Microchip University

Lab Manual for 16-Bit Bootloaders Using MCC

Table of Contents

MICROCHIP
UNIVERSITY

Introduction:

The purpose of this lab is to go into more detail on creating a generic 16-bit Bootloader using MCC. Interrupts and other features of the device will be used to further show the details of the 16-Bit Bootloader.

Required Development Tools:

- One of the two boards below
    a. PIC-IoT WA Development Board (EV54Y39A)
    b. PIC-IoT WG Development Board (AC164164)
    c. Explorer 16/32 Development Board (DM240001-2) with a Device/PIM that is supported by MCC
        i. If using an Explorer 16/32 board, a *Secure Click 4* with an ATECC608 is required for the two ECDSA labs
- MPLAB® X version 6.0 or later
- MPLAB Code Configurator (MCC) version 5.1.1 or later
- 16-Bit Bootloader MCC module version 1.19.1 or later
    a. Version 1.19.0 will not work for this lab
- MPLAB XC16 version 1.60 or later
- Universal Bootloader Host Application found at www.microchip.com/16-bit-bootloader

Upon completion, you will:

- Use the MPLAB Code Configurator (MCC) to generate a bootloader project.
- Use the MPLAB Code Configurator (MCC) to generate an application project that can be loaded by the bootloader.
- Use the bootloader to verify the integrity and authenticity of an application image using the Elliptical Curve Digital Signature Algorithm (ECDSA).

Prerequisites:

The lab material assumes you have prior experience with:
- MPLAB X IDE
- MPLAB based Programming/Debugging fundamentals
- C language programming
- MPLAB Code Configurator (MCC) basic usage

Lab 1 – Creating a Bootloader Project

Purpose:

Create a bootloader project that can load an example application that will be created in lab 2.

Overview:

This lab will use the PIC-IoT WA Development Board, part number EV54Y39A. The lab will also work on the PIC-IoT WG Development Board, part number AC164164.

This lab will create a bootloader project that will be used to load an application that we will build in lab 2. To differentiate when the bootloader is running and when the application is running, the bootloader will toggle the Yellow Data LED connected to pin RC3 while running. This way we can tell when we are in the bootloader.

Code will be added so that a holding push button SW0 on reset will enter bootloader mode.

Procedure:

STEP 1: Create an MPLABX Project for the PIC24FJ128GA705

Step 1a: Open MPLABX

Step 1b: Create a new project by selecting “File – New Project”

Step 1c: In the new project window, select “Standalone Project” in the “Microchip Embedded” folder and press “Next”.

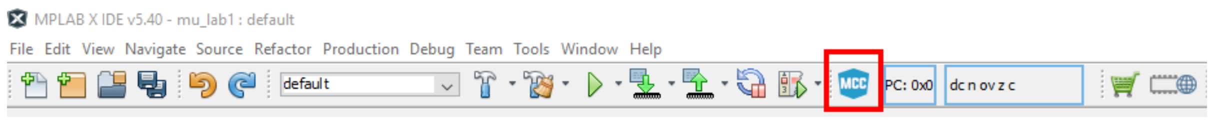
Step 1d: Select the PIC24FJ128GA705 device in the device menu and click “Next”

Step 1e: Select the compiler you want to use and click “Next”

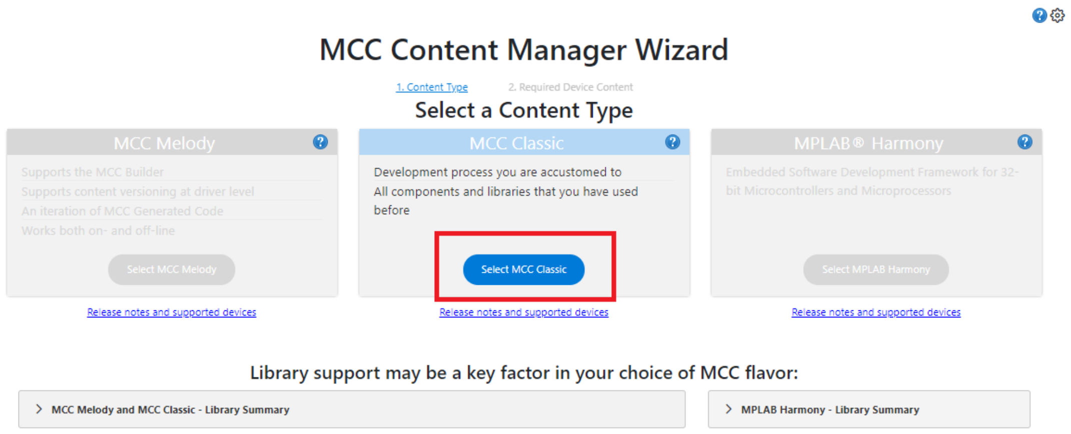
Step 1f: Type in “boot” for the project name, select a folder for the project, and click “Finish”

STEP 2: Configure MCC for the Board

Step 2a: Open MCC by clicking the MCC button in MPLAB X


MPLAB X IDE v5.40 - mu_lab1 : default
File Edit View Navigate Source Refactor Production Debug Team Tools Window Help
default
MCC
PC: 0x0
dc n ov z c

Step 2b: If prompted with the MCC Content Manager Wizard, Select the “MCC Classic”


MCC Content Manager Wizard

1. Content Type 2. Required Device Content

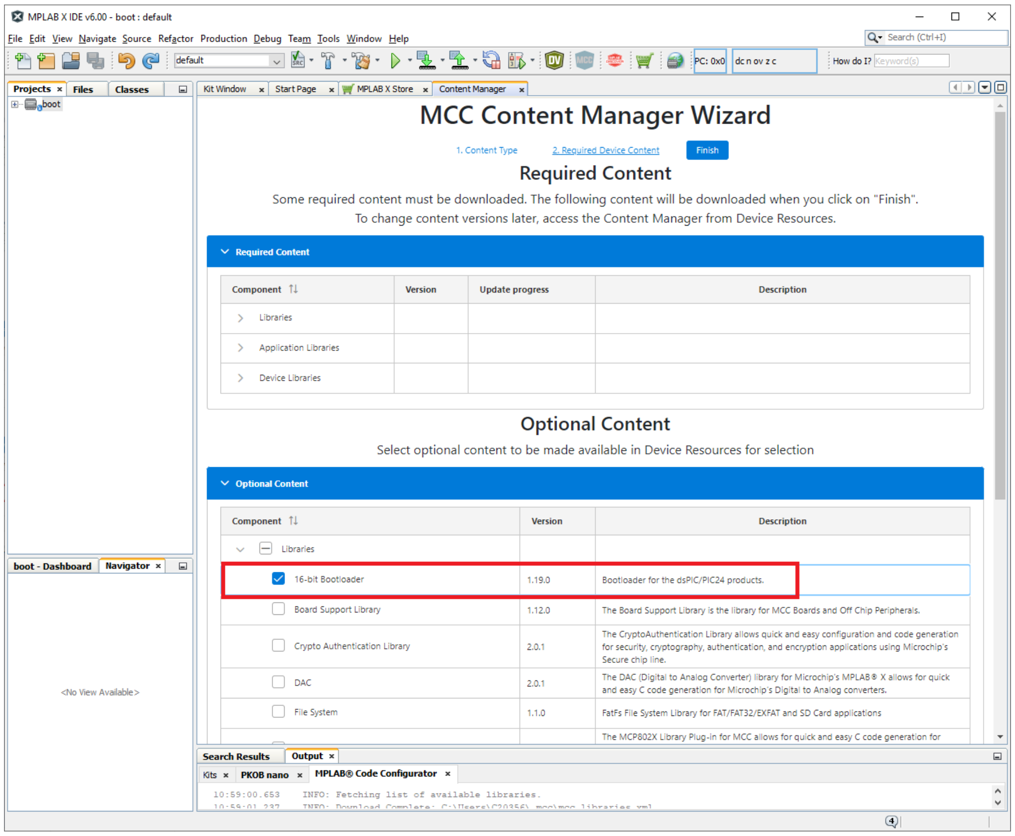
Select a Content Type

MCC Melody
Supports the MCC Builder
Supports content versioning at driver level
An iteration of MCC Generated Code
Works both on- and off-line
Select MCC Melody
Release notes and supported devices
MCC Classic
Development process you are accustomed to
All components and libraries that you have used before
Select MCC Classic
Release notes and supported devices
MPLAB® Harmony
Embedded Software Development Framework for 32-bit Microcontrollers and Microprocessors
Select MPLAB Harmony
Release notes and supported devices

Library support may be a key factor in your choice of MCC flavor:

MCC Melody and MCC Classic - Library Summary
MPLAB Harmony - Library Summary

Step 2c: If this is a new installation, under optional content, Libraries, select the 16-bit Bootloader and press Finish.


MPLAB X IDE v6.00 - boot : default
File Edit View Navigate Source Refactor Production Debug Team Tools Window Help
Projects Files Classes
boot
boot - Dashboard Navigator
<No View Available>
Kit Window Start Page MPLAB X Store Content Manager
1. Content Type 2. Required Device Content Finish

MCC Content Manager Wizard

Required Content

Some required content must be downloaded. The following content will be downloaded when you click on "Finish".
To change content versions later, access the Content Manager from Device Resources.

Required Content

Component	Version	Update progress	Description
Libraries			
Application Libraries			
Device Libraries			

Optional Content

Select optional content to be made available in Device Resources for selection

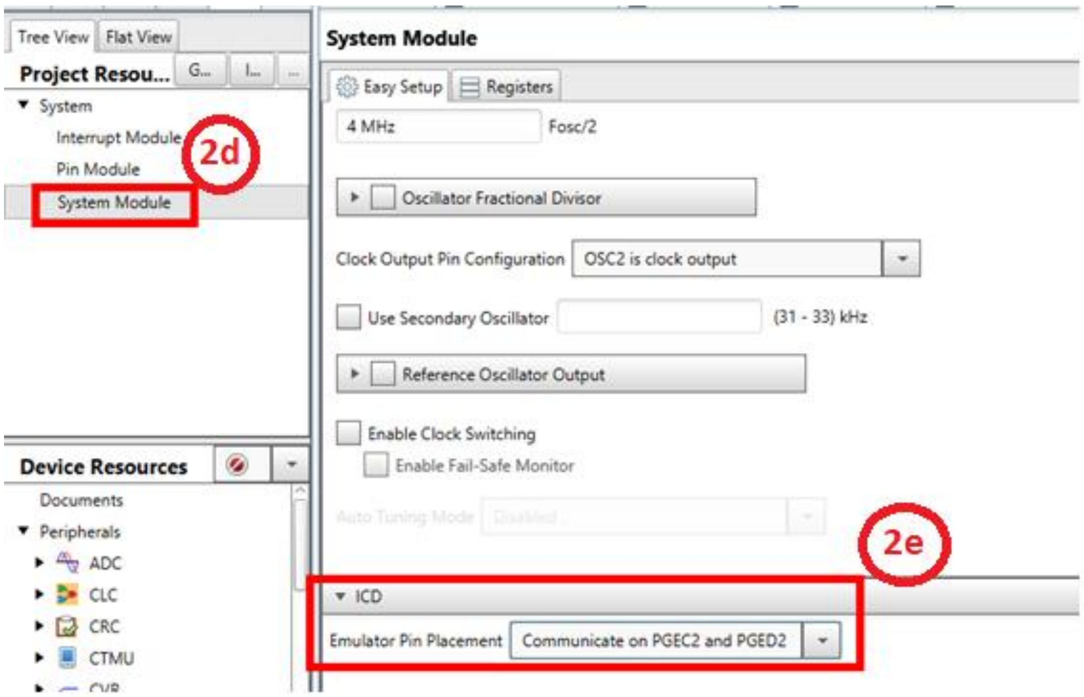
Optional Content

Component	Version	Description
Libraries		
✓ 16-bit Bootloader	1.19.0	Bootloader for the dsPIC/PIC24 products.
Board Support Library	1.12.0	The Board Support Library is the library for MCC Boards and Off Chip Peripherals.
Crypto Authentication Library	2.0.1	The CryptoAuthentication Library allows quick and easy configuration and code generation for security, cryptography, authentication, and encryption applications using Microchip's Secure chip line.
DAC	2.0.1	The DAC (Digital to Analog Converter) library for Microchip's MPLAB® X allows for quick and easy C code generation for Microchip's Digital to Analog converters.
File System	1.1.0	FatFs File System Library for FAT/FAT32/EXFAT and SD Card applications
		The MCP802X Library Plug-in for MCC allows for quick and easy C code generation for

```
10:59:00.653    INFO: Fetching list of available libraries.
```

Step 2d: Select the “System Module” in the Project Resourced pane

Step 2e: Configure the emulator pins to PGEC2 and PGED2



4 MHz Fosc/2

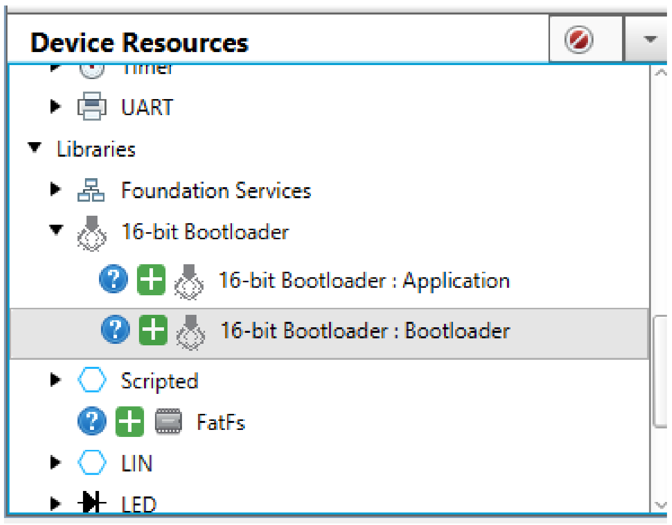
Clock Output Pin Configuration: OSC2 is clock output

Use Secondary Oscillator (31 - 33) kHz

Emulator Pin Placement: Communicate on PGEC2 and PGED2

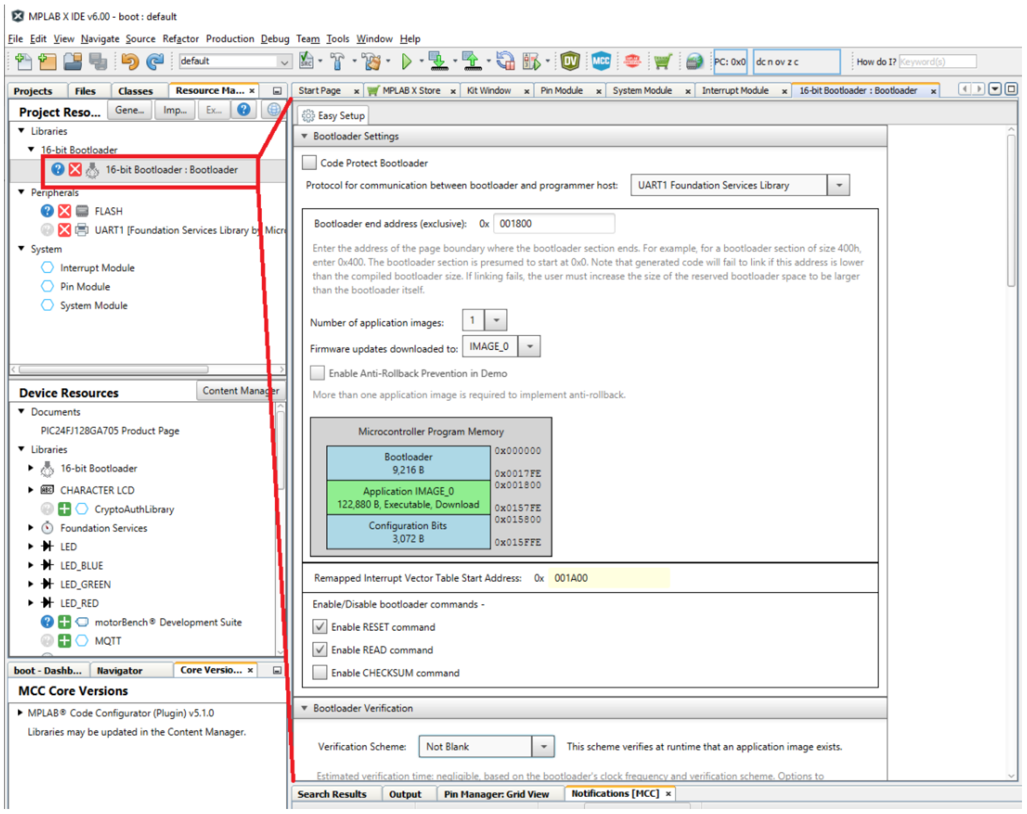
STEP 3: Add the bootloader module to the project

Step 3a: Add the “16-bit Bootloader : Bootloader” module to the project by clicking the “+” sign next to the module in the “Device Resources” pane.



STEP 4: Configure the bootloader module

Step 4a: Open the bootloader module pane by clicking on the “16-bit Bootloader : Bootloader” node in the “Project Resources” pane.



Easy Setup

Bootloader Settings

☐ Code Protect Bootloader

Protocol for communication between bootloader and programmer host: UART1 Foundation Services Library

Bootloader end address (exclusive): 0x 001800

Enter the address of the page boundary where the bootloader section ends. For example, for a bootloader section of size 400h, enter 0x400. The bootloader section is presumed to start at 0x0. Note that generated code will fail to link if this address is lower than the compiled bootloader size. If linking fails, the user must increase the size of the reserved bootloader space to be larger than the bootloader itself.

Number of application images: 1

Firmware updates downloaded to: IMAGE_0

☐ Enable Anti-Rollback Prevention in Demo

More than one application image is required to implement anti-rollback.

Microcontroller Program Memory

Bootloader 9,216 B	0x000000 – 0x0017FE
Application IMAGE_0 122,880 B, Executable, Download	0x001800 – 0x0157FE
Configuration Bits 3,072 B	0x015800 – 0x015FFE

Remapped Interrupt Vector Table Start Address: 0x 001A00

Enable/Disable bootloader commands -

☑ Enable RESET command

☑ Enable READ command

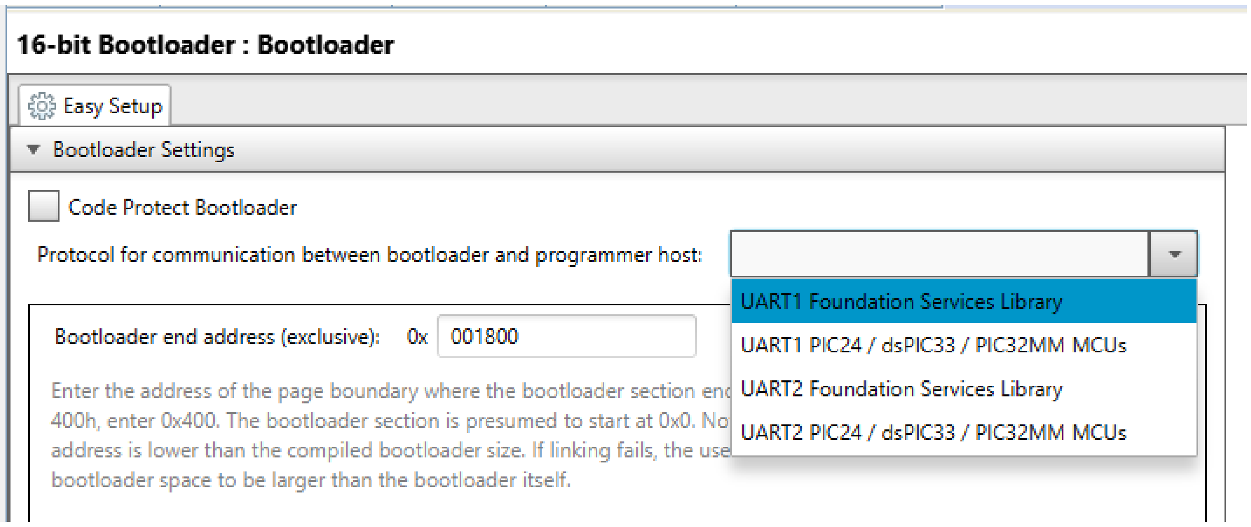
☐ Enable CHECKSUM command

Bootloader Verification

Verification Scheme: Not Blank — This scheme verifies at runtime that an application image exists.

Estimated verification time: negligible, based on the bootloader's clock frequency and verification scheme. Options to

Step 4b: In the Protocol selection drop-down, select the “UART1 Foundation Services Library” option.



16-bit Bootloader : Bootloader

Easy Setup

Bootloader Settings

☐ Code Protect Bootloader

Protocol for communication between bootloader and programmer host:

- UART1 Foundation Services Library
- UART1 PIC24 / dsPIC33 / PIC32MM MCUs
- UART2 Foundation Services Library
- UART2 PIC24 / dsPIC33 / PIC32MM MCUs

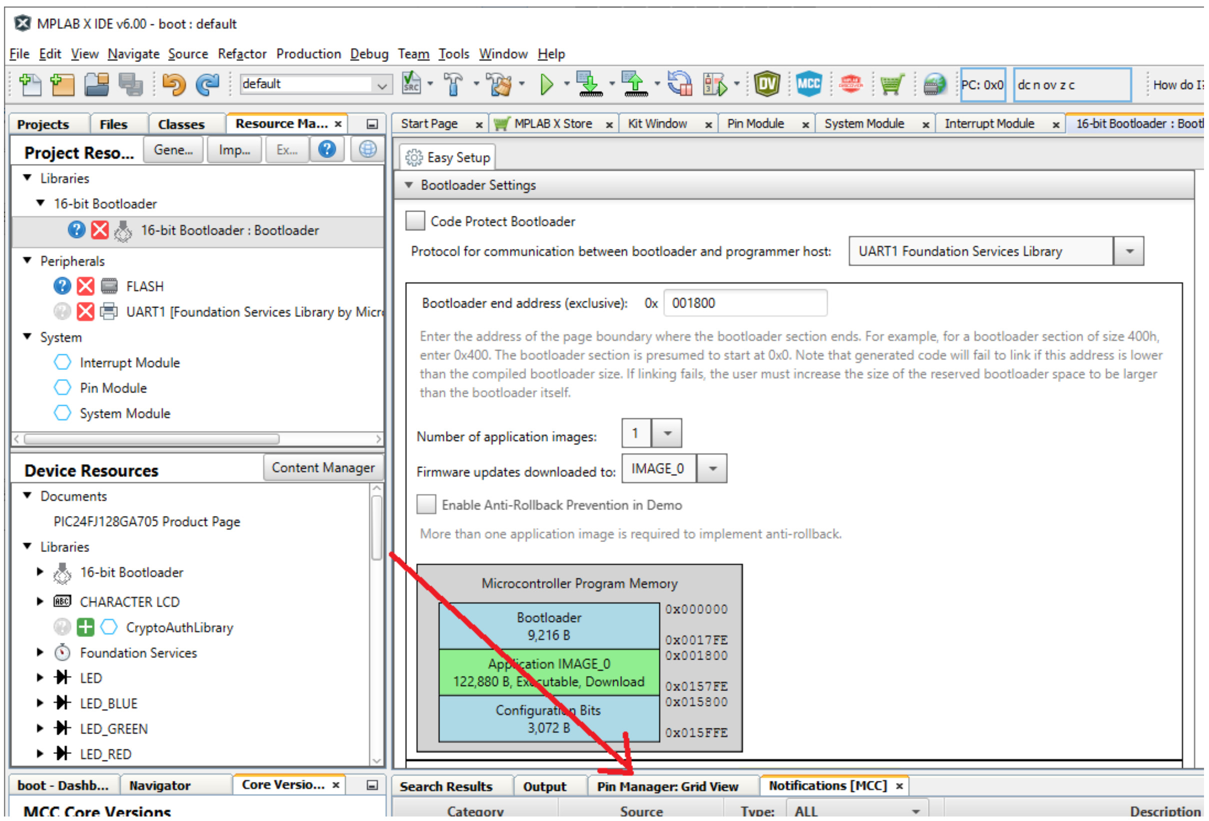
Bootloader end address (exclusive): 0x 001800

Enter the address of the page boundary where the bootloader section end 400h, enter 0x400. The bootloader section is presumed to start at 0x0. No address is lower than the compiled bootloader size. If linking fails, the use bootloader space to be larger than the bootloader itself.

STEP 5: Configure the UART

Step 5a: Verify the UART module settings. The UART module should have been automatically added to the "Project Resources" pane. Select the UART module in this pane. It should be configured to 9600:8:N:1 by default. Leave these settings. If the settings aren't for 9600:8:N:1, then change them.

Step 5b: Open the pin manager grid view



Code Protect Bootloader

Protocol for communication between bootloader and programmer host: UART1 Foundation Services Library

Bootloader end address (exclusive): 0x 001800

Enter the address of the page boundary where the bootloader section ends. For example, for a bootloader section of size 400h, enter 0x400. The bootloader section is presumed to start at 0x0. Note that generated code will fail to link if this address is lower than the compiled bootloader size. If linking fails, the user must increase the size of the reserved bootloader space to be larger than the bootloader itself.

Number of application images: 1

Firmware updates downloaded to: IMAGE_0

Enable Anti-Rollback Prevention in Demo

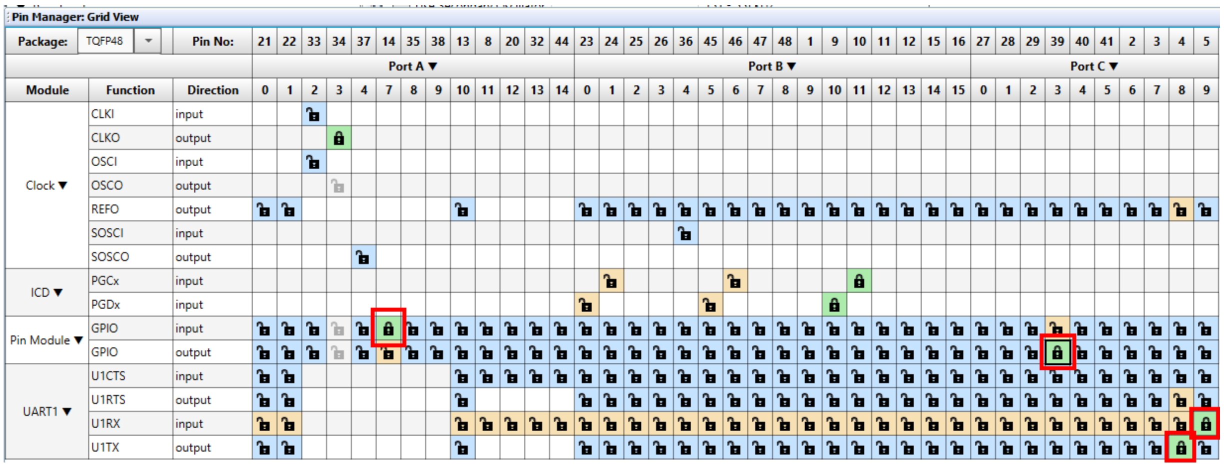
More than one application image is required to implement anti-rollback.

Microcontroller Program Memory

Bootloader 9,216 B	0x000000 0x0017FE
Application IMAGE_0 122,880 B, Executable, Download	0x001800 0x0157FE
Configuration Bits 3,072 B	0x015800 0x015FFE

Step 5c: Configure the following pins:

a. RC3 GPIO as an output. This is connected to the yellow “Data LED”.
b. RA7 as an input. This is the SW0 push button pin.
c. UART TX pin is RC8 on this board.
d. UART RX pin is RC9 on this board.

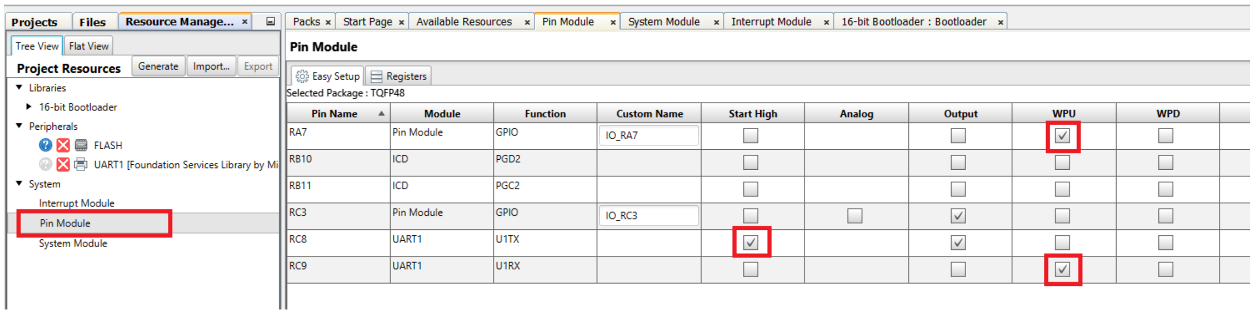


Pin Manager: Grid View

Package:	TQFP48	Pin No:	21	22	33	34	37	14	35	38	13	8	20	32	44	23	24	25	26	36	45	46	47	48	1	9	10	11	12	15	16	27	28	29	39	40	41	2	3	4	5
			Port A ▼													Port B ▼																Port C ▼									
Module	Function	Direction	0	1	2	3	4	7	8	9	10	11	12	13	14	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	0	1	2	3	4	5	6	7	8	9
Clock ▼	CLKI	input																																							
	CLKO	output				🔒																																			
	OSCI	input																																							
	OSCO	output																																							
	REFO	output																																							
	SOSCI	input																																							
	SOSCO	output																																							
ICD ▼	PGCx	input																										🔒													
	PGDx	input																									🔒														
Pin Module ▼	GPIO	input						🔒																																	
	GPIO	output																																	🔒						
UART1 ▼	U1CTS	input																																							
	U1RTS	output																																							
	U1RX	input																																							🔒
	U1TX	output																																						🔒	

Step 5d: Open the pin module and make the following configuration changes:

a. Enable the weak pull-up on RA7 (the push button pin)
b. Enable the weak pull-up on RC9 (the U1RX line)
c. Enable Start High on RC8 (the U1TX line)



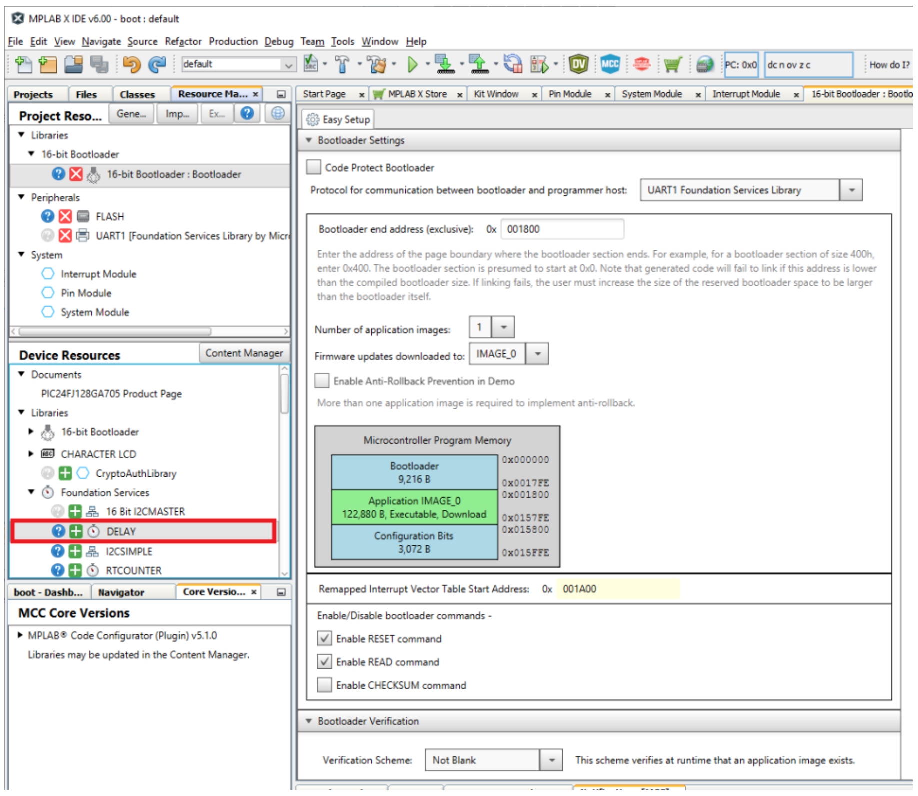
Pin Module

Selected Package : TQFP48

Pin Name ▲	Module	Function	Custom Name	Start High	Analog	Output	WPU	WPD
RA7	Pin Module	GPIO	IO_RA7	☐		☐	✓	☐
RB10	ICD	PGD2		☐		☐	☐	☐
RB11	ICD	PGC2		☐		☐	☐	☐
RC3	Pin Module	GPIO	IO_RC3	☐	☐	✓	☐	☐
RC8	UART1	U1TX		✓		✓	☐	☐
RC9	UART1	U1RX		☐		☐	✓	☐

STEP 6: Add delay module to project

Step 6a: This project will use a delay function to generate the LED blink delay. Add the Foundation Services -> DELAY library to the project by double clicking on it in the “Device Resources” pane under the “Libraries” folder shown below:


MPLAB X IDE v6.00 - boot : default
File Edit View Navigate Source Refactor Production Debug Team Tools Window Help
Projects Files Classes Resource Ma...
Start Page MPLAB X Store Kit Window Pin Module System Module Interrupt Module 16-bit Bootloader : Bootlo
Project Reso...
Libraries
16-bit Bootloader
16-bit Bootloader : Bootloader
Peripherals
FLASH
UART1 [Foundation Services Library by Micr
System
Interrupt Module
Pin Module
System Module
Device Resources
Content Manager
Documents
PIC24FJ128GA705 Product Page
Libraries
16-bit Bootloader
CHARACTER LCD
CryptoAuthLibrary
Foundation Services
16 Bit I2CMASTER
DELAY
I2CSIMPLE
RTCOUNTER
MCC Core Versions
MPLAB® Code Configurator (Plugin) v5.1.0
Libraries may be updated in the Content Manager.
Easy Setup
Bootloader Settings
Microcontroller Program Memory
Bootloader 9,216 B
Application IMAGE_0 122,880 B, Executable, Download
Configuration Bits 3,072 B
0x000000
0x0017FE
0x001800
0x0157FE
0x015800
0x015FFE
Bootloader Verification

Code Protect Bootloader

Protocol for communication between bootloader and programmer host: UART1 Foundation Services Library

Bootloader end address (exclusive): 0x 001800

Enter the address of the page boundary where the bootloader section ends. For example, for a bootloader section of size 400h, enter 0x400. The bootloader section is presumed to start at 0x0. Note that generated code will fail to link if this address is lower than the compiled bootloader size. If linking fails, the user must increase the size of the reserved bootloader space to be larger than the bootloader itself.

Number of application images: 1

Firmware updates downloaded to: IMAGE_0

Enable Anti-Rollback Prevention in Demo

More than one application image is required to implement anti-rollback.

Remapped Interrupt Vector Table Start Address: 0x 001A00

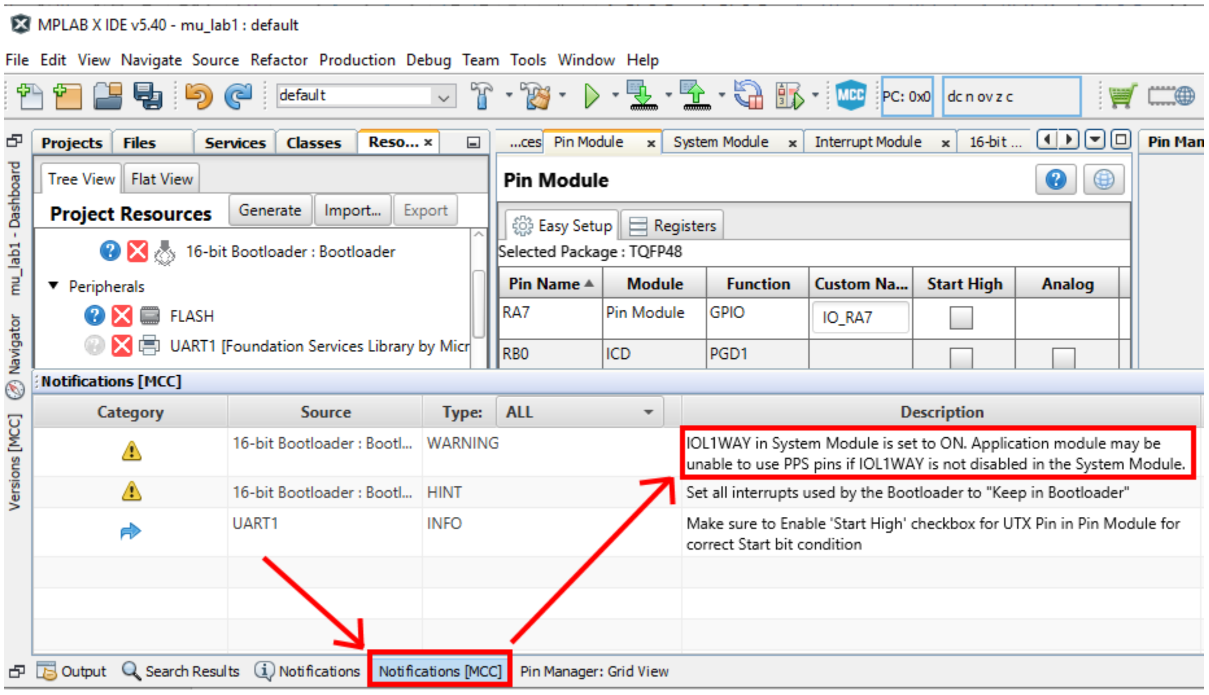
Enable/Disable bootloader commands -

- [x] Enable RESET command
- [x] Enable READ command
- [ ] Enable CHECKSUM command

Verification Scheme: Not Blank This scheme verifies at runtime that an application image exists.

STEP 7: Resolve project warnings

Note the warning for “IOL1WAY in System Module is set”. This message is saying that once the boot loader configures the PPS pins, the application will no longer be able to modify the PPS settings. For some customers, this will be okay. The boot loader may configure all of the pin configuration for the system. For other customers, they may want the application code to be able to configure the PPS settings. If the user does want to be able re-configure the PPS pins in the application, then the IOL1WAY bit needs to be changed to “Allow multiple reconfigurations”.


MPLAB X IDE v5.40 - mu_lab1 : default
File Edit View Navigate Source Refactor Production Debug Team Tools Window Help
Projects Files Services Classes Reso...
Tree View Flat View
Project Resources Generate Import... Export
16-bit Bootloader : Bootloader
Peripherals
FLASH
UART1 [Foundation Services Library by Micr
Pin Module System Module Interrupt Module 16-bit ...
Pin Module
Easy Setup Registers
Selected Package : TQFP48
Notifications [MCC]
Output Search Results Notifications Notifications [MCC] Pin Manager: Grid View

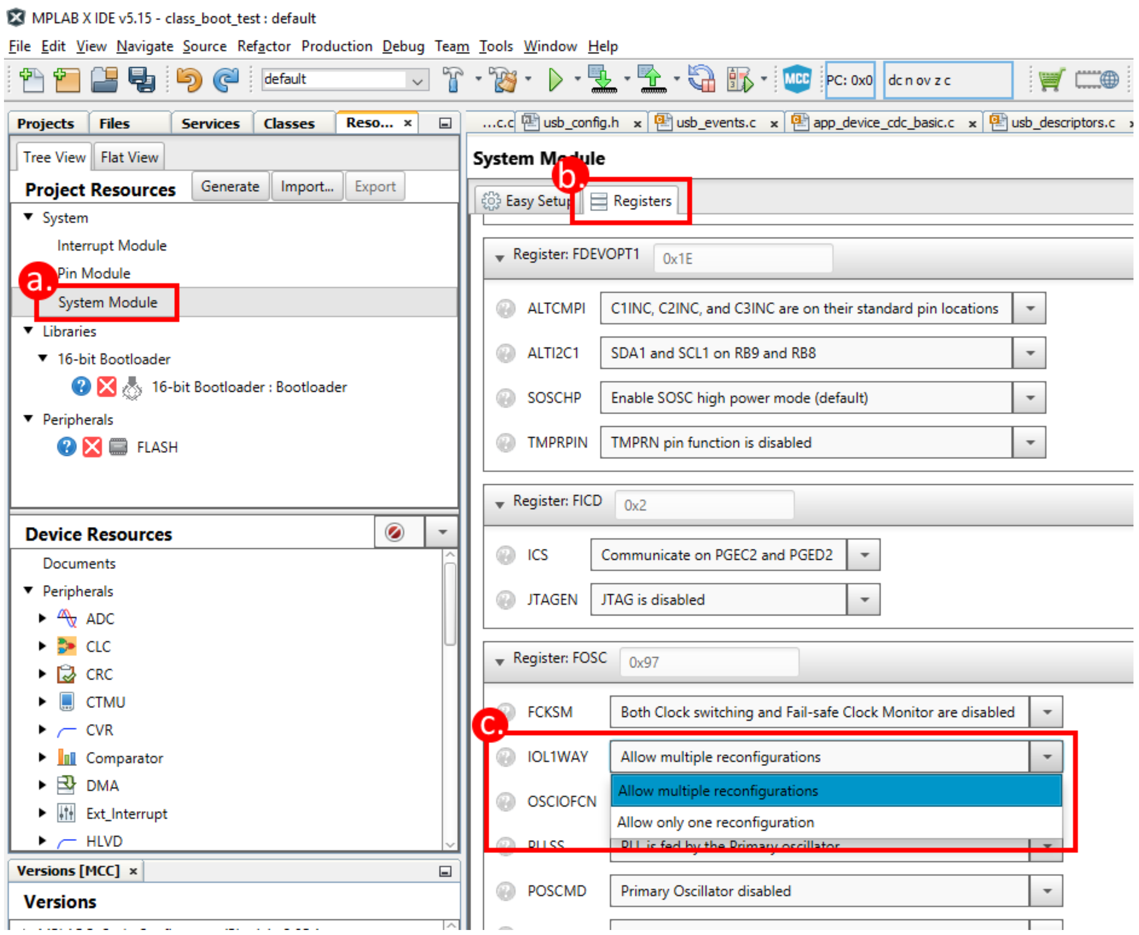
Pin Name	Module	Function	Custom Na...	Start High	Analog
RA7	Pin Module	GPIO	IO_RA7	☐	
RB0	ICD	PGD1		☐	☐

Category	Source	Type: ALL	Description
⚠	16-bit Bootloader : Bootl...	WARNING	IOL1WAY in System Module is set to ON. Application module may be unable to use PPS pins if IOL1WAY is not disabled in the System Module.
⚠	16-bit Bootloader : Bootl...	HINT	Set all interrupts used by the Bootloader to "Keep in Bootloader"
➡	UART1	INFO	Make sure to Enable 'Start High' checkbox for UTX Pin in Pin Module for correct Start bit condition

Step 7a: In the “Project Resources” pane, click on the “System Module” node.

Step 7b: Select the Registers view tab.

Step 7c: Change the IOL1WAY bit to “Allow multiple reconfigurations”

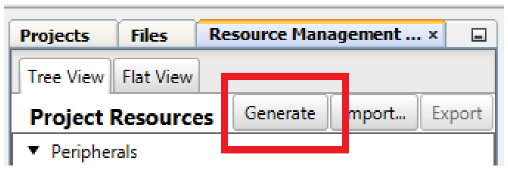


Register: FDEVOPT1	0x1E
ALTCMPI	C1INC, C2INC, and C3INC are on their standard pin locations
ALTI2C1	SDA1 and SCL1 on RB9 and RB8
SOSCHP	Enable SOSC high power mode (default)
TMPRPIN	TMPRN pin function is disabled
Register: FICD	0x2
ICS	Communicate on PGEC2 and PGED2
JTAGEN	JTAG is disabled
Register: FOSC	0x97
FCKSM	Both Clock switching and Fail-safe Clock Monitor are disabled
IOL1WAY	Allow multiple reconfigurations
OSCIOFCN	
PLLSS	PLL is fed by the Primary oscillator
POSCMD	Primary Oscillator disabled

Dropdown options: Allow multiple reconfigurations; Allow only one reconfiguration

STEP 8: Generate the project

Step 8a: At this point the bootloader is completely configured in MCC. Click the “Generate” button to generate the code.

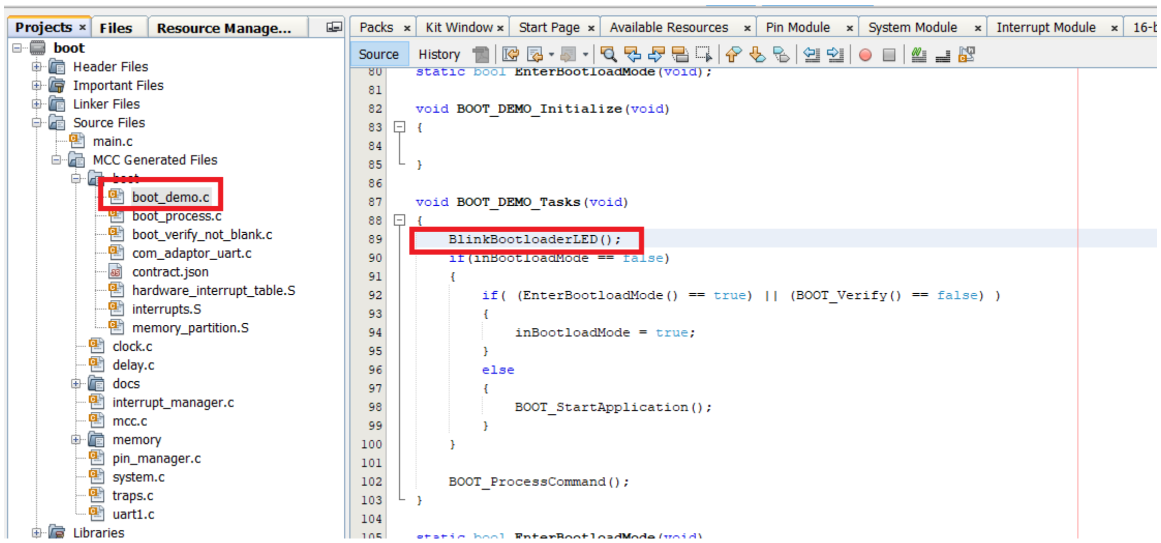


STEP 9: Add demo code to the project use the LED and pushbuttons in bootloader mode

Step 9a: This boot loader will blink the yellow LED to be able to differentiate when we are in bootloader mode verses in the end application mode. Add the code below required to toggle the yellow LED in the boot loader to the file /Source Files/MCC Generated Files/boot/boot_demo.c

```
#include "../pin_manager.h"
#include "../delay.h"

void BlinkBootloaderLED()
{
#define RATE 250
    static unsigned  count = 0;

    count++;
    DELAY_milliseconds(1);
    if(count == RATE)
    {
        IO_RC3_Toggle();
        count = 0;
    }
}
```

Step 9b: Add a call to the new `BlinkBootloaderLED()` function into the top of the `BOOT_DEMO_Tasks()` function in `boot_demo.c` so it gets called. If you don't call the function, the LED won't blink.



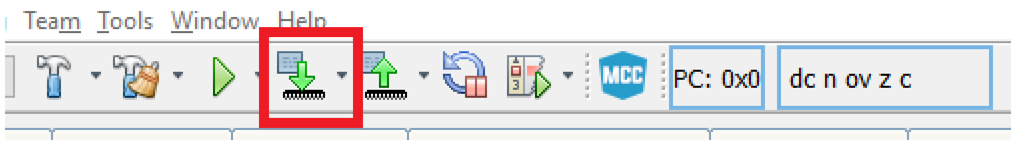
```
static bool EnterBootloadMode(void);

void BOOT_DEMO_Initialize(void)
{


}

void BOOT_DEMO_Tasks(void)
{
    BlinkBootloaderLED();
    if(inBootloadMode == false)
    {
        if( (EnterBootloadMode() == true) || (BOOT_Verify() == false) )
        {
            inBootloadMode = true;
        }
        else
        {
            BOOT_StartApplication();
        }
    }

    BOOT_ProcessCommand();
}

static bool EnterBootloadMode(void)
```

Step 9c: The default demo code doesn't know when the user wants to enter bootload mode verses running the application code. Add code to the `EnterBootLoadMode()` function in `boot_demo.c` to check RA7 pin.

```
static bool EnterBootLoadMode(void)
{
    return (IO_RA7_GetValue()==0);
}
```

STEP 10: Run the demo

Step 10a: Build and program the code by pressing thebuild and download button below. If it's running correctly, the yellow LED should be blinking.



Results:

If the bootloader demo was created correctly, the yellow LED should be linking twice per second. We will verify the UART connectivity in later labs when we have an application to try to load.

Summary:

We now have a bootloader project that should be able to load new code. The next lab we will create an example application that we can load using the bootloader.

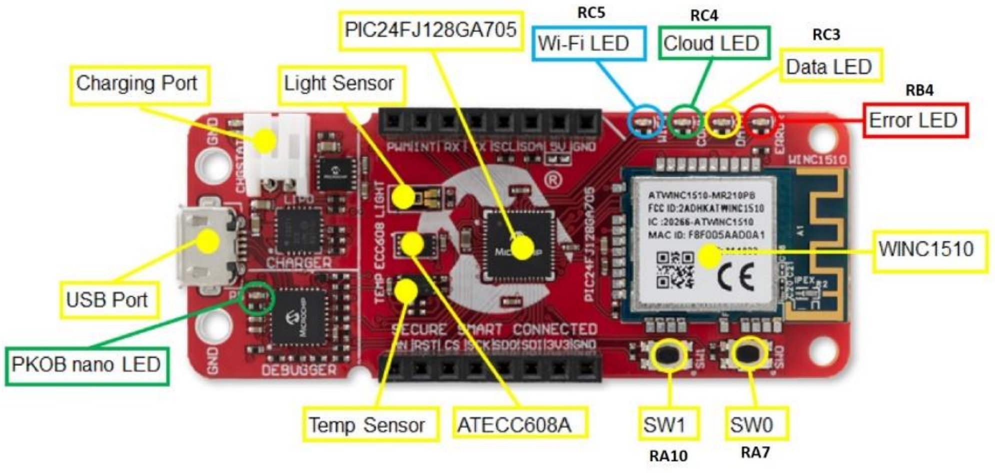
Lab 2 – Create an Application

Purpose:

Create a simple example application that can serve as an example "end product" and then enable it to be used with the bootloader generated in lab 1.

Overview:

For the simple example application, we will just toggle the blue "Wi-Fi" LED on the PIC-IoT board seen below using a timer interrupt. Then when this is working, we will use MCC to read the bootloader configuration from lab 1 and update this the application project to match the bootloader.



Procedure:

STEP 1: Create an MPLABX Project for the PIC24FJ128GA705

Step 1a: Open MPLABX

Step 1b: Create a new project by selecting "File – New Project"

Step 1c: In the new project window, select "Standalone Project" in the "Microchip Embedded" folder and press "Next".

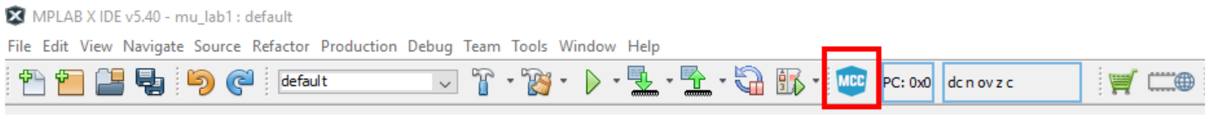
Step 1d: Select the PIC24FJ128GA705 device in the device menu and click "Next"

Step 1e: Select the compiler you want to use and click "Next"

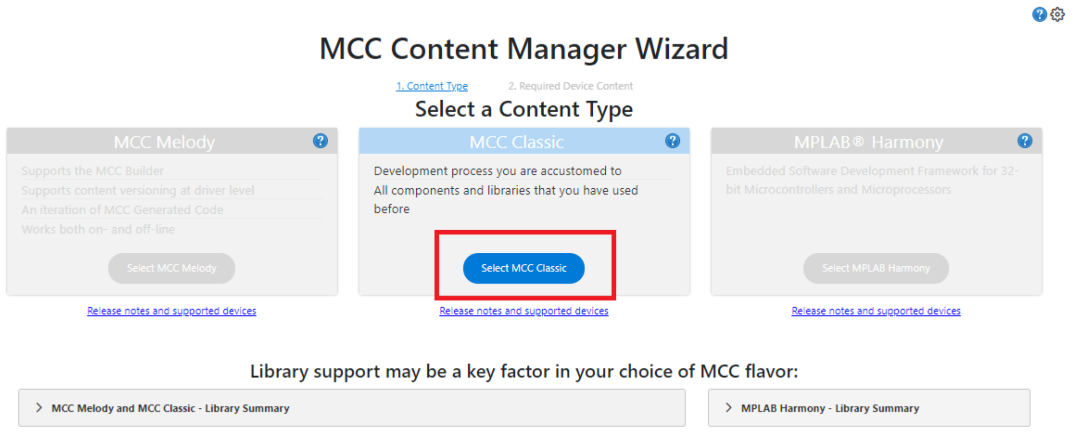
Step 1f: Type in "app" for the project name, select a folder for the project, and click "Finish"

STEP 2: Configure MCC for the Board

Step 2a: Open MCC by clicking the MCC button in MPLAB X


MPLAB X IDE v5.40 - mu_lab1 : default
File Edit View Navigate Source Refactor Production Debug Team Tools Window Help
default
MCC
PC: 0x0
dc n ov z c

Step 2b: If the MCC Content Manage Wizard comes up, select the MCC Classic option.



MCC Content Manager Wizard

1. Content Type 2. Required Device Content

Select a Content Type

MCC Melody
- Supports the MCC Builder
- Supports content versioning at driver level
- An iteration of MCC Generated Code
- Works both on- and off-line

Select MCC Melody

Release notes and supported devices

MCC Classic
- Development process you are accustomed to
- All components and libraries that you have used before

Select MCC Classic

Release notes and supported devices

MPLAB® Harmony
- Embedded Software Development Framework for 32-bit Microcontrollers and Microprocessors

Select MPLAB Harmony

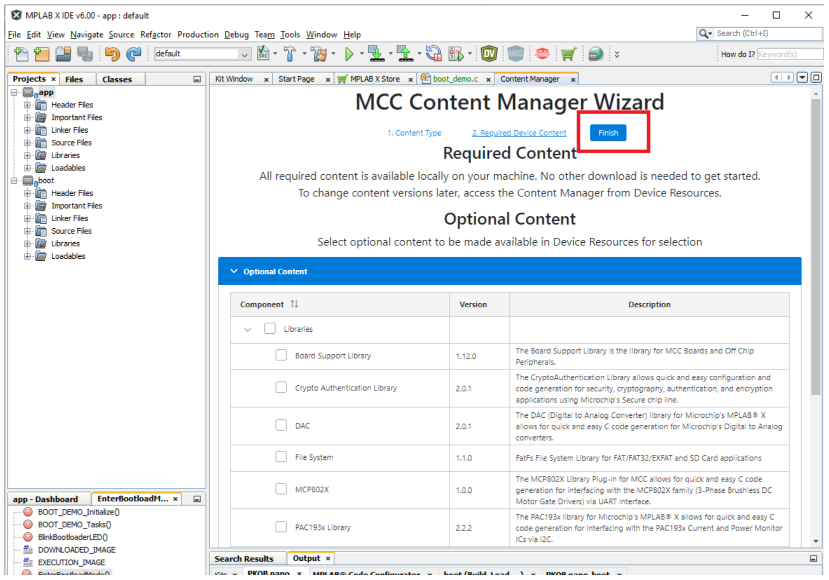
Release notes and supported devices

Library support may be a key factor in your choice of MCC flavor:

> MCC Melody and MCC Classic - Library Summary

> MPLAB Harmony - Library Summary

Step 2c: If the MCC Content Manage Wizard did come up, press Finish.



MCC Content Manager Wizard

1. Content Type 2. Required Device Content Finish

Required Content

All required content is available locally on your machine. No other download is needed to get started.
To change content versions later, access the Content Manager from Device Resources.

Optional Content

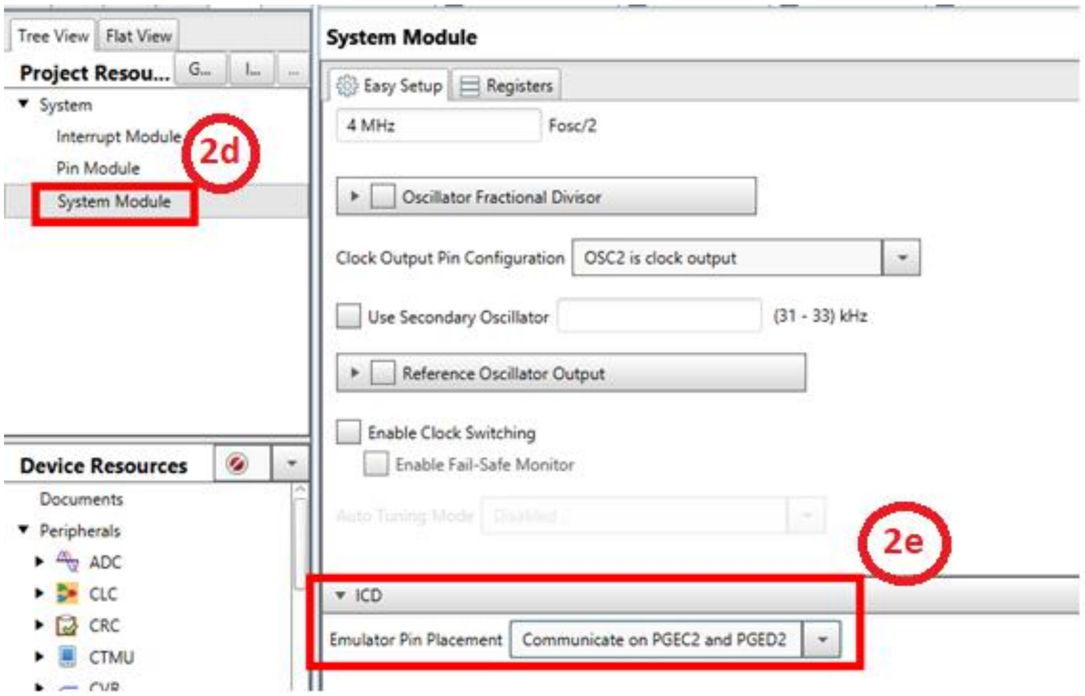
Select optional content to be made available in Device Resources for selection

Optional Content

Component	Version	Description
☐ Libraries		
☐ Board Support Library	1.12.0	The Board Support Library is the library for MCC Boards and Off Chip Peripherals.
☐ Crypto Authentication Library	2.0.1	The CryptoAuthentication Library allows quick and easy configuration and code generation for security, cryptography, authentication, and encryption applications using Microchip's Secure chip line.
☐ DAC	2.0.1	The DAC (Digital to Analog Converter) library for Microchip's MPLAB® X allows for quick and easy C code generation for Microchip's Digital to Analog converters.
☐ File System	1.1.0	FatFs File System Library for FAT/FAT32/EXFAT and SD Card applications
☐ MCP802X	1.0.0	The MCP802X Library Plug-in for MCC allows for quick and easy C code generation for interfacing with the MCP802X family (3-Phase Brushless DC Motor Gate Drivers) via UART interface.
☐ PAC193x Library	2.2.2	The PAC193x library for Microchip's MPLAB® X allows for quick and easy C code generation for interfacing with the PAC193x Current and Power Monitor ICs via I2C.

Step 2d: Select the "System Module" in the Project Resourced pane

Step 2e: Configure the emulator pins to PGEC2 and PGED2



4 MHz Fosc/2

Oscillator Fractional Divisor

Clock Output Pin Configuration: OSC2 is clock output

Use Secondary Oscillator (31 - 33) kHz

Reference Oscillator Output

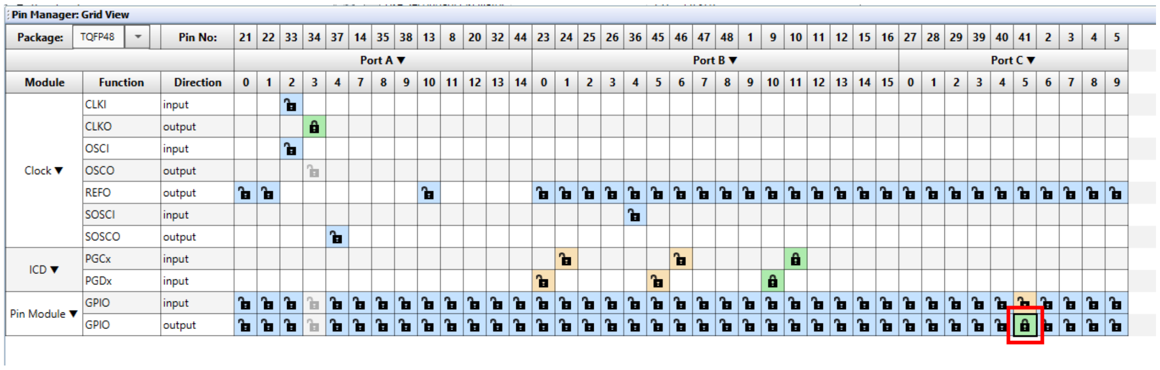
Enable Clock Switching

Enable Fail-Safe Monitor

Auto Tuning Mode: Disabled

Emulator Pin Placement: Communicate on PGEC2 and PGED2

Step 2f: Open the pin manager grid view and configure the RC5 GPIO as an output. This is connected to the blue "Wi-Fi LED".

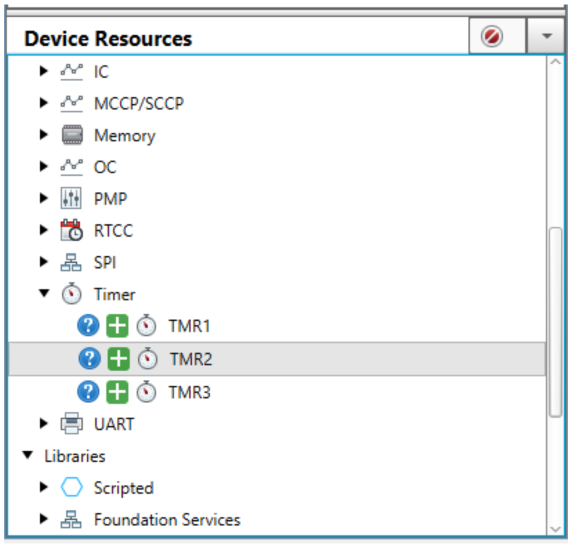


Pin Manager: Grid View

Package:	TQFP48	Pin No:	21	22	33	34	37	14	35	38	13	8	20	32	44	23	24	25	26	36	45	46	47	48	1	9	10	11	12	15	16	27	28	29	39	40	41	2	3	4	5
			Port A ▼													Port B ▼																Port C ▼									
Module	Function	Direction	0	1	2	3	4	7	8	9	10	11	12	13	14	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	0	1	2	3	4	5	6	7	8	9
Clock ▼	CLKI	input			🔓																																				
	CLKO	output				🔒																																			
	OSCI	input			🔓																																				
	OSCO	output				🔓																																			
	REFO	output	🔓	🔓							🔓					🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓
	SOSCI	input																		🔓																					
	SOSCO	output					🔓																																		
ICD ▼	PGCx	input															🔓					🔓					🔒														
	PGDx	input														🔓					🔓					🔒															
Pin Module ▼	GPIO	input	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓
	GPIO	output	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔓	🔒	🔓	🔓	🔓	🔓

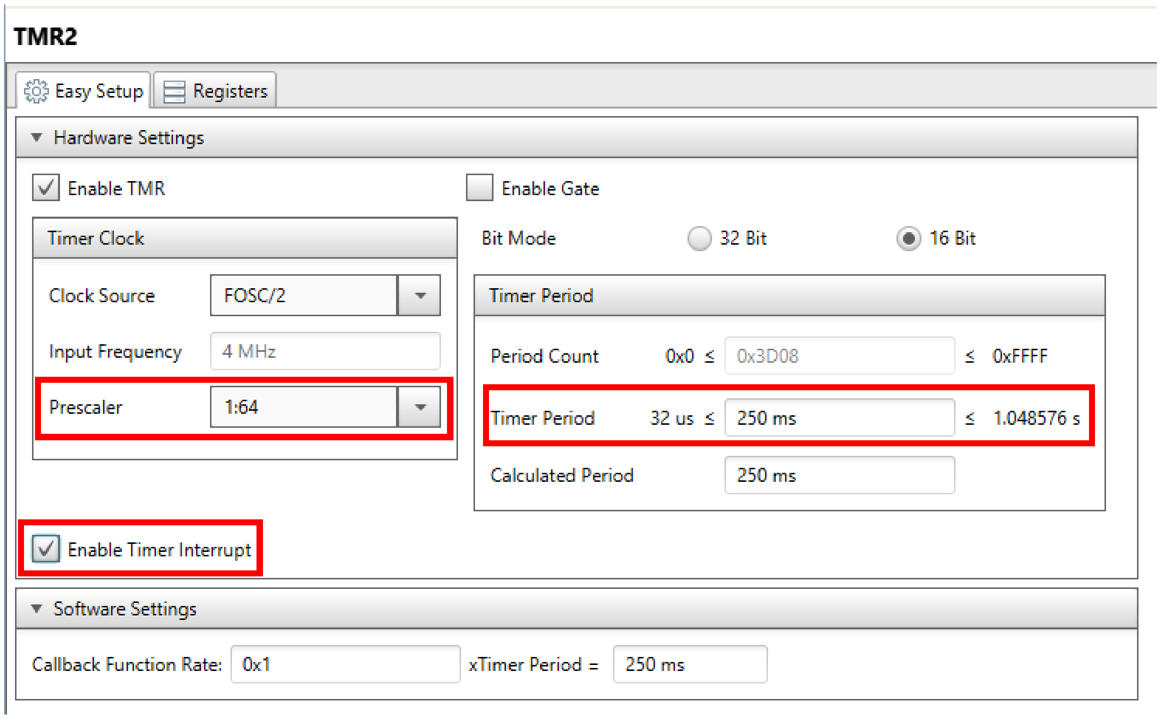
STEP 3: Create the example application code that blinks the LED

Step 3a: Go to Device Resources and add the TMR2 module to the project by clicking on the "+" symbol.



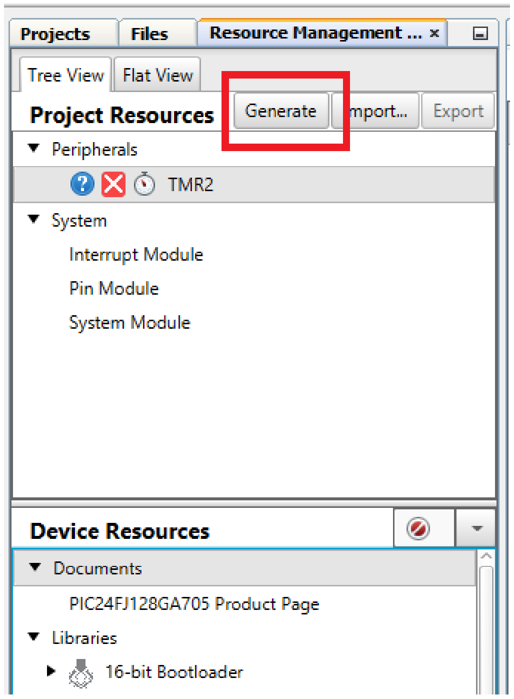
Step 3b: Open TMR2 Device Resource module if it wasn't automatically opened on the step above. Make the following changes to the TMR2 module configuration:

a. Configure pre-scaler to 1:64
b. Change Timer Period to 250ms
c. Enable Timer Interrupt.



Enable TMR ✓ | Enable Gate ☐
Bit Mode: ○ 32 Bit ● 16 Bit
Clock Source: FOSC/2
Input Frequency: 4 MHz
Prescaler: 1:64
Period Count: 0x0 ≤ 0x3D08 ≤ 0xFFFF
Timer Period: 32 us ≤ 250 ms ≤ 1.048576 s
Calculated Period: 250 ms
Enable Timer Interrupt ✓
Callback Function Rate: 0x1 xTimer Period = 250 ms

Step 3c: Generate the code by clicking the "Generate" button



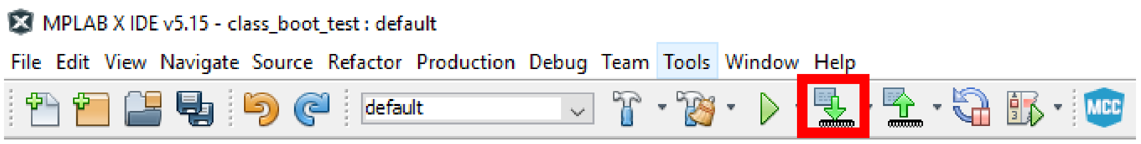
Step 3d: The generated code for TMR2 has a weakly prototyped function

`void TMR2_CallBack(void)`. To have code called on the timer interrupt, we need to create a function named `TMR2_CallBack(void)` and it will be called automatically on a timer interrupt.

Add a function in your `main.c` file called `TMR2_CallBack()`. In this function, add code to toggle the RC5 pin.

```
#include "mcc_generated_files/pin_manager.h"
#include "mcc_generated_files/tmr2.h"

void TMR2_CallBack(void)
{
    IO_RC5_Toggle();
}
```

Step 3e: Compile code and then download and run this code on the PIC-IoT board. Verify that the Blue LED is toggling at the correct speed. Verify that the blue LED toggles every 250 milliseconds. We now have an example application.

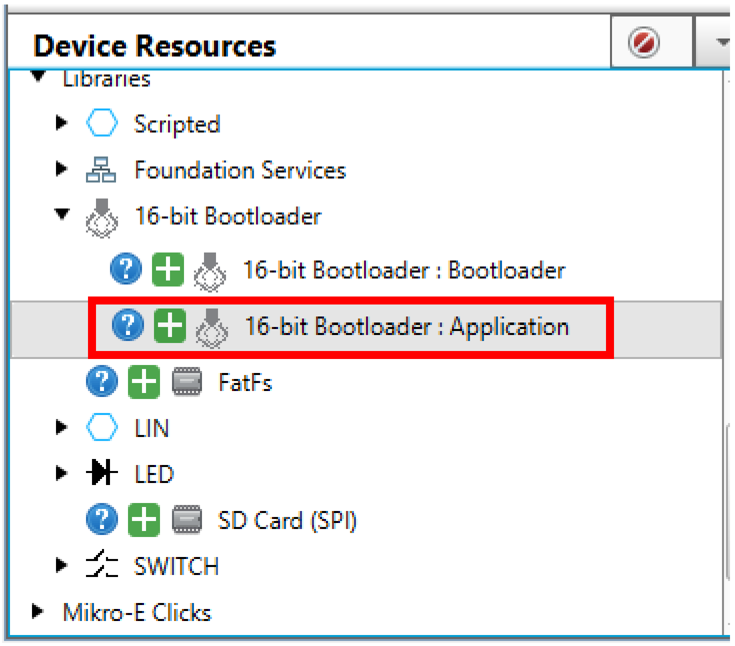

MPLAB X IDE v5.15 - class_boot_test : default
File Edit View Navigate Source Refactor Production Debug Team Tools Window Help
default

STEP 4: Modify application to work with the bootloader

In many boot loader solutions this is where custom linker scripts would have to be created. With this boot loader, no linker file modifications are required.

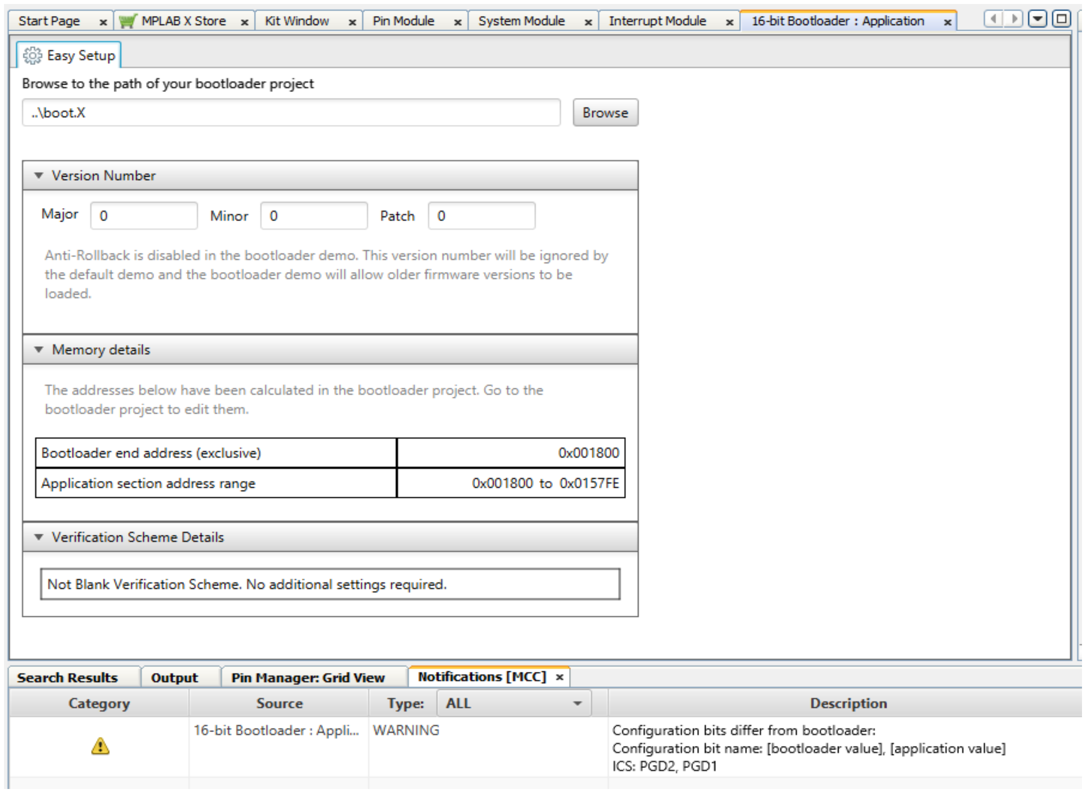
Step 4a: If MCC is not still open then re-open MCC.

Step 4b: Add the Bootloader : Application module to the project by clicking the "+" next to it in the Device Resourced pane. Make sure to click the "Application" module and not the "Bootloader" module.



This will open the Bootloader : Application module configuration window. If it does not, you can click on the module in the device resources pane.

Step 4b: The next step is to import the settings from the bootloader project. Press the browse button and browse to the location of the bootloader and then press select.



Browse to the path of your bootloader project

..\boot.X

Major 0 Minor 0 Patch 0

Anti-Rollback is disabled in the bootloader demo. This version number will be ignored by the default demo and the bootloader demo will allow older firmware versions to be loaded.

The addresses below have been calculated in the bootloader project. Go to the bootloader project to edit them.

Bootloader end address (exclusive)	0x001800
Application section address range	0x001800 to 0x0157FE

Not Blank Verification Scheme. No additional settings required.

Category	Source	Type: ALL	Description
⚠	16-bit Bootloader : Appli...	WARNING	Configuration bits differ from bootloader: Configuration bit name: [bootloader value], [application value] ICS: PGD2, PGD1

When the application library builder reads the bootloader project, it will read all of the addresses, interrupt and configuration bit settings and will create the code to match. This also means that if the user ever changes the bootloader settings, configuration bits or interrupts, the application library needs to be re-synced with the changes. Once the settings are imported, the user will see the screen show in the figure below. By importing these values here and adjusting the addresses in the *.s files, we do not need to modify the default linker scripts in any way.

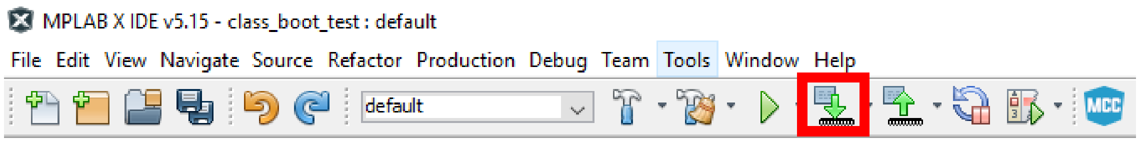
If the user now tries to generate the project, the user will get a notification error saying the configuration bits are different. In this case, MCC is detecting that the IOL1WAY and the debugger pins in this project do not match the settings in the bootloader.

Step 4c: Update the IOL1WAY setting in this project to “Allow Multiple Configurations” just like you did in Lab 1, Step 7 (in the system module under the register tab).

Step 4d: Update the debugger pins in project by going to the System Module and change the ICD pins

Step 4e: Press the Generate button again to generate the updated code for the application.

Step 4e: Compile code and then download and run this code on the PIC-IoT board. Verify that the Blue LED is toggling at the correct speed. Verify that the blue LED toggles every 250 milliseconds just as it did before.


MPLAB X IDE v5.15 - class_boot_test : default
File Edit View Navigate Source Refactor Production Debug Team Tools Window Help
default
MCC

Results:

We saw that when the application was programmed into the board, that the blue WiFi LED blinked every 250 ms.

When the bootloader files were added to the application project, we were still able to program the application into the board and run it, without the bootloader.

Summary:

When the bootloader files were added to the project in Step 4, this related the application code to the memory ranges we specified in the bootloader. The application is not ready to be loaded by the bootloader.

It's key to note here that the application can be both run independently from the bootloader, as we did in step 4, or using the bootloader, as we will see in lab 3. This means the application can be developed independently from the bootloader project. The reset and interrupts are forwarded to the application and have the same latency as they would when using the bootloader. The application code also sits in the exact memory locations as it would. The same .hex file can be loaded by a programmer or a bootloader and will work regardless without modification.

Lab 3 – Downloading the application example with the Universal Bootloader Host Application (UBHA)

Purpose:

Become familiar with the Universal Bootloader Host Application (UBHA) program used to bootload the application code.

Overview:

In this lab we will load the example application we created in lab 2 with the bootloader we created in lab 1. This lab will give experience using the bootloader PC program.

Procedure:

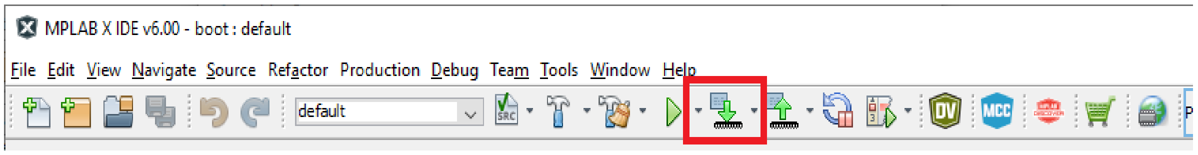
STEP 1: Reprogram the bootloader into the board

In Lab 2 steps 3 and 4 we programmed our example application in the board. We need to re-program the bootloader into the board

Step 1a: Close the "app" MPLAB X project

Step 1b: Open the "boot" MPLAB X project created in lab 1.

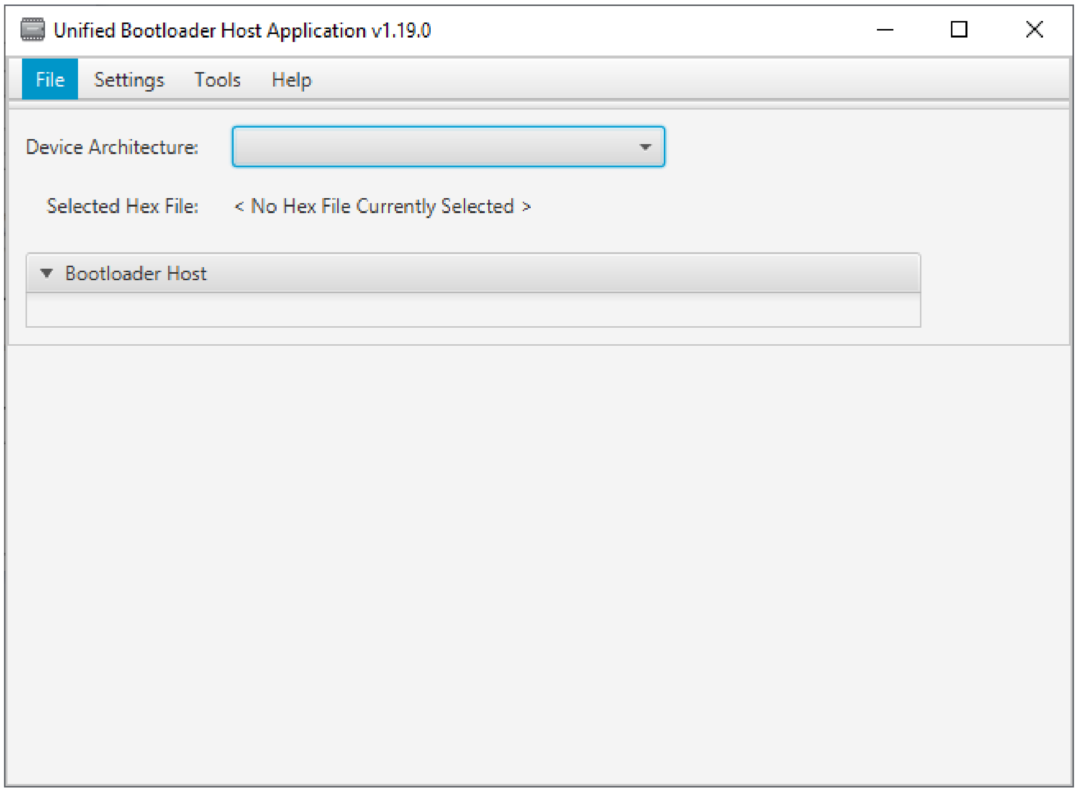
Step 1c: Compile and program the project into the PIC-IoT board by pressing the program button. After this step is complete, the yellow LED should be blinking on the board.


MPLAB X IDE v6.00 - boot : default
File Edit View Navigate Source Refactor Production Debug Team Tools Window Help
default

STEP 2: Bootloading an Application

In this step we will use the bootloader we created in lab 1 to load the application we created in lab 2. This lab will use the Unifier Bootloader Host Application found at www.microchip.com/16-bit-bootloader

Step 2a: Open the Unified Bootloader Host Application (UBHA).

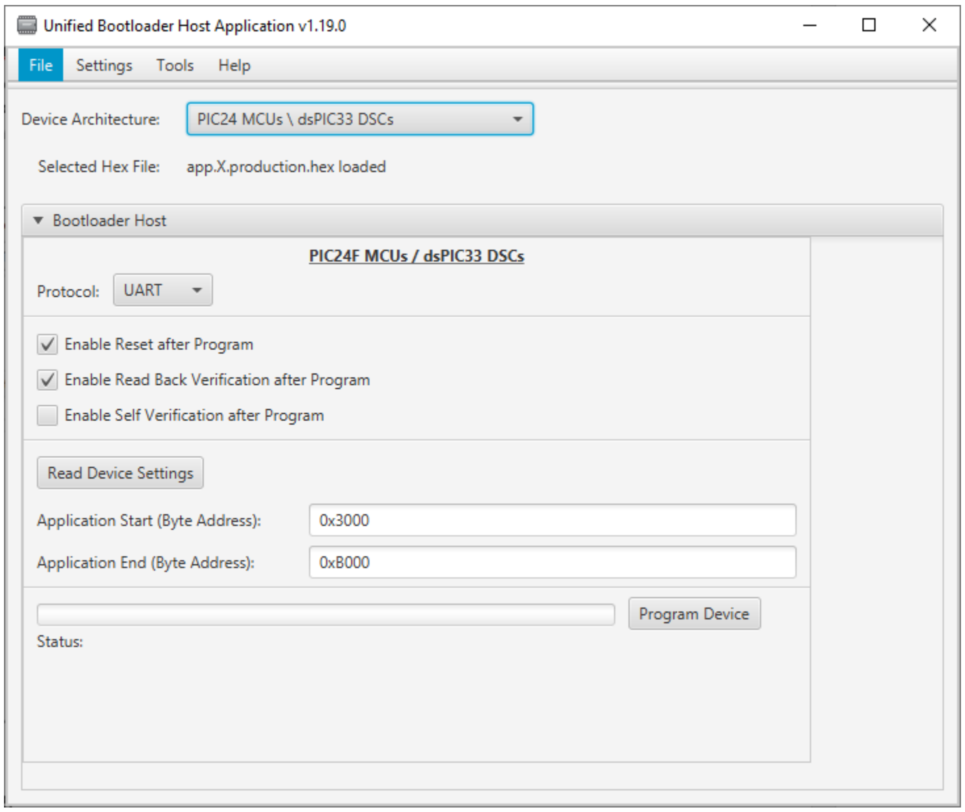

Unified Bootloader Host Application v1.19.0
File
Settings
Tools
Help
Device Architecture:
Selected Hex File:
< No Hex File Currently Selected >
Bootloader Host

Step 2b: Change the device architecture to “PIC24/dsPIC33/PIC33MM” or “PIC24 MCUs / dsPIC33 DSCs” depending on which version you are using.

Step 2c: Under the “Settings” menu, select the “Serial” option. This will bring up a window where you can change the UART port settings. Change the UART to the comm port where the IoT is connected to. This will be specific to your PC. Change baud rate to 9600. If you changed the UART port settings in lab 1, change this application to match.

Step 2d: Under the “File” menu, select the “Open/Load .hex file” option. Select the application hex file from the `app` project folder. The application hex file is in the `dist\default\production` of the application project.

At this point in time, the UHBA should look like the screen below.



Device Architecture: PIC24 MCUs \ dsPIC33 DSCs

Selected Hex File: app.X.production.hex loaded

Protocol: UART

- [✓] Enable Reset after Program
- [✓] Enable Read Back Verification after Program
- [ ] Enable Self Verification after Program

Application Start (Byte Address): 0x3000

Application End (Byte Address): 0xB000

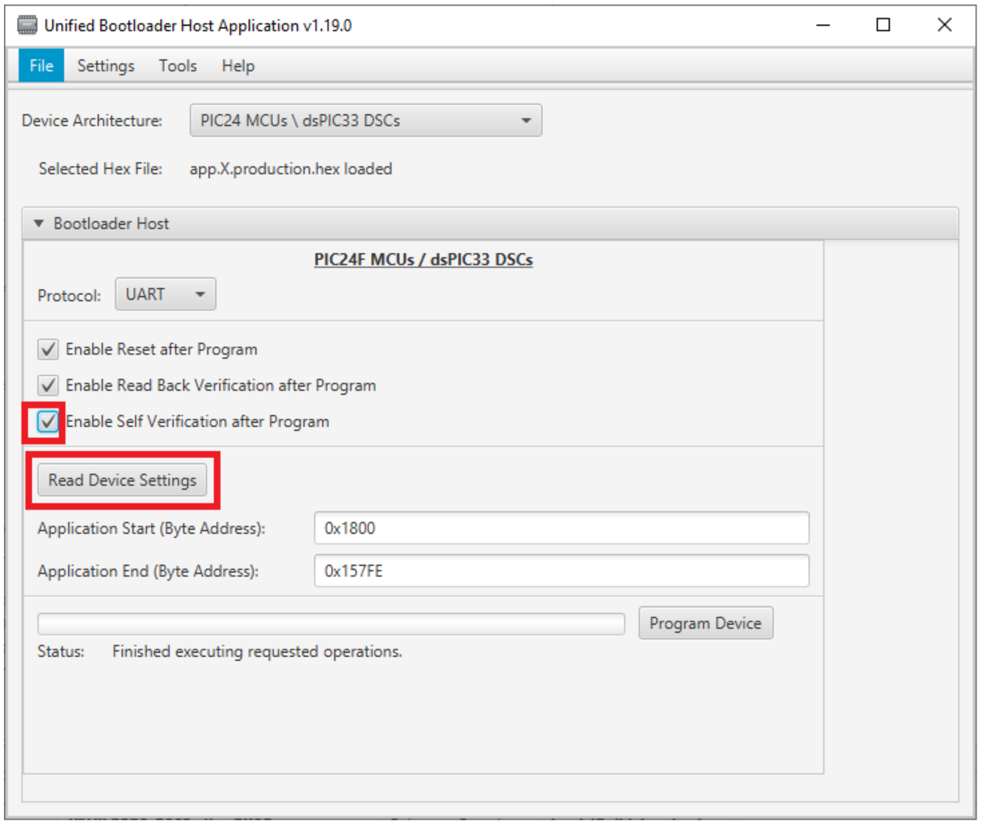
Step 2e: Update the start and end address to match what we configured in the boot loader project. These settings can be manually entered, or the user can press the Read Device Settings button on the UBHA which will query the bootloader for these values.

If no changes were made in the bootloader project, the values should be the following:

a) The Application Start address is 0x1800
b) The Application End address is 0x157FE

Step 2f: Next, set the “Enable Self Verification after Program” checkbox which will cause the UBHA to test out the self verify function of the bootloader and application.

At this point the system is ready to program and the fields should match the values shown below.


Unified Bootloader Host Application v1.19.0

Device Architecture: PIC24 MCUs \ dsPIC33 DSCs

Selected Hex File: app.X.production.hex loaded

PIC24F MCUs / dsPIC33 DSCs

Protocol: UART

[✓] Enable Reset after Program

[✓] Enable Read Back Verification after Program

[✓] Enable Self Verification after Program

Application Start (Byte Address): 0x1800

Application End (Byte Address): 0x157FE

Status: Finished executing requested operations.

Step 2g: Press the “Program Device” button.

During programming, the status bar and messages will sequence through different messages. Once the programming is completed the last command sent to the device is the command to reset itself. This will cause the LED pattern to change to the Application pattern.

If the application does not run successfully, open the menu option “Tools->Console”. You will see a of the events and any detailed errors messages if errors were encountered. The messages here may be able to give you an indication as to what might have gone wrong.

STEP 3: Re-entering the Bootloader

Next, we will test out the ability to re-enter the bootloader mode and download a new application after a previous application has been downloaded. This feature was enabled in the bootloader by adding the code to the function EnterBootloadMode() that checks the status of SW0 in GPIO pin RA7.

Step 3a: Unplug the IOT Board from the USB port and plug the board back in again

You should see the application start almost immediately since the application verification is very fast

Step 3c: Unplug the IOT Board from the USB port again

Step 3d: Hold the SW0 button while re-connecting the board to the USB port again.

The bootloader Yellow LED should start blinking because the bootloader detected SW0 was pressed after reset and the bootloader did not even attempt to verify the application code and stayed in firmware update mode.

Step 3e: Re-download the application with the UBHA by pressing the program button again. This should program the part as before. Once complete, it will issue a reset to the bootloader allowing the application code to run again.

Results:

We were able to use the bootloader to load a new application into the device memory. We were able to re-enter the bootloader after the application has been programmed using a pushbutton.

Summary:

This lab showed how to use the Unified Bootloader Host Application (UBHA) tool to load a hex file and program it into the device.

Lab 4 – Interrupts on Devices without an Alternate Interrupt Table

Purpose:

To understand the different types of interrupts and the interrupt controllers and how they affect the bootloader.

Overview:

This lab will start by using the bootloader from the lab1 project from before which toggles the Yellow “Data” LED and add a timer interrupt to toggle the red “Error” LED from the bootloader and then update Lab2 results to work with the updated bootloader.

On devices that don’t have an Alternate Interrupt Vector Table (AIVT), there is only one set of vectors for an interrupt. This means that interrupts can only be consumed by either the bootloader or the end application. We’ll see in this lab how to change modify an interrupt source to be used by the bootloader instead of the application on one of these devices.

NOTE: It is always critical that a bootloader return the processor to a state that can be used by the application correctly. This includes, but is not limited to,

- Disabling peripherals that might conflict with the applications usage
- Disabling interrupt sources
- Returning modified registers to their reset state

Most of the time these can all be accomplished by resetting the CPU after bootloading is complete and jumping to the application before making any modifications to the controller in the boot loader.

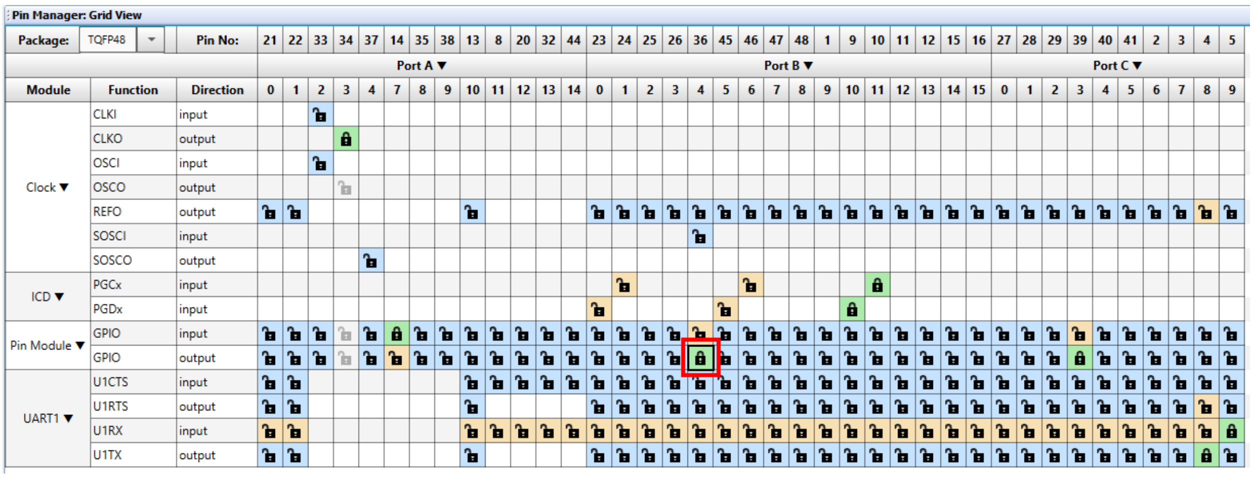
Procedure:

STEP 1: Update bootloader MCC settings required to blink LED using a timer interrupt

Step 1a: Open the “boot” project that we created in Lab 1.

Step 1b: Open MCC

Step 1c: Configure the GPIO pin RB4 (Red LED) to be an output. Keep all the other pin settings we had from previous labs.


Pin Manager: Grid View

Package: TQFP48 Pin No: 21 22 33 34 37 14 35 38 13 8 20 32 44 23 24 25 26 36 45 46 47 48 1 9 10 11 12 15 16 27 28 29 39 40 41 2 3 4 5

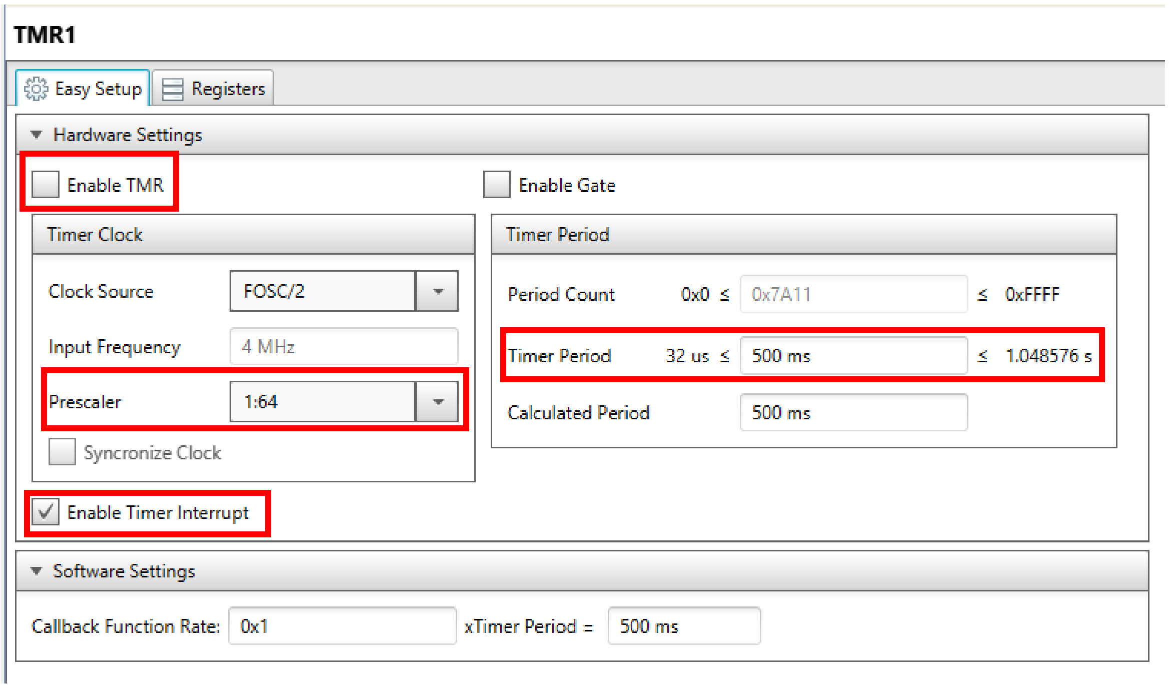
Port A ▼ 0 1 2 3 4 7 8 9 10 11 12 13 14; Port B ▼ 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15; Port C ▼ 0 1 2 3 4 5 6 7 8 9

Module	Function	Direction
Clock ▼	CLKI	input
	CLKO	output
	OSCI	input
	OSCO	output
	REFO	output
	SOSCI	input
	SOSCO	output
ICD ▼	PGCx	input
	PGDx	input
Pin Module ▼	GPIO	input
	GPIO	output
UART1 ▼	U1CTS	input
	U1RTS	output
	U1RX	input
	U1TX	output

Step 1d: From the Device Resources pane, add TMR1 to the project by clicking the "+" next to it.

Step 1e: Modify the following settings to the TMR1 module:

a. Change the prescaler to 1:64
b. Change the timer period to 500ms
c. Enable the timer interrupt
d. Disable the timer by default by unchecking the "Enable Timer" option


TMR1

Easy Setup | Registers

Hardware Settings

☐ Enable TMR ☐ Enable Gate

Timer Clock

Clock Source	FOSC/2
Input Frequency	4 MHz
Prescaler	1:64

☐ Syncronize Clock

Timer Period

Period Count	0x0 ≤	0x7A11	≤	0xFFFF
Timer Period	32 us ≤	500 ms	≤	1.048576 s
Calculated Period		500 ms		

☑ Enable Timer Interrupt

Software Settings

Callback Function Rate: 0x1 xTimer Period = 500 ms

Step 1f: Generate the updated code by pressing the "Generate" button at the top of the Project Resources pane.

STEP 2: Update bootloader demo code to blink the LED

Step 2a: Add the `TMR1_CallBack()` code to the top of the file main.c to toggle RB4 pin as shown below.

```
#include "mcc_generated_files/pin_manager.h"
#include "mcc_generated_files/tmr1.h"
void TMR1_CallBack(void)
{
    IO_RB4_Toggle();

}
```

Step 2b: To enable the Timer1, go to the file `boot_demo.c`, the function `BOOT_DEMO_Initialize()` and add the `TMR1_Start()` code shown below and add the `#include` for `tmr1.h` at the top of the file.

```
#include "../tmr1.h"
void BOOT_DEMO_Initialize(void)
{
    TMR1_Start();
}
```

Step 2c: The code to disable the Timer, add a call to `TMR1_Stop()` prior to starting the application as shown below. This is done in the `BOOT_DEMO_Tasks()` function in the `boot_demo.c` file

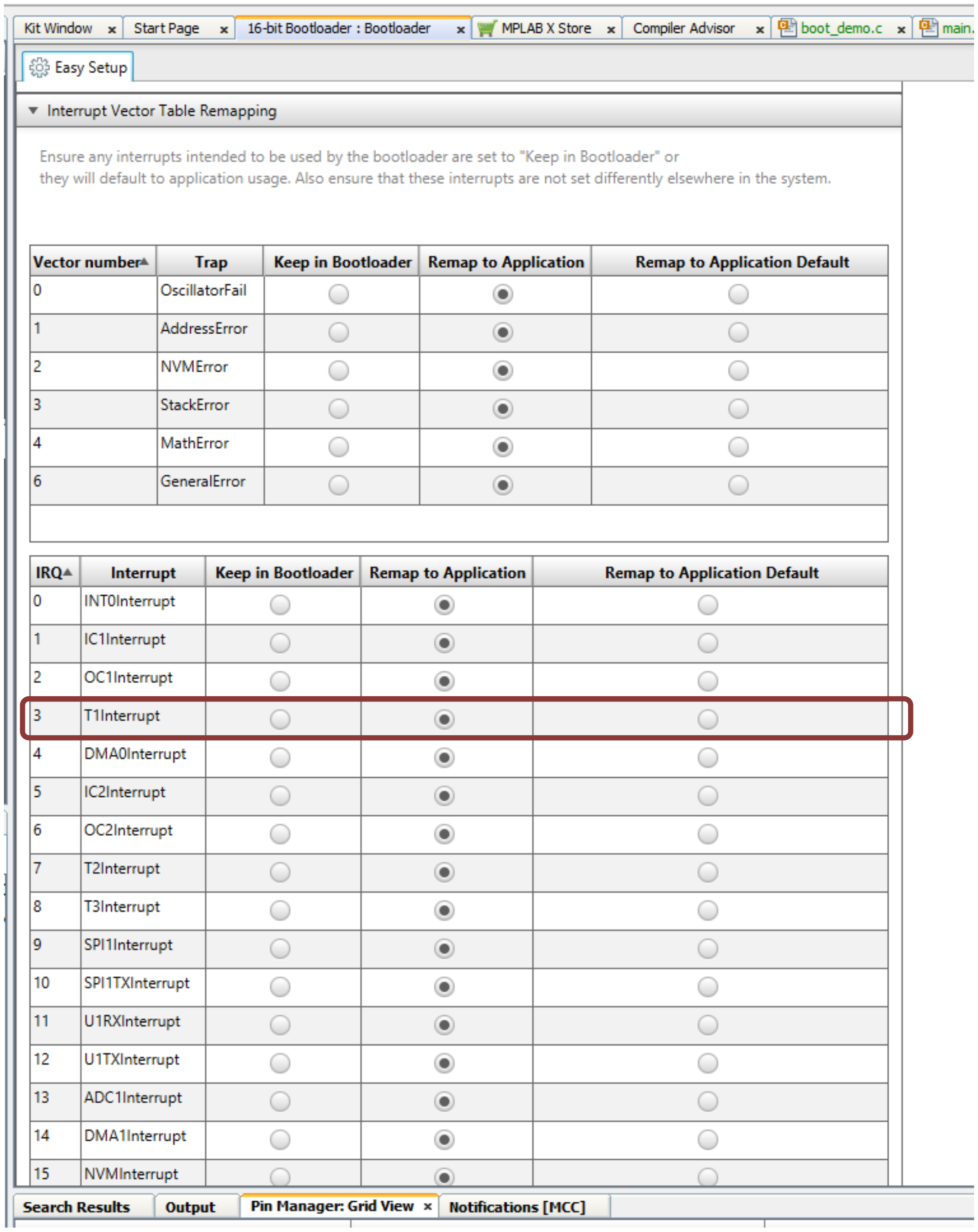
```
void BOOT_DEMO_Tasks(void)
{
    BlinkBootloaderLED();
    if(inBootloadMode == false)
    {
        if((EnterBootloadMode() == true) ||
            (BOOT_Verify() == false))
        {
            InBootloadMode = true;
        }
        else
        {
            TMR1_Stop();
            BOOT_StartApplication();
        }
    }
    BOOT_ProcessCommand();
}
```

Step 2d: Finally, lets change the frequency of the yellow LED, by changing the value of `RATE` in the function `BlinkBootloaderLED()`. Change `RATE` from 250 to 500.

Step 2e: Compile and program the code into the board by pressing the “Make and Program” button.

Notice now that the LEDs just come on and stay on and the red LED isn’t blinking. Stop and take a moment to consider why neither the original “Yellow” LED is no longer blinking and the new “Red” is not blinking.

To understand why, open MCC and go to the 16-bit Bootloader:Bootloader library dialog shown below. Scroll down to the Interrupt Vector Table Remapping section, seen below.



Interrupt Vector Table Remapping

Ensure any interrupts intended to be used by the bootloader are set to "Keep in Bootloader" or they will default to application usage. Also ensure that these interrupts are not set differently elsewhere in the system.

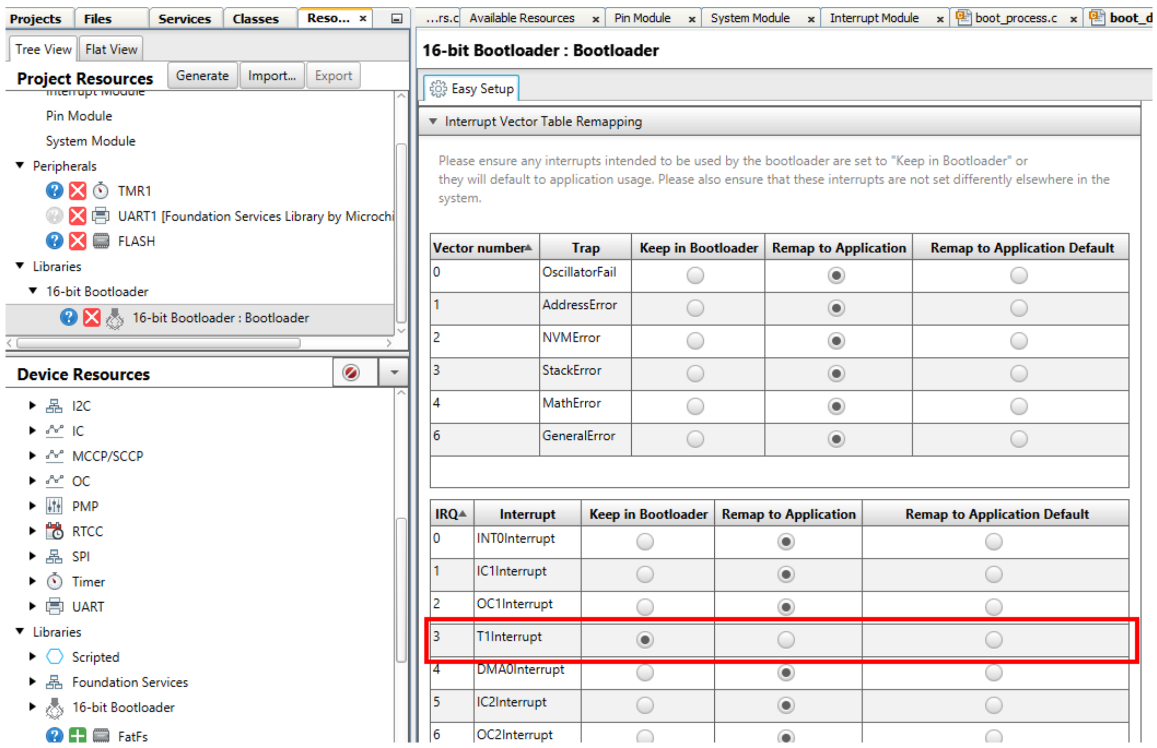
Vector number	Trap	Keep in Bootloader	Remap to Application	Remap to Application Default
0	OscillatorFail	○	●	○
1	AddressError	○	●	○
2	NVMError	○	●	○
3	StackError	○	●	○
4	MathError	○	●	○
6	GeneralError	○	●	○

IRQ	Interrupt	Keep in Bootloader	Remap to Application	Remap to Application Default
0	INT0Interrupt	○	●	○
1	IC1Interrupt	○	●	○
2	OC1Interrupt	○	●	○
3	T1Interrupt	○	●	○
4	DMA0Interrupt	○	●	○
5	IC2Interrupt	○	●	○
6	OC2Interrupt	○	●	○
7	T2Interrupt	○	●	○
8	T3Interrupt	○	●	○
9	SPI1Interrupt	○	●	○
10	SPI1TXInterrupt	○	●	○
11	U1RXInterrupt	○	●	○
12	U1TXInterrupt	○	●	○
13	ADC1Interrupt	○	●	○
14	DMA1Interrupt	○	●	○
15	NVMInterrupt	○	●	○

Look at how T1Interrupt is configured. It’s configured to forward all TMR1 interrupts to the application. However, there currently is no application loaded. So, when a TMR1 interrupt occurs it’s being forwarded to blank application code which currently has nothing in it. This results in a device reset. This then causes the code to reconfigure the part, re-enables this interrupt and starts again. This then repeats over and over

We are forwarding the TMR1 interrupts to the application when the bootloader needs it.

Step 2f: To solve this problem, we need to stop the interrupt forwarding for this interrupt. To do this, find the T1Interrupt option in the Interrupt Vector Table Remapping section and change it to "Keep In Bootloader". This will cause the interrupt to be vectored to the bootloader ISR for TMR1 instead of to the application.



16-bit Bootloader : Bootloader

Interrupt Vector Table Remapping

Please ensure any interrupts intended to be used by the bootloader are set to "Keep in Bootloader" or they will default to application usage. Please also ensure that these interrupts are not set differently elsewhere in the system.

Vector number	Trap	Keep in Bootloader	Remap to Application	Remap to Application Default
0	OscillatorFail	○	●	○
1	AddressError	○	●	○
2	NVMError	○	●	○
3	StackError	○	●	○
4	MathError	○	●	○
6	GeneralError	○	●	○

IRQ	Interrupt	Keep in Bootloader	Remap to Application	Remap to Application Default
0	INT0Interrupt	○	●	○
1	IC1Interrupt	○	●	○
2	OC1Interrupt	○	●	○
3	T1Interrupt	●	○	○
4	DMA0Interrupt	○	●	○
5	IC2Interrupt	○	●	○
6	OC2Interrupt	○	●	○

Step 2g: Now, Generate the code again, Rebuild and then download and program the code.

Do both the Red and Yellow LED toggle at different rates? They should. This means the bootloader is now keeping the TMR1 interrupt correctly.

Step 2h: Since the bootloader configuration has been changed, the application needs to be re-generated to account for the change in the interrupt table. This is because the interrupt settings are a shared resource/setting between the bootloader and the application. So now lets open up the project "app" from lab 2.

a. Open the app project
b. Open MCC
c. Click "Generate"
d. Rebuild the application, but don't download the code to the device.

Step 2i: Use the UBHA to download the new application as we did in Lab 3. After downloading and running, bootloader should stop blinking the Yellow and Red LEDs. The Blue LED in the application should be blinking. If this did not happen, open the Tools->Console and see if there are any error messages.

Results:

We how to identify an issue where an interrupt is not happening when expected in a the bootloader of a device that doesn’t have an alternate interrupt table.

We showed how to have that interrupt kept in the bootloader rather than forwarded to the application.

Summary:

In devices without an AIVT table, the hardware interrupt vector table can only be programmed for one location. That can either be in the bootloader or in the application space.

The 16-bit bootloader MCC module allows the designer to select if they want the interrupt to be forwarded to the application or the bootloader. By default all interrupts are forwarded to the application.

In some devices, the AIVT table is only accessible when the CodeGuard™ security module is enabled. In Lab 5 we will explore how enabling this module changes the bootloader module.

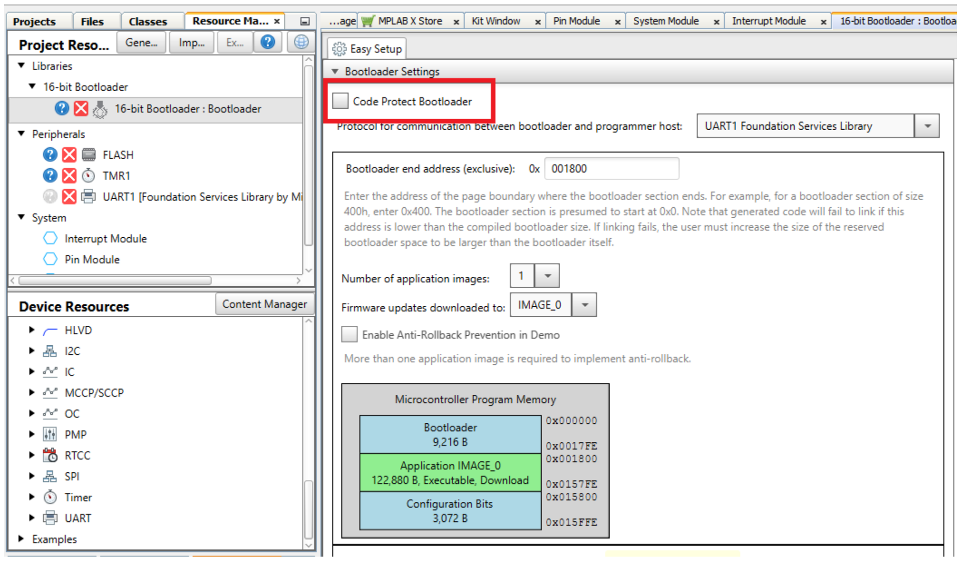
Lab 5 – Bootloading while using the CodeGuard™ Security Module

Purpose:

To understand how enabling the CodeGuard™ flash security module can impact a bootloader design using the 16-bit bootloader solution in MCC.

Overview:

In the previous lab we saw how a specific interrupt could either be used by the bootloader or by the application but not both. This is because in some devices there is only a single IVT interrupt table. One solution to this is to enable the CodeGuard™ flash security module by enabling "Code Protect" option in the bootloader configuration screen below.



Code Protect Bootloader

Protocol for communication between bootloader and programmer host: UART1 Foundation Services Library

Bootloader end address (exclusive): 0x 001800

Enter the address of the page boundary where the bootloader section ends. For example, for a bootloader section of size 400h, enter 0x400. The bootloader section is presumed to start at 0x0. Note that generated code will fail to link if this address is lower than the compiled bootloader size. If linking fails, the user must increase the size of the reserved bootloader space to be larger than the bootloader itself.

Number of application images: 1

Firmware updates downloaded to: IMAGE_0

Enable Anti-Rollback Prevention in Demo

More than one application image is required to implement anti-rollback.

Microcontroller Program Memory

Bootloader 9,216 B	0x000000 – 0x0017FE
Application IMAGE_0 122,880 B, Executable, Download	0x001800 – 0x0157FE
Configuration Bits 3,072 B	0x015800 – 0x015FFE

When CodeGuard™ is enabled, it allows an Alternate Interrupt Vector Table(AIVT) to be enabled as well which is a second interrupt vector table that can be used by the application while the bootloader continues to use the original IVT.

To demonstrate this feature, this lab will modify the previous code to just use the Timer 1 Interrupt in both the bootloader and application. In the bootloader we will blink the red LED and in the application code we will blink the blue LED, both using timer 1.

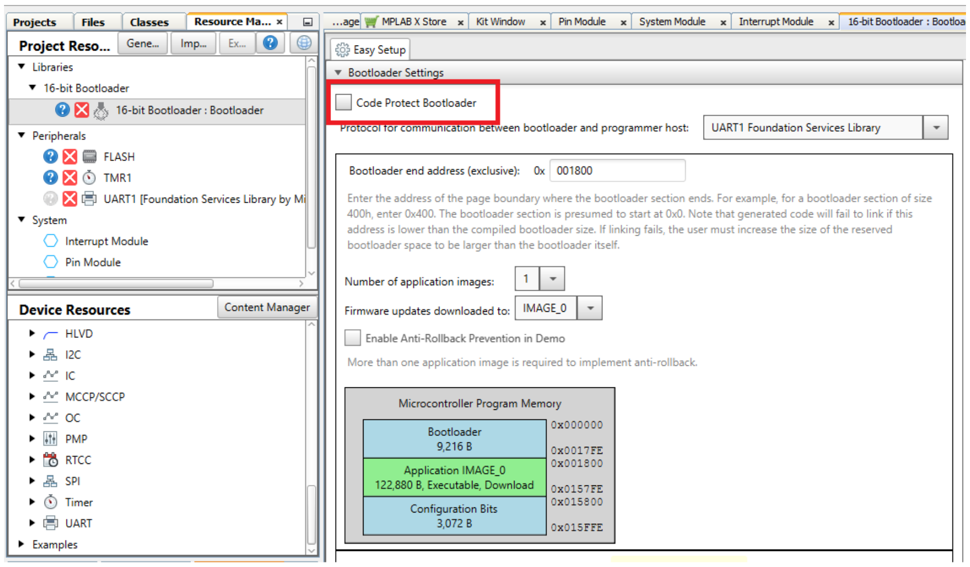
Procedure:

STEP 1: Update bootloader MCC settings to enable the code protect feature

Step 1a: Open the boot project from lab 4.

Step 1b: Open MCC.

Step 1c: Select Bootloader:Bootloader in the Project Resources. This will open up the bootloader configuration screen below.



Code Protect Bootloader

Protocol for communication between bootloader and programmer host: UART1 Foundation Services Library

Bootloader end address (exclusive): 0x 001800

Enter the address of the page boundary where the bootloader section ends. For example, for a bootloader section of size 400h, enter 0x400. The bootloader section is presumed to start at 0x0. Note that generated code will fail to link if this address is lower than the compiled bootloader size. If linking fails, the user must increase the size of the reserved bootloader space to be larger than the bootloader itself.

Number of application images: 1

Firmware updates downloaded to: IMAGE_0

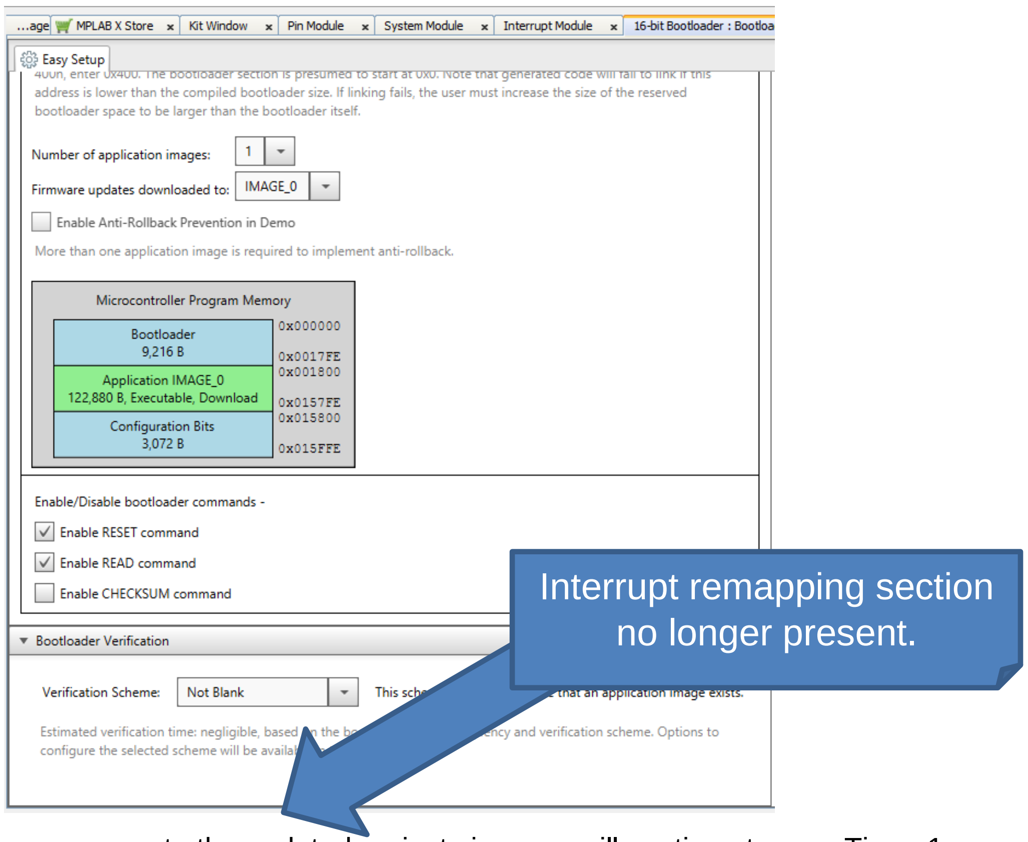
Enable Anti-Rollback Prevention in Demo

More than one application image is required to implement anti-rollback.

Microcontroller Program Memory

Bootloader 9,216 B	0x000000 0x0017FE
Application IMAGE_0 122,880 B, Executable, Download	0x001800 0x0157FE
Configuration Bits 3,072 B	0x015800 0x015FFE

Step 1d: Enable Code Protect which will enable CodeGuard™. Notice that when this is done the interrupt mapping section of the configuration screen disappears. This is because since we now have separate IVT and AIVT tables, we no longer need to pick where each interrupt goes to.



400h, enter 0x400. The bootloader section is presumed to start at 0x0. Note that generated code will fail to link if this address is lower than the compiled bootloader size. If linking fails, the user must increase the size of the reserved bootloader space to be larger than the bootloader itself.

Number of application images: 1

Firmware updates downloaded to: IMAGE_0

Enable Anti-Rollback Prevention in Demo

More than one application image is required to implement anti-rollback.

Microcontroller Program Memory

Bootloader 9,216 B	0x000000 0x0017FE
Application IMAGE_0 122,880 B, Executable, Download	0x001800 0x0157FE
Configuration Bits 3,072 B	0x015800 0x015FFE

Enable/Disable bootloader commands -

- [✓] Enable RESET command
- [✓] Enable READ command
- [ ] Enable CHECKSUM command

Bootloader Verification

Verification Scheme: Not Blank   This sch... that an application image exists.

Estimated verification time: negligible, based on the b... ency and verification scheme. Options to configure the selected scheme will be availa...

Step 1e: We can now generate the updated project since we will continue to user Timer 1 as we did previously. This should generate with no errors.

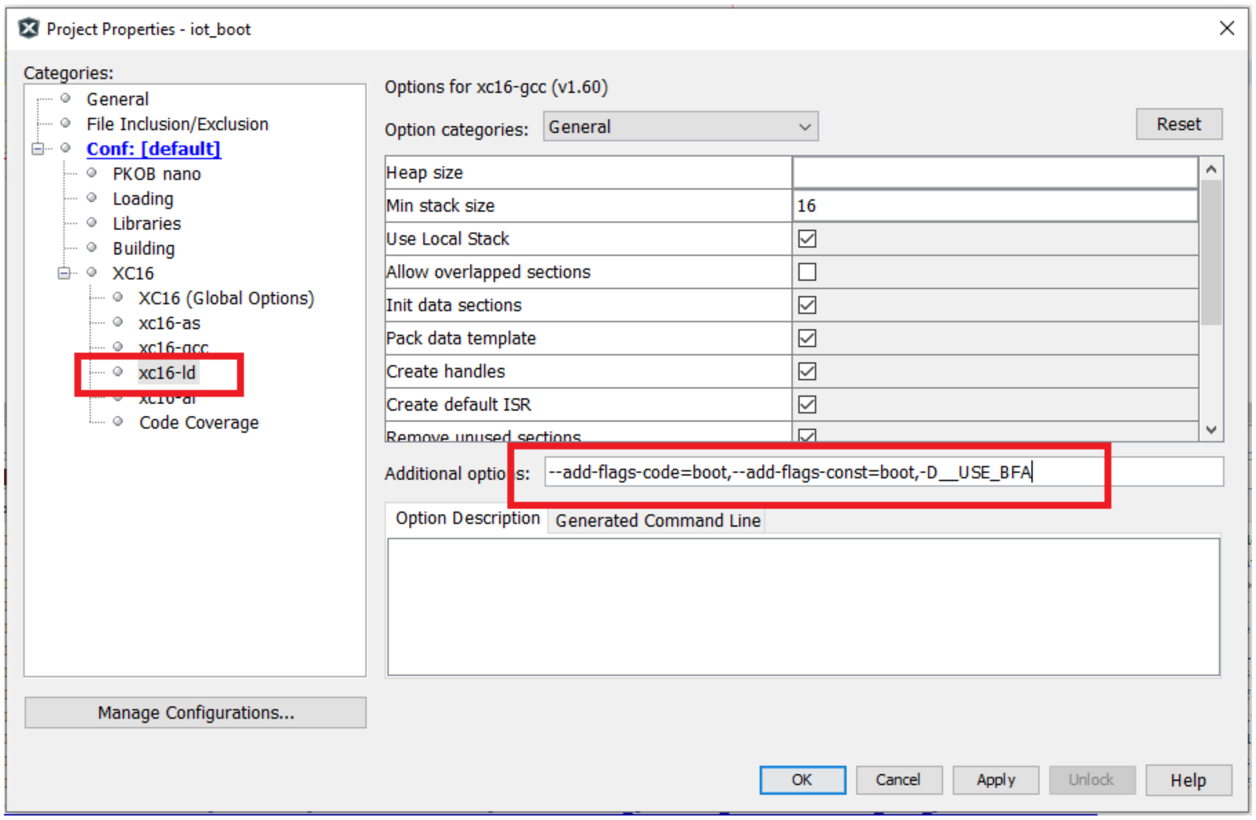
Press the “Generate” button to regenerated the code with the code protect feature enabled.

STEP 2: Update project linker settings to link to newly created boot section

Step 2a: Re-compile the project with the new changes.

Step 2b: Notice that while the code is compiling, we get an `#error` in the file `boot_code_protect.c` shown below. This error is telling us to add the linker options `--add-flags-code=boot,--add-flags-const=boot,-D__USE_BFA` to the linker command line. So following the first three steps in the comment field below, we will update the linker options show below as well. Its easiest to copy the command line below and paste them into the project properties.

```
#error Add the following to the linker command line: --add-flags-
code=boot,--add-flags-const=boot,-D__USE_BFA
/*
1. Goto Project Properties
2. Goto XC16->xc16-ld
3. Under additional options add the following: --add-flags-code=boot,--
add-flags-const=boot,-D__USE_BFA
4. Comment out the line with the #error message when finished
*/
```



Options for xc16-gcc (v1.60)

Option categories: General

Heap size	
Min stack size	16
Use Local Stack	☑
Allow overlapped sections	☐
Init data sections	☑
Pack data template	☑
Create handles	☑
Create default ISR	☑
Remove unused sections	☑

Additional options: --add-flags-code=boot,--add-flags-const=boot,-D__USE_BFA

Step 2c: Once these changes are made, please re-compile the project. This should compile cleanly.

Note to comment out the #error once the modifications are made, step 4 above.

Step 2d: Download the project to the board and it should blink the red and yellow LEDs

STEP 3: Update the application project

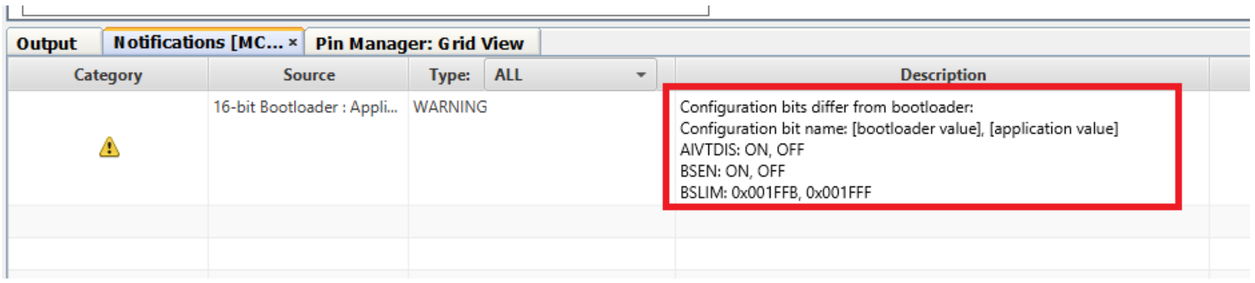
The changes we made in step 2 will cause changes that need to be updated in the device. Additionally we need to update the demo to use the same interrupt source to verify we can use the same interrupt source in both the bootloader and the application project.

Step 3a: Open the app project previously created.

Step 3b: Open MCC

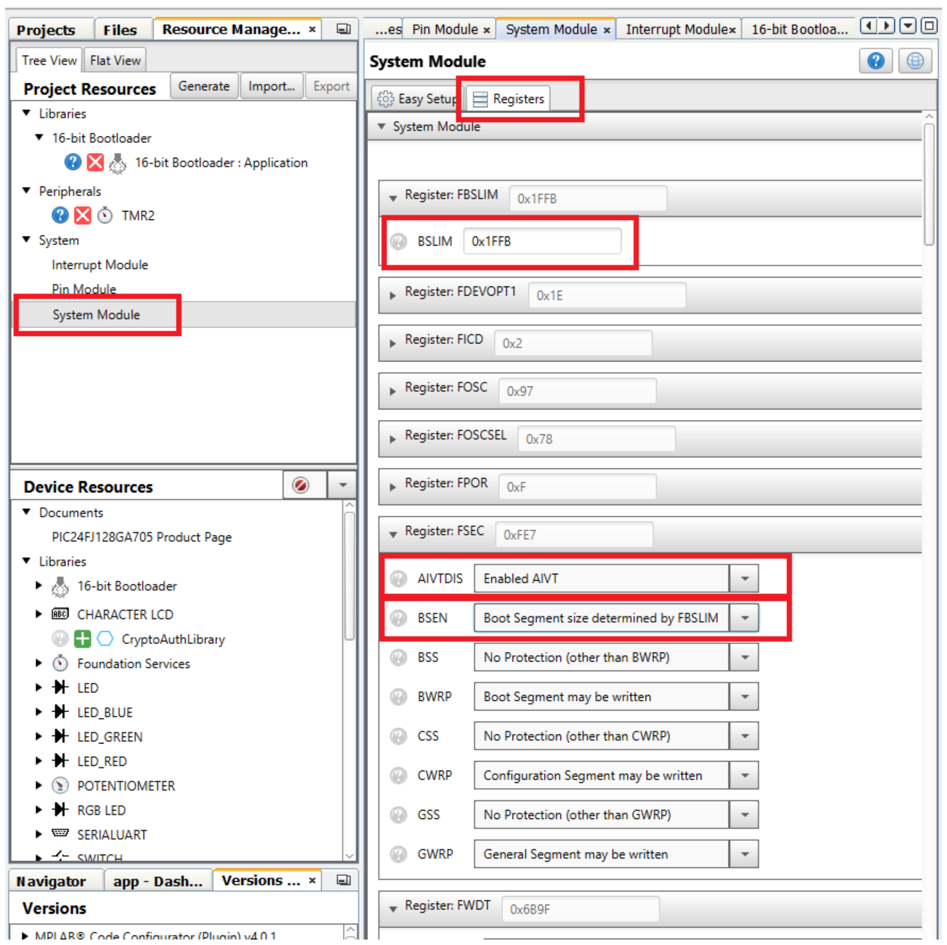
Step 3c: Look at the notification box in MCC.

MCC has detected that the configuration words in the bootloader has changed and we should change them here. If this was a new project, MCC would have pulled in the CodeGuard™ settings automatically, however since we are modifying the previous project, we need to change them manually.



Category	Source	Type: ALL	Description
⚠	16-bit Bootloader : Appli...	WARNING	Configuration bits differ from bootloader: Configuration bit name: [bootloader value], [application value] AIVTDIS: ON, OFF BSEN: ON, OFF BSLIM: 0x001FFB, 0x001FFF

Step 3d: To change them, we need to go to Project Resources, select the System Module. Then select the Registers tab. Then change the three configuration items show below.



Register: FBSLIM 0x1FFB

BSLIM	0x1FFB

Register: FDEVOPT1 0x1E

Register: FICD 0x2

Register: FOSC 0x97

Register: FOSCSEL 0x78

Register: FPOR 0xF

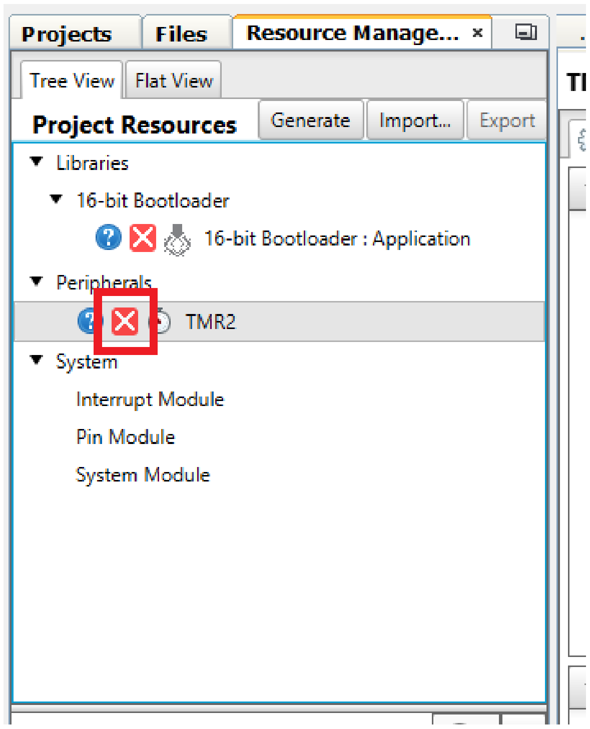
Register: FSEC 0xFE7

AIVTDIS	Enabled AIVT
BSEN	Boot Segment size determined by FBSLIM
BSS	No Protection (other than BWRP)
BWRP	Boot Segment may be written
CSS	No Protection (other than CWRP)
CWRP	Configuration Segment may be written
GSS	No Protection (other than GWRP)
GWRP	General Segment may be written

Register: FWDT 0x6B9F

Step 3e: After these are changed the notifications in the Notification tab should clear.

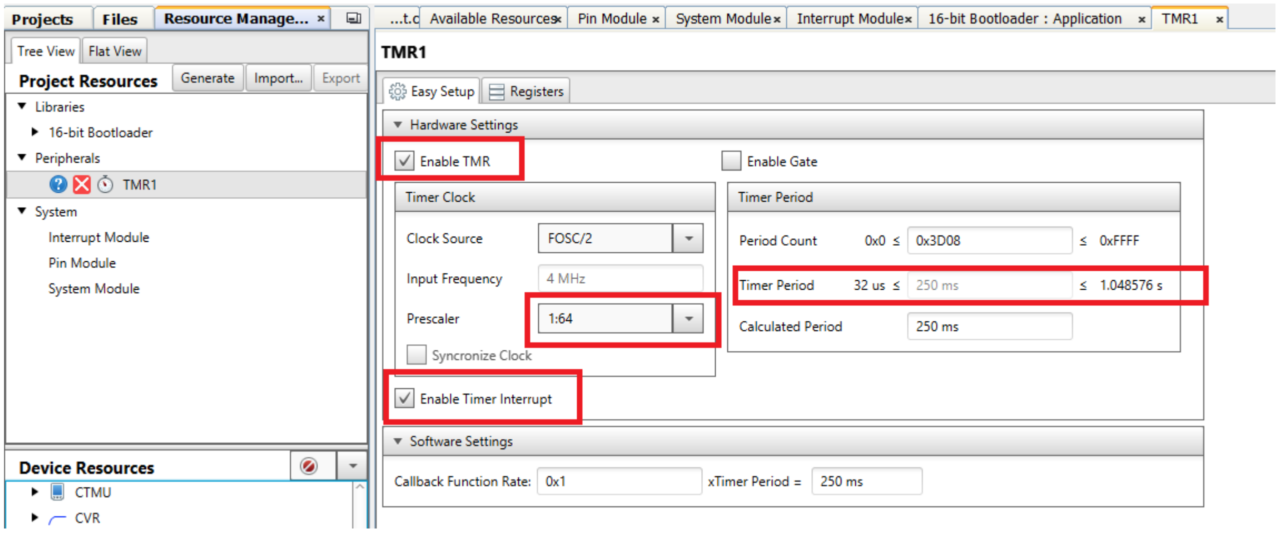
Step 3f: Remove Timer 2 from the project, by going back to the Project Resources, under peripherals and clicking the red “X” shown below. This will remove Timer 2 from the project.



Step 3g: Add Timer 1 to the project like we did for the bootloader. From the Device Resources pane, add TMR1 to the project by clicking the “+” next to it.

Step 3h: Modify the following settings to the TMR1 module:

a. Change the prescaler to 1:64
b. Change the timer period to 250ms
c. Enable the timer interrupt
d. Enable the timer by default by leaving the “Enable Timer” option enabled



TMR1

Hardware Settings

[✓] Enable TMR
[ ] Enable Gate

Timer Clock

Clock Source	FOSC/2
Input Frequency	4 MHz
Prescaler	1:64

[ ] Syncronize Clock

Timer Period

Period Count	0x0 ≤	0x3D08	≤ 0xFFFF
Timer Period	32 us ≤	250 ms	≤ 1.048576 s
Calculated Period		250 ms	

[✓] Enable Timer Interrupt

Software Settings

Callback Function Rate: 0x1   xTimer Period = 250 ms

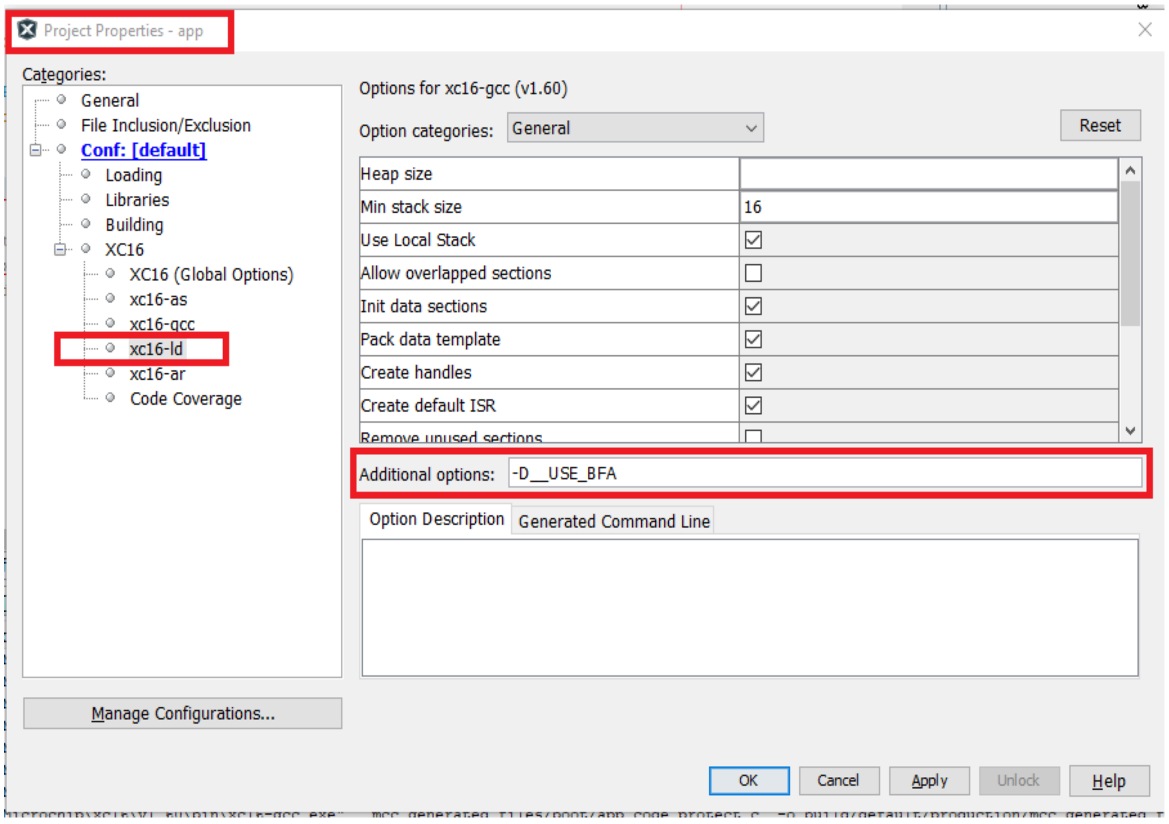
Step 3i: Go to the file main.c and remove the following code since we are no longer using Timer 2.

```
#include "mcc_generated_files/tmr2.h"

void TMR2_CallBack(void)
{
    IO_RC5_Toggle();
}
```

Step 3j: Generate the updated code by pressing the "Generate" button at the top of the Project Resources pane.

Step 3k: Compile the code. You should see an error like the bootloader error shown below. This message is telling us to update the linker command line for CodeGuard and gives us the steps to do it. The message and the configuration screens are show below.

```
#error Add the following to the linker command line: -D__USE_BFA
/*
1. Goto Project Properties
2. Goto XC16->xc16-ld
3. Under additional options add the following: -D__USE_BFA
4. Delete this #error message when finished
*/
```



Options for xc16-gcc (v1.60)

Option categories: General

Heap size	
Min stack size	16
Use Local Stack	☑
Allow overlapped sections	☐
Init data sections	☑
Pack data template	☑
Create handles	☑
Create default ISR	☑
Remove unused sections	☐

Additional options: -D__USE_BFA

Step 3l: Comment out the #error line and recompile. This time it should compile cleanly.

Step 3m: Add the `TMR1_CallBack()` code to the top of the file `main.c` to toggle RC5 pin as shown below.

```
      #include "mcc_generated_files/tmr1.h"
      void TMR1_CallBack(void)
      {
          IO_RC5_Toggle();
      }
```

Step 3n: Compile the code and again, it should compile cleanly. We can now just download and run the code with MPLAB X or if the bootloader with CodeGuard is running, we can just use the UBHA to download the code.

Step 3o: When downloaded, the blue LED should be blinking. This shows that the same interrupt source can now be used in not the bootloader and application.

Results:

The bootloader is now protected by the CodeGuard flash security module. This allows the bootloader code to be unmodifiable by changing boot sector write protection bits. This is a critical first step towards implementing a secure boot solution.

Summary:

With devices that support the CodeGuard flash security module, it is beneficial to enable that feature to protect the bootloader from either accidental or intentional modification. This is a critical first step towards implementing a secure boot solution. Secure boot is increasingly becoming a system require due to various security regulations being put in place in both the US and the EU for network connected devices.

Lab 6 – Application Verification

Purpose:

Configure the bootloader and application projects to use CRC32 verification to verify the integrity of the application image before running it.

Overview:

In the previous labs we have not really looked at the verification scheme used. The default verification used in the previous labs has been the "Not Blank" verification. This method is by far, the simplest method used. The bootloader simply checks to see if there is code at the application reset vector and if so, it branches to the application. Obviously, this not a robust way to verify the application.

The 16-Bit Bootloader module supports the ability to supports multiple methods of verification including:

- Not Blank
- Checksum
- CRC32
- SHA256
- ECDSA

For this lab we are going to focus on using the CRC32 verification method. The Checksum and SHA256 will have identical flows, just the algorithm will be different.

Procedure:

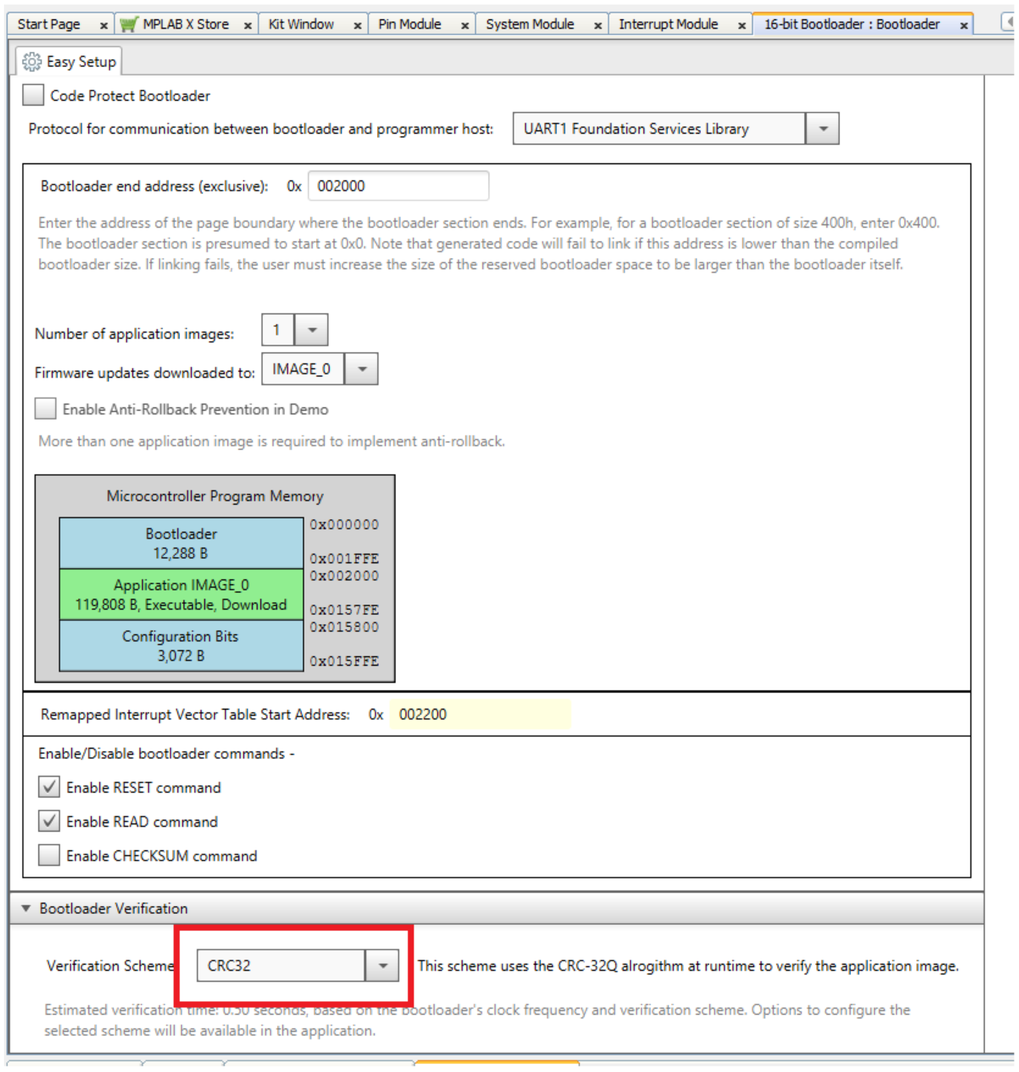
STEP 1: Update bootloader project to use verification method

Step 1a: Open the "boot" project from Lab 4

Step 1b: Open MCC

Step 1c: Open the "16-bit Bootloader : Bootloader" module by clicking it in the "Project Resources" pane.

Step 1d: Scroll down to the Bootloader Verification section shown below and change the Verification Scheme from "Not Blank" to CRC32. Notice that the bootloader end address changed when the CRC32 verification method was selected. This is because the CRC32 verification code increased the size of the bootloader.



Code Protect Bootloader

Protocol for communication between bootloader and programmer host: UART1 Foundation Services Library

Bootloader end address (exclusive): 0x 002000

Enter the address of the page boundary where the bootloader section ends. For example, for a bootloader section of size 400h, enter 0x400. The bootloader section is presumed to start at 0x0. Note that generated code will fail to link if this address is lower than the compiled bootloader size. If linking fails, the user must increase the size of the reserved bootloader space to be larger than the bootloader itself.

Number of application images: 1

Firmware updates downloaded to: IMAGE_0

Enable Anti-Rollback Prevention in Demo

More than one application image is required to implement anti-rollback.

Microcontroller Program Memory

Region	Start	End
Bootloader 12,288 B	0x000000	0x001FFE
Application IMAGE_0 119,808 B, Executable, Download	0x002000	0x0157FE
Configuration Bits 3,072 B	0x015800	0x015FFE

Remapped Interrupt Vector Table Start Address: 0x 002200

Enable/Disable bootloader commands -

☑ Enable RESET command

☑ Enable READ command

☐ Enable CHECKSUM command

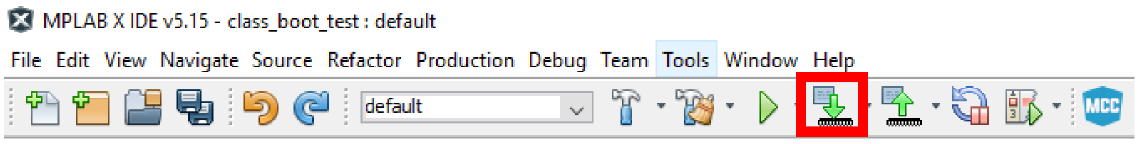
Bootloader Verification

Verification Scheme CRC32 This scheme uses the CRC-32Q alrogithm at runtime to verify the application image.

Estimated verification time: 0.50 seconds, based on the bootloader's clock frequency and verification scheme. Options to configure the selected scheme will be available in the application.

Step 1e: Once CRC32 is selected, go to the Project Resources tab and press the Generate button again. This will generate the new bootloader code needed for the CRC32 verification of the application. The address range being check is shown in the "Application section address range" shown in the dialog above.

Step 1f: Build and program the new bootloader code into the board. At this point both the yellow and red LEDs should be blinking.



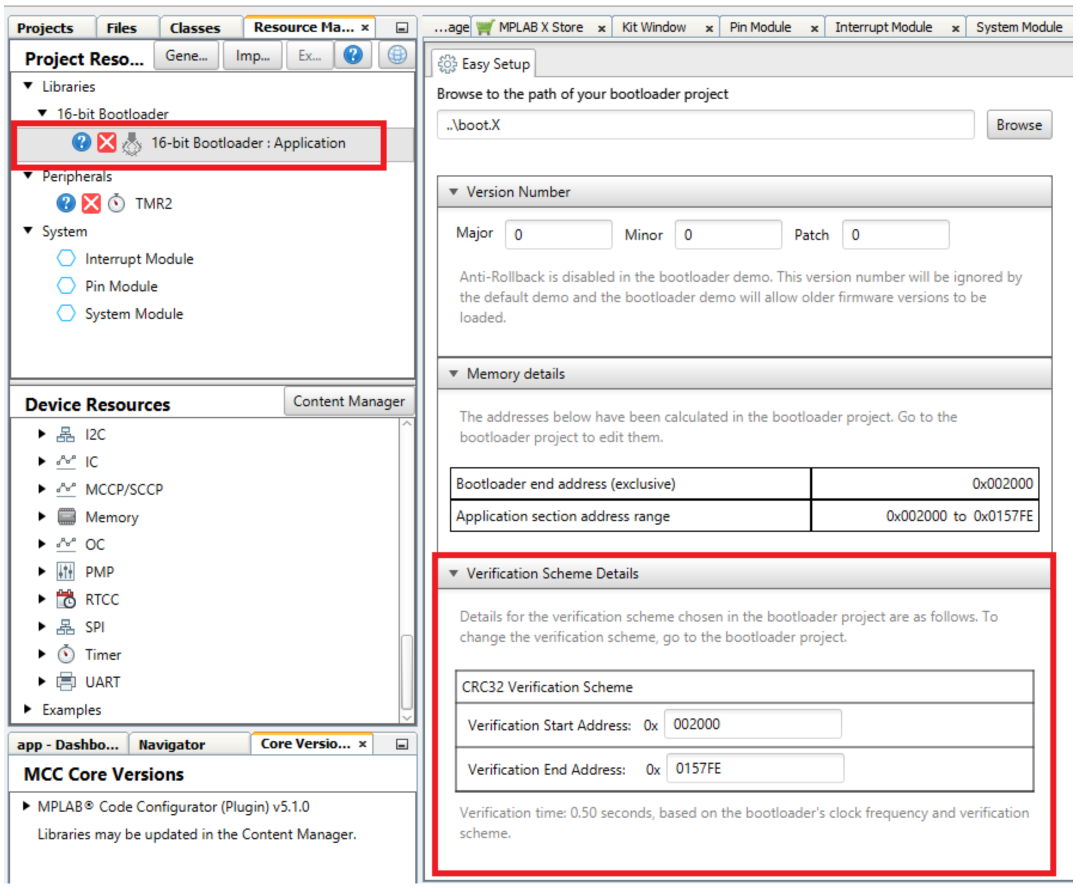
STEP 2: Update the application project to generate CRC field

In step 1 the bootloader was updated to verify the CRC value of the application before allowing it to run. Now the application needs to be updated to generate the CRC field that the bootloader will use to verify the code integrity. In this step we will update the application project to generate an application header that contains the CRC value the bootloader should compare against.

Step 2a: Open the “app” project from Lab 4

Step 2b: Open MCC

Step 2c: Go to Project Resources and open the Bootloader:Application library shown below. Notice that the verification details have been updated to CRC32 and the address range has been updated.



Browse to the path of your bootloader project

..\boot.X

Version Number

Major 0 Minor 0 Patch 0

Anti-Rollback is disabled in the bootloader demo. This version number will be ignored by the default demo and the bootloader demo will allow older firmware versions to be loaded.

Memory details

The addresses below have been calculated in the bootloader project. Go to the bootloader project to edit them.

Bootloader end address (exclusive)	0x002000
Application section address range	0x002000 to 0x0157FE

Verification Scheme Details

Details for the verification scheme chosen in the bootloader project are as follows. To change the verification scheme, go to the bootloader project.

CRC32 Verification Scheme

Verification Start Address: 0x 002000

Verification End Address: 0x 0157FE

Verification time: 0.50 seconds, based on the bootloader's clock frequency and verification scheme.

Step 2d: Go to Project Resources and press Generate button to update the software. This will regenerate the new application code required to generate the required CRC32 value in the application header.

Step 2e: Once the application project has been regenerated, do a clean build of the application project.

NOTE: Do not program the application project. Only build the project. If you program the application project, the bootloader project will need to be reprogrammed into the board before continuing.

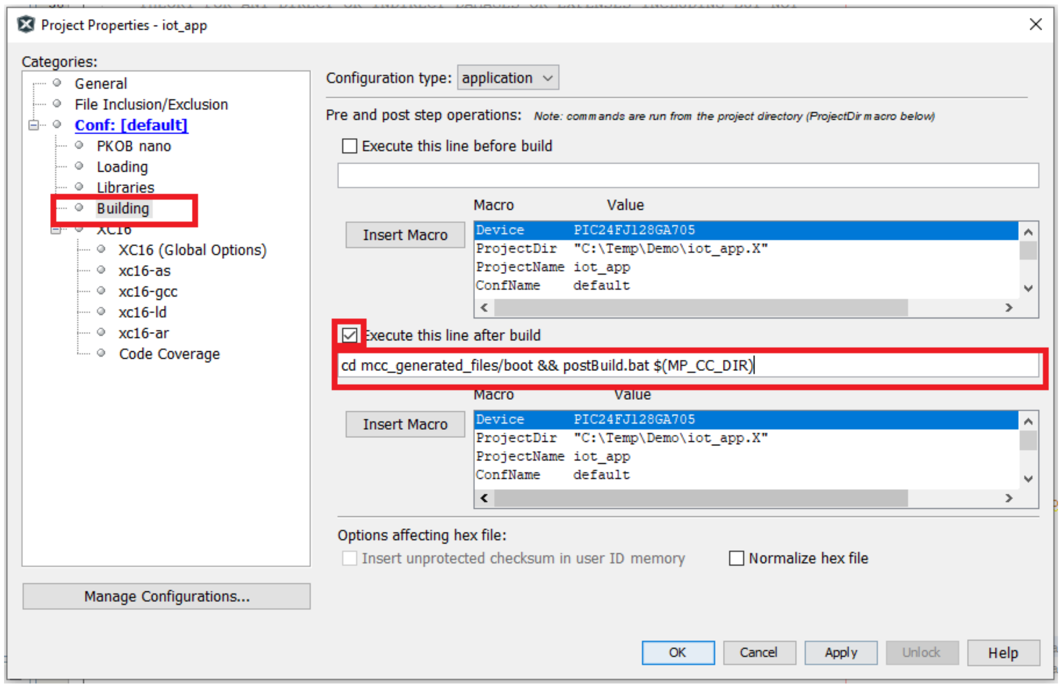
Step 2f: This should generate a compile error which was caused by an #error message in the file certificate_crc32.S.

```
#error "A script to append verification checksum has been added to your
project. You must add a call to it as a post-build step in your project
properties. Delete this warning after you have done so."
/*
* Steps to append verification checksum to the project hex file:
* 1. Right click on your project, and click 'Properties'.
* 2. Select the 'Building' left navigation node.
* 3. Check the box next to 'Execute this line after the build'.
* 4. In the text field below,
*      add "cd mcc_generated_files/boot && postBuild.bat $(MP_CC_DIR)"
(without quotes) if you are on a Windows machine,
*      or "cd mcc_generated_files/boot && ./postBuild.sh $(MP_CC_DIR)"
(without quotes) if you are on a Linux/Unix/Mac machine.
* 5. Delete or comment out the #error message above
*/
```

Step 2g: Click on the error message.

This will take you to the file and #error line causing the error. Read the comments surrounding the error message. These comments are telling us to add a post build step to the build process. This post build step will take the hex file that was generated, compute the CRC32 of the application area and then insert the computed CRC32 value back into the hex file at the predefined address of the application header. This "expected" checksum along with the start and end addresses is stored in the header and will be used by the bootloader as the expected CRC32 of the application image in flash.

Step 2h: Following #1 & #2 of the instructions above will open the Project Properties dialog shown below where we need to select the "Execute this line after build" checkbox and fill in the command. The command line can be copied from the C file and pasted here



Configuration type: application

Pre and post step operations: *Note: commands are run from the project directory (ProjectDir macro below)*

☐ Execute this line before build

```
Device       PIC24FJ128GA705
ProjectDir   "C:\Temp\Demo\iot_app.X"
ProjectName  iot_app
ConfName     default
```

☑ Execute this line after build

cd mcc_generated_files/boot && postBuild.bat $(MP_CC_DIR)

```
Device       PIC24FJ128GA705
ProjectDir   "C:\Temp\Demo\iot_app.X"
ProjectName  iot_app
ConfName     default
```

Options affecting hex file:

☐ Insert unprotected checksum in user ID memory ☐ Normalize hex file

Step 2i: Once done, press OK to close this dialog box. Then go back the file, certificate_crc32.S, and comment out the line with the #error on it.

Step 2j: Now clean and rebuild the project again. This time it should generate without errors and you should see the below message in the Output Window towards the end of the build messages. This message confirms that we are running the post build script.

```
"--------------------------------------"
"User defined post-build step: [cd mcc_generated_files/boot && postBuild.bat "C:\Program Files\Microchip\xc16\v1.60\bin"]"
"--------------------------------------"
```

NOTE: Do not program the application project. Only build the project. If you program the application project, the bootloader project will need to be reprogrammed into the board before continuing.

Step 2k: Make sure the updated bootloader with the new verification method is running on the board. If it is not, switch back the bootloader project created in the first part of this lab and program the board with the updated bootloader. When its running we should see the both the red and yellow LEDs blinking.

Step 2l: Open the UBHA program.

Step 2m: Click the “Read Device Settings” button. Remember, the bootloader size has changed and thus so has the valid application range. Alternatively, you can manually enter the updated application addresses.

Step 2n: Selecting the “Enable Self Verification after Program” request that the bootloader run the CRC32 verification method after the firmware programming is complete and verify it matches what is in the application header.

Step 2o: Select the new application file using the “File->Open” option.

Step 2p: Press the Program button. If this fails, common failure modes are forgetting to add the post build step (step 2h above) to the application build process or not updating the bootloader.

If it completes successfully, the bootloader has verified the calculated CRC32 value against the value stored in the application header and they match. At this point, the application code should start running and the blue LED should be toggling.

NOTE: When the bootloader completes the verification process after programming, it will reset the device. This will re-enter the bootloader. The bootloader will re-verify the application image again before letting it run, as it will after any reset. This will cause a delay before the application can start running.

Results:

The application code is now verified by a CRC32 check before it is launched after every reset.

This lab showed how to change the verification method of the application. With the exception of ECDSA which requires additional external hardware, the process can be used for most of the other verification methods.

Summary:

Verifying the applications integrity before starting the application is often a requirement. This lab has showed how that can be achieved using a CRC32 check. The CRC32 value is automatically calculated and injected back into the application hex file. The bootloader verifies the application CRC value on power up before allowing it to execute.

Lab 7 – Combine the Bootloader and Application Hex Files for Production

Purpose:

Combine the Bootloader and Application hex files into a single hex file that can be given used for production.

Overview:

To speed up production, the factory will often want just a single hex file that contains the production bootloader and a working application. This lab will show how to update the application project from the previous lab by adding a step to generate the combined hex file.

Procedure:

STEP 1: Generating a Combined Hex File

Step 1a: Switch to the app project from the previous lab 6.

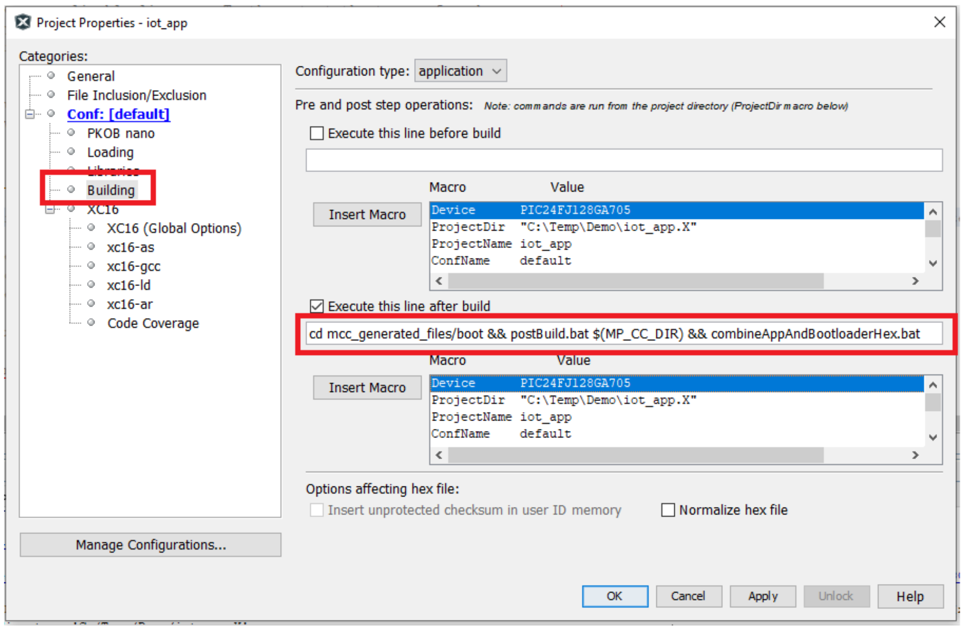
~~**Step 1b:** Open the "boot" MPLAB X project created in lab 1.~~

Step 1c: In the projects window, right click on the app project, and click 'Properties'.

Step 1d: Select the 'Building' in the 'Categories' pane. In the 'Building' options window, append the command:

```
&& combineAppAndBootloaderHex.bat
```

to the end of the current command in the post build command line options shown below:


Project Properties - iot_app

Categories:
- General
- File Inclusion/Exclusion
- Conf: [default]
  - PKOB nano
  - Loading
  - Libraries
  - Building
  - XC16
    - XC16 (Global Options)
    - xc16-as
    - xc16-gcc
    - xc16-ld
    - xc16-ar
    - Code Coverage

Configuration type: application

Pre and post step operations: *Note: commands are run from the project directory (ProjectDir macro below)*

☐ Execute this line before build

Macro	Value
Device	PIC24FJ128GA705
ProjectDir	"C:\Temp\Demo\iot_app.X"
ProjectName	iot_app
ConfName	default

☑ Execute this line after build

```
cd mcc_generated_files/boot && postBuild.bat $(MP_CC_DIR) && combineAppAndBootloaderHex.bat
```

Macro	Value
Device	PIC24FJ128GA705
ProjectDir	"C:\Temp\Demo\iot_app.X"
ProjectName	iot_app
ConfName	default

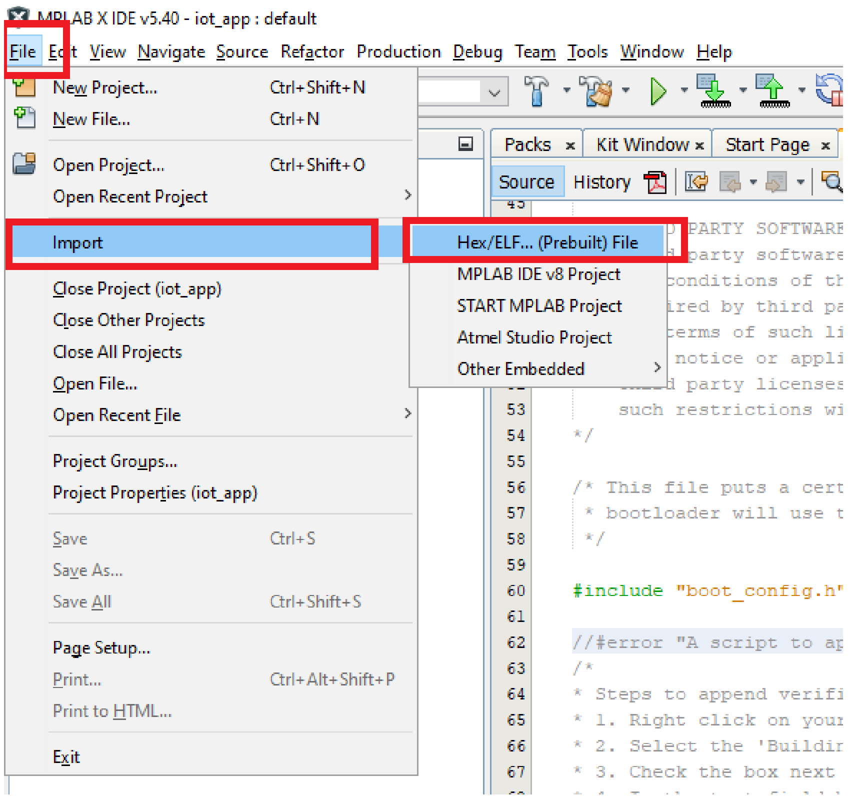
Options affecting hex file:
☐ Insert unprotected checksum in user ID memory ☐ Normalize hex file

Step 1e: Rebuild the project. When the build completes, a new combined hex file is created named `dist\default\production\combined.production.hex`

STEP 2: Programming the Combined Hex File

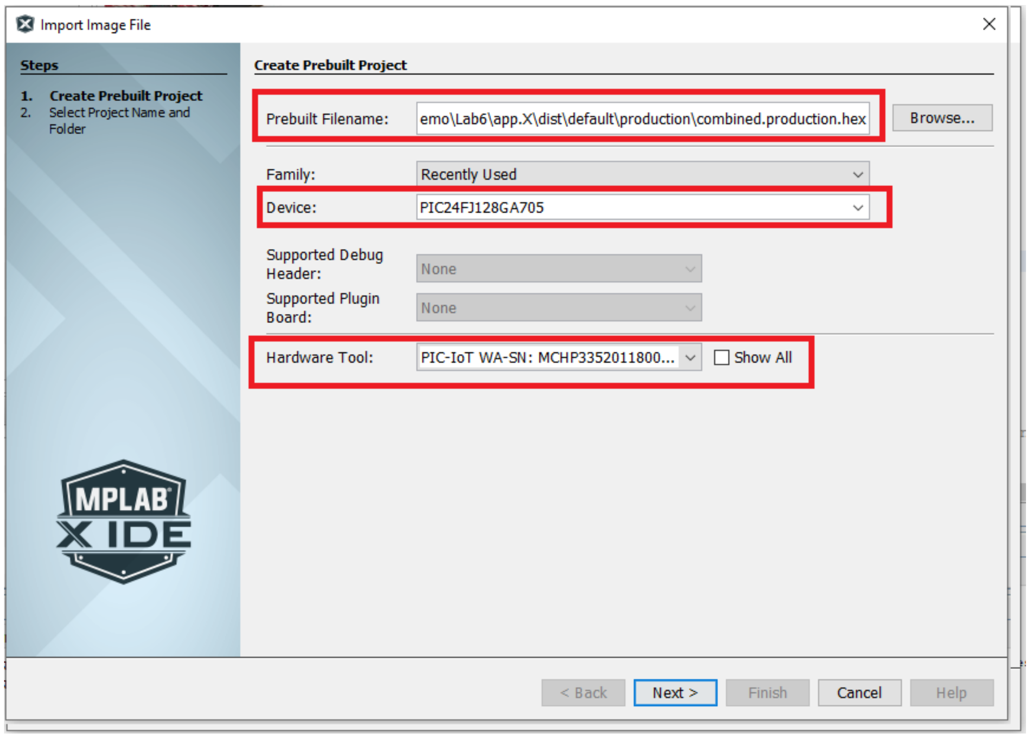
The next step is programming the combined hex file into the part. Normally, we would use the IPE tool which is part of the MPLAB® X install. However, at time of writing this lab, IPE does not support this board so we will use an alternative method using MPLAB X.

Step 2a: We need to first import the combined hex file produced above into MPLAB X. This is done in MPLAB X by going to "File->Import->Hex/Elf (Prebuilt) File" as shown below.



```
    PARTY SOFTWARE
    party software
    conditions of th
    ired by third pa
    terms of such li
     notice or appli
    d party licenses
    such restrictions wi
 */

/* This file puts a cert
 * bootloader will use t
 */

#include "boot_config.h"

//#error "A script to ap
/*
 * Steps to append verifi
 * 1. Right click on your
 * 2. Select the 'Buildir
 * 3. Check the box next
```

Step 2b: This will open the Import Image File Dialog shown below and we need to enter the location of the Prebuilt hex filename, the Device and the Hardware Tool to use.



Prebuilt Filename: emo\Lab6\app.X\dist\default\production\combined.production.hex

Family: Recently Used

Device: PIC24FJ128GA705

Supported Debug Header: None

Supported Plugin Board: None

Hardware Tool: PIC-IoT WA-SN: MCHP3352011800... ☐ Show All

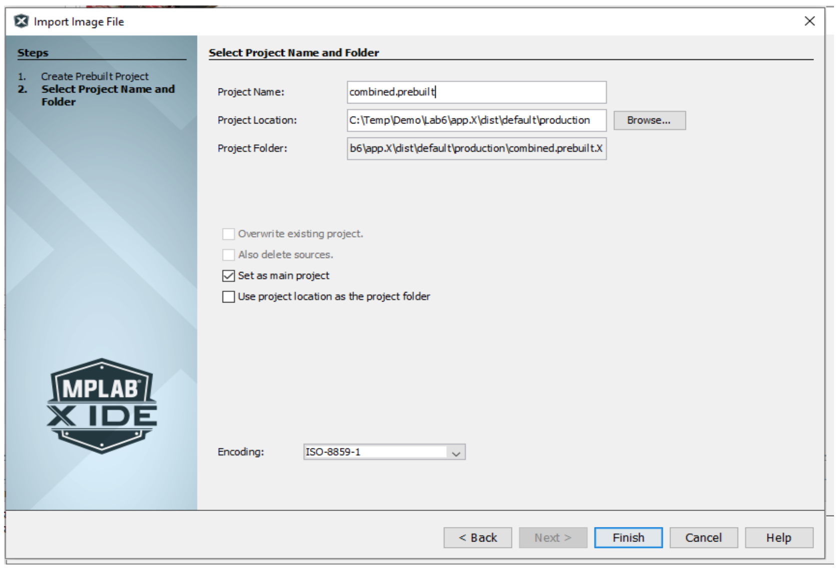
Step 2c: The combined hex file will be located in the application folder at:
`app.X\dist\default\production\combined.production.hex`

Step 2d: The device is PIC24FJ128GA705.

Step 2e: And the HW tool is the PIC-IoT WA board that we have been using.

Step 2f: Press “Next”.

Step 2g: This will display the Project Creation Dialog below showing us where this project is going to be built. For this lab I will leave the defaults and press Finish.


Import Image File
Steps
1. Create Prebuilt Project
2. Select Project Name and Folder
Select Project Name and Folder
Browse...
MPLAB X IDE
< Back
Next >
Finish
Cancel
Help

Project Name: combined.prebuilt

Project Location: C:\Temp\Demo\Lab6\app.X\dist\default\production

Project Folder: b6\app.X\dist\default\production\combined.prebuilt.X

☐ Overwrite existing project.

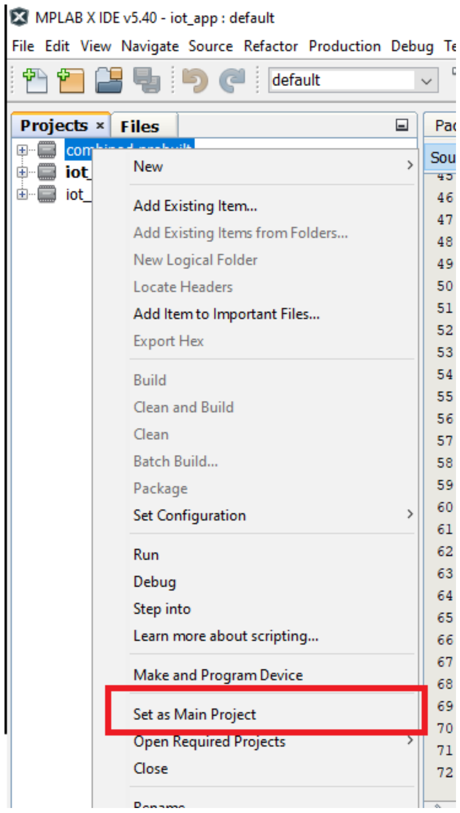
☐ Also delete sources.

☑ Set as main project

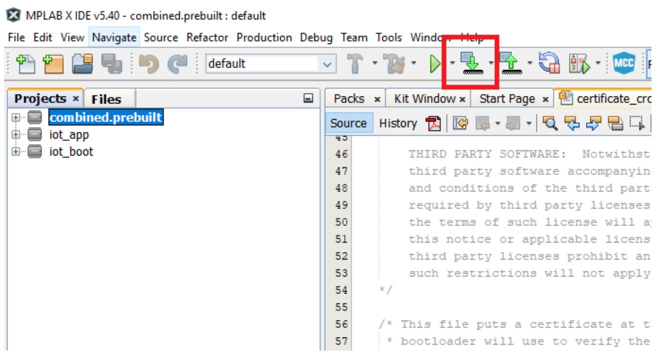
☐ Use project location as the project folder

Encoding: ISO-8859-1

Step 2h: This created a new project call combined.prebuilt as shown below. Switch to this project by right clicking on the combined.prebuilt project and selecting "Set as Main Project" as shown below.



Step 2i: We can now program the combined hex file into the part by pressing the Make and Program icon shown below. Pressing this will program the entire hex file into the part and then reset the part.


MPLAB X IDE v5.40 - combined.prebuilt : default
File Edit View Navigate Source Refactor Production Debug Team Tools Window Help
default
Projects × Files
combined.prebuilt
iot_app
iot_boot
Packs × Kit Window × Start Page × certificate_crc
Source History

```
   THIRD PARTY SOFTWARE:  Notwithst
   third party software accompanyin
   and conditions of the third part
   required by third party licenses
   the terms of such license will a
   this notice or applicable licens
   third party licenses prohibit an
   such restrictions will not apply
*/

/* This file puts a certificate at t
 * bootloader will use to verify the
```

Step 2j: After programming, the part will:

a. Reset
b. Run the bootloader
c. The bootloader will verify the application
d. Once the application is verified the bootloader will then run the application.

Step 2k: The user can force entry into the bootloader update mode by unplugging the USB, then hold down the SW0 button and plug the board back into USB. This should cause the application update mode to be entered as indicated by the LEDs.

Results:

A hex file has produced that has both the bootloader and the application code in it. This file can be easily used for production so that only one programming sequence is required.

We learned how this combined hex could be programmed into a device using the MPLAB X IDE. The MPLABX IPE can also be used to program the hex file directly. For information about how to use the IPE to program a hex file, please refer to the IPE user's manual.

Summary:

For production it is often desired to have a single hex file that has both the application and the bootloader in it so that only one programming sequence is required and there isn't a need to run the bootloader at all. This lab showed how to combine the bootloader and application hex files into a single hex file for production using the scripts provided by the MCC bootloader solution.

Trademarks