Microchip University

Lab Manual for 16-Bit ECDSA Secure Boot Bootloaders Using MCC

Table of Contents

Introduction:

The labs in this document are a continuation of the previous labs but are focused on creating a Secure Boot bootloader using the ECDSA encryption. They will use the same board and MCC modules as in the previous labs.

Required Development Tools:

- One of the two boards below
    a. PIC-IoT WA Development Board (EV54Y39A)
    b. PIC-IoT WG Development Board (AC164164)
- MPLAB® X version 6.0 or later
- MPLAB Code Configurator (MCC) version 5.01 or later
- *16-Bit Bootloader MCC module version 1.19.1 or later.

    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 8 – ECDSA Secure Boot Bootloader

Purpose:

Create a bootloader using ECDSA secure boot.

Overview:

In the previous labs we have used verification methods to verify the integrity of the application. In this lab we will also verify the authenticity of the application using ECDSA verification. This will verify that the application was signed by with the private key that was used to generate the public key that the bootloader uses. In addition, we will also enable Code Protect which configure CodeGuard™ on the PIC24FJ128GA705 allowing the bootloader to be write protected. CodeGuard™ also enables the Alternate Interrupt Vector Table(AIVT) which can be used by the application while the bootloader used the primary Interrupt Vector Table(IVT).

This lab is written to work on one of the following boards:

1. PIC-IoT WA Development Board, part number EV54Y39A
2. PIC-IoT WG Development Board, part number AC164164

Both of the above boards have a ATECC608A on board for security and signing. A user can also use external ATECC608A located on the Secure 4 Click (https://www.mikroe.com/secure-4-click). The only difference between these two options is the configuring the I2C. Currently, the bootloader16 module is setup to default to the external option. If the user wants to use the on board ATECC608A, a couple extra steps are required and are documented when needed.

For this lab we are going to start by creating a new bootloader project called GA705_boot. Just like in Lab 4, it will

1. blink the Yellow LED (RC3) via a SW loop
2. Use Timer 1 to blink the Red LED on RB4
3. Use SW0 on RA7 to detect when we want to enter the application update mode at reset.

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”.

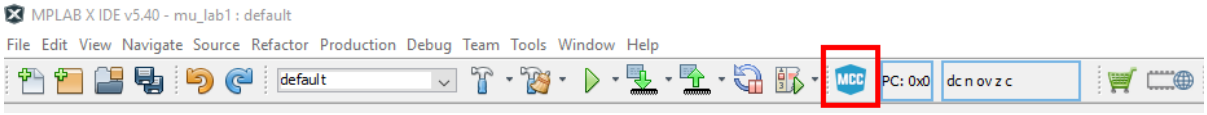
Step 1d: Select the PIC24FJ128GA705 device in the device menu and click “Next”

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

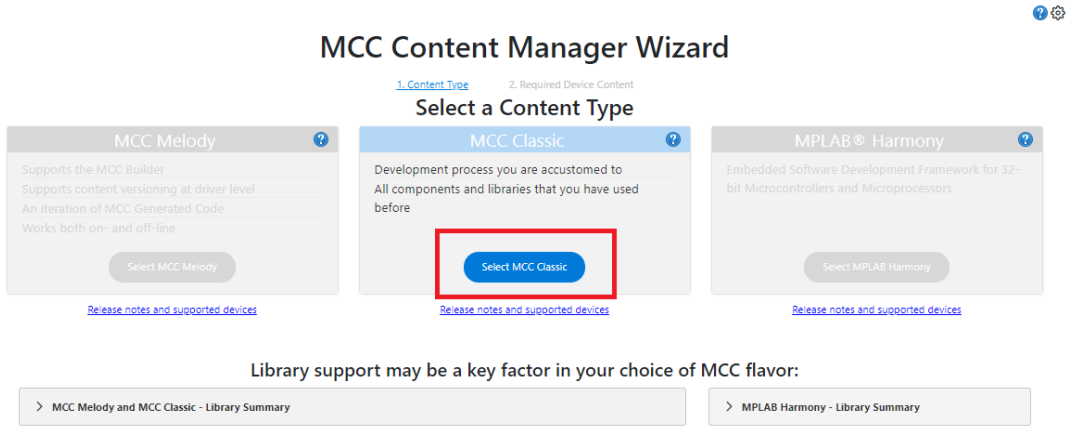
Step 1f: Type in “iot_boot” for the project name, select a folder for the project, and click “Finish”

STEP 2: Configure MCC for the Board

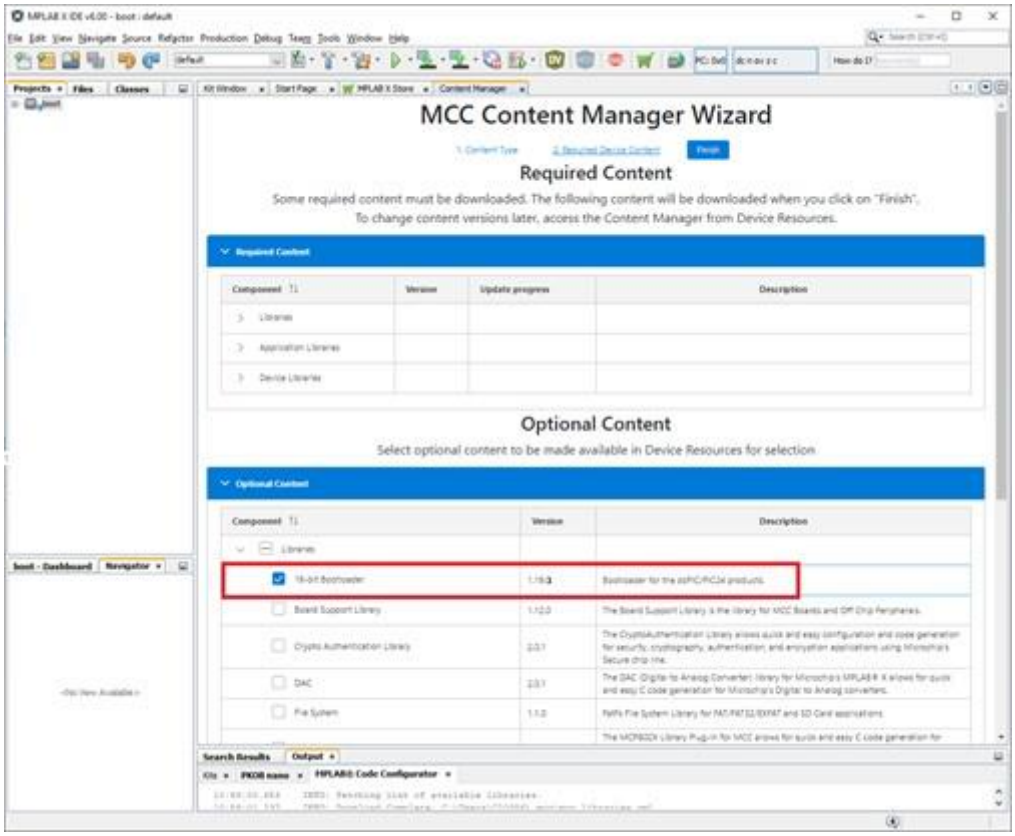
Step 2a: Open MCC by clicking the MCC button in MPLAB X



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

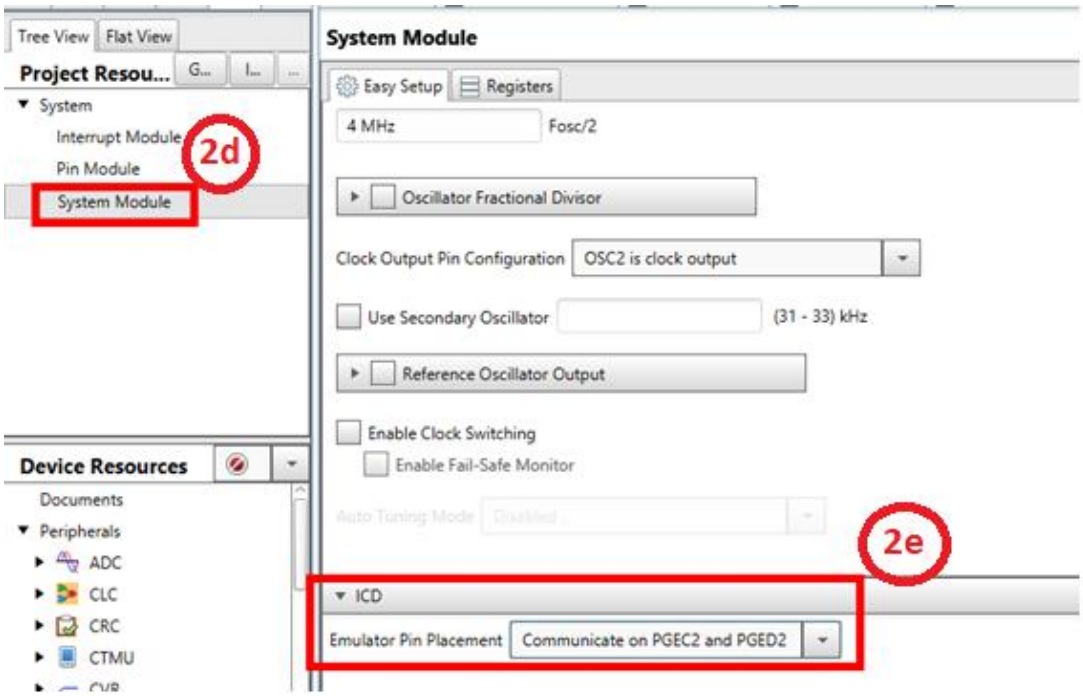


Step 2c: If this is a new installation, under optional content, Libraries, select the 16-bit Bootloader and press Finish.



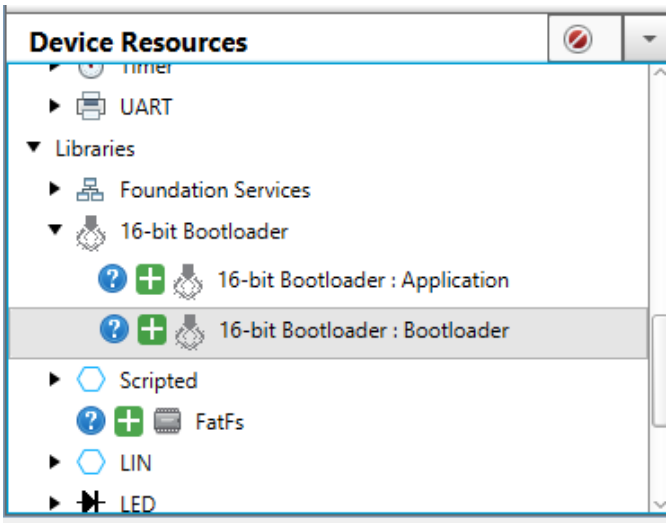
Step 2d: Select the “System Module” in the Project Resourced pane

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



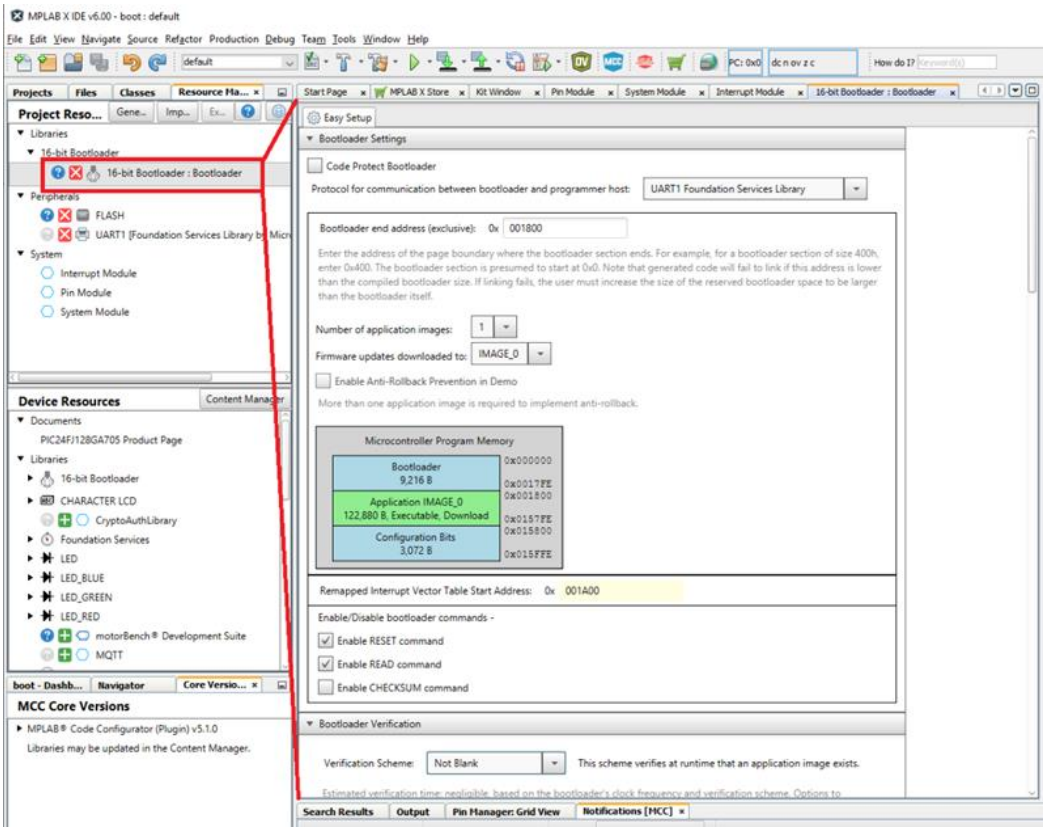
STEP 3: Add the bootloader module to the project

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.

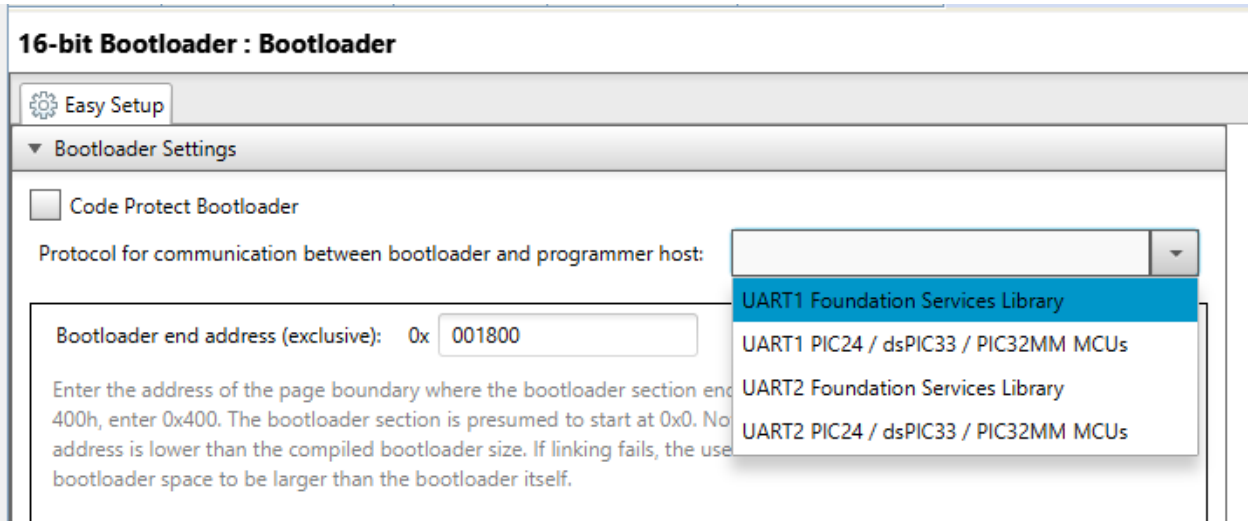


STEP 4: Configure the bootloader module

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



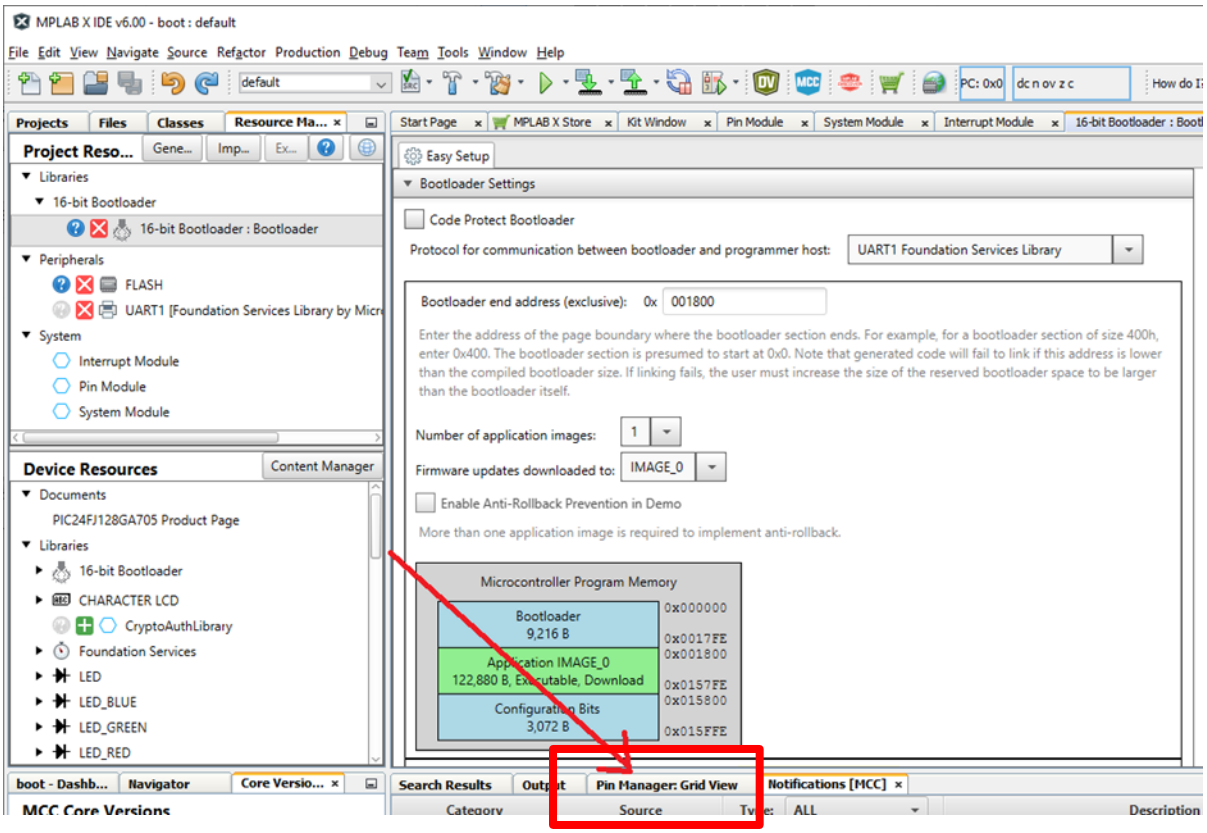
Step 4b: In the Protocol selection drop-down, select the “UART1 Foundation Services Library” option.



STEP 5: Configure the UART

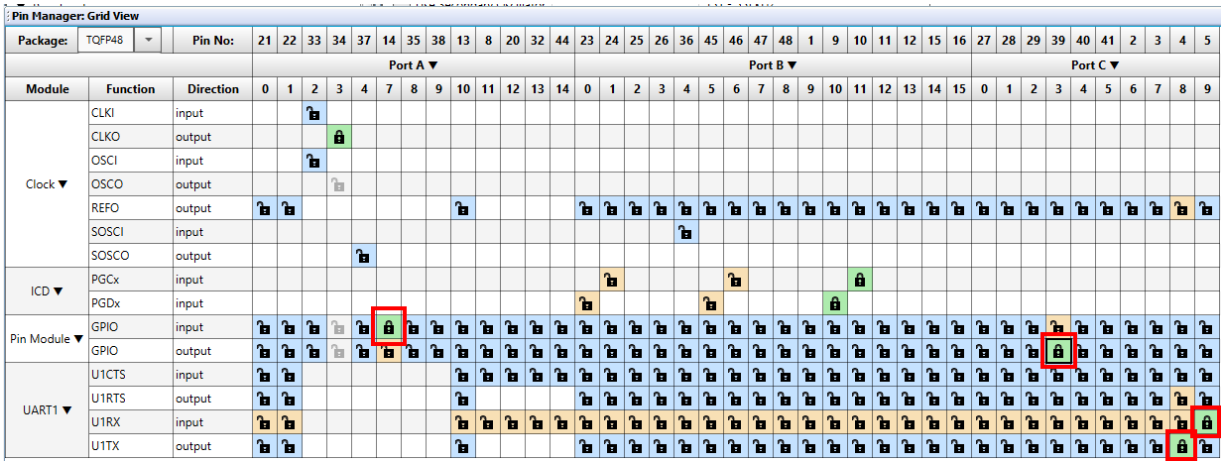
Step 5a: Verify the UART module settings. The UART module should have been automatically added to the “Project Resources” pane. Select the UART module in this pane. It should be configured to 9600:8:N:1 by default. Leave these settings. If the settings aren’t for 9600:8:N:1, then change them.

Step 5b: Open the pin manager grid view



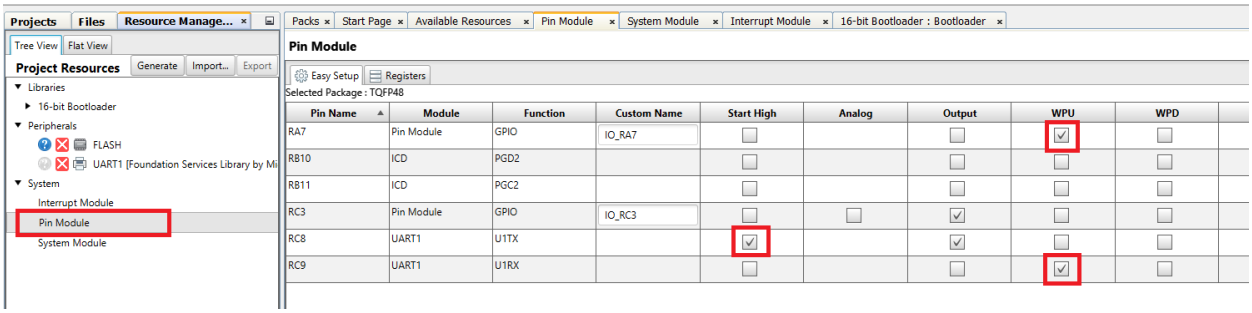
Step 5c: Configure the following pins:

a. RC3 GPIO as an output. This is connected to the yellow “Data LED”.
b. RB4 GPIO as an output. This is connected to the red “Data LED”.
c. RA7 as an input. This is the SW0 push button pin.
d. UART TX pin is RC8 on this board.
e. UART RX pin is RC9 on this board.



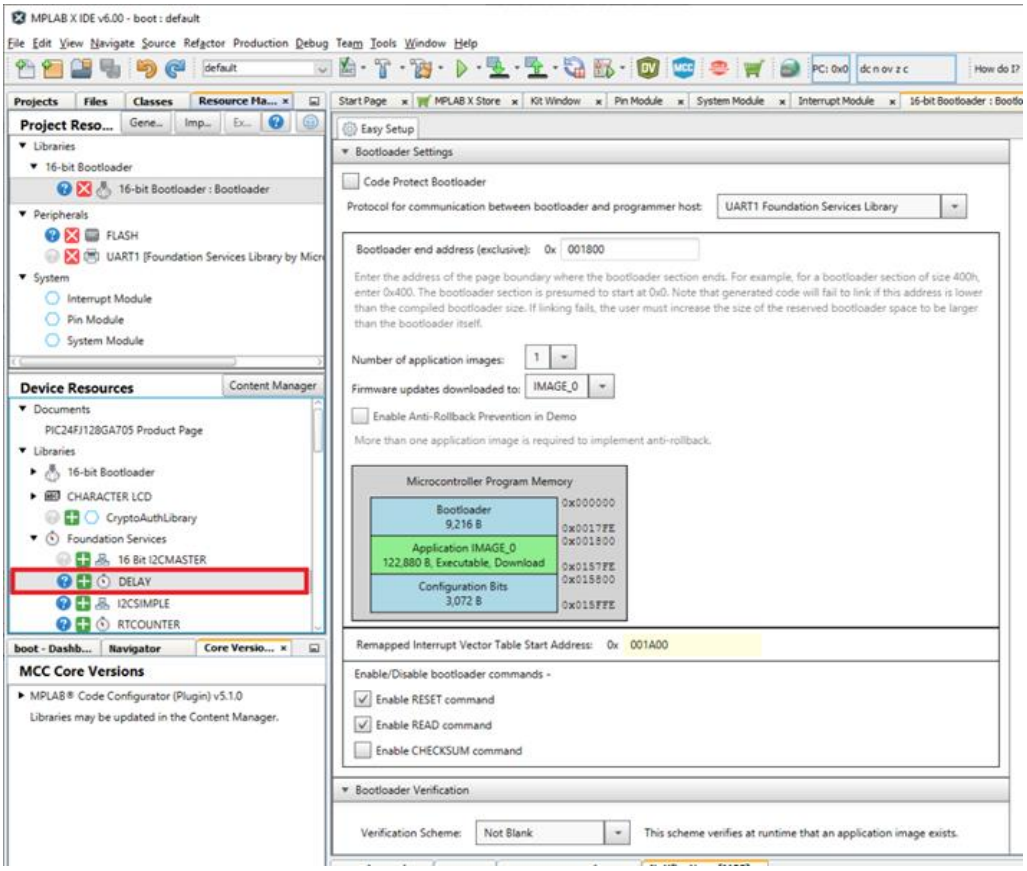
Step 5d: Open the pin module and make the following configuration changes:

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)



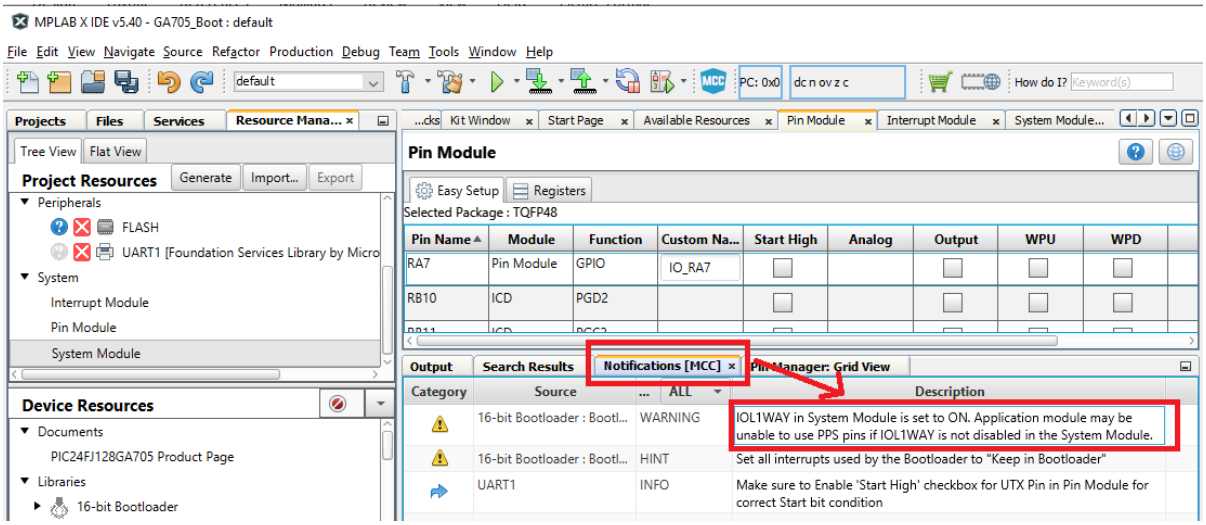
STEP 6: Add delay module to project

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:



STEP 7: Resolve project warnings

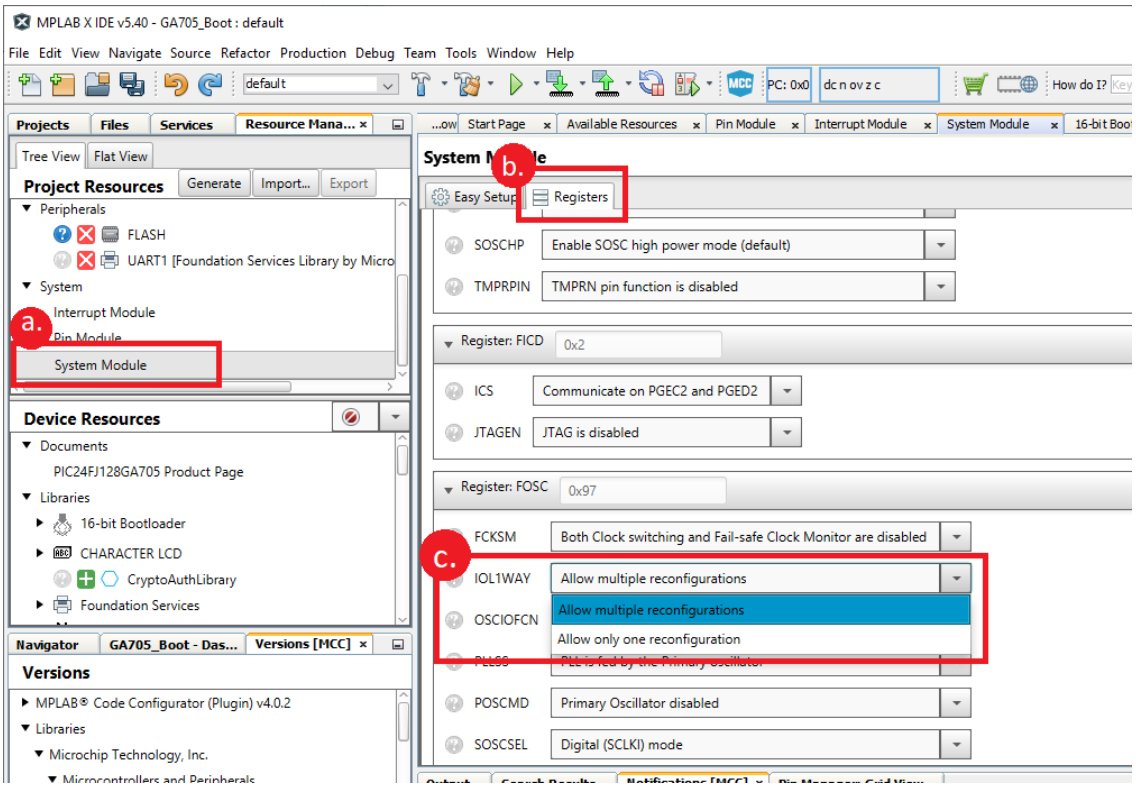
Note the warning for “IOL1WAY in System Module is set”. This message is saying that once the boot loader configures the PPS pins, the application will no longer be able to modify the PPS settings. For some customers, this will be okay. The boot loader may configure all of the pin configuration for the 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”.



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

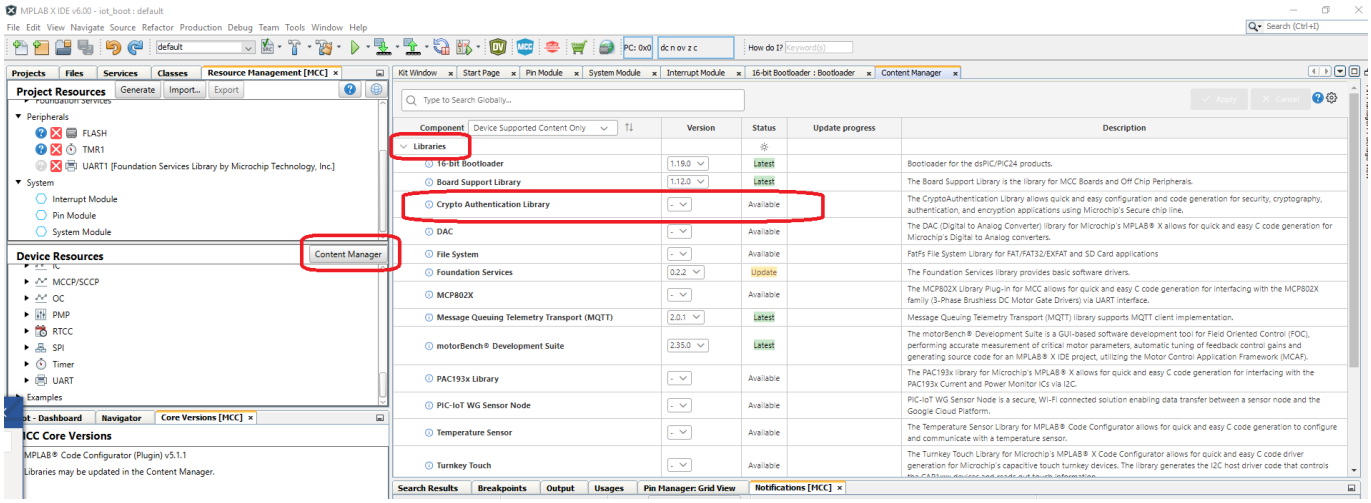
Step 7b: Select the Registers view tab.

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

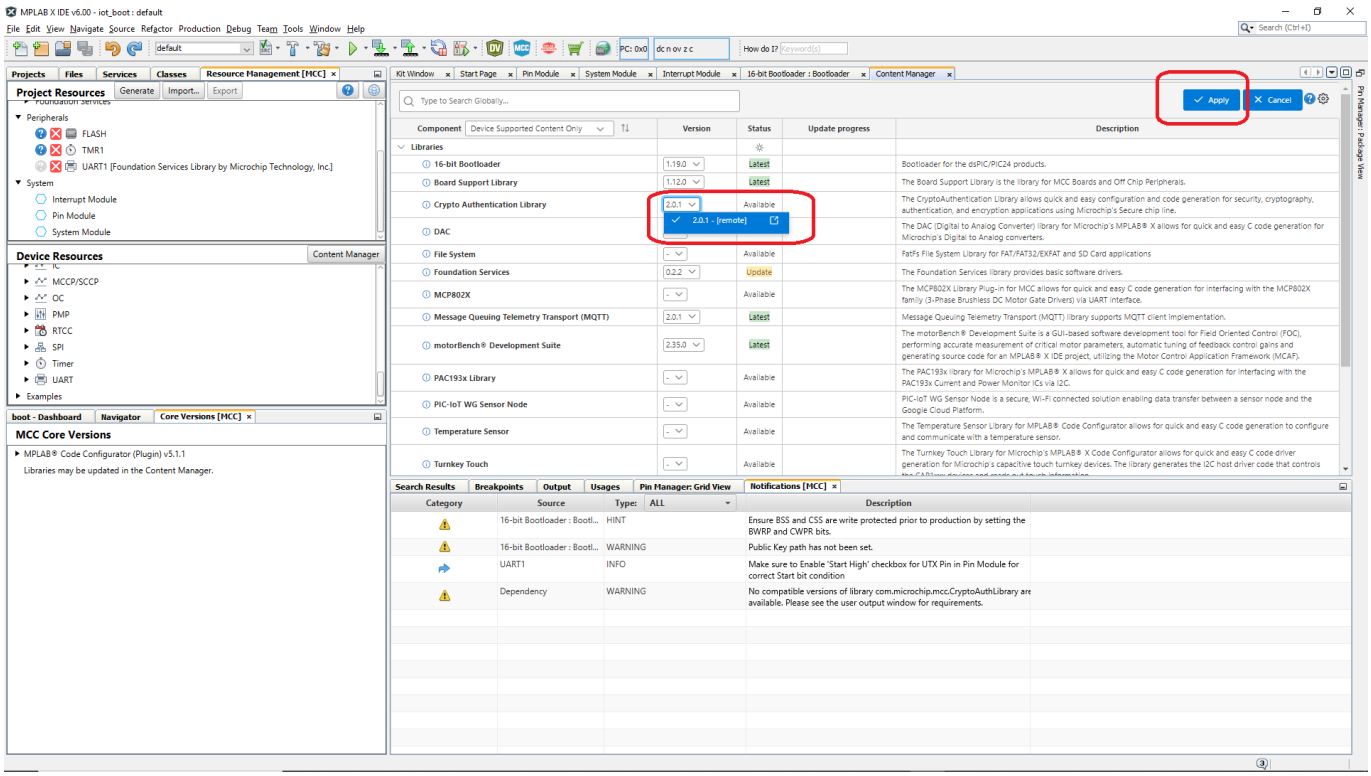


STEP 8: Make sure Crypto Library is installed

Before we go further, we need to make sure that the Cypto auth library is installed. This can be done by going to “Device Resources”, and pressing “Content Manager” as show below. Then select libraries and verify the Crypto Authentication Library is installed.



If it is not installed, press the download box and get the latest driver and then press Apply as shown below.

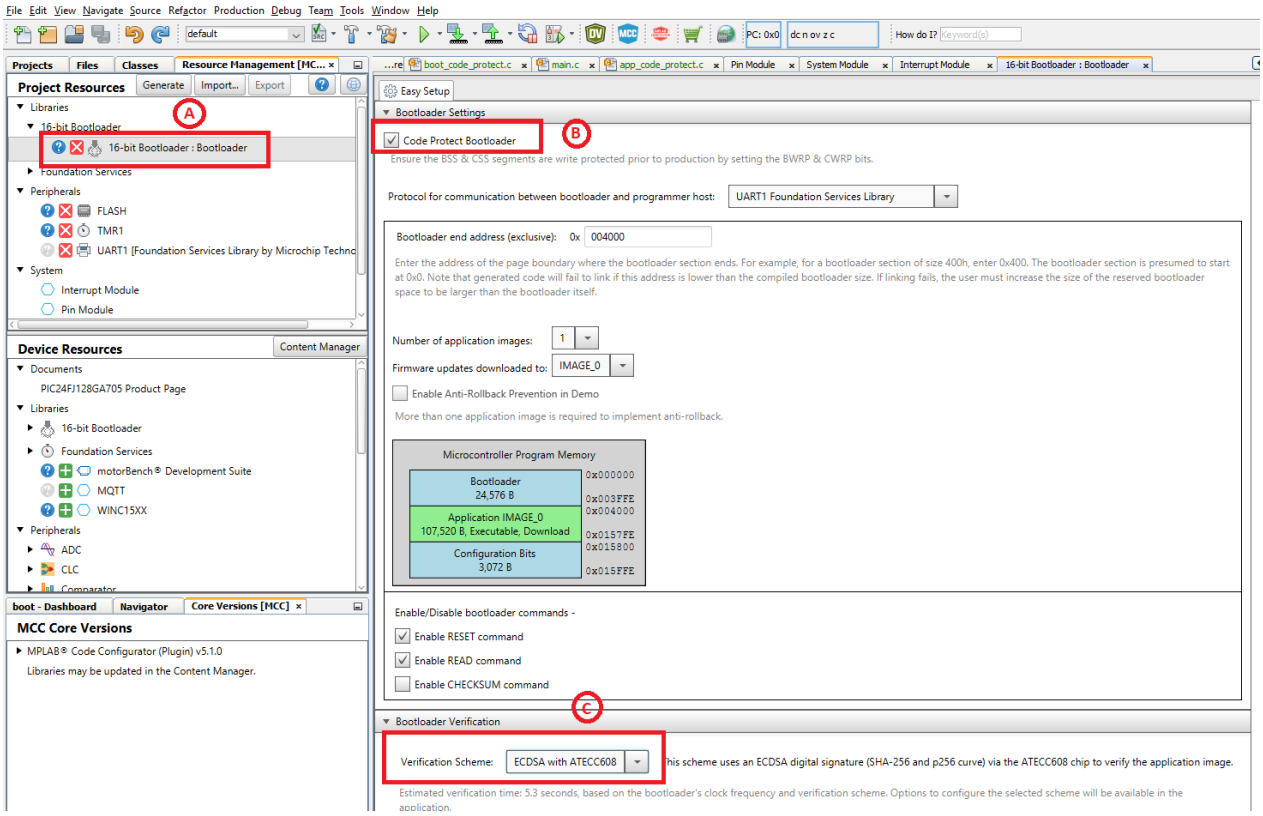


STEP 9: Enable ECDSA Verification

Step 9a: Select the “16-bit Bootloader : Bootloader” module in the “Project Resources” pane

Step 9b: Check the “Code Protect Bootloader” box.

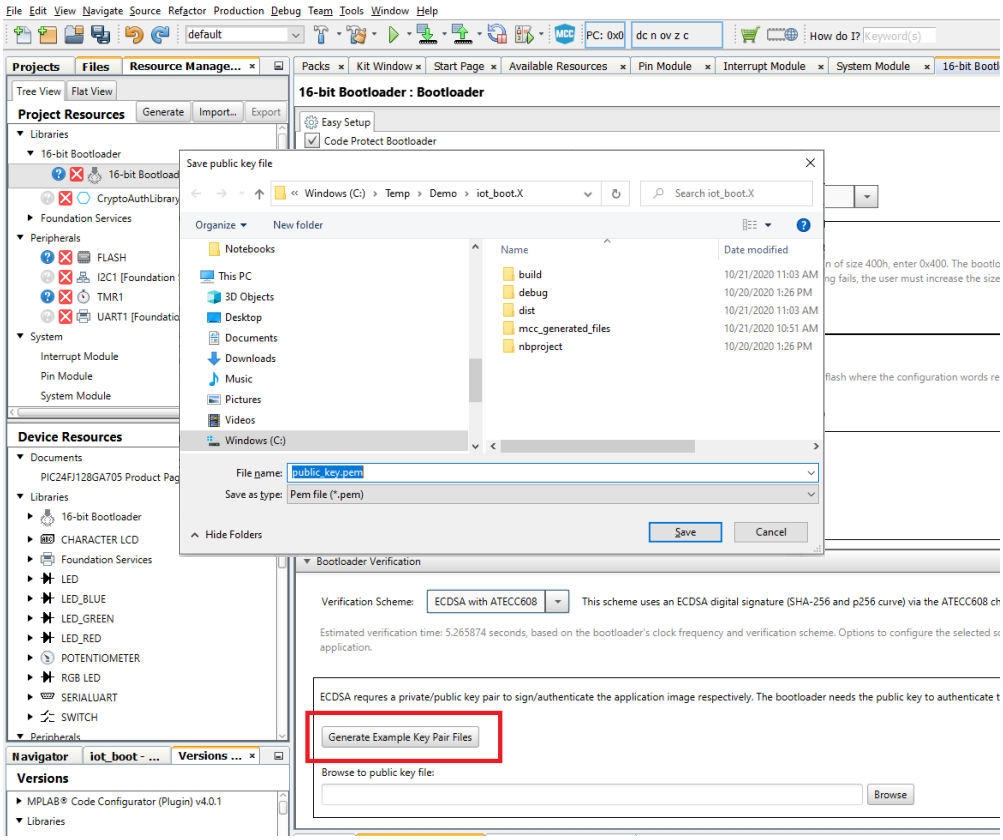
Step 9c: Change the verification method to “ECDSA with ATECC608”



STEP 10: Configure ECDSA Verification

Step 10a: Once the verification scheme is changed to ECDSA, the bootloader configuration dialog is updated with new features. This includes adding the CyptoAuthLibrary and I2C1 to the Project Resources and adding a place to add or generate keys shown below. Also, notice that the size of the bootloader has increased as well. This is because it takes a lot more code for this type of verification and room needs to set aside for it.

The next step is to generate both public and private keys for the ECDSA algorithm by pressing the Generate Example Key Pair button highlighted below.



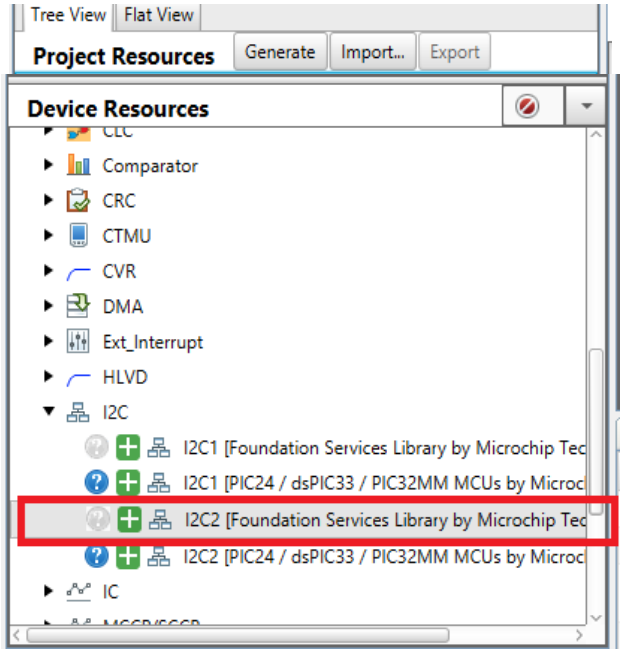
Step 10b: After pressing the “Generate Example Key Pair Files” button, you will be prompted to select where the save the public and private keys. For this demo, we will just use the default location for both keys which is the root of our bootloader project as shown below.

STEP 11: Configure I2C for On-Board ECC608A

NOTE: This step is not required if a secure Click Board is used instead of the on-board ECC608. This step is only required when using the on-board ECC608.

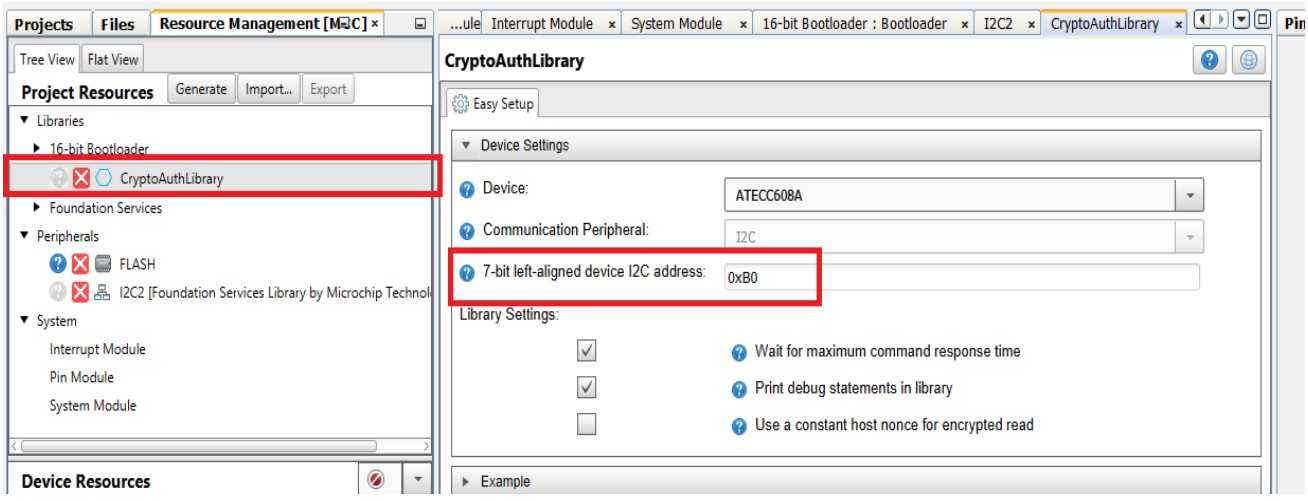
Step 11a: The onboard ECC608A part is connected to a different I2C port than the Click version so we first need to change which I2C port is being used. This is done by first removing the I2C1 peripheral in the project resources by clicking on the red "X" shown below:

Step 11b: Add the I2C2 module that the ECC608A is connected to by going to the Device Resources, expanding the I2C group and selecting the I2C2 [Foundation Services Library]

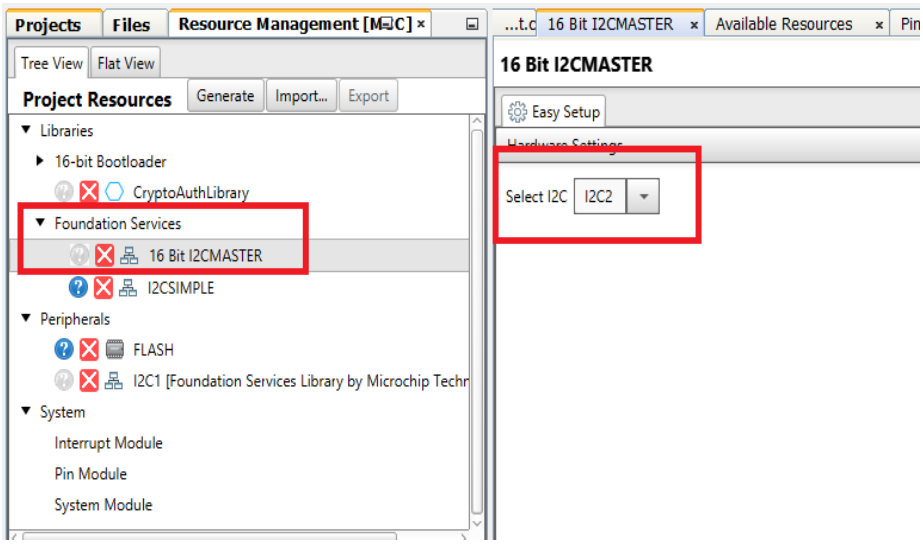


shown below by pressing the "+" sign. Then remove I2C1 from the list of peripherals in you project

Step 11c: Then change the I2C address used to communicate with the ECC608A device since the onboard ECC608A does not use the factory default address. To do this, go back to Project Resources, Select the CyptoAuthLibrary. This will open the CryptoAuthLibrary configure dialog below and we need to change the I2C address from 0xC0 to 0xB0.



Step 11d: Finally, we need to tell the driver to use the new I2C2 controller. This is done by going back to Project Resources, Selecting the Foundation Services and then selecting the 16 Bit I2CMASTER option shown below and then change the “16 Bit I2CMaster” from I2C1 □ I2C2 using the Select I2C dropdown box also outlined below.



STEP 12: Resolve Build and Link Errors

Step 12a: Press Generate which should generate the code with no issues.

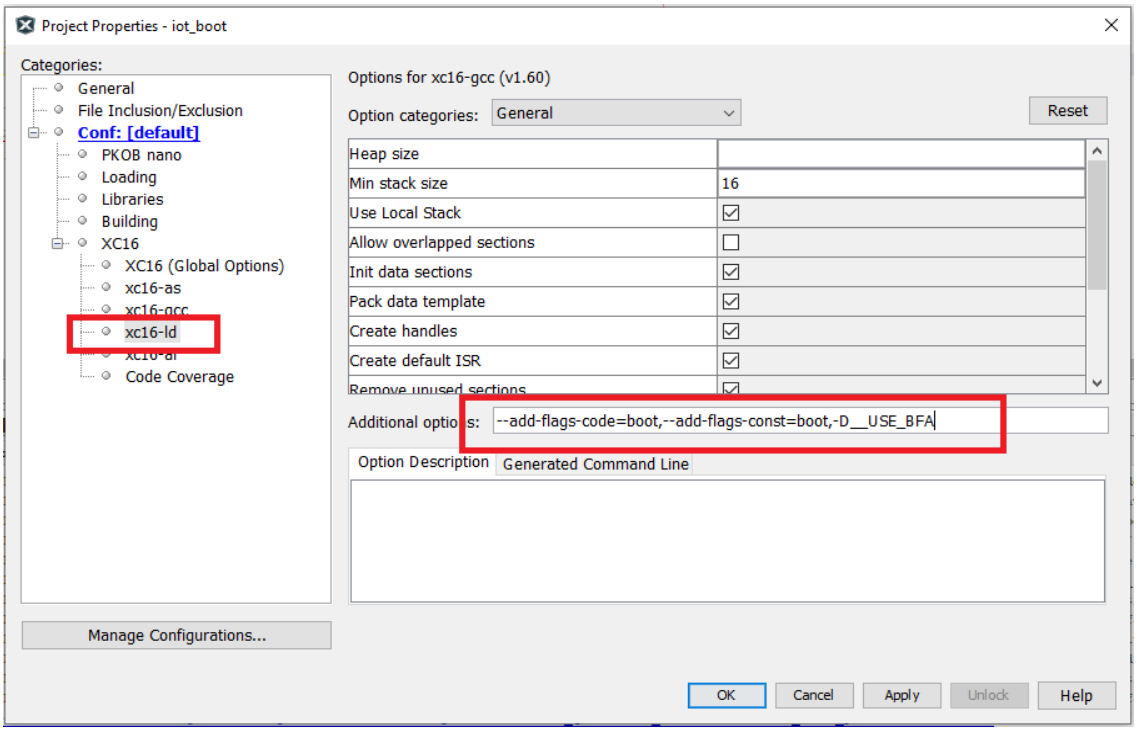
Step 12b: After the code is generated, we can compile the code by pressing the Clean and Build icon below.

Step 12c: 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
*/
```



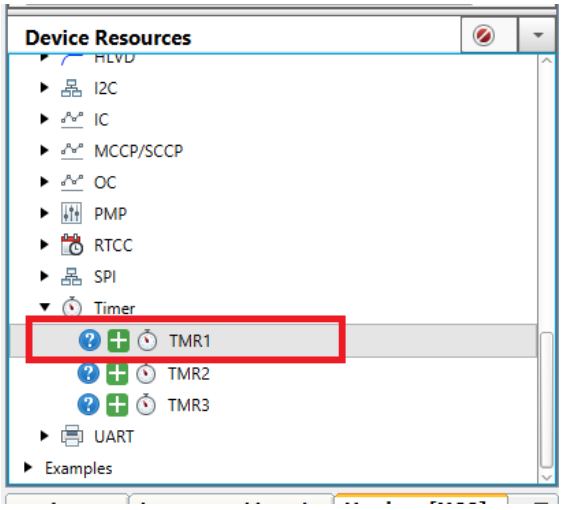
Step 12d: Once these changes are made, please comment out the line with the #error in the file `boot_code_protect.c` and recompile the code.

STEP 11: Add Demo Functionality – Blink LEDs

Step 13a: After the code compiles cleanly we will then add the code to blink the yellow and red LEDs as was done in Lab 4. First, add the code below required to toggle the yellow LED to the file `boot_demo.c`

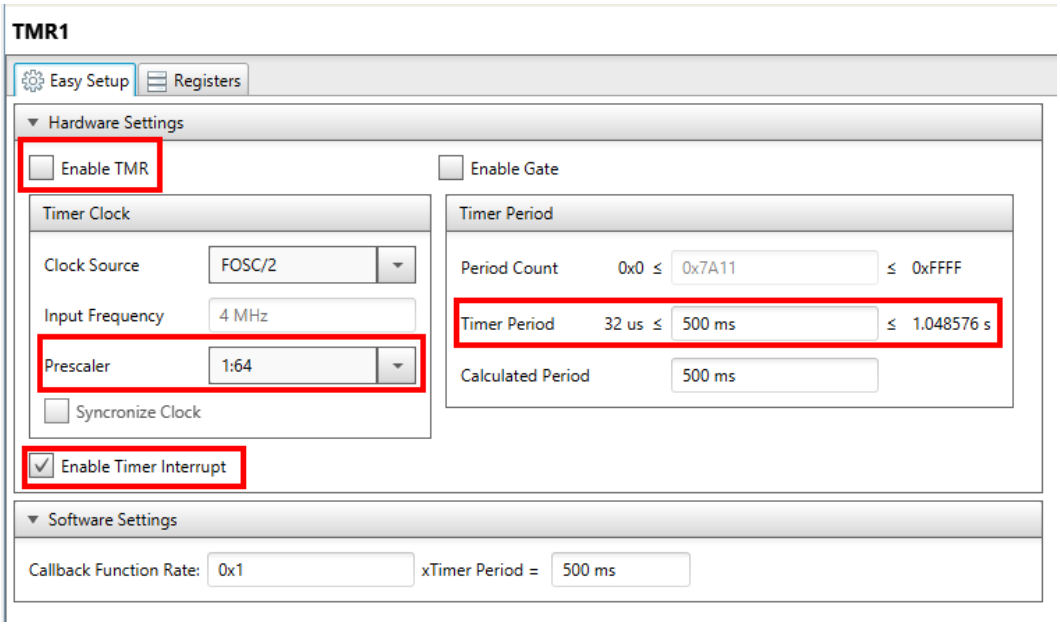
```
#include "../pin_manager.h"
#include "../delay.h"

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

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

Step 13b: From the Device Resources pane, add TMR1 to the project by clicking the "+" next to it.



Step 13c: 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



Step 13d: Generate the updated code by pressing the “Generate” button at the top of the Project Resources pane.

Step 13e: Add the `TMR1_CallBack()` code to `main.c` to toggle RB4 pin as shown below. Make sure to add the line `#include "mcc_generated_files/pin_manager.h"`.

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

    IO_RB4_Toggle();
}
```

Step 13f: 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 13g: Add the call, `BlinkBootloaderLED()` , to blink the yellow LED and 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 13h: Finally, add the code to check to see if SW0 is pressed to the `EnterBootloadMode()` function in the `boot_demo.c` file as shown below:

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

Step 13i: Now, download and run the code. You should have the Yellow and Red LEDs blinking.

Step 13j: We can also do a quick test to verify the bootloader kernel is running by using UBHA to read the Device Settings like we did in Lab 3. We would just want to press the Read Device Settings and verify we are able to communicate with the device.

Results:

We now have a bootloader that will do an ECDSA verification of the application before allowing it to run. This will make sure that application is unmodified and from an authentic provider.

Summary:

In this lab we created a bootloader that will ECDSA verification of an application image. This is only the first half of the solution. In the next lab we will show how to create an application that is signed so that the bootloader will accept the application image.

Lab 9 – Creating an ECDSA Signed Application

Purpose:

Create a simple example application which uses ECDSA verification that can serve as an example “end product” and then enable it to be used with the bootloader generated in lab 8.

Overview:

This lab closely matches Lab 2. For this lab we will enabling ECDA verification as well. 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 8 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”.

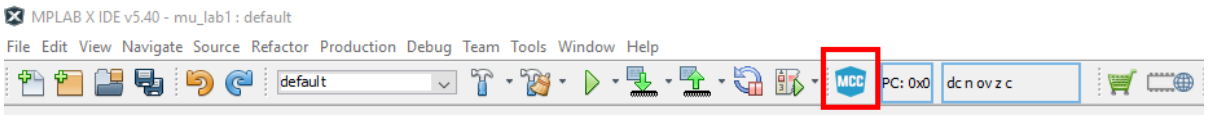
Step 1d: Select the PIC24FJ128GA705 device in the device menu and click “Next”

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

Step 1f: Type in “GA705_app” for the project name, select a folder for the project, and click “Finish”

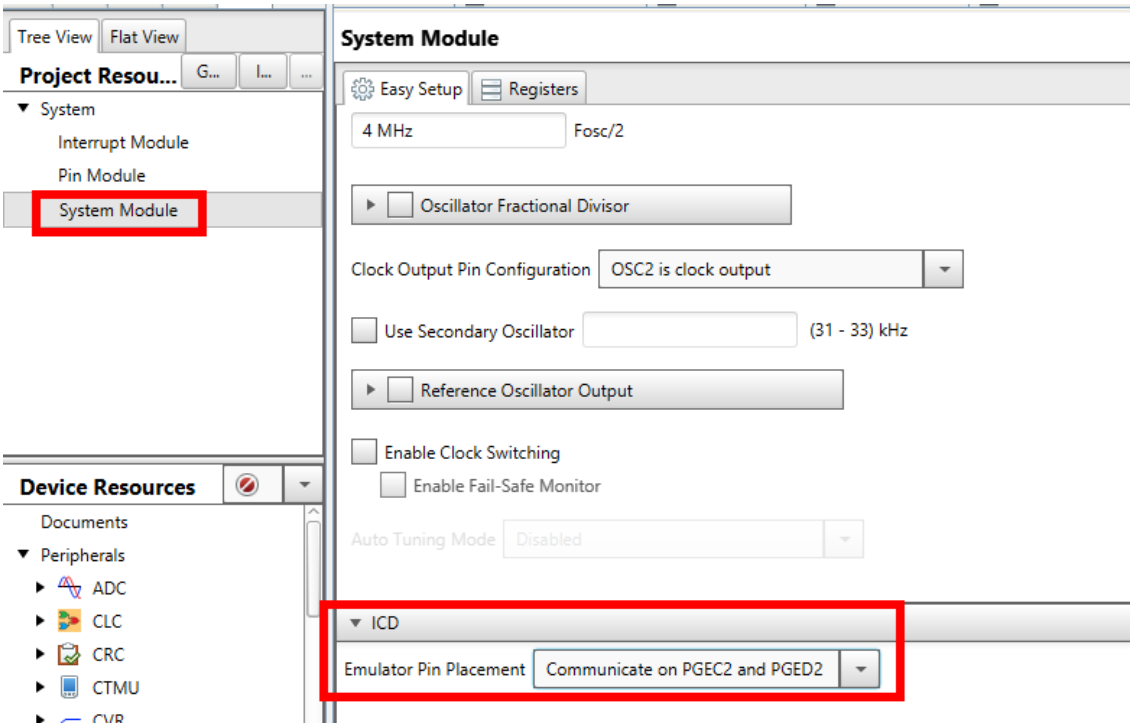
STEP 2: Configure MCC for the Board

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

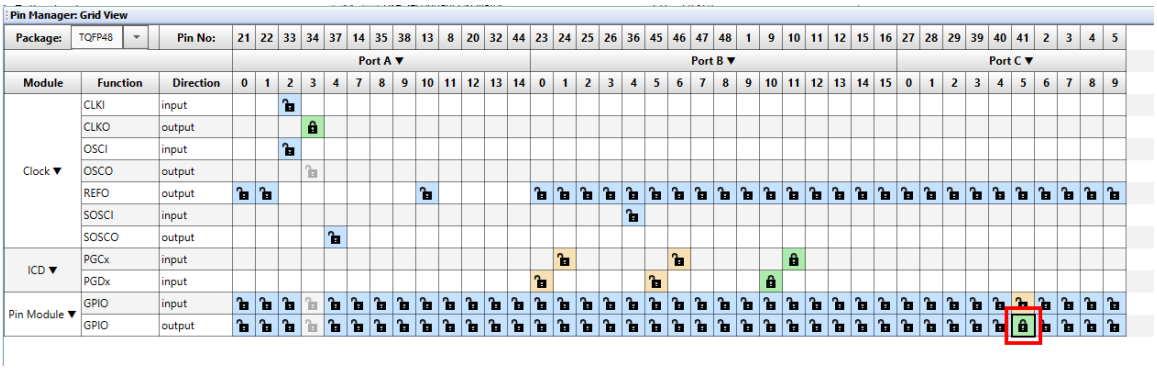


Step 2b: Select the “System Module” in the Project Resourced pane

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

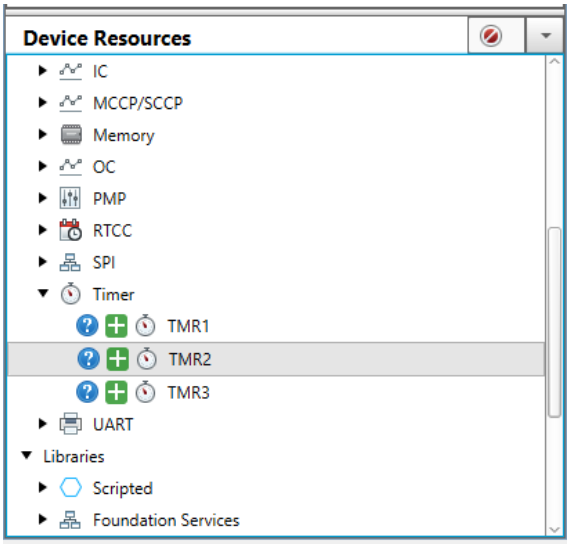


Step 2d: Open the pin manager grid view and configure the RC5 GPIO as an output. This is connected to the blue “Wi-Fi LED”.



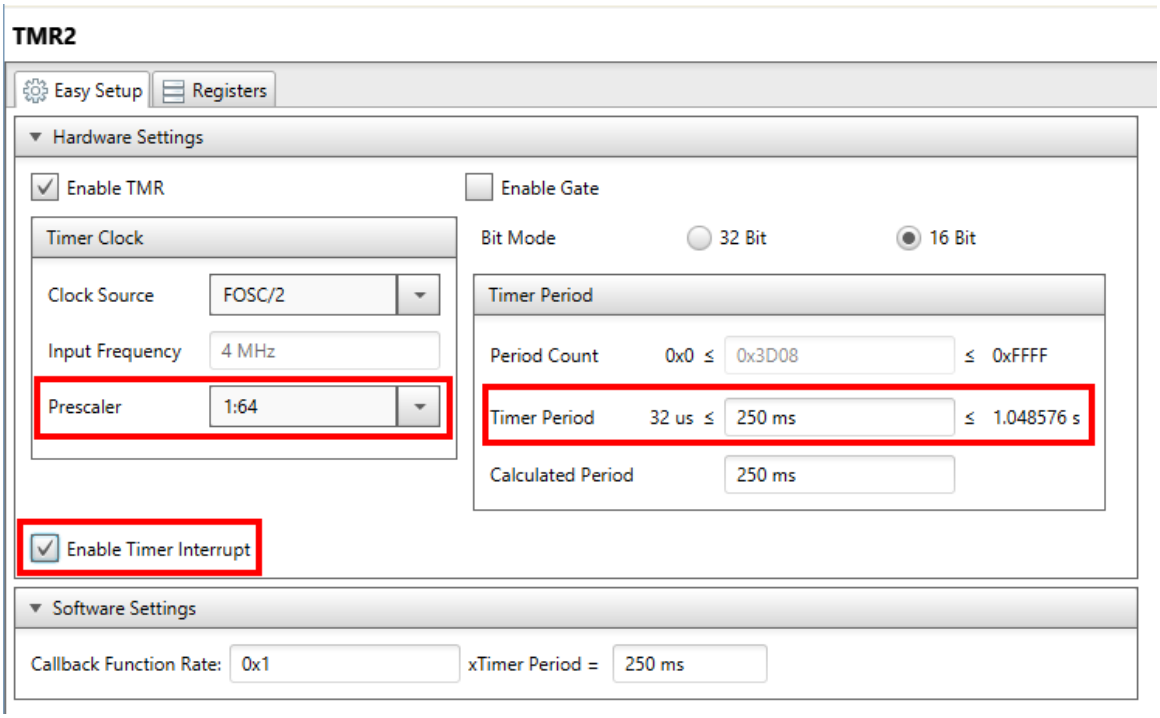
STEP 3: Create the example application code that blinks the LED

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

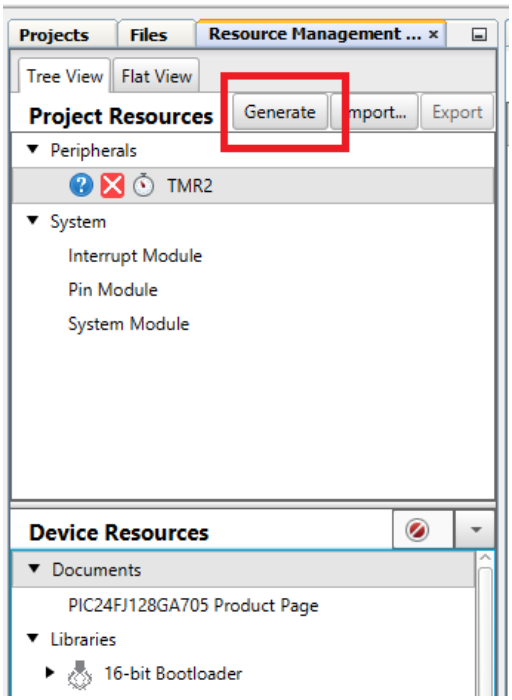


Step 3b: Open TMR2 Device Resource module if it wasn't automatically opened on the step above. Make the following changes to the TMR2 module configuration:

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



Step 3c: Generate the code by clicking the “Generate” button

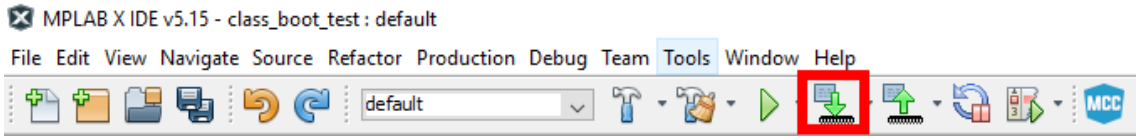


Step 3d: The generated code for TMR2 has a weakly prototyped function `void TMR2_CallBack(void)`. To have code called on the timer interrupt, just create a function named `TMR2_CallBack(void)` and it will be called automatically on a timer interrupt.

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.

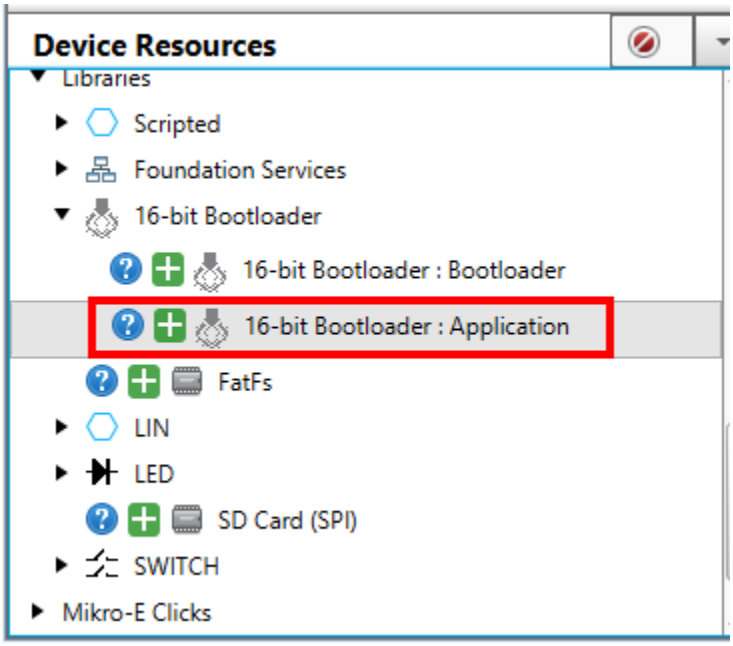


STEP 4: Modify application to work with the bootloader

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.

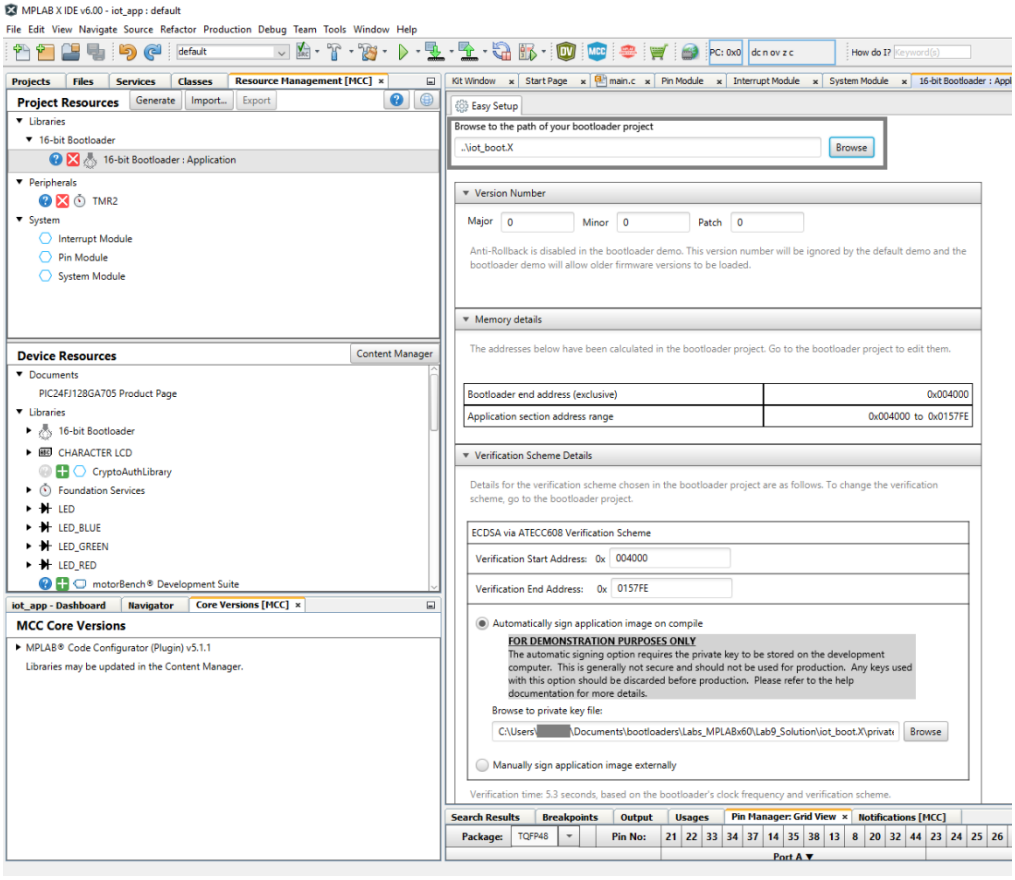
Step 4a: If MCC is not still open then re-open MCC.

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.



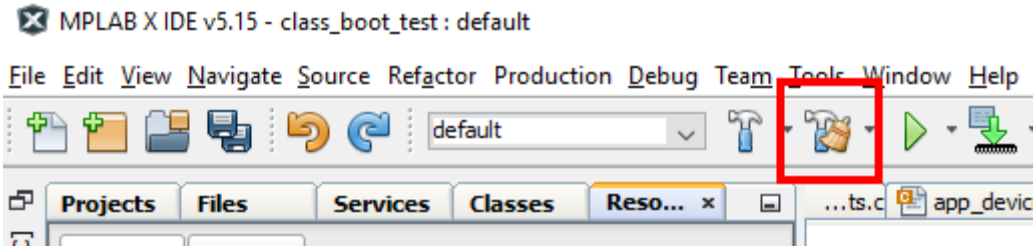
Just like before, after loading the details of the bootloader project, the application is updated with these settings as well. In this case it's detected that were using ECDSA Verification Scheme as well as updating the location of the ECDSA Private Key File.
If the user now tries to generate the project, the user will get a notification error saying the configuration bits are different. In this case, MCC is detecting that the IOL1WAY bits 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 8, Step 7 (in the system module under the register tab).

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

STEP 5: Resolve Build Errors

Step 4a: Compile the project by hitting the clean and build button shown below. DO NOT press the program button. This will generate the new application hex file that is offset correctly to load with the boot loader. We'll use the bootloader to load this application in the next lab.



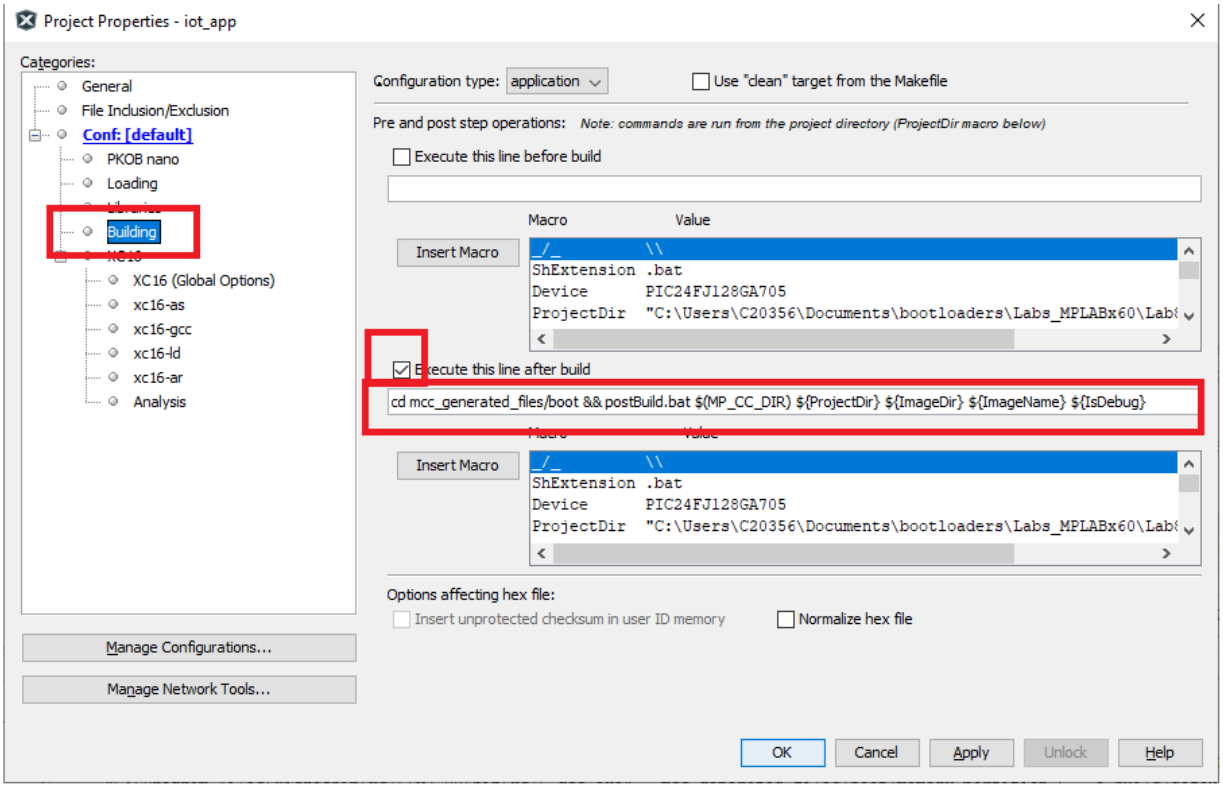
Notice that we get two error messages during the compiling forcing two changes.

Step 5b: The first error shown below, is telling the user that a script that signs the file has been added to the system and that the user needs to add it to the post build step like we did in Lab 5. Clicking on the error message will take you to the file "certificate_atecc608.S" shown below:

```
#error "A script to append a verification signature has been added
to your project. You must add a call to it as a post-build step in
your project properties. Delete this error after you have done so."

/*

* Steps to append verification hash 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)
${ProjectDir} ${ImageDir} ${ImageName} ${IsDebug}" (without quotes)
if you are on a Windows machine, or
```

Step 5c: 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, sign it with the ECDSA signature of the application area and then insert the computed signature value back into the hex file at the predefined address of the application header. This signature along with the start and end addresses is stored in the header will be used by the bootloader and the ATECC608A device to verify the authenticity of the application image in flash.

Step 5d: Following the steps 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.



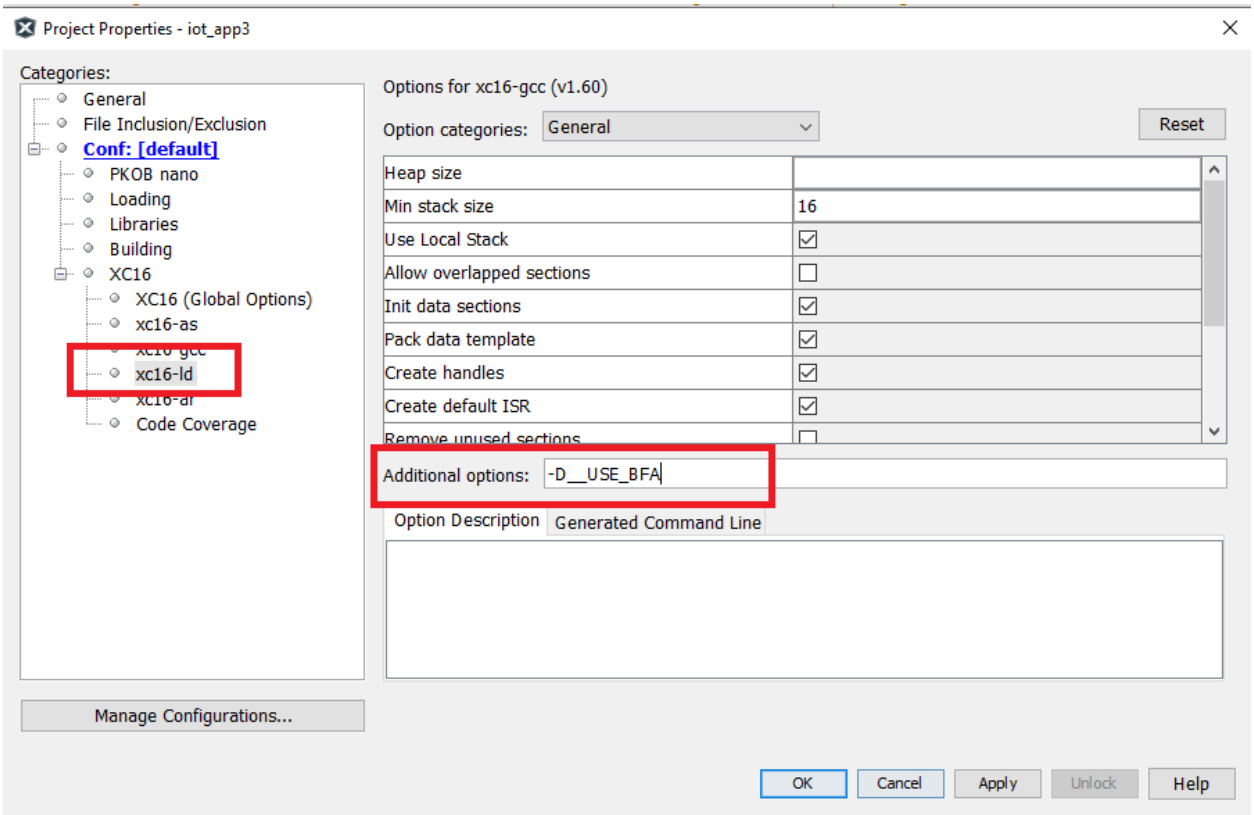
Step 5e: Once done, press OK to close this dialog box. Then go back the file, `certificate_atecc608.S`, and comment out the line with the `#error` on it.

Step 5f: Clean and rebuild the project again. This should remove the first error, but there will be one error remaining telling us to update the linker options.

Step 5g: Follow the directions provided by the #error link message text, seen 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
*/
```

After making the modifications, the linker options panel in MPLAB X should look like this:



Step 5h: One this is done, comment out the line in `app_code_protect.c` with the `#error` message and recompile again.

This time the code compiles cleanly. We are now ready to download the application using the UBHA.

NOTE: You can program this application image into the board and it will still run. If you do program the updated application firmware into the board (either on purpose or by accident), you will need to re-program the bootloader firmware into the board so we can load the application using the bootloader instead.

STEP 6: Bootload new Application with Bootloader

Step 6a: 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 6b: Open UBHA and program.

Step 6c: Click the “Read Device Settings” button and verify it updates the start and end addresses.

Step 6d: Selecting the “Enable Self Verification after Program” will request that the bootloader run the ECDSA verification method after the firmware programming is complete and verify it matches what is in the application header.

Step 6e: Select the new application file using the “File->Open” option.

Step 6f: Press the Program button. If this fails, common failure modes are forgetting to add the post build step (step 8 above) to the application build process or not updating the bootloader.

If it completes successfully, the bootloader has verified image in the flash was signed using the private key. 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:

At the end of this lab, we've created an application that was signed with an ECDSA signature, bootloaded it, and had the booloader verify the application before running. The ECDSA signature was automatically generated on compilation using a script that we added to the post build step of the project.

Summary:

This lab completes a very simple secure boot application. This application gets checked for integrity (no modifications) via the SHA-256 hash as part of the ECDSA signing process. It gets checked for authenticity via the ECDSA signature.

In this lab we used a private key on the local machine for the signing process. If a more secure signing flow is required, the "manual signing" option can be selected in MCC and two scripts are created. The first pre-processes the hex file into a binary file to sign. The second script post-processes the signature file and reinjects it back into the hex file. The signing process is then left as an exercise for the developer.

Trademarks