSense, VectorBlox, VeriPHY, ViewSpan, WiperLock, XpressConnect, and ZENA

**The following is a service mark of Microchip in the U.S.A.:** SQTP

**The following are logos of Microchip in the U.S.A. and other countries:** The Adaptec logo, Frequency on Demand, Silicon Storage Technology, Symmcom, and Trusted Time

**The following is a registered trademark of Microchip Technology Germany II GmbH & Co. KG, a subsidiary of Microchip Technology Inc., in other countries:** GestIC

MICROCHIP