LAB Manual for 21016 DEV5

Creating Simple PIC32 Embedded Applications using MPLAB® Harmony

Table of Contents

Notes:

Introduction:

The lab exercises in this manual are intended to reinforce key MPLAB® Harmony concepts described in the Microchip MASTERs 2017, “*21016-DEV5: Creating Simple PIC32 Embedded Applications using MPLAB® Harmony*” class.

This manual provides specific instructions to configure the MPLAB Harmony projects and the PIC32 MCU to successfully complete the exercises. However, due to the limited time allotted to each lab, it does not provide all necessary background details as to why each configuration item is required. Upon completion of the class you are encouraged to use additional resources (identified in the class materials) to further your learning and understanding of MPLAB Harmony and the PIC32 MCU.

Upon completion, you will:

1. Be able to create an MPLAB Harmony project.
2. Have a fundamental understanding of MPLAB Harmony project layout and execution flow.
3. Have hands-on experience with the MPLAB Harmony Configurator (MHC).
4. Have hands-on experience with MPLAB Harmony Framework core components like Drivers and System services.
5. Have hands-on experience with MPLAB Harmony BSP.
6. Have a good understanding of key features and benefits of MPLAB Harmony.

Prerequisites:

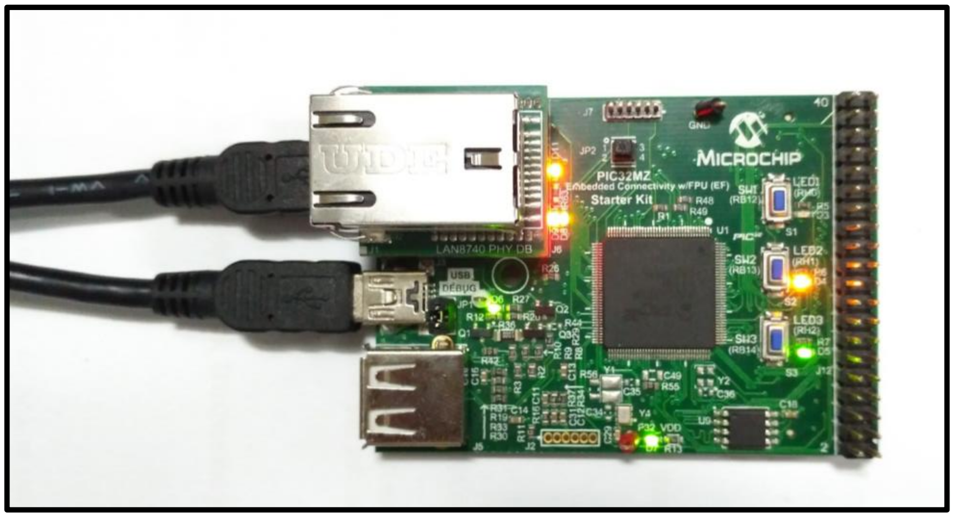
The class assumes the attendee has prior experience with:

1. MPLAB® X IDE basics
2. MPLAB based Programming/Debugging experience
3. C language programming knowledge
4. Basic knowledge of PIC32 microcontrollers

Required Development Tools:

1. PIC32 MZ EF Starter Kit.
   These labs are designed to support either of these PIC32MZ EF Starter Kits:
    a. PIC32MZ Embedded Connectivity with FPU (EF) Starter Kit, part # DM320007 (PIC32MZ2048EFH144 MCU).
    b. PIC32MZ Embedded Connectivity with FPU (EF) Starter Kit (Crypto), part # DM320007-C, (PIC32MZ2048EFM144 MCU).
2. MPLAB® X IDE v3.61.
3. MPLAB XC32 Compiler v1.43.
4. MPLAB Harmony Integrated Software Framework v2.03b.
5. MPLAB Harmony Configurator (MHC) v2.0.3.5.
   (Included in MPLAB Harmony v2.03b installation.)

Hardware Setup:



LAB 1

Creating a Harmony Project

-Toggle LED

LAB 1: Creating a Harmony Project –Toggle LED

Purpose:

After completing Lab 1, you will have a basic understanding of the fundamental elements, layout, and execution model of a MPLAB® Harmony project. You will also have gained the experience of using the MPLAB Harmony Configurator tool (MHC) to configure and generate the MPLAB Harmony project.

Overview:

In this lab, you will create a MPLAB Harmony project using the MHC. The application you will create will utilize the BSP to control LED toggle (ON/OFF) using SWITCH state. While you achieve this, you will understand the project layout and the execution flow of a Harmony project.

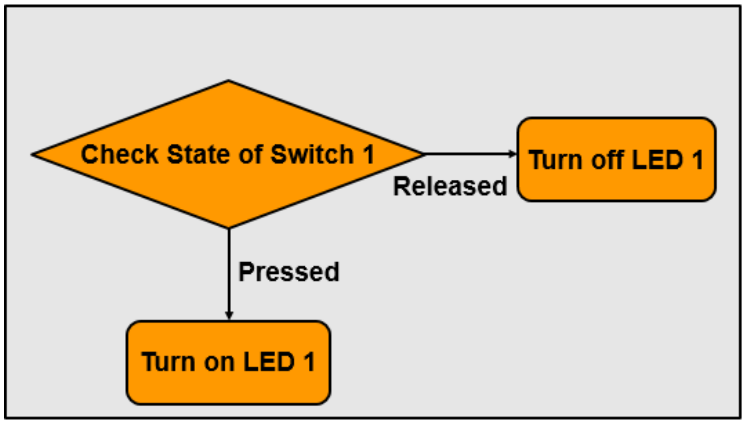
This lab is divided in two parts:

Part 1: Create Project
- Step 1 – Install the MHC Plug-in in MPLAB® X IDE
- Step 2 – Create MPLAB® Harmony project

Part 2: Add Application code and Execute
- Step 3 – Add code for BSP SWITCH and LED toggle logic.
- Step 4 – Explore Harmony Project and its Execution flow.

Note: Each step has a short procedure that you must follow to complete the step.

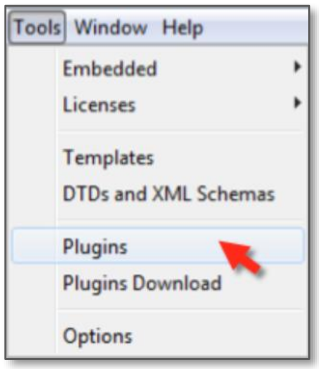
Application logic:



Step 1: Install the MHC Plug-in in MPLAB® X IDE

This step can be skipped if the correct version of the MHC is already installed (See Required Development Tools). See following steps 1. and 2.a. to verify the installed version.

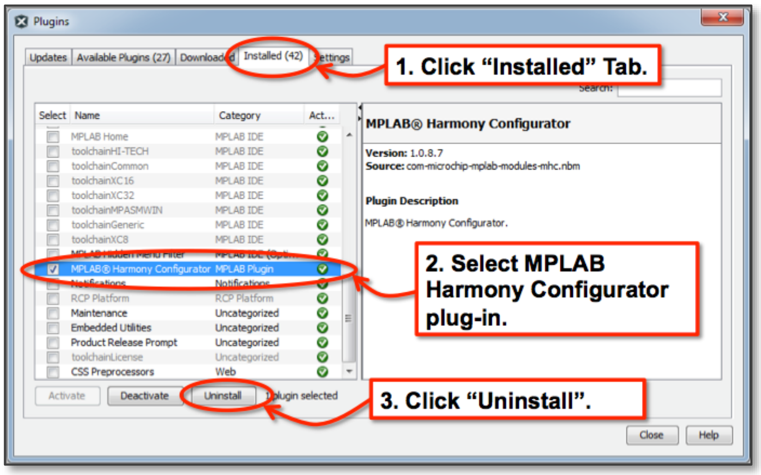
Procedure:



1. Launch MPLAB X IDE from the Windows Start Menu. Close any currently open projects and files.

2. Ensure that MPLABX has the correct version of the MHC, otherwise uninstall the current version of the MHC plug-in from the MPLAB X IDE and install the version needed by these labs (See Required Development Tools).
   a. Choose ***Tools > plugins*** from the IDE's top-level menu as shown above.

   b. *"Uninstall"* the MPLAB Harmony Configurator plugin from the IDE using the "*Installed"* tab of the Plugins dialog box as shown below.
   After uninstalling, select "*Restart"* and click "*Finish"*.



Select	Name	Category	Act...
☐	MPLAB Home	MPLAB IDE	✓
☐	toolchainHI-TECH	MPLAB IDE	✓
☐	toolchainCommon	MPLAB IDE	✓
☐	toolchainXC16	MPLAB IDE	✓
☐	toolchainXC32	MPLAB IDE	✓
☐	toolchainMPASMWIN	MPLAB IDE	✓
☐	toolchainGeneric	MPLAB IDE	✓
☐	toolchainXC8	MPLAB IDE	✓
☐	MPLAB Hidden Menu Filter	MPLAB IDE (Opti...	✓
☑	MPLAB® Harmony Configurator	MPLAB Plugin	✓
☐	Notifications	Notifications	✓
☐	RCP Platform	RCP Platform	✓
☐	Maintenance	Uncategorized	✓
☐	Embedded Utilities	Uncategorized	✓
☐	Product Release Prompt	Uncategorized	✓
☐	toolchainLicense	Uncategorized	✓
☐	CSS Preprocessors	Web	✓

MPLAB® Harmony Configurator

Version: 1.0.8.7
Source: com-microchip-mplab-modules-mhc.nbm

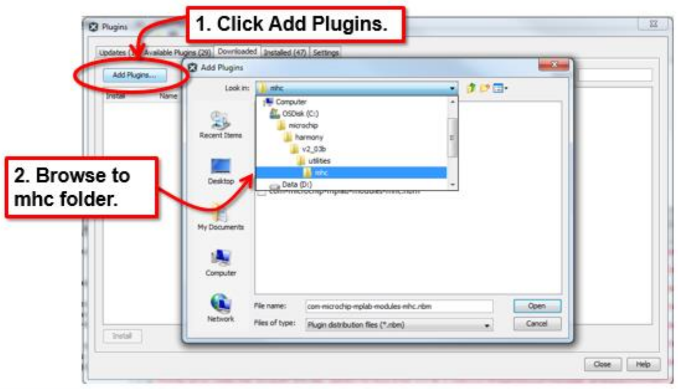
Plugin Description

MPLAB® Harmony Configurator.

   c. Choose ***Tools > Plugins*** from the IDE's top-level menu again.

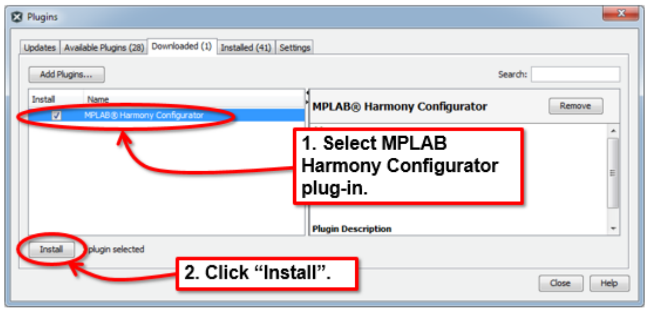
d. Under the "*Downloaded"* tab, click "*Add Plugins"* and browse to the version of the MPLAB® Harmony Configuration plugin that is used in this class (See Required Development Tools) as shown below. Click "Open".

Path: **`C:\Microchip\harmony\<version>\utilities\mhc`**
File: **`com-microchip-mplab-modules-mhc.nbm`**



3. *"Install"* the MHC plugin as shown below (You will need to accept the license agreement).

Note: After installing, ensure "*Restart Now"* is selected and click "*Finish"*.



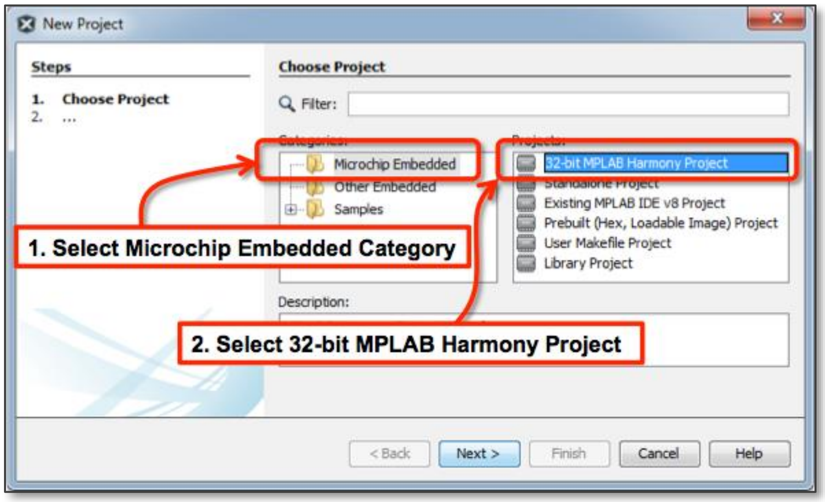
Step 2: Create MPLAB® Harmony Project

Procedure:

1. Create a new MPLAB Harmony Project.

   a. Select ***File > New Project*** from the MPLABX IDE main menu.

b. In the “*New Project”* dialog box, select “*32-bit MPLAB® Harmony Project”* from the “*Microchip Embedded”* category and click “*Next”* as shown below.



New Project

Steps
1. Choose Project
2. ...

Choose Project

Filter:

Categories:
- Microchip Embedded
- Other Embedded
- Samples

Projects:
- 32-bit MPLAB Harmony Project
- Standalone Project
- Existing MPLAB IDE v8 Project
- Prebuilt (Hex, Loadable Image) Project
- User Makefile Project
- Library Project

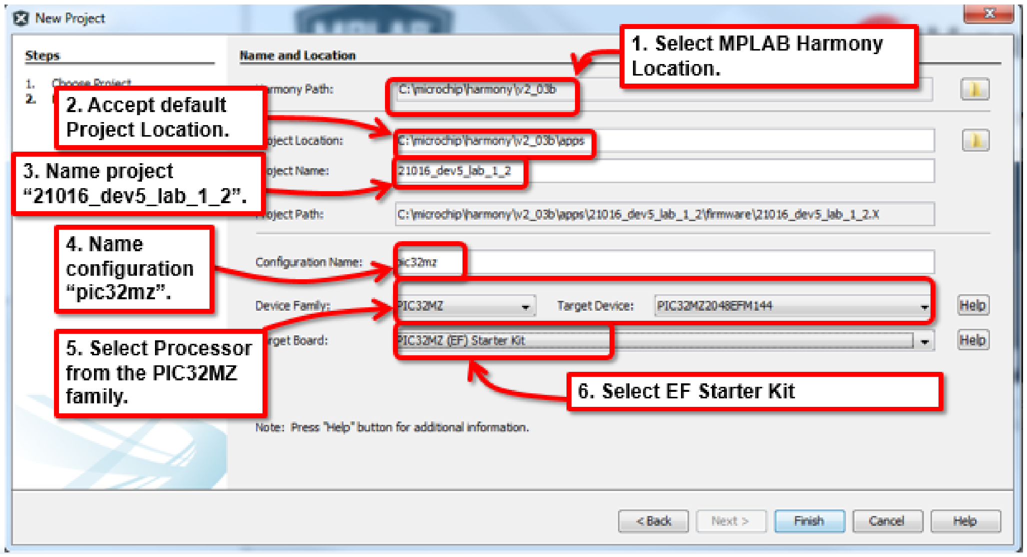
Description:

1. Select Microchip Embedded Category

2. Select 32-bit MPLAB Harmony Project

c. Create the “*New Project”* as follows:

Harmony Path:	**`C:\Microchip\harmony\<version>`**
Project Location:	**`C:\Microchip\harmony\<version>\apps`**
Name:	**21016_dev5_lab_1_2**
Configuration:	**pic32mz**
Device:	**PIC32MZ2048EFM144**
Target Board:	**PIC32MZ (EF) Starter Kit**



New Project

Steps
1. Choose Project
2. Name and Location

Name and Location

Harmony Path: C:\microchip\harmony\v2_03b
Project Location: C:\microchip\harmony\v2_03b\apps
Project Name: 21016_dev5_lab_1_2
Project Path: C:\microchip\harmony\v2_03b\apps\21016_dev5_lab_1_2\firmware\21016_dev5_lab_1_2.X

Configuration Name: pic32mz
Device Family: PIC32MZ
Target Device: PIC32MZ2048EFM144
Target Board: PIC32MZ (EF) Starter Kit

Note: Press "Help" button for additional information.

1. Select MPLAB Harmony Location.

2. Accept default Project Location.

3. Name project “21016_dev5_lab_1_2”.

4. Name configuration “pic32mz”.

5. Select Processor from the PIC32MZ family.

6. Select EF Starter Kit

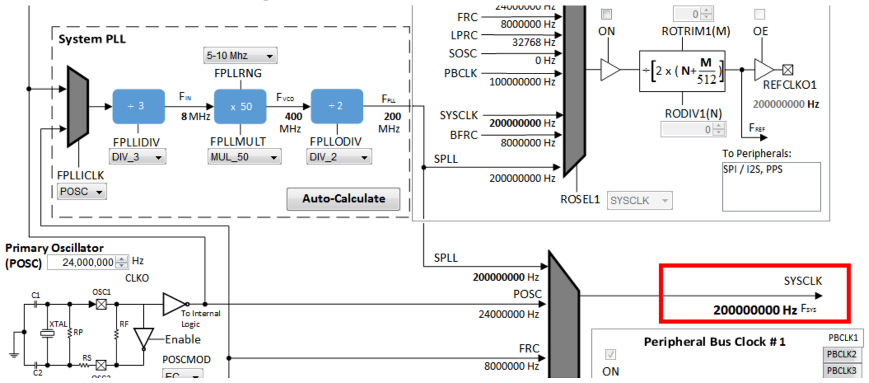
Note: Be sure to select the **EF** starter kit, not the EC starter kit.

d. Click "*Finish"* when ready.
MHC launches on its own for the fresh Harmony project. Please wait until it is loaded.

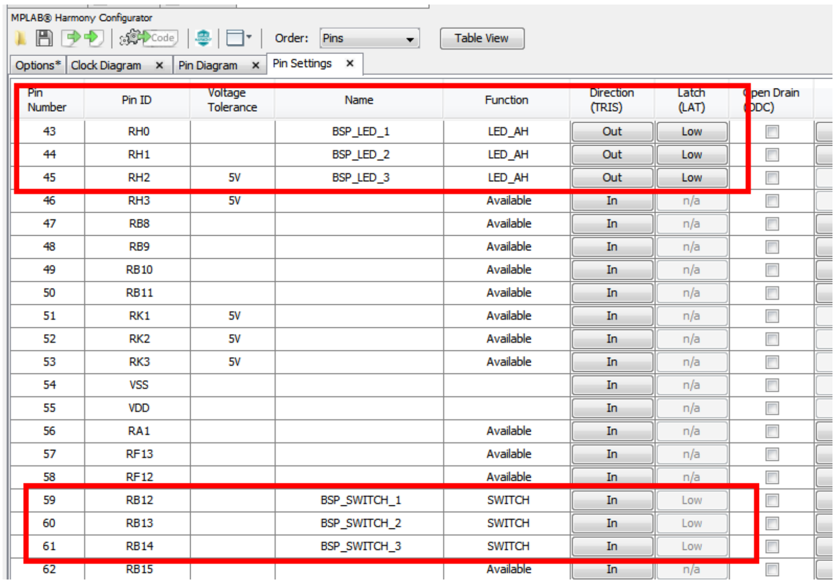
Note: If MHC doesn't launch automatically or if you need to reopen it for an existing project. You can launch MHC by setting the project as "*Main Project*" and launch MHC from the MPLAB® X toolbar:
Tools > Embedded > MPLAB Harmony Configurator.

Note: The default initial target board configuration sets the processor clock at its highest rate and sets up the IO pins for switches and LEDs. Feel free to explore these settings in the "*Clock Diagram"* and "*Pin Settings"* tabs in the MHC window as shown below.

Default ***Clock Diagram***:



Default ***Pin Settings***:


MPLAB® Harmony Configurator

Order: Pins | Table View

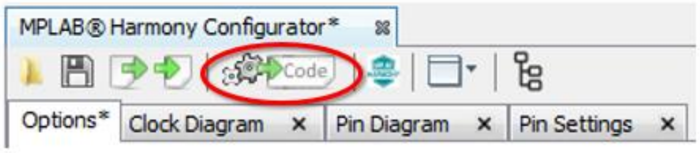
Options* | Clock Diagram | Pin Diagram | Pin Settings

Pin Number	Pin ID	Voltage Tolerance	Name	Function	Direction (TRIS)	Latch (LAT)	Open Drain (ODC)
43	RH0		BSP_LED_1	LED_AH	Out	Low	☐
44	RH1		BSP_LED_2	LED_AH	Out	Low	☐
45	RH2	5V	BSP_LED_3	LED_AH	Out	Low	☐
46	RH3	5V		Available	In	n/a	☐
47	RB8			Available	In	n/a	☐
48	RB9			Available	In	n/a	☐
49	RB10			Available	In	n/a	☐
50	RB11			Available	In	n/a	☐
51	RK1	5V		Available	In	n/a	☐
52	RK2	5V		Available	In	n/a	☐
53	RK3	5V		Available	In	n/a	☐
54	VSS				In	n/a	☐
55	VDD				In	n/a	☐
56	RA1			Available	In	n/a	☐
57	RF13			Available	In	n/a	☐
58	RF12			Available	In	n/a	☐
59	RB12		BSP_SWITCH_1	SWITCH	In	Low	☐
60	RB13		BSP_SWITCH_2	SWITCH	In	Low	☐
61	RB14		BSP_SWITCH_3	SWITCH	In	Low	☐
62	RB15			Available	In	n/a	☐

2. Generate Harmony project files and code:

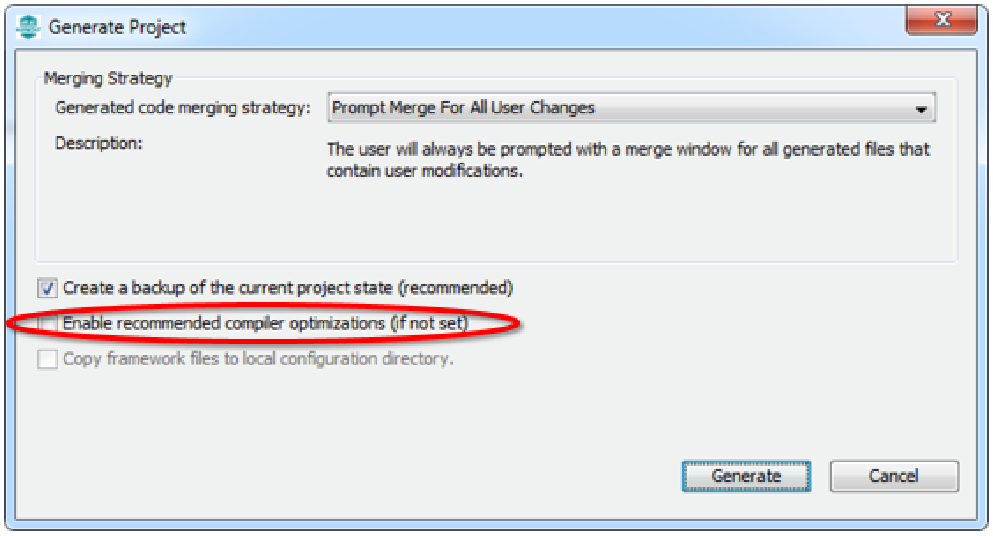
   The MHC will generate the basic system code, the default configuration code and a template application file.

   a. Click the "*Generate Code*" button as shown on the right (save the configuration in its default location, when asked). Complete "*step b"* to generate code.



   b. Deselect the "*Enable recommended compiler optimizations*" setting. **Note:** This setting enables optimization level **1** (-O1). For this class, we are using optimization level **0** (-O0, whenever you are debugging).

      Click on "*Generate*" when ready.



Generate Project

Merging Strategy

Generated code merging strategy: Prompt Merge For All User Changes

Description: The user will always be prompted with a merge window for all generated files that contain user modifications.

[✓] Create a backup of the current project state (recommended)

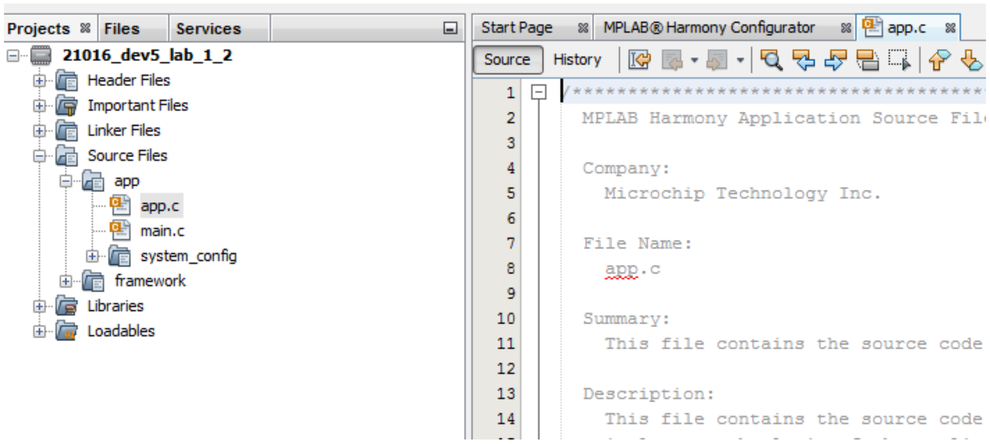
[ ] Enable recommended compiler optimizations (if not set)

[ ] Copy framework files to local configuration directory.

Step 3: Add code for BSP SWITCH and LED toggle logic

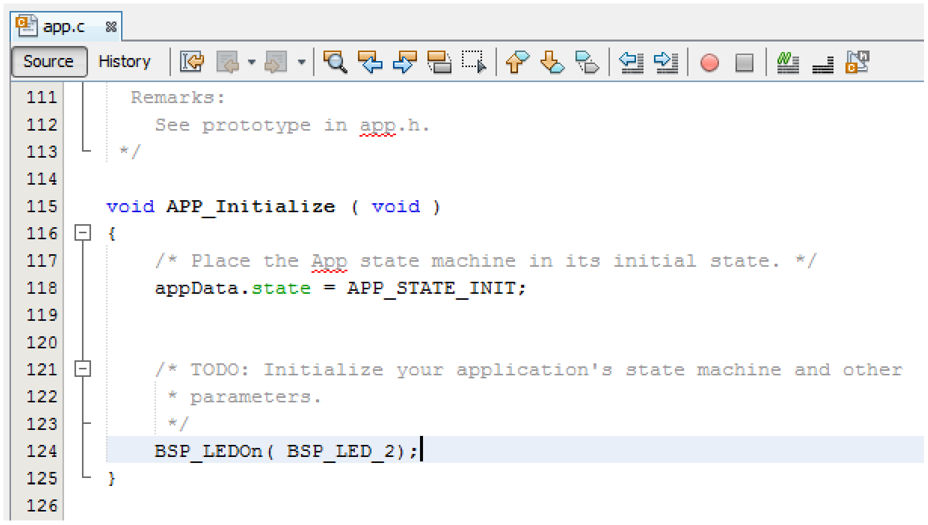
Procedure:

1. Open **`app.c`** file from the project Source Files.



```
/*************************************
  MPLAB Harmony Application Source Fil

  Company:
    Microchip Technology Inc.

  File Name:
    app.c

  Summary:
    This file contains the source code

  Description:
    This file contains the source code
```

2. Navigate to **`APP_Initialize`** function and call the **`BSP_LEDOn`** function to turn on **`BSP_LED_2`** as shown below. The **`APP_Initialize`** function initializes the application state and indicates application ON status.



```
  Remarks:
    See prototype in app.h.
 */

void APP_Initialize ( void )
{
    /* Place the App state machine in its initial state. */
    appData.state = APP_STATE_INIT;


    /* TODO: Initialize your application's state machine and other
     * parameters.
     */
    BSP_LEDOn( BSP_LED_2);
}
```

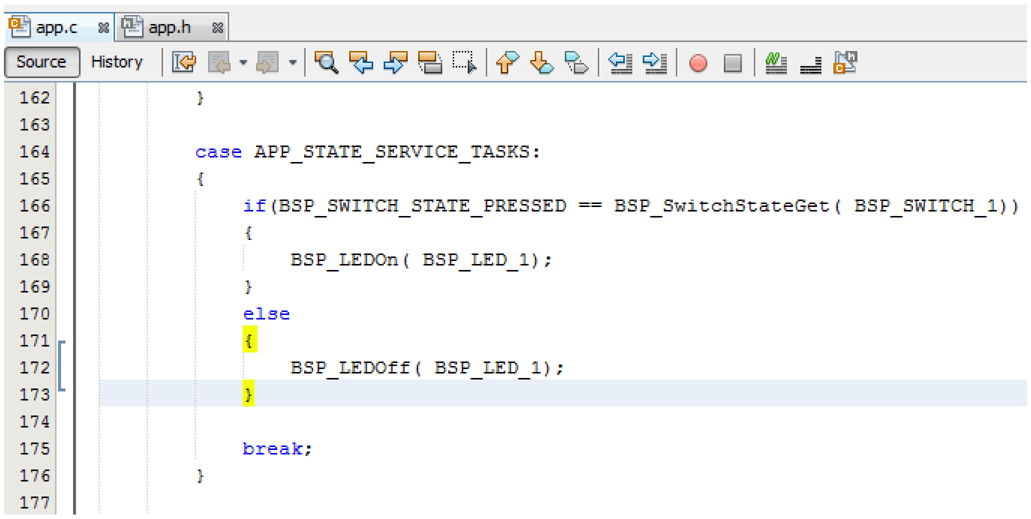
Code snippet:

```
BSP_LEDOn( BSP_LED_2);
```

3. Add application logic in **`APP_Tasks`** function.
    a. Navigate to **`APP_Tasks`** function switch case **`"APP_STATE_SERVICE_TASKS"`**

    b. Call **`BSP_SwitchStateGet`** function to check the state of **`BSP_SWITCH_1`**.

    c. Based on the **`BSP_SWITCH_1`** status, toggle **`BSP_LED_1`** by calling **`BSP_LEDOn`** and **`BSP_LEDOff`** functions as shown below. That is, if the SWITCH is pressed, then turn ON the LED otherwise turn it OFF.



```
            }

            case APP_STATE_SERVICE_TASKS:
            {
                if(BSP_SWITCH_STATE_PRESSED == BSP_SwitchStateGet( BSP_SWITCH_1))
                {
                    BSP_LEDOn( BSP_LED_1);
                }
                else
                {
                    BSP_LEDOff( BSP_LED_1);
                }

                break;
            }

```

Code snippet:

```
if(BSP_SWITCH_STATE_PRESSED == BSP_SwitchStateGet( BSP_SWITCH_1))
{
    BSP_LEDOn( BSP_LED_1);
}
else
{
    BSP_LEDOff( BSP_LED_1);
}
```

<span>Step 4:</span> Explore Harmony Project and its Execution flow

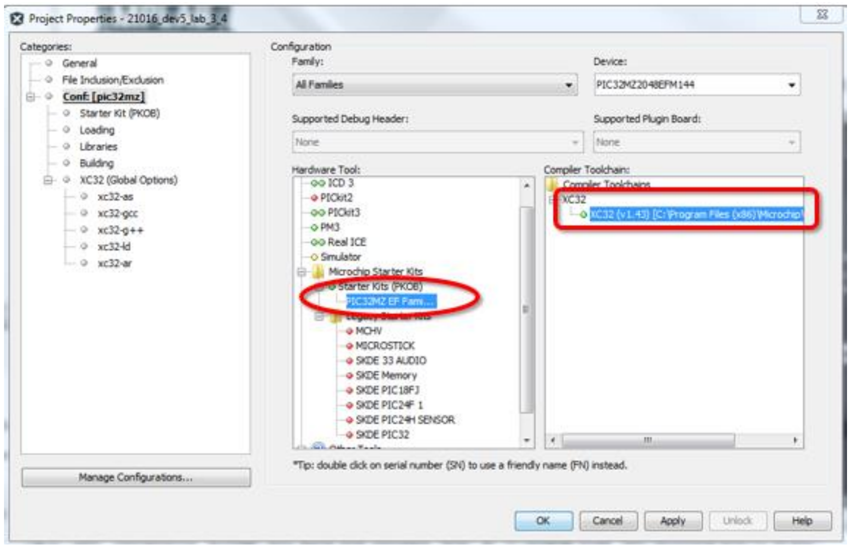
Procedure:
Connect the Starter Kit to the Computer as shown in the "*Hardware Setup"* (Page 4).

1. Ensure that the correct hardware tool and compiler are selected as shown below.

   a. Right-click on "***21016_dev5_lab_1_2***" in the "*Projects"* window pane and choose "*Properties"*.

   b. Ensure that the "***PIC32MZ EF starter kit****"* Hardware tool is selected.

   c. Ensure that the correct compiler "***XC32 (v1.43)***" is selected. Click "*Apply"* and exit the properties window by clicking "*OK"*.



Project Properties - 21016_dev5_lab_3_4

Categories:
- General
- File Inclusion/Exclusion
- Conf: [pic32mz]
  - Starter Kit (PKOB)
  - Loading
  - Libraries
  - Building
  - XC32 (Global Options)
    - xc32-as
    - xc32-gcc
    - xc32-g++
    - xc32-ld
    - xc32-ar

Configuration
Family: All Families
Device: PIC32MZ2048EFM144
Supported Debug Header: None
Supported Plugin Board: None

Hardware Tool:
- ICD 3
- PICkit2
- PICkit3
- PM3
- Real ICE
- Simulator
- Microchip Starter Kits
  - Starter Kits (PKOB)
    - PIC32MZ EF Fami...
  - Legacy Starter Kits
    - MCHV
    - MICROSTICK
    - SKDE 33 AUDIO
    - SKDE Memory
    - SKDE PIC18FJ
    - SKDE PIC24F 1
    - SKDE PIC24H SENSOR
    - SKDE PIC32
- Other Tools

Compiler Toolchain:
- Compiler Toolchains
  - XC32
    - XC32 (v1.43) [C:\Program Files (x86)\Microchip\

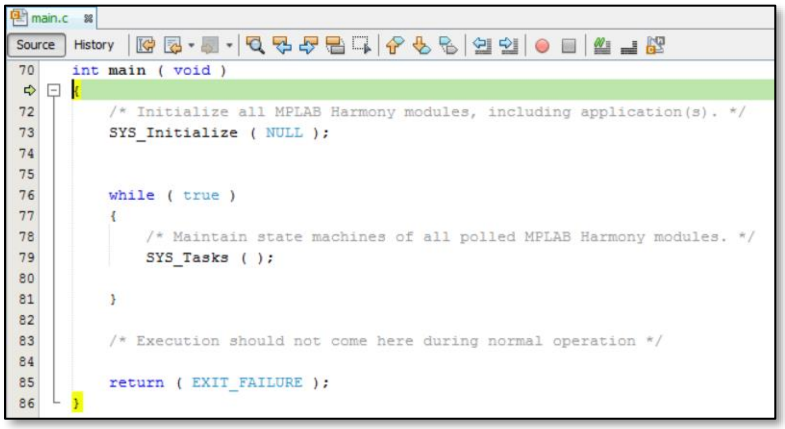
*Tip: double click on serial number (SN) to use a friendly name (FN) instead.

   d. Click the *Debug* button on MPLAB® X toolbar to build, program and run (in debug mode) the application.
   Note: If you get a Device ID mismatch warning, please ignore it.

   e. Once the program starts running, LED 2 should be lit by default. When you press SWITCH 1, LED 1 should turn ON. If you release it, LED 1 should turn OFF.

   f. You are able to toggle the LED 1 using SWITCH 1. You can now move on to explore the execution flow.

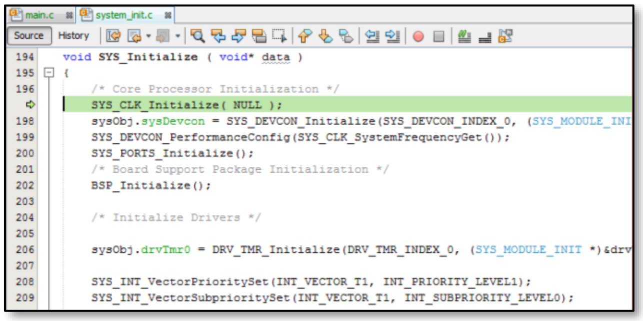
2. Once you have the project running in debug mode, pause and reset the controller.
   a. First, click the "*Pause*" button on the IDE toolbar.
   b. Then, click the "*Reset*" button on the IDE toolbar.
   c. Observe that the debugger breaks at the entry point **`main`** as shown below.



```
int main ( void )
{
    /* Initialize all MPLAB Harmony modules, including application(s). */
    SYS_Initialize ( NULL );


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

    }

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

    return ( EXIT_FAILURE );
}
```

   d. Hover the mouse pointer over the file name to locate the **`main.c`** file on computer.

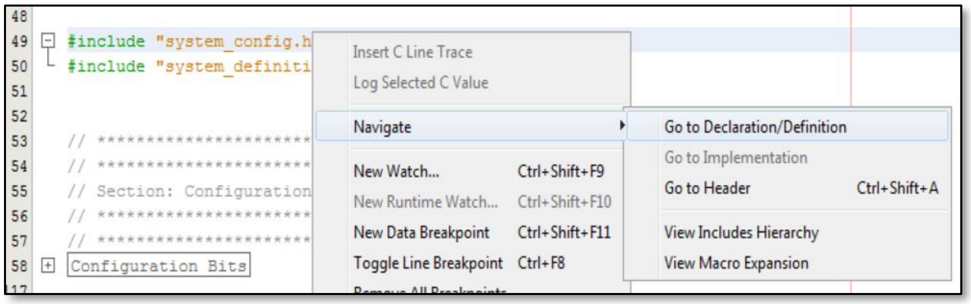
3. Step into the **`SYS_Initialize`** function and locate the **`system_init.c`** file.
   a. Click the "*Step Into*" button, repeatedly until the debugger enters the **`SYS_Initialize`** function.
   b. Locate the **`system_init.c`** file in the project and on the computer.
   c. Explore **`system_init.c`**. It defines the **`SYS_Initialize`** function that initializes the Harmony system (Including drivers, libraries, and the application). It also defines the processor's configuration bits and any data structures required to initialize the libraries used in the system.



```
void SYS_Initialize ( void* data )
{
    /* Core Processor Initialization */
    SYS_CLK_Initialize( NULL );
    sysObj.sysDevcon = SYS_DEVCON_Initialize(SYS_DEVCON_INDEX_0, (SYS_MODULE_INI
    SYS_DEVCON_PerformanceConfig(SYS_CLK_SystemFrequencyGet());
    SYS_PORTS_Initialize();
    /* Board Support Package Initialization */
    BSP_Initialize();

    /* Initialize Drivers */

    sysObj.drvTmr0 = DRV_TMR_Initialize(DRV_TMR_INDEX_0, (SYS_MODULE_INIT *)&drv

    SYS_INT_VectorPrioritySet(INT_VECTOR_T1, INT_PRIORITY_LEVEL1);
    SYS_INT_VectorSubprioritySet(INT_VECTOR_T1, INT_SUBPRIORITY_LEVEL0);
```

4. Locate and explore the system configuration and definitions header files.

   a. In **`system_init.c`**, scroll to the top of the file, Right-click on the include statement for the **`system_config.h`**. Go to the file's definition as shown below (or hold the "Ctrl" key and click over the name).



```
#include "system_config.h
#include "system_definiti
// **********************
// **********************
// Section: Configuration
// **********************
// **********************
Configuration Bits
```

Insert C Line Trace
Log Selected C Value
Navigate
New Watch... Ctrl+Shift+F9
New Runtime Watch... Ctrl+Shift+F10
New Data Breakpoint Ctrl+Shift+F11
Toggle Line Breakpoint Ctrl+F8
Go to Declaration/Definition
Go to Implementation
Go to Header Ctrl+Shift+A
View Includes Hierarchy
View Macro Expansion

   b. Locate the **`system_config.h`** file in the project and on the computer.

   c. Search for **`SYS_CLK_FREQ`**. This macro identifies the *System Clock Frequency*.

   d. Explore **`system_config.h`**.
      It defines configuration macros for all the Harmony components (including drivers, system service, other libraries and the application). It also contains the application defined BSP macros which can be used in the application.

   e. Go back to **`system_init.c`** and open **`system_definitions.h`** file. Locate the file in the project and on the computer.

   f. Explore **`system_definitions.h`**:
      This file contains definitions and prototypes that are generated specifically for this system configuration.
      This file is included by all system-configuration-specific files (files in the **`<application>/firmware/src/system_config/<configuration>`** folder) so that the system has access to the generated definitions and prototypes. Locate this file in the project and on the computer.

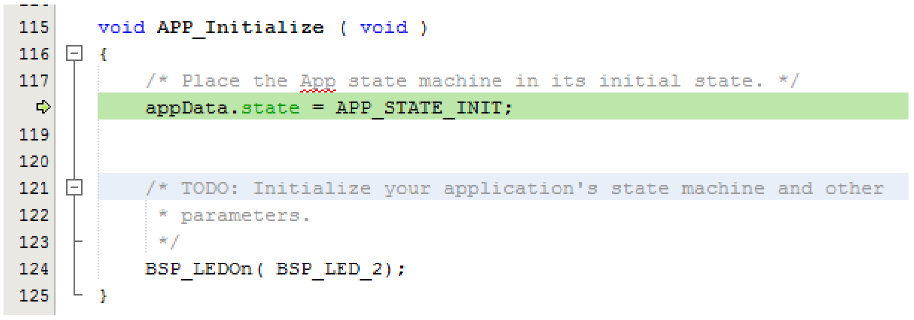
5. Explore **`APP_Initialize`** function and locate the application's source files.

   a. Switch back to **`system_init.c`**.

   b. Set a breakpoint on the call to **`APP_Initialize`** function and run until the debugger hits the breakpoint.

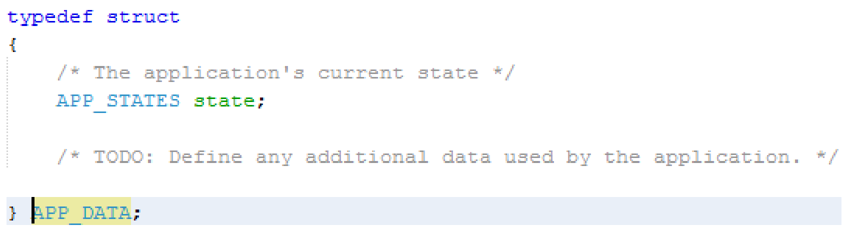
   c. Click the "*Step Into*" button to single step into the function.

   d. Locate the application's source file (**`app.c`**) and observe that the application's state is initialized here, as shown on below.



```
void APP_Initialize ( void )
{
    /* Place the App state machine in its initial state. */
    appData.state = APP_STATE_INIT;


    /* TODO: Initialize your application's state machine and other
     * parameters.
     */
    BSP_LEDOn( BSP_LED_2);
}
```

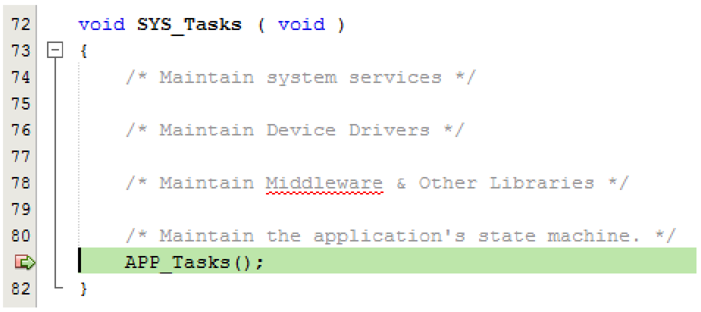
   e. Right-click on the application data structure variable (**`appData`**) and navigate to its declaration/allocation (in **`app.c`**) and the definition of its data structure type in the **`app.h`** header file, as shown on the right.



```
typedef struct
{
    /* The application's current state */
    APP_STATES state;

    /* TODO: Define any additional data used by the application. */

} APP_DATA;
```

Go back to **`app.c`** and for now step over the call to BSP LEDOn(BSP_LED_2) function. You can see how BSP call turn ON LED 2 on the board.

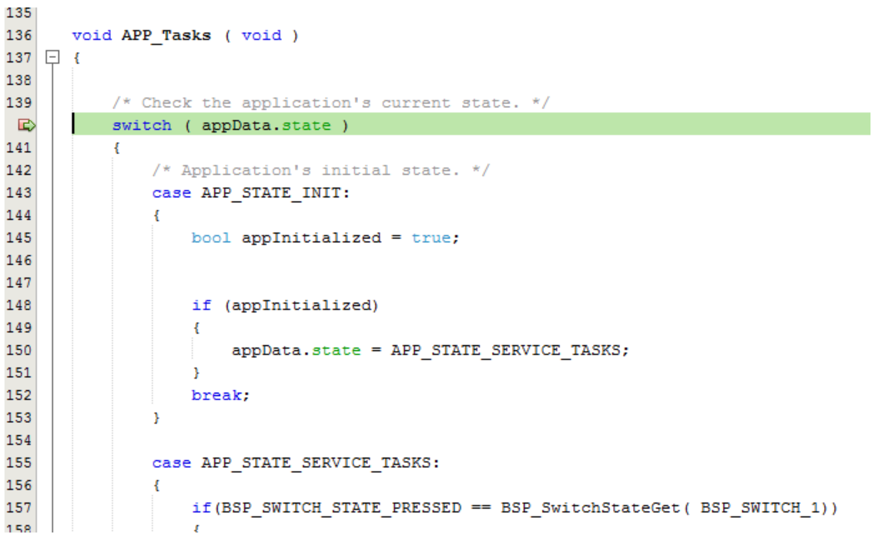
6. Continue stepping through system initialization, back to **`main`**, into the system tasks function (**`SYS_Tasks`**), and into the application's task (**`APP_Tasks`**) function.



```
void SYS_Tasks ( void )
{
    /* Maintain system services */

    /* Maintain Device Drivers */

    /* Maintain Middleware & Other Libraries */

    /* Maintain the application's state machine. */
    APP_Tasks();
}
```

7. Set a breakpoint on the switch statement in the application's tasks function and run the program, stopping in the application's tasks function to explore the application's state machine.

   a. Set a breakpoint at the application state machine's main switch statement, as shown below.



```

void APP_Tasks ( void )
{

    /* Check the application's current state. */
    switch ( appData.state )
    {
        /* Application's initial state. */
        case APP_STATE_INIT:
        {
            bool appInitialized = true;


            if (appInitialized)
            {
                appData.state = APP_STATE_SERVICE_TASKS;
            }
            break;
        }

        case APP_STATE_SERVICE_TASKS:
        {
            if(BSP_SWITCH_STATE_PRESSED == BSP_SwitchStateGet( BSP_SWITCH_1))
```

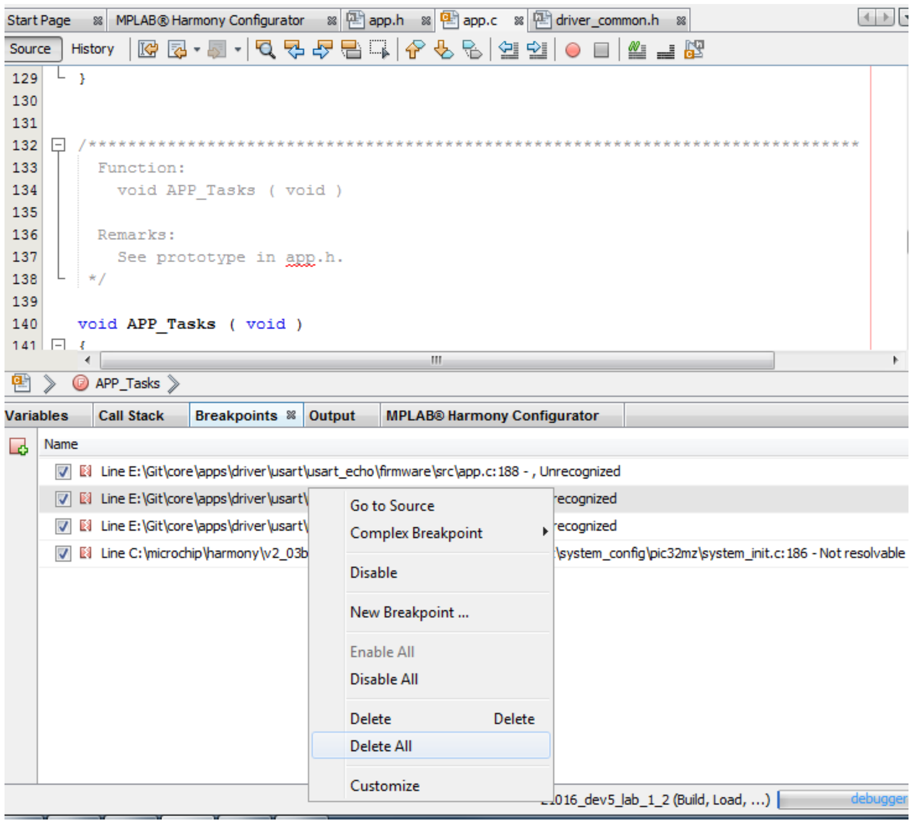
   b. *"Run"* to the breakpoint and step through the switch case executed. Each state in the state machine implements one state transition.

   c. When you reach a **`break`** statement in the application's main state machine, click the *"Run"* button to allow the entire program to execute and return back to the top of the application's state machine where you set the breakpoint.

   "Step Over" the calls to BSP functions. You can see how the BSP SWITCH and LED calls work by doing trials with SWITCH 1 pressed and released cases.

8. Delete all breakpoints and end the debug session.

    a. Go to "*Breakpoints"* window as shown below, right click and select "*Delete All*".



```
}


/*******************************************************************************
  Function:
    void APP_Tasks ( void )

  Remarks:
    See prototype in app.h.
 */

void APP_Tasks ( void )
{
```

```
Line E:\Git\core\apps\driver\usart\usart_echo\firmware\src\app.c:188 - , Unrecognized
Line E:\Git\core\apps\driver\usart\                                  ecognized
Line E:\Git\core\apps\driver\usart\                                  ecognized
Line C:\microchip\harmony\v2_03b                                     \system_config\pic32mz\system_init.c:186 - Not resolvable
```

```
016_dev5_lab_1_2 (Build, Load, ...)
```

    b. Click the "*Stop"* button in the MPLAB® X toolbar.

********************** Congratulations! You're done with Lab 1 **********************

You had hands on experience of the Harmony project layout and execution flow. You also gained experience on how to use MHC to generate a Harmony project.

You are now a MPLAB Harmony developer!

****

Notes:

LAB 2
Using Harmony Drivers
-Timer Driver to Implement Delay

LAB 2: Using Harmony Drivers – Timer Driver to Implement Delay

Purpose:

After completing Lab 2, you will have hands-on experience of ***how to configure*** a Harmony Driver using MHC and ***how to use*** it in your application. You will keep the driver logic simple and basic for this lab. You can build on this foundational knowledge later.

Overview:

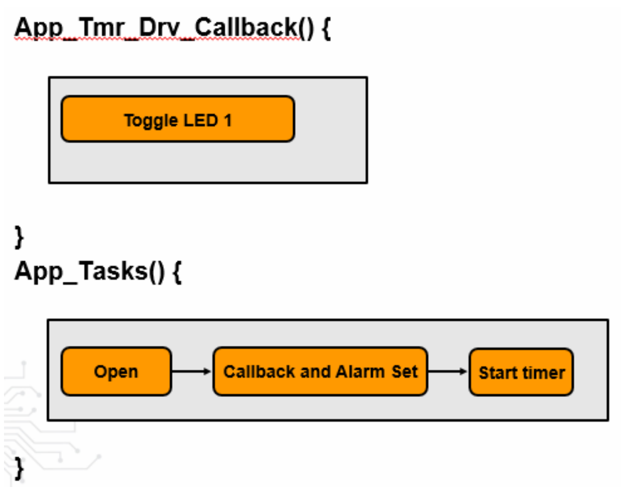
In this lab, you will start with the project created in Lab 1. Using the MHC, you will select and configure the **Timer** driver and then use it to implement a delay to toggle LED 3 periodically.

This lab is completed in the following steps:

Step 1: Enable and configure Timer driver using MHC and generate code.
Step 2: Open the Timer driver.
Step 3: Setup clock and calculate Timer period for 1 second delay.
Step 4: Register a callback function and request for a periodic alarm every 1 second.
Step 5: Explore the driver working flow.

Extra Credit! Try configuring different delay.

Application logic:



Step 1: Enable and configure Timer driver using MHC and generate code

Procedure:

Continue using the project “***21016_dev5_lab_1_2***” from Lab 1.

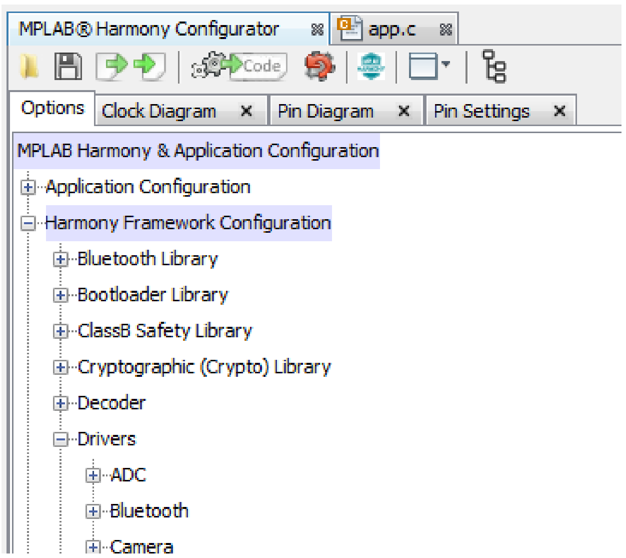
1. Go to MHC options window and navigate to “*Harmony Framework Configuration*” then “*Drivers*” as shown on the right.

2. Scroll down to “*Timer*” and expand the menu and enable it as shown on the right.

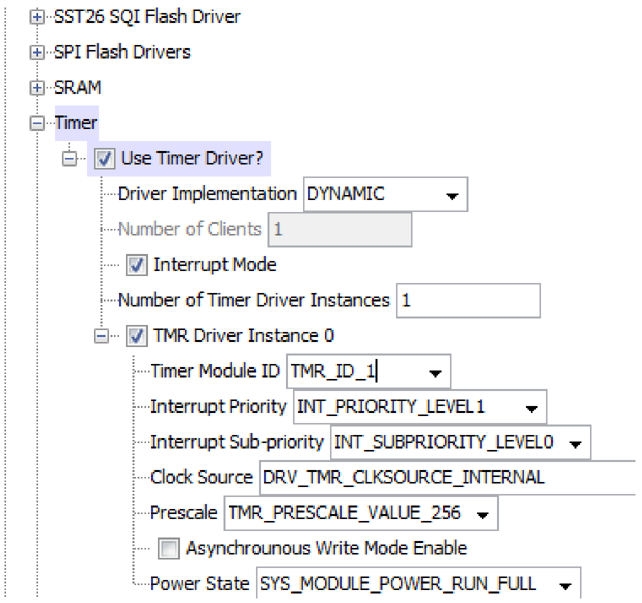
3. Keep the default configuration of Timer driver, which is the commonly used configuration for timers. Feel free to explore the options and the corresponding help available in MHC.

   Note: For now, you can ignore the “*Driver Implementation*” option. This will be discussed in the advanced Harmony class and does not have any effect on the application development.

4. Click on “*Generate code*” when ready. Then click ***Save > Generate.***
   Now Timer driver is ready to use in your application.



MPLAB Harmony & Application Configuration
- Application Configuration
- Harmony Framework Configuration
  - Bluetooth Library
  - Bootloader Library
  - ClassB Safety Library
  - Cryptographic (Crypto) Library
  - Decoder
  - Drivers
    - ADC
    - Bluetooth
    - Camera



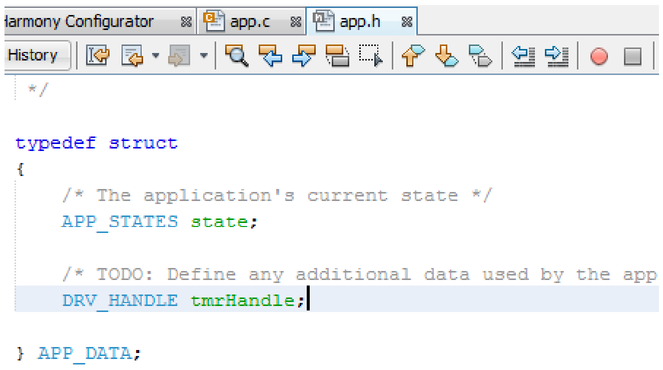
- SST26 SQI Flash Driver
- SPI Flash Drivers
- SRAM
- Timer
  - [✓] Use Timer Driver?
    - Driver Implementation: DYNAMIC
    - Number of Clients: 1
    - [✓] Interrupt Mode
    - Number of Timer Driver Instances: 1
    - [✓] TMR Driver Instance 0
      - Timer Module ID: TMR_ID_1
      - Interrupt Priority: INT_PRIORITY_LEVEL1
      - Interrupt Sub-priority: INT_SUBPRIORITY_LEVEL0
      - Clock Source: DRV_TMR_CLKSOURCE_INTERNAL
      - Prescale: TMR_PRESCALE_VALUE_256
      - [ ] Asynchrounous Write Mode Enable
      - Power State: SYS_MODULE_POWER_RUN_FULL

Step 2: Open the Timer driver.

Procedure:

The driver open function creates a relationship between the driver and application. The function returns a driver handle, which must be used in all subsequent calls to the driver functions.

1. Open **`app.h`** file and navigate to **`APP_DATA`** structure. Declare an application variable "**`tmrHandle`**" of type **`DRV_HANDLE`** as shown below. Here you are adding data to the app data structure that was discussed in the Lab 1.

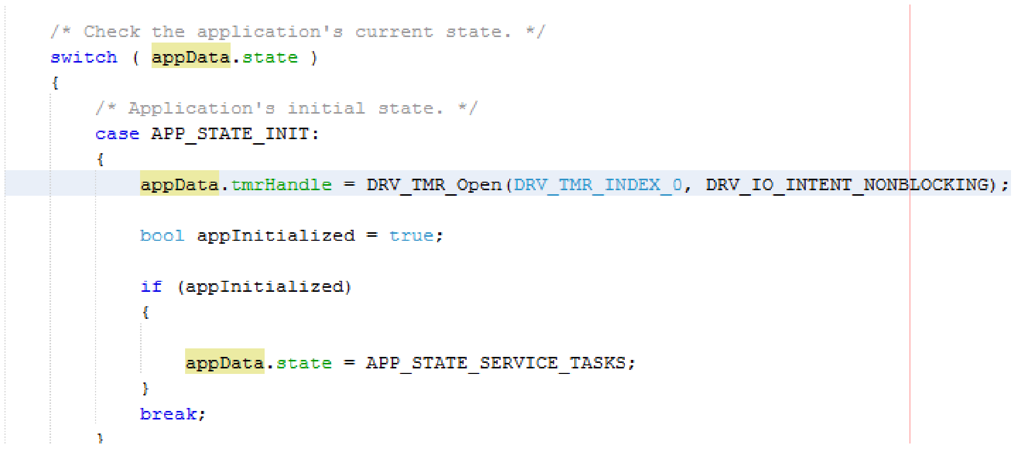


```
 */

typedef struct
{
    /* The application's current state */
    APP_STATES state;

    /* TODO: Define any additional data used by the app
    DRV_HANDLE tmrHandle;

} APP_DATA;
```

Code snippet:

```
DRV_HANDLE tmrHandle;
```

2. Open **`app.c`** file and navigate to "**`APP_STATE_INIT`**" state in **`APP_Tasks`** function. Call **`DRV_TMR_Open`** function to link "**`tmrHandle`**" and "**`DRV_TMR_INDEX_0`**" (timer driver instance 0) that is enabled in the MHC.



```
    /* Check the application's current state. */
    switch ( appData.state )
    {
        /* Application's initial state. */
        case APP_STATE_INIT:
        {
            appData.tmrHandle = DRV_TMR_Open(DRV_TMR_INDEX_0, DRV_IO_INTENT_NONBLOCKING);

            bool appInitialized = true;

            if (appInitialized)
            {

                appData.state = APP_STATE_SERVICE_TASKS;
            }
            break;
        }
```

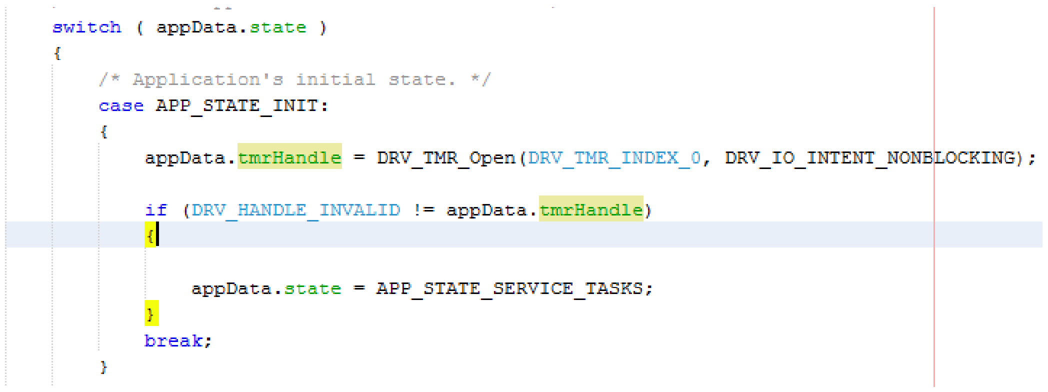
Code snippet:

```
appData.tmrHandle = DRV_TMR_Open(DRV_TMR_INDEX_0, DRV_IO_INTENT_NONBLOCKING);
```

3. Check if you are able to get a valid driver handle. If so, then update application state to "**`APP_STATE_SERVICE_TASKS`**", otherwise stay in the same state.

   To do this, remove the default line:
   "**`bool appInitialized = true;`**"

   And "**`if(appInitialized)`**" logic has to be replaced with
   "**`if(appData.tmrHandle)`**" logic as shown below.



```
switch ( appData.state )
{
    /* Application's initial state. */
    case APP_STATE_INIT:
    {
        appData.tmrHandle = DRV_TMR_Open(DRV_TMR_INDEX_0, DRV_IO_INTENT_NONBLOCKING);

        if (DRV_HANDLE_INVALID != appData.tmrHandle)
        {

            appData.state = APP_STATE_SERVICE_TASKS;
        }
        break;
    }
```

   Code snippet:

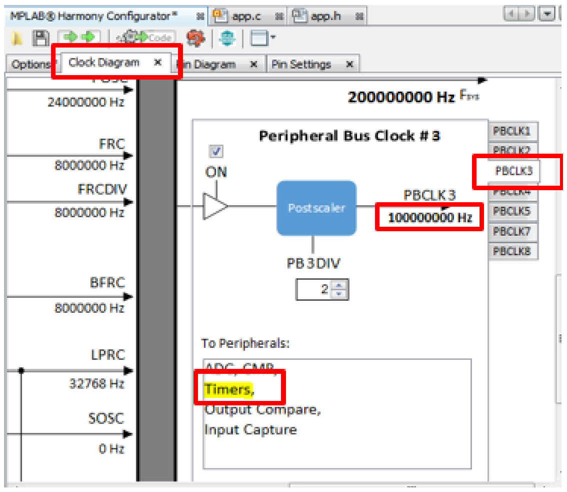
```
if (DRV_HANDLE_INVALID != appData.tmrHandle)
{
    appData.state = APP_STATE_SERVICE_TASKS;
}
```

Step 3: Setup clock and calculate Timer period for 1 second delay.

Procedure:

The timer driver can provide a periodic callback when the desired time period has expired. To use this feature, the application must set an "alarm" in the timer driver. The desired alarm period is configured by defining the required number of timer's input clock cycles that the timer should count before executing the callback function. The bus clock frequency must be known in order to calculate the desired alarm period.

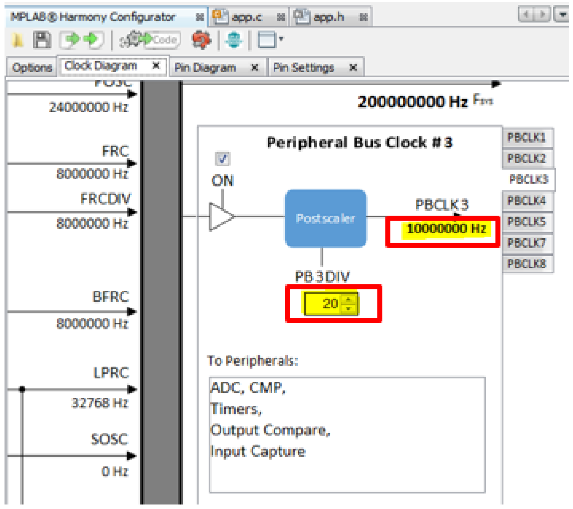
1. The Timer driver by default is configured for "**`DRV_TMR_CLK_SRC_INTERNAL`**" (Please see the Timer driver options in the MHC). Let's understand the peripheral clock connected to the Timer. You can easily find it by using the MHC ***Clock diagram*** as shown on the right. Try selecting other peripheral clock number and see the peripherals connected.



In the ***Clock diagram***, it will be noted that Timers are connected to "*Peripheral Bus Clock(PBCLK) 3"*. Note down the frequency of "*PBCLK3"*.

PBCLK3 = 100000000 Hz (100 MHz)

With this frequency you won't be able to achieve 1 second delay with a 16-bit Timer (Timer 1 feature), so lower the "*PBCLK3"* by changing the "*PB3DIV"* divider in the MHC clock manager to 20 and make sure to press *ENTER* to refresh the value. Now, the new value of "*PBCLK3"* should be 10MHz.



PBCLK3(new) = 10000000 Hz (10 MHz)

The Timer driver by default is configured for "**`TMR_PRESCALE_VALUE_256`**" (Please see the Timer driver options in the MHC). So, the clock source (or the clock ticks per second) available for the Timer module is,
Peripheral Clock / Prescale Value.

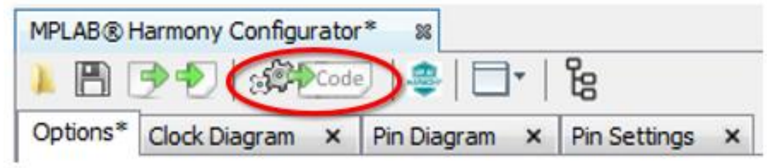
(10000000 / 256) = 39062 or 0x9892.

That is, the Timer module will count 0x9892 per second. Hence, you must set Timer period to 0x9892 to get Timer expiry event for every 1 second.

Timer Period value = 0x9892

Note: No MHC update is needed for the Timer driver. We will use the **"***Timer Period value***"** later to set the period using the Timer driver API.

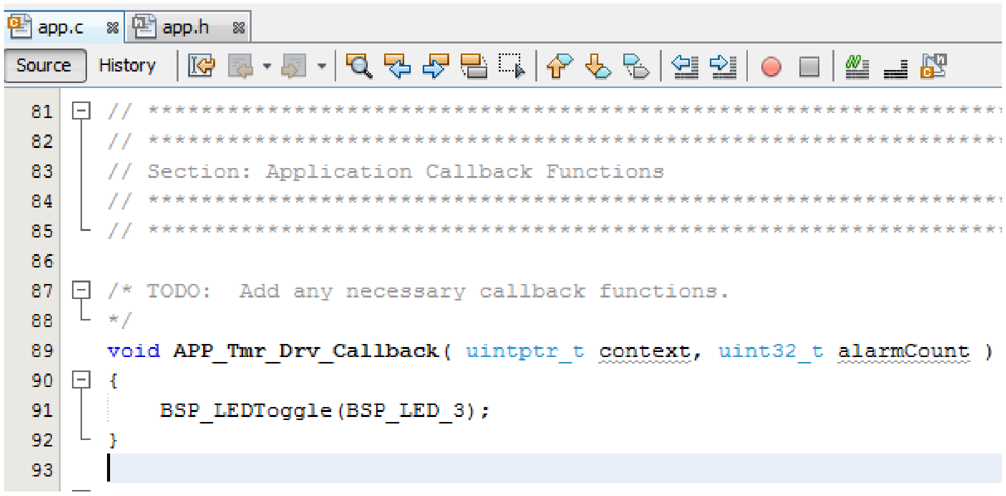
2. Regenerate the code as clock settings are updated. Click ***Save > Generate***.



Step 4: Register a callback function and request for a periodic alarm every 1 second.

Procedure:

1. Define a callback function in **`app.c`** under "*Section: application callback functions*".
   As discussed in the lab overview, you will Toggle LED 3 periodically. For this, implement the callback function as shown below,

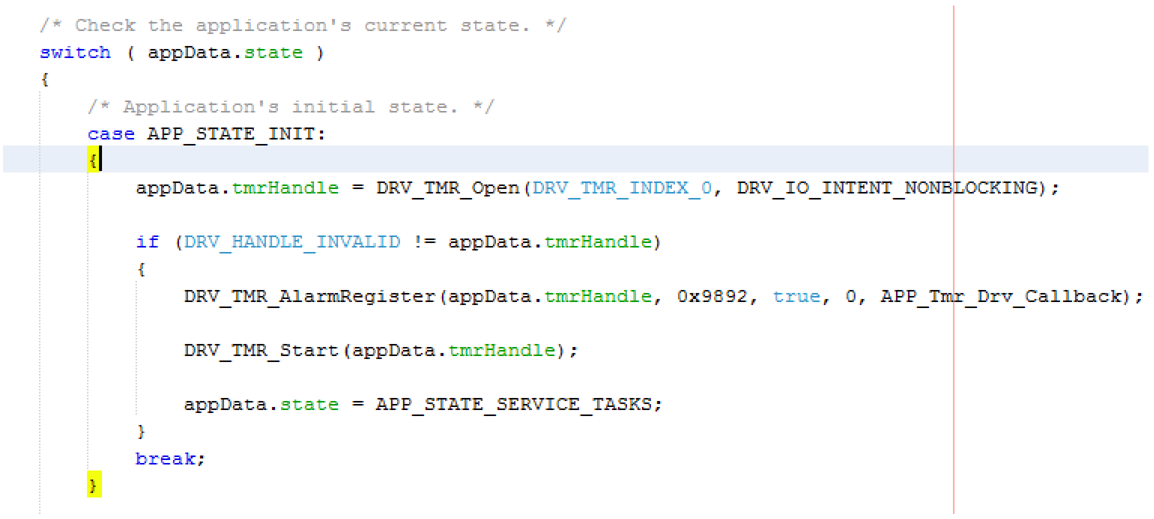


```
// *****************************************************************
// *****************************************************************
// Section: Application Callback Functions
// *****************************************************************
// *****************************************************************

/* TODO:  Add any necessary callback functions.
*/
void APP_Tmr_Drv_Callback( uintptr_t context, uint32_t alarmCount )
{
    BSP_LEDToggle(BSP_LED_3);
}

```

Code snippet:

```
void APP_Tmr_Drv_Callback( uintptr_t context, uint32_t alarmCount )
{
    BSP_LEDToggle(BSP_LED_3);
}
```

2. After the callback function is defined, register this callback function with the Timer driver. The Timer driver provides an API, **`DRV_TMR_AlarmRegister`** function to register an alarm as well as the callback function. Call this API in the application initial state "**`APP_STATE_INIT`**" and start the Timer as shown below. **Note:** Notice that the Timer driver APIs are called only if the handle obtained from **`DRV_TMR_Open`** function is valid.



```
/* Check the application's current state. */
switch ( appData.state )
{
    /* Application's initial state. */
    case APP_STATE_INIT:
    {
        appData.tmrHandle = DRV_TMR_Open(DRV_TMR_INDEX_0, DRV_IO_INTENT_NONBLOCKING);

        if (DRV_HANDLE_INVALID != appData.tmrHandle)
        {
            DRV_TMR_AlarmRegister(appData.tmrHandle, 0x9892, true, 0, APP_Tmr_Drv_Callback);

            DRV_TMR_Start(appData.tmrHandle);

            appData.state = APP_STATE_SERVICE_TASKS;
        }
        break;
    }
```

Code snippet:

```
DRV_TMR_AlarmRegister(appData.tmrHandle, 0x9892, true, 0, APP_Tmr_Drv_Callback);

DRV_TMR_Start(appData.tmrHandle);
```

3. Click the "*Debug*" button to build, program and run (in debug mode) the application. The application logic implemented in Lab 1 continues to operate unaffected. That is, LED 2 should be ON by default and pressing SWITCH 1 toggles LED 1. Along with this operation, notice the Lab 2 output. That is LED 3 toggles at 1 second intervals, using the timer driver.

Step 5: **Explore the driver working flow.**

In Lab 1, you have explored the project layout and execution flow of Harmony project. As a continuation, let’s see how interrupt handlers(ISRs) are managed and how the driver callback function works.

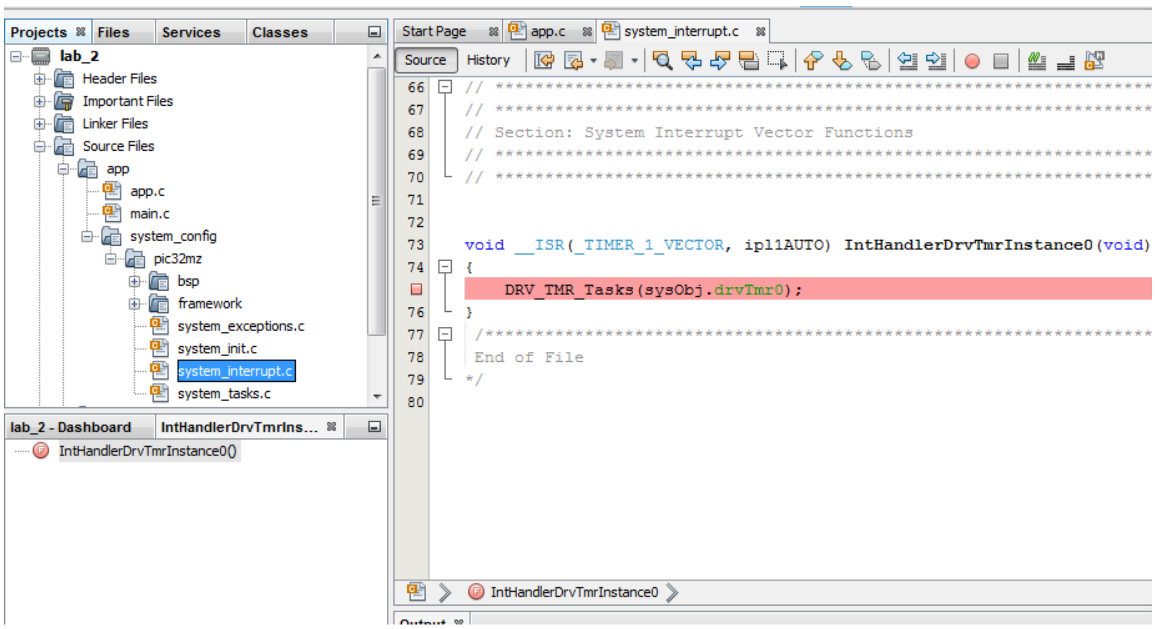
Procedure:

1. Reset the Debug session:

   a. First, click the “*Pause”* button on the IDE toolbar.

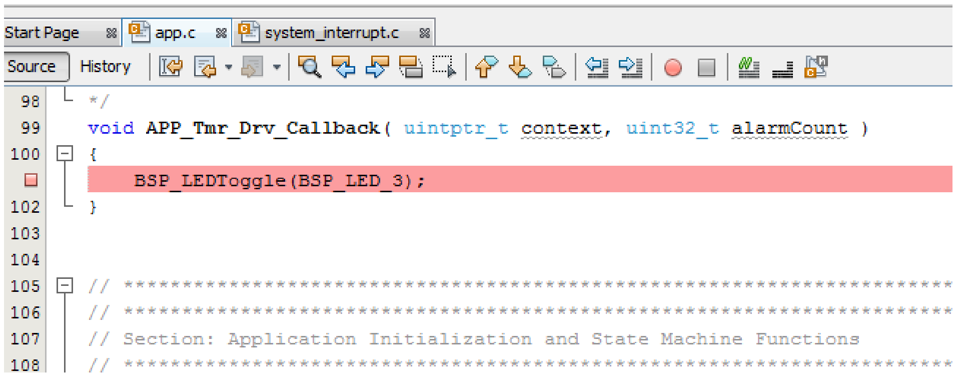
   b. Then, click the “*Reset”* button on the IDE toolbar.

2. Set a breakpoint at the Timer ISR as shown below by navigating to `system_interrupt.c` file.

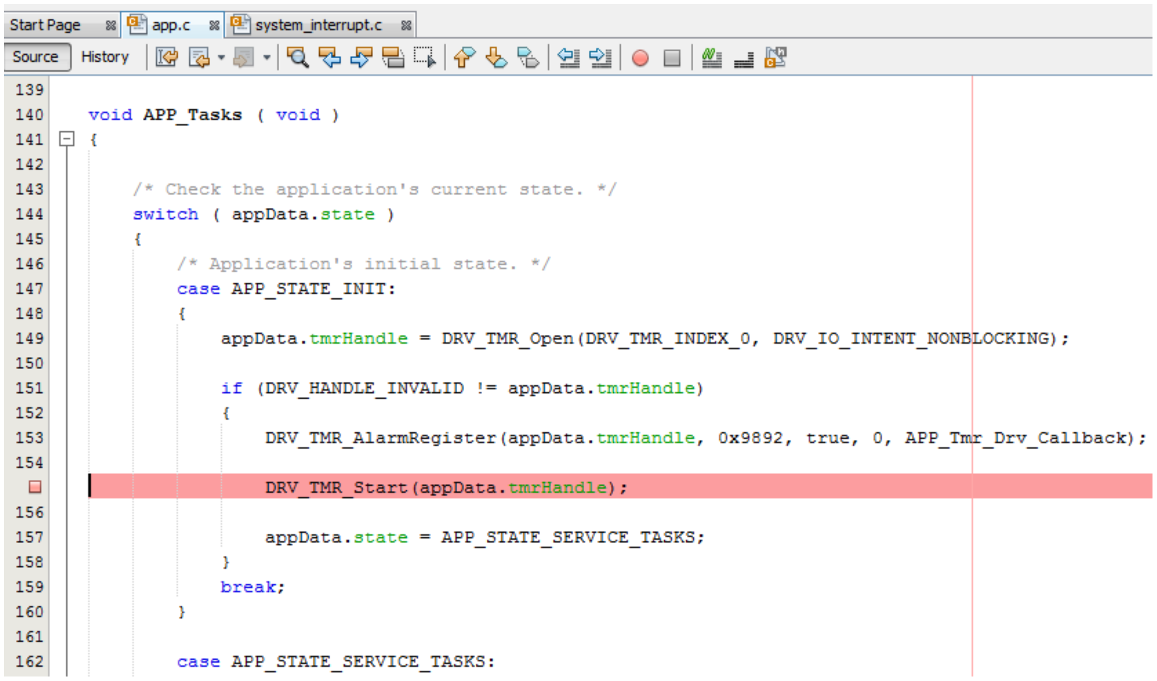


```
// *****************************************************************
// *****************************************************************
// Section: System Interrupt Vector Functions
// *****************************************************************
// *****************************************************************


void __ISR(_TIMER_1_VECTOR, ipl1AUTO) IntHandlerDrvTmrInstance0(void)
{
    DRV_TMR_Tasks(sysObj.drvTmr0);
}
 /*****************************************************************
 End of File
*/
```

3. Set two more breakpoints in **`app.c`**, one at **`DRV_TMR_Start`** function and the other in the callback function as shown below.



```
*/
void APP_Tmr_Drv_Callback( uintptr_t context, uint32_t alarmCount )
{
    BSP_LEDToggle(BSP_LED_3);
}


// *****************************************************************************
// *****************************************************************************
// Section: Application Initialization and State Machine Functions
// *****************************************************************************
```



```

void APP_Tasks ( void )
{

    /* Check the application's current state. */
    switch ( appData.state )
    {
        /* Application's initial state. */
        case APP_STATE_INIT:
        {
            appData.tmrHandle = DRV_TMR_Open(DRV_TMR_INDEX_0, DRV_IO_INTENT_NONBLOCKING);

            if (DRV_HANDLE_INVALID != appData.tmrHandle)
            {
                DRV_TMR_AlarmRegister(appData.tmrHandle, 0x9892, true, 0, APP_Tmr_Drv_Callback);

                DRV_TMR_Start(appData.tmrHandle);

                appData.state = APP_STATE_SERVICE_TASKS;
            }
            break;
        }

        case APP_STATE_SERVICE_TASKS:
```

Now you are all set to Run the program.

4. Click the “*Run”* button ▶ to run the application in debug mode.

    a. The program control will execute the **`DRV_TMR_Start`** function first where it starts Timer 1. Observe that you are able to get a proper handle from **`DRV_TMR_Open`** function and relation is established between the driver instance and the application.

    b. Continue to “*Run”* ▶. As “interrupt mode” is the default configuration for the Timer driver in MHC, the program control hits Timer 1 Interrupt Handler when the timer period expires.

    c. The interrupt handler calls the **`DRV_TMR_Tasks`** function, which will service the interrupt and will call the registered callback function.

    d. Continue to “*Run”* to enter the callback function **`APP_Tmr_Drv_Callback`** breakpoint. Note that this is the alarm callback function.

    e. Now, relate the flow with the flow diagram discussed in the “Using Drivers” section of the presentation. Use debug reset and continue once again to understand the flow.

5. Delete all the breakpoints and end the debug session.

    a. Go to “*Breakpoints”* window and right click to select “*Delete All”* as done in Lab 1.

    b. Click on “*Stop*” ■ button in MPLAB® X toolbar.

********************** Congratulations! You’re done with Lab 2 **********************

You have gained experience on how to configure drivers using MHC and how to use them in the Harmony application.

****

Extra Credit: **Try configuring different delay.**

Calculate the Timer period value for a different delay, let’s say 0.5 seconds and try it using the Timer driver.

Notes:

LAB 3

Using Harmony System Services –Timer System Service to Implement Delays

LAB 3: Using Harmony System Services – Timer System Service to Implement Delays

Purpose:

After completing Lab 3, you will have hands-on experience of how to configure Harmony system services in the MHC and how to use it in your application. You will keep the system service logic simple and basic for this lab. You can build on this foundational knowledge later.

Overview:

In this lab, you will start with a new Harmony project. Using the MHC, you will select and configure the Timer system service. And use it to implement two different periodic delays using one Timer peripheral.
For extra credit, you can configure and use Console system service to send log to the computer over UART every time the LED is toggled.

This lab is completed in the following steps:

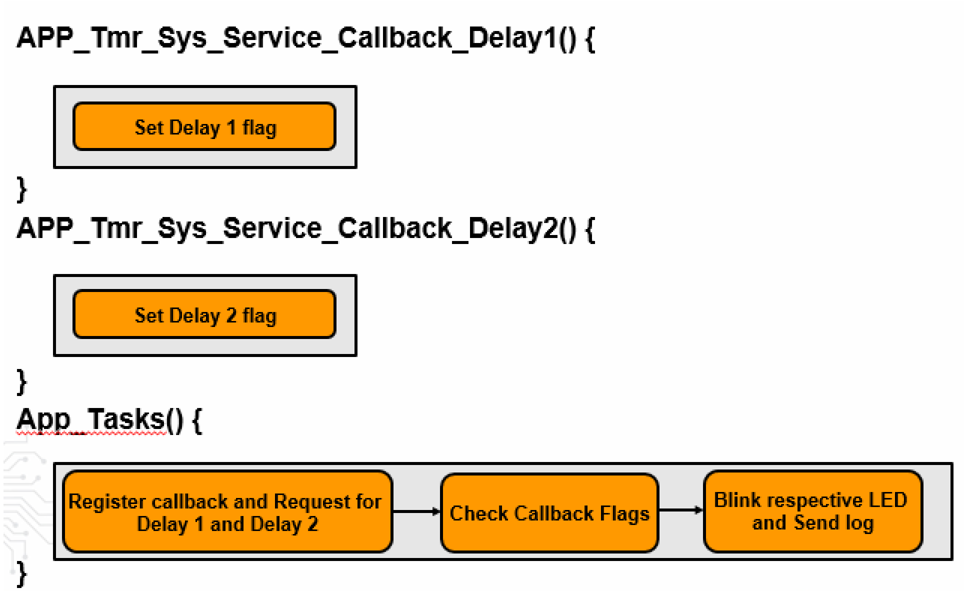
Step 1: Create a new project
Step 2: Enable and configure Timer system service in the MHC
Step 3: Request for two periodic alarms every 0.5 and 2 seconds respectively
Step 4: Toggle LED 1 and LED 3 for 0.5 and 2 seconds callback respectively.
Step 5: Run the application.

Extra Credit 1: Using Console system service.
- Step 1: Enable and configure Console system service in the MHC
- Step 2: Send log messages using Console system service for each delay expiry
- Step 3: Run the application.

Extra Credit 2: Try different delays.

Note: Each step has a short procedure that you must follow to complete the step.

Application logic:


APP_Tmr_Sys_Service_Callback_Delay1() {
Set Delay 1 flag
}
APP_Tmr_Sys_Service_Callback_Delay2() {
Set Delay 2 flag
}
App_Tasks() {
Register callback and Request for Delay 1 and Delay 2
Check Callback Flags
Blink respective LED and Send log
}

Step 1: Create a new project

Procedure:

This step will be a refresher on how to generate a Harmony project.

1. Start by creating a new Harmony project by following "*Step 2*" of Lab 1 (page 8), use "***21026_dev5_lab_3_4***" instead of "*21016_dev5_lab_1_2*" for the project name.

2. Generate the empty project, you will add system service to the project in the next steps.

Step 2: Enable and configure Timer system service in the MHC

Procedure:

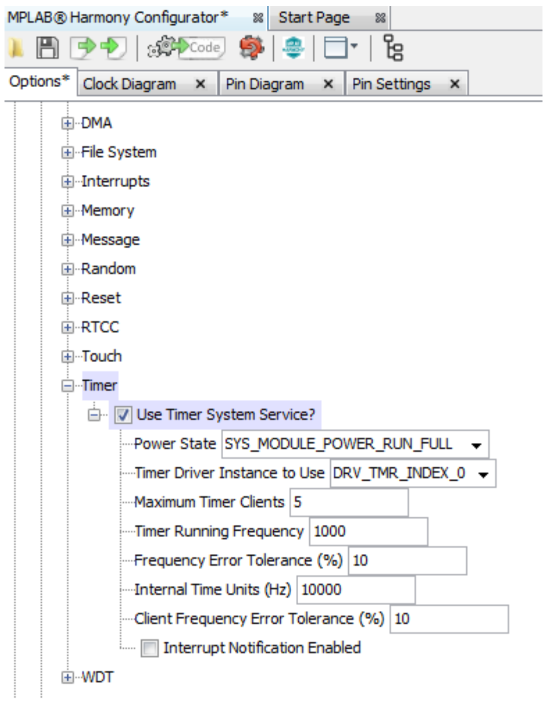
1. In MHC Options window, navigate to "*Harmony Framework Configuration*" then "*System Services*". Scroll down to "*Timer*" and expand the menu. Enable it as shown on the right.

2. Keep the default configuration of Timer system service, which is the commonly used configuration for timers. Feel free to check options and the corresponding help.

   a. As you are toggling two LEDs at two different delays, you can set "*Maximum Timer Clients*" to 2.

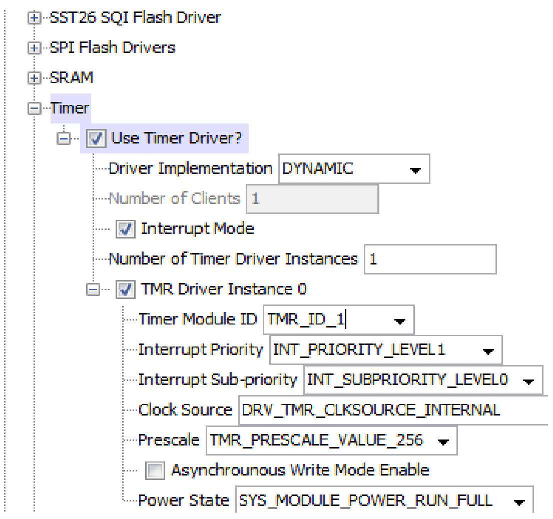
   b. Check the help for each option for more details.

   c. Notice that Timer system service uses Timer driver instance 0, that is "**`DRV_TMR_INDEX_0`**" as a timer source. The driver will be enabled and configured by system service itself.



- DMA
- File System
- Interrupts
- Memory
- Message
- Random
- Reset
- RTCC
- Touch
- Timer
  - [✓] Use Timer System Service?
    - Power State: SYS_MODULE_POWER_RUN_FULL
    - Timer Driver Instance to Use: DRV_TMR_INDEX_0
    - Maximum Timer Clients: 5
    - Timer Running Frequency: 1000
    - Frequency Error Tolerance (%): 10
    - Internal Time Units (Hz): 10000
    - Client Frequency Error Tolerance (%): 10
    - [ ] Interrupt Notification Enabled
- WDT

3. Verify the Timer driver configuration as you did in Lab 2. Navigate to "*Harmony Framework Configuration*" then "*Drivers*", then "Timer". Selecting the Timer system service also configures the needed Timer driver settings. So, you don't have to update anything in the Timer driver.



```
SST26 SQI Flash Driver
SPI Flash Drivers
SRAM
Timer
  [✓] Use Timer Driver?
      Driver Implementation  DYNAMIC
      Number of Clients  1
      [✓] Interrupt Mode
      Number of Timer Driver Instances  1
      [✓] TMR Driver Instance 0
          Timer Module ID  TMR_ID_1
          Interrupt Priority  INT_PRIORITY_LEVEL1
          Interrupt Sub-priority  INT_SUBPRIORITY_LEVEL0
          Clock Source  DRV_TMR_CLKSOURCE_INTERNAL
          Prescale  TMR_PRESCALE_VALUE_256
          [ ] Asynchrounous Write Mode Enable
          Power State  SYS_MODULE_POWER_RUN_FULL
```

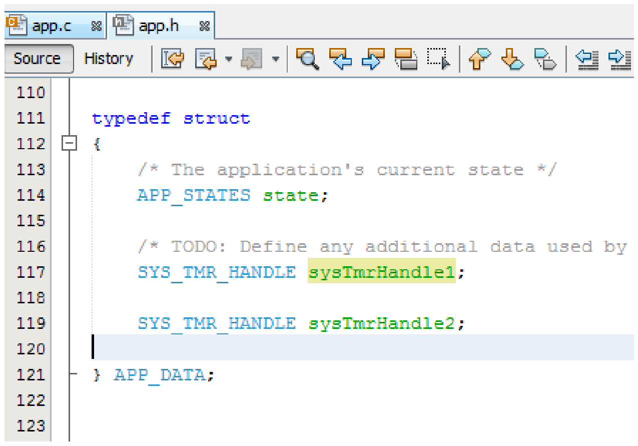
4. Click on "*Generate code*" Code when ready. Now, the Timer system service is ready to be used in your application.

Step 3: Request for two periodic alarms every 0.5 and 2 seconds respectively

Procedure:

The Timer system service periodic callback request function returns a "*handle*". This is the handle for the delay requested.

1. Open the **`app.h`** file. As you need two periodic delays in your application, declare two application variables "**`sysTmrHandle1`**" and "**`sysTmrHandle2`**" of type "**`SYS_TMR_HANDLE`**" in **`APP_DATA`** as shown on the right for Delay 1(0.5 second) and Delay 2 (2 seconds) respectively.



```
typedef struct
{
    /* The application's current state */
    APP_STATES state;

    /* TODO: Define any additional data used by
    SYS_TMR_HANDLE sysTmrHandle1;

    SYS_TMR_HANDLE sysTmrHandle2;

} APP_DATA;
```

Here you are adding data to the app data structure that you have already used in Lab 1 and Lab 2.

Code snippet:

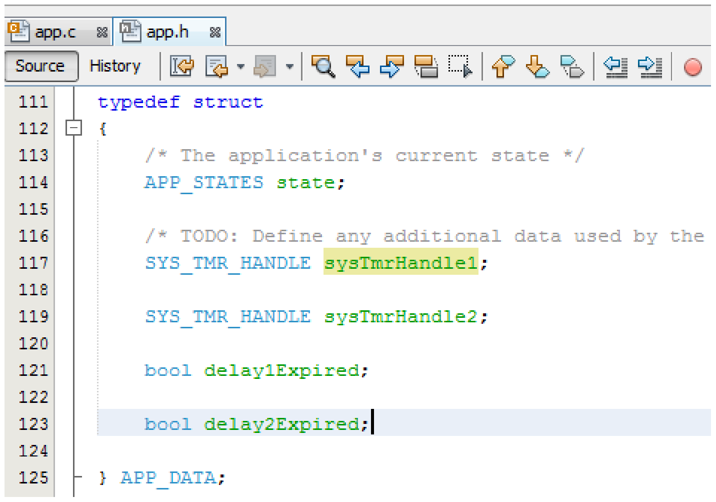
```
SYS_TMR_HANDLE sysTmrHandle1;

SYS_TMR_HANDLE sysTmrHandle2;
```

2. Define the callback functions.

   a. Unlike Lab 2, instead of toggling the LED in the callback function itself, you will set a flag here and then toggle the LED in the main **`APP_Tasks`** function loop based on the flag status.

      Note: The callback function commonly executes in an interrupt context. It should therefore be treated like an interrupt service routine. An application should not call blocking functions or perform computationally intensive functions in the callback functions. Such tasks must be differed to **`APP_Tasks`** function.

   b. Declare two application flags "**`delay1Expired`**" and "**`delay2Expired`**" of type "**`bool`**" in **`APP_DATA`** as shown below.

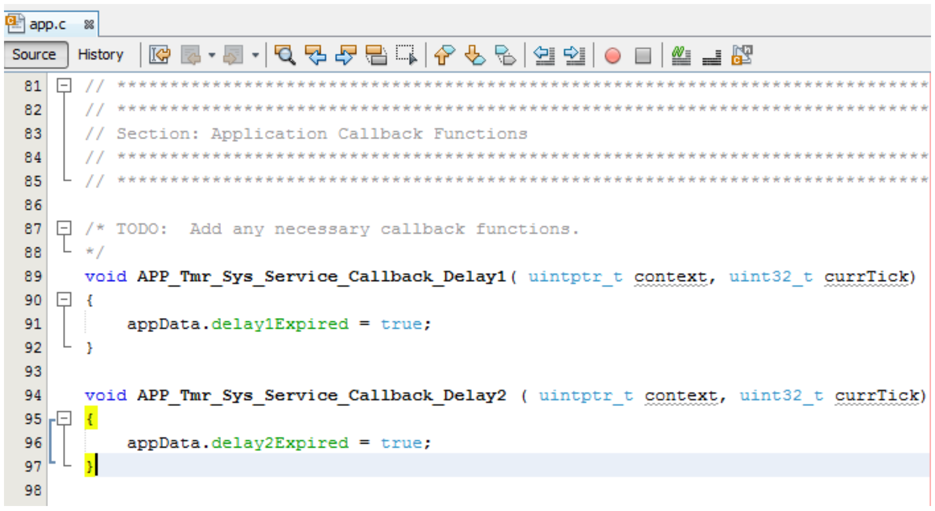


```
typedef struct
{
    /* The application's current state */
    APP_STATES state;

    /* TODO: Define any additional data used by the
    SYS_TMR_HANDLE sysTmrHandle1;

    SYS_TMR_HANDLE sysTmrHandle2;

    bool delay1Expired;

    bool delay2Expired;

} APP_DATA;
```

   Code snippet:

```
    bool delay1Expired;

    bool delay2Expired;
```

   c. Now, define the callback functions "**`APP_Tmr_Sys_Service_Callback_Delay1`**" and "**`APP_Tmr_Sys_Service_Callback_Delay2`**" in **`app.c`** file under callback functions section as shown below.

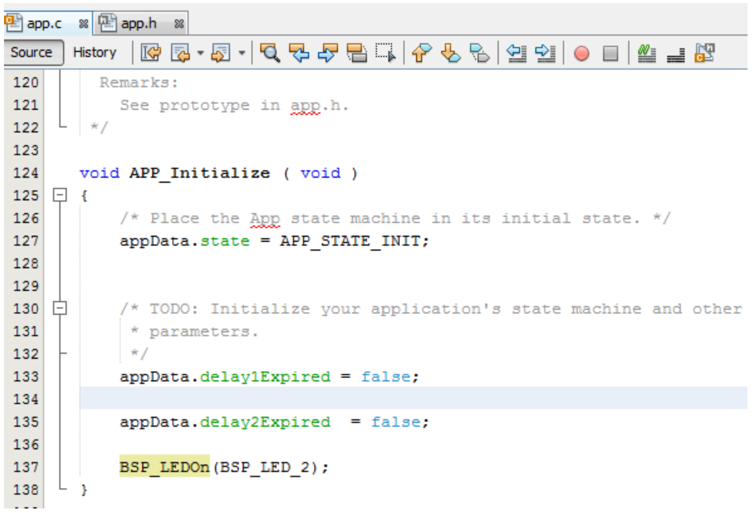


```
// *****************************************************************************
// *****************************************************************************
// Section: Application Callback Functions
// *****************************************************************************
// *****************************************************************************

/* TODO:  Add any necessary callback functions.
*/
void APP_Tmr_Sys_Service_Callback_Delay1( uintptr_t context, uint32_t currTick)
{
    appData.delay1Expired = true;
}

void APP_Tmr_Sys_Service_Callback_Delay2  ( uintptr_t context, uint32_t currTick)
{
    appData.delay2Expired = true;
}
```

Code snippet:

```
void APP_Tmr_Sys_Service_Callback_Delay1( uintptr_t context, uint32_t currTick)
{
    appData.delay1Expired = true;
}

void APP_Tmr_Sys_Service_Callback_Delay2( uintptr_t context, uint32_t currTick)
{
    appData.delay2Expired = true;
}
```

3. Initialize the application by setting the delay flags "**delay1Expired**" and "**delay2Expired**" to false and turning on LED 2 in **APP_Initialize** function as shown on the right.



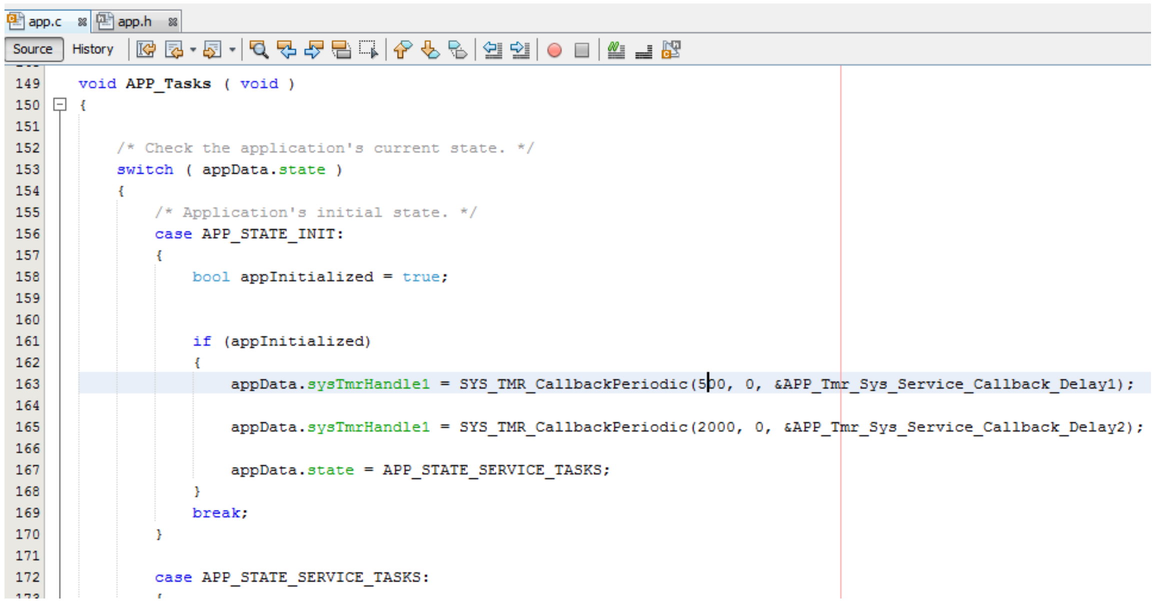
```
    Remarks:
      See prototype in app.h.
  */

void APP_Initialize ( void )
{
    /* Place the App state machine in its initial state. */
    appData.state = APP_STATE_INIT;


    /* TODO: Initialize your application's state machine and other
     * parameters.
     */
    appData.delay1Expired = false;

    appData.delay2Expired  = false;

    BSP_LEDOn(BSP_LED_2);
}
```

Code snippet:

```
appData.delay1Expired = false;

appData.delay2Expired = false;

BSP_LEDOn(BSP_LED_2);
```

4. Now that the callback functions are defined and the app is initialized, register these callback functions with delay 1(0.5 second) and delay 2 (2 seconds) using Timer system service request API **`SYS_TMR_CallbackPeriodic`** in "**`APP_STATE_INIT`**" state of **`APP_Tasks`** function as shown below.

   Note: Notice that there is no need to Open the Timer driver as the Timer system service takes care of this internally.
   Timer system service need required delay information in milli-seconds unit. Hence, use values 500 and 2000 for 0.5 and 2 seconds respectively.



```
void APP_Tasks ( void )
{

    /* Check the application's current state. */
    switch ( appData.state )
    {
        /* Application's initial state. */
        case APP_STATE_INIT:
        {
            bool appInitialized = true;


            if (appInitialized)
            {
                appData.sysTmrHandle1 = SYS_TMR_CallbackPeriodic(500, 0, &APP_Tmr_Sys_Service_Callback_Delay1);

                appData.sysTmrHandle1 = SYS_TMR_CallbackPeriodic(2000, 0, &APP_Tmr_Sys_Service_Callback_Delay2);

                appData.state = APP_STATE_SERVICE_TASKS;
            }
            break;
        }

        case APP_STATE_SERVICE_TASKS:
```

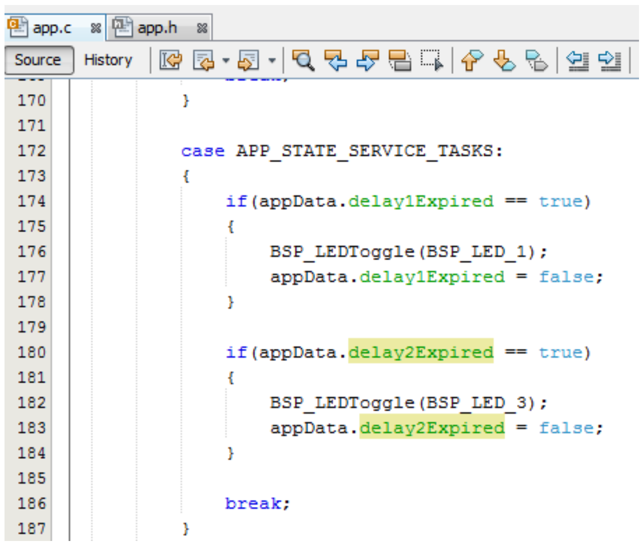
Code snippet:

```
appData.sysTmrHandle1 = SYS_TMR_CallbackPeriodic(500, 0,
&APP_Tmr_Sys_Service_Callback_Delay1);

appData.sysTmrHandle1 = SYS_TMR_CallbackPeriodic(2000, 0,
&APP_Tmr_Sys_Service_Callback_Delay2);
```

Step 4: Toggle LED 1 and LED 3 for 0.5 and 2 seconds callback respectively.

Procedure:

1. Check flags "**`delay1Expired`**" and "**`delay2Expired`**". If they are set, then toggle LED 1 and LED 3 respectively in "**`APP_STATE_SERVICE_TASKS`**" state of **`APP_Tasks`** function as shown below.



```
            }

            case APP_STATE_SERVICE_TASKS:
            {
                if(appData.delay1Expired == true)
                {
                    BSP_LEDToggle(BSP_LED_1);
                    appData.delay1Expired = false;
                }

                if(appData.delay2Expired == true)
                {
                    BSP_LEDToggle(BSP_LED_3);
                    appData.delay2Expired = false;
                }

                break;
            }
```

Code snippet:

```
if(appData.delay1Expired == true)
{
    BSP_LEDToggle(BSP_LED_1);
    appData.delay1Expired = false;
}

if(appData.delay2Expired == true)
{
    BSP_LEDToggle(BSP_LED_3);
    appData.delay2Expired = false;
}
```

Step 5: Run the Application.

Procedure:
With all the MHC configuration done, code generated and application logic added, now run the demo.

1. Click on the “*Program”* button to build, program and run the application.

2. Select the correct hardware (“*PIC32MZ EF Family*”) in the pop-up window.

3. LED 2 should be lit by default to show the system is ON. LED 1 and LED 3 should be toggling at 0.5 and 2 second intervals using the same Timer peripheral (Timer 1) without any conflict.

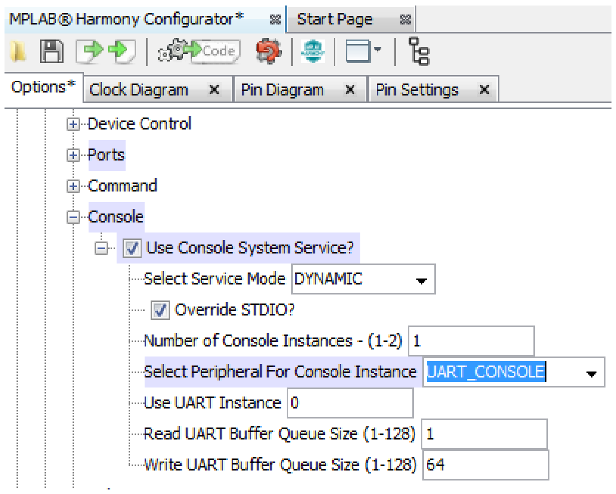
******************** **Congratulations! You’re done with Lab 3** ********************
You have learned how to configure system service using MHC and how to use it in the Harmony application.
****

Extra Credit 1: Using Console system service

Step 1: Enable and configure Console system service in the MHC

Procedure:

1. Go to "*Options*" window and navigate to "*Harmony Framework Configuration*" then "*System Services*". Scroll down to "*Console*" and expand the menu. Enable and Configure it as shown below.



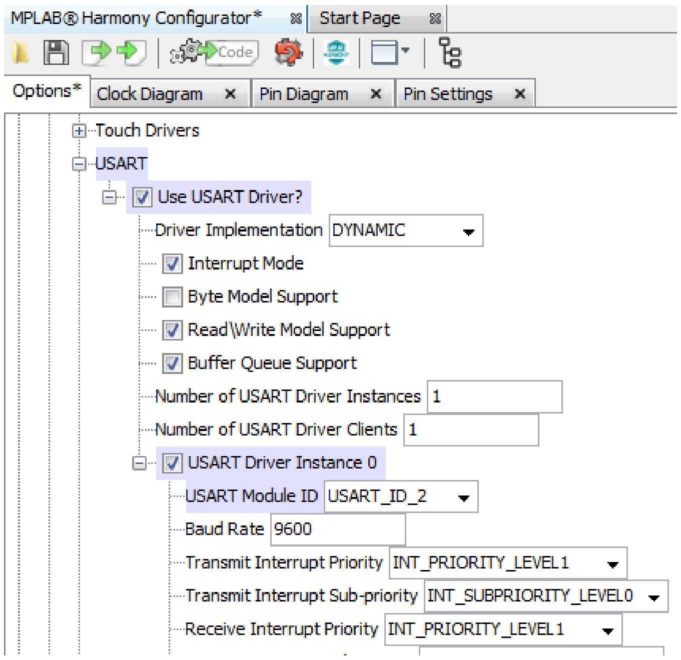
- [✓] Use Console System Service?
  - Select Service Mode: DYNAMIC
  - [✓] Override STDIO?
  - Number of Console Instances - (1-2): 1
  - Select Peripheral For Console Instance: UART_CONSOLE
  - Use UART Instance: 0
  - Read UART Buffer Queue Size (1-128): 1
  - Write UART Buffer Queue Size (1-128): 64

2. Keep the default configuration of Console system service, which is the commonly used configuration for Console with just one update.

    a. The Console uses UART as the communication medium. Therefore update the "*Select Peripheral For Console Instance*" to "**`UART_CONSOLE`**".

    Note: Accept the default write queue size of 64, as the application performs several console write operations.
    Check the Harmony help window pane for the more details on each option.

3. Check the UART driver configuration, navigate to "*Harmony Framework Configuration*" > "*Drivers*" > "*USART*". Selecting the Console system service automatically configures USART driver as needed.



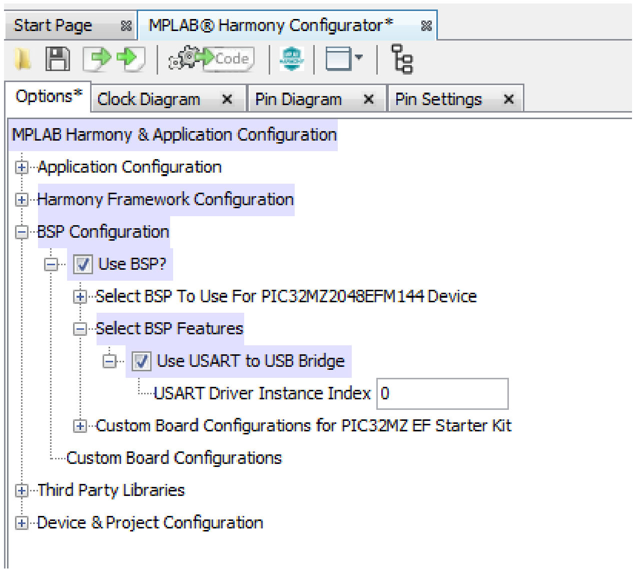
Touch Drivers
USART
[✓] Use USART Driver?
Driver Implementation DYNAMIC
[✓] Interrupt Mode
[ ] Byte Model Support
[✓] Read\Write Model Support
[✓] Buffer Queue Support
Number of USART Driver Instances 1
Number of USART Driver Clients 1
[✓] USART Driver Instance 0
USART Module ID USART_ID_2
Baud Rate 9600
Transmit Interrupt Priority INT_PRIORITY_LEVEL1
Transmit Interrupt Sub-priority INT_SUBPRIORITY_LEVEL0
Receive Interrupt Priority INT_PRIORITY_LEVEL1

4. Update the "*USART Module ID*" to **`USART_ID_2`** as shown below. This is needed because, the USB-to-USART converter on the Starter Kit is connected to USART 2 of the PIC32MZ device.

5. The PIC32MZ EF device port pins have Peripheral Pin Select (PPS) feature. This requires the USART TX and RX functions to be mapped to the appropriate pins. As discussed in the class, the ***MHC Pin manager*** plugin allows selecting the pin functions graphically. Additionally, the BSP options also allow easy selection of this feature.
In this lab, you will use the BSP option to select the pins and then verify the selection in the pin table and pin diagram windows.

   a. Select USART to USB bridge option under BSP as shown on the right

      Note: Selecting this BSP feature automatically sets the USART module ID option of the selected USART driver instance to **`USART_ID_2`**.

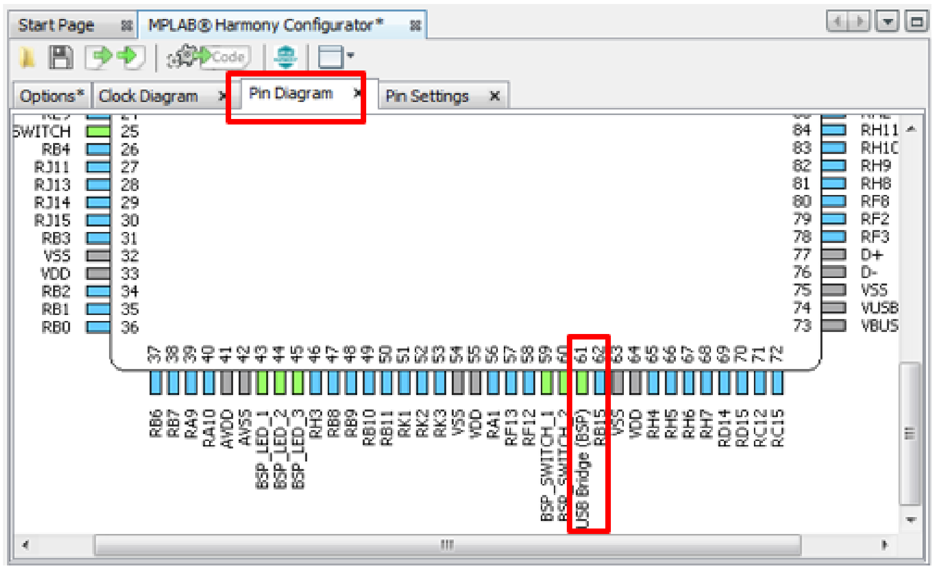


MPLAB Harmony & Application Configuration
Application Configuration
Harmony Framework Configuration
BSP Configuration
[✓] Use BSP?
Select BSP To Use For PIC32MZ2048EFM144 Device
Select BSP Features
[✓] Use USART to USB Bridge
USART Driver Instance Index 0
Custom Board Configurations for PIC32MZ EF Starter Kit
Custom Board Configurations
Third Party Libraries
Device & Project Configuration

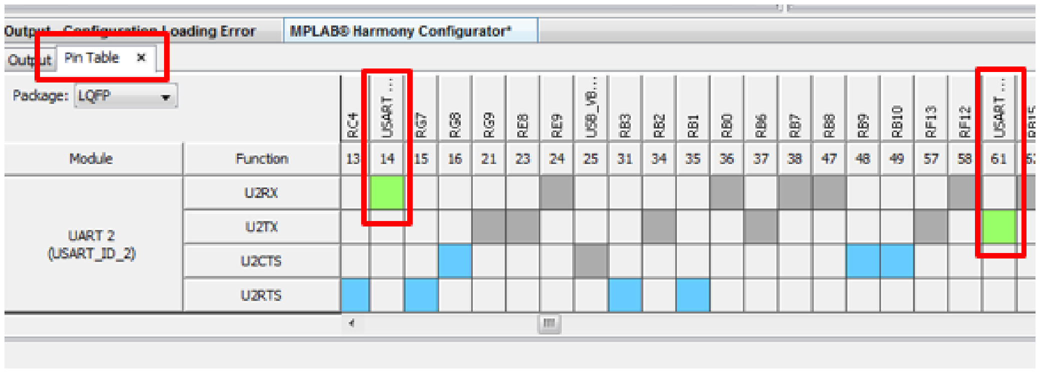
b. Verify that U2RX and U2TX pins are enabled:

U2RX	Pin 14	RG 6
U2TX	Pin 61	RB 14

Pin Diagram tab:



Pin Table tab (bottom window of MPLAB® X):


Output - Configuration Loading Error
MPLAB® Harmony Configurator*
Output
Pin Table

Package: LQFP

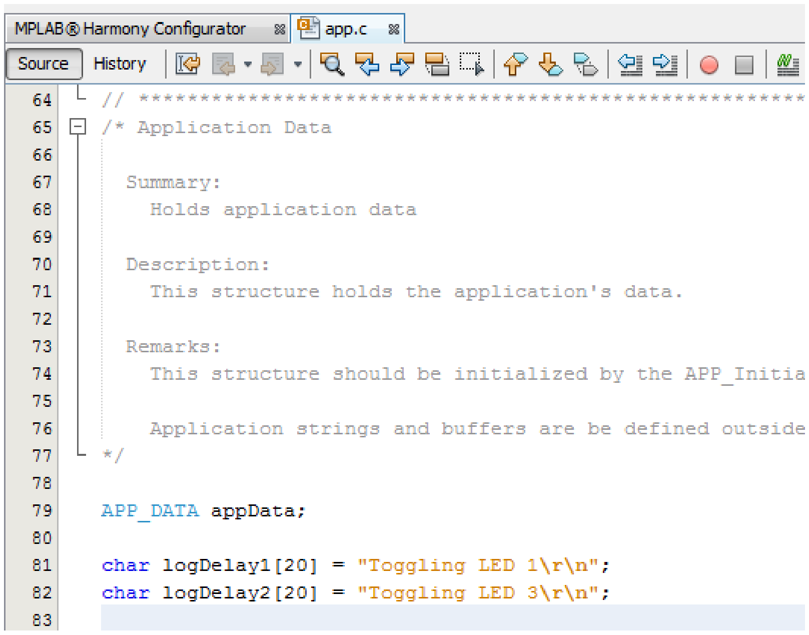
		RC4	USART ...	RG7	RG8	RG9	RE8	RE9	USB_VB...	RB3	RB2	RB1	RB0	RB6	RB7	RB8	RB9	RB10	RF13	RF12	USART ...
Module	Function	13	14	15	16	21	23	24	25	31	34	35	36	37	38	47	48	49	57	58	61
UART 2 (USART_ID_2)	U2RX		🟩					▨					▨		▨	▨				▨	
	U2TX					▨	▨				▨			▨					▨		🟩
	U2CTS				🟦				▨								🟦	🟦			
	U2RTS	🟦		🟦						🟦		🟦									

6. Click on "*Generate code*" Code when ready. Click ***Save > Generate.*** Now Console system service is ready to use in the application.

Step 2: Send log messages using Console system service for each delay expiry

Procedure:
Define log messages and send them to PC over UART using Console system service API.

1. Define the log messages as shown below.

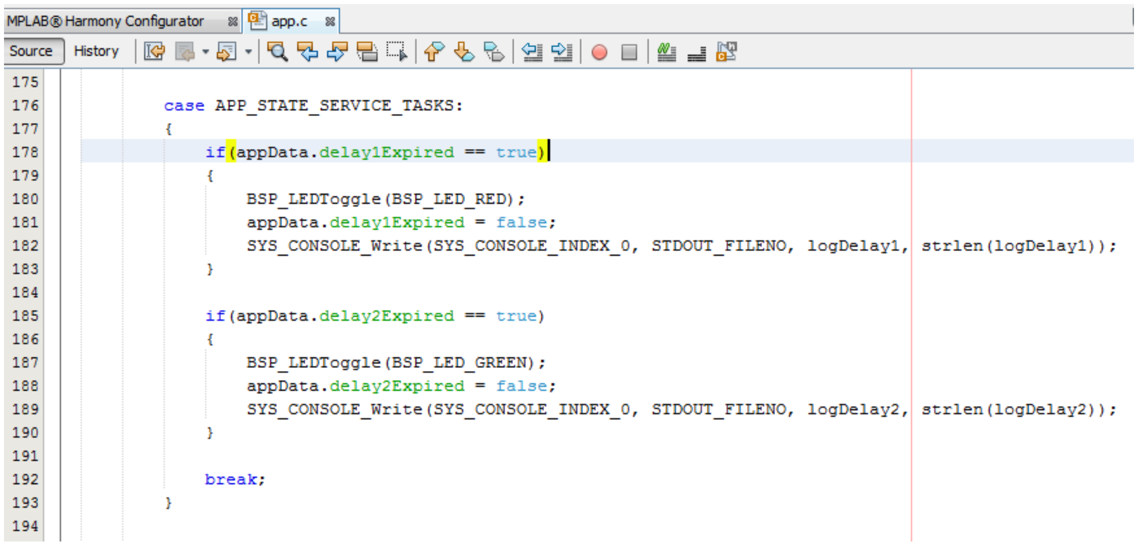


```
// ************************************************************
/* Application Data

  Summary:
    Holds application data

  Description:
    This structure holds the application's data.

  Remarks:
    This structure should be initialized by the APP_Initia

    Application strings and buffers are be defined outside
*/

APP_DATA appData;

char logDelay1[20] = "Toggling LED 1\r\n";
char logDelay2[20] = "Toggling LED 3\r\n";

```

Code snippet:

```
char logDelay1[20] = "Toggling LED 1\r\n";
char logDelay2[20] = "Toggling LED 3\r\n";
```

2. The **`APP_Tasks`** function already contains logic to check if the delay has expired and then toggle the LED. Use **`SYS_CONSOLE_Write`** function to send the respective log messages as shown below,



```
            case APP_STATE_SERVICE_TASKS:
            {
                if(appData.delay1Expired == true)
                {
                    BSP_LEDToggle(BSP_LED_RED);
                    appData.delay1Expired = false;
                    SYS_CONSOLE_Write(SYS_CONSOLE_INDEX_0, STDOUT_FILENO, logDelay1, strlen(logDelay1));
                }

                if(appData.delay2Expired == true)
                {
                    BSP_LEDToggle(BSP_LED_GREEN);
                    appData.delay2Expired = false;
                    SYS_CONSOLE_Write(SYS_CONSOLE_INDEX_0, STDOUT_FILENO, logDelay2, strlen(logDelay2));
                }

                break;
            }
```

Code snippet:

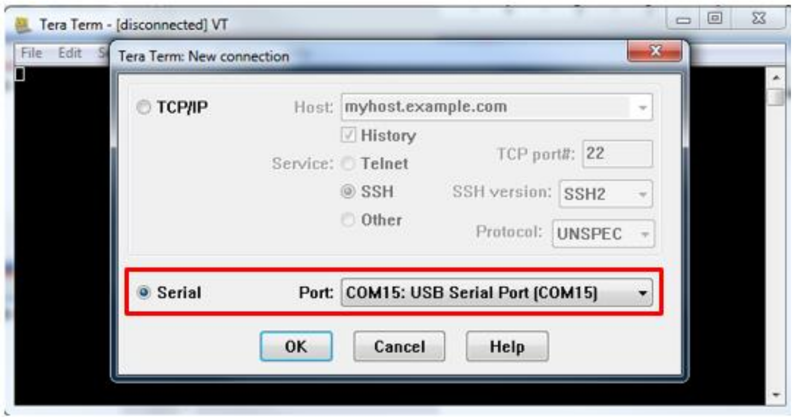
```
SYS_CONSOLE_Write(SYS_CONSOLE_INDEX_0, STDOUT_FILENO, logDelay1, strlen(logDelay1));
SYS_CONSOLE_Write(SYS_CONSOLE_INDEX_0, STDOUT_FILENO, logDelay2, strlen(logDelay2));
```

Step 3: Run the Application.

Procedure:
With all the required MHC configuration completed, code generated and added application logic, now setup hardware environment to run the demo.

1. Using the available USB cable, connect USB port J11 on the Starter Kit (mini-B connector beneath the Ethernet port) to one of your computer USB port.

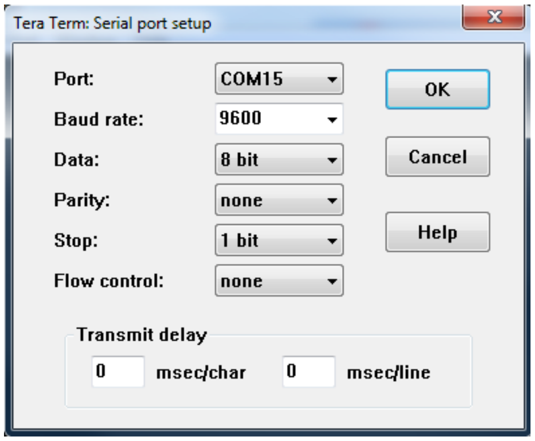
2. Launch “Tera Term”, serial terminal application on your Computer, which can be located on your desktop. Select the serial port connected as shown on the right.



TCP/IP Host: myhost.example.com
History
Service: Telnet
[✓ SSH]
Other
TCP port#: 22
SSH version: SSH2
Protocol: UNSPEC
[✓ Serial] Port: COM15: USB Serial Port (COM15)

Note: You may see a different COM port, that is OK.

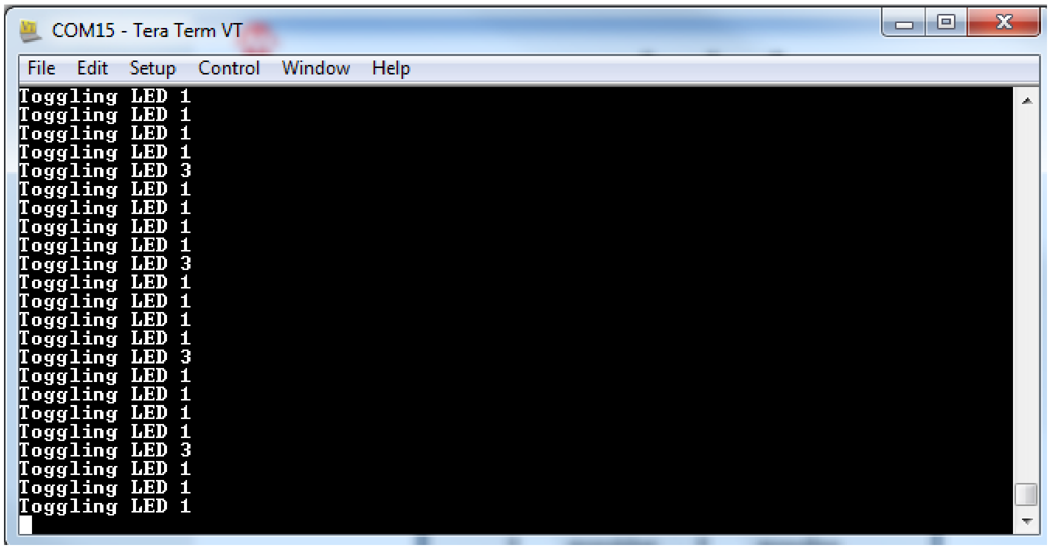
3. Keep the default serial settings on Tera Term, that is 9600 baud-rate and 8-N-1 frame setting. Go to ***Setup > Serial port*** in the Tera Term toolbar to verify the same as shown on the right.



Port:	COM15
Baud rate:	9600
Data:	8 bit
Parity:	none
Stop:	1 bit
Flow control:	none

Transmit delay
0 msec/char 0 msec/line

4. Go back to MPLAB® X and click the Program button to build, program and run the application. LED 2 should be lit by default to show system is ON. LED 1 and LED 3 should be toggling at 0.5 and 2 second intervals as configured. See the log messages on the Tera Term window running on your Computer.



```
Toggling LED 1
Toggling LED 1
Toggling LED 1
Toggling LED 1
Toggling LED 3
Toggling LED 1
Toggling LED 1
Toggling LED 1
Toggling LED 1
Toggling LED 3
Toggling LED 1
Toggling LED 1
Toggling LED 1
Toggling LED 1
Toggling LED 3
Toggling LED 1
Toggling LED 1
Toggling LED 1
Toggling LED 1
Toggling LED 3
Toggling LED 1
Toggling LED 1
Toggling LED 1
```

Extra Credit 2: Try different delays

Try using different delays and multiple delay requests for the Timer system service.

LAB 4
Create Custom BSP

LAB 4: Create Custom BSP

Purpose:

After completing Lab 4, you will have hands-on experience about how to create a new/custom Board Support Package (BSP).

Overview:

In this lab, you will continue with Lab 3 project. As Lab 3 is already using BSP, you will be creating a custom BSP based on existing "**`PIC32MZ EF Starter Kit`**" BSP. For extra credit, you can create a new project and import the custom BSP that you created and use the new BSP definitions in the application.

This lab is completed in the following steps:

Step 1 – Explore the existing BSP features and customize LED names
Step 2 – Save the custom BSP
Step 3 – Use new LED names in the existing application
Step 4 – Run the Application

Extra Credit 1 – Create a new project, import custom BSP and use it!

Extra Credit 2 – Change system clock frequency and see if the application is affected.

Note: Each step has a short procedure that you must follow to complete the step.

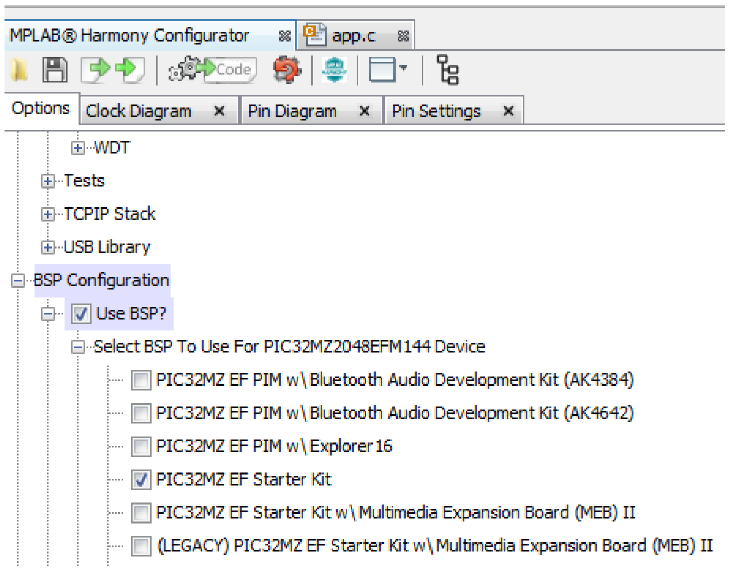
Application logic:

Default BSP	Custom BSP
BSP_LED_1	BSP_LED_RED
BSP_LED_2	BSP_LED_YELLOW
BSP_LED_3	BSP_LED_GREEN

Step 1: Explore the existing BSP features and customize LED names

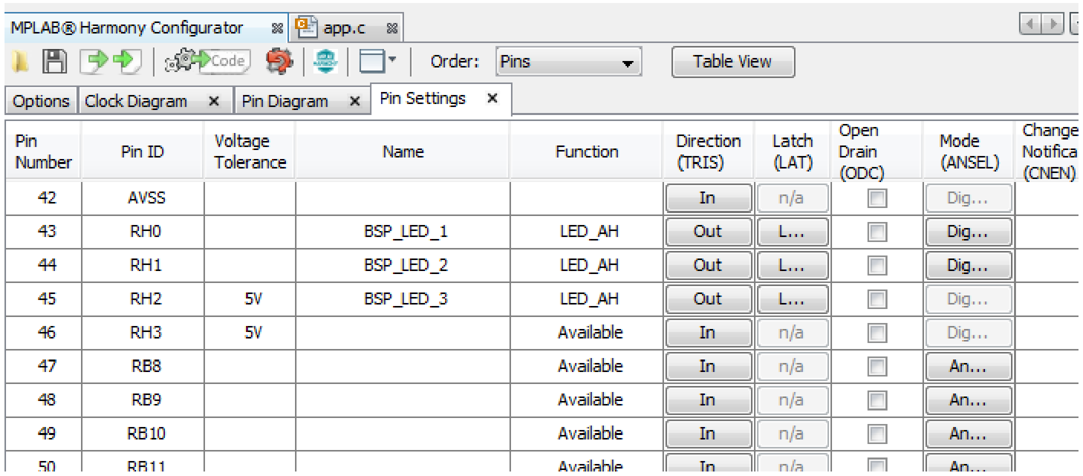
Procedure:

1. Go to "*BSP Configuration*" section of MHC "*Options"* window and verify that "**`PIC32MZ EF Starter Kit`**" is selected as BSP. This was chosen while creating the project for Lab 3. You will be using this BSP as a base and create a custom BSP.



```
⊞ WDT
⊞ Tests
⊞ TCPIP Stack
⊞ USB Library
⊟ BSP Configuration
  ⊟ [✓] Use BSP?
    ⊟ Select BSP To Use For PIC32MZ2048EFM144 Device
        [ ] PIC32MZ EF PIM w\ Bluetooth Audio Development Kit (AK4384)
        [ ] PIC32MZ EF PIM w\ Bluetooth Audio Development Kit (AK4642)
        [ ] PIC32MZ EF PIM w\ Explorer16
        [✓] PIC32MZ EF Starter Kit
        [ ] PIC32MZ EF Starter Kit w\ Multimedia Expansion Board (MEB) II
        [ ] (LEGACY) PIC32MZ EF Starter Kit w\ Multimedia Expansion Board (MEB) II
```

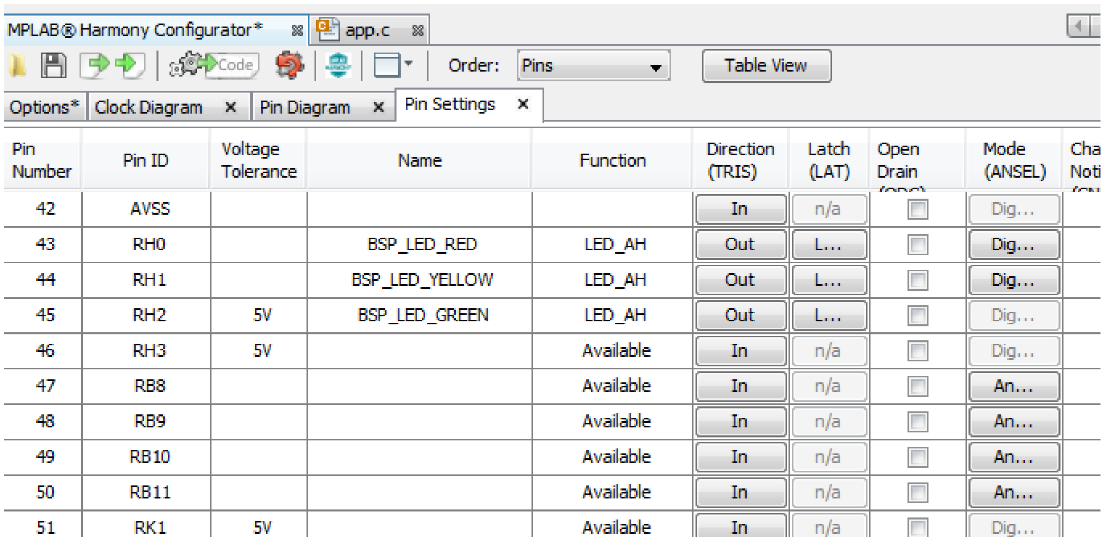
2. Check the existing LEDs in the Pin settings window as shown below and explore the other BSP features provided by "**`PIC32MZ EF Starter Kit`**" BSP.



Pin Number	Pin ID	Voltage Tolerance	Name	Function	Direction (TRIS)	Latch (LAT)	Open Drain (ODC)	Mode (ANSEL)	Change Notifica (CNEN)
42	AVSS				In	n/a	☐	Dig...	
43	RH0		BSP_LED_1	LED_AH	Out	L...	☐	Dig...	
44	RH1		BSP_LED_2	LED_AH	Out	L...	☐	Dig...	
45	RH2	5V	BSP_LED_3	LED_AH	Out	L...	☐	Dig...	
46	RH3	5V		Available	In	n/a	☐	Dig...	
47	RB8			Available	In	n/a	☐	An...	
48	RB9			Available	In	n/a	☐	An...	
49	RB10			Available	In	n/a	☐	An...	
50	RB11			Available	In	n/a	☐	An...	

3. Existing BSP uses LED names available on the board, namely LED 1, LED 2 and LED 3. Let's say the new board uses LED RED, LED YELLOW and LED GREEN instead and for convenience, you want to use those names for LED operation. In Pin Settings tab, you can configure the pins graphically. In this case, you will only assign new names to the LEDs. Edit the LED names as shown below (Double click on pin name to edit).

Existing LED name	New LED name
BSP_LED_1	BSP_LED_RED
BSP_LED_2	BSP_LED_YELLOW
BSP_LED_3	BSP_LED_GREEN



Order: Pins

Pin Number	Pin ID	Voltage Tolerance	Name	Function	Direction (TRIS)	Latch (LAT)	Open Drain	Mode (ANSEL)	Cha Noti
42	AVSS				In	n/a	☐	Dig...	
43	RH0		BSP_LED_RED	LED_AH	Out	L...	☐	Dig...	
44	RH1		BSP_LED_YELLOW	LED_AH	Out	L...	☐	Dig...	
45	RH2	5V	BSP_LED_GREEN	LED_AH	Out	L...	☐	Dig...	
46	RH3	5V		Available	In	n/a	☐	Dig...	
47	RB8			Available	In	n/a	☐	An...	
48	RB9			Available	In	n/a	☐	An...	
49	RB10			Available	In	n/a	☐	An...	
50	RB11			Available	In	n/a	☐	An...	
51	RK1	5V		Available	In	n/a	☐	Dig...	

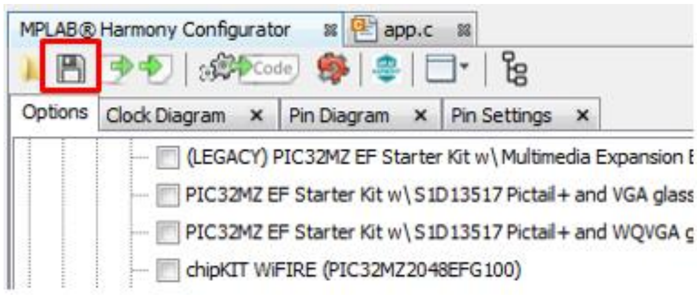
Note: For this lab only existing LEDs are renamed. If the new board has LEDs on different pins you can configure the pin settings used by the existing LEDs and assign a convenient name. Similarly, other features of the BSP can be modified.

Step 2: Save the custom BSP

Procedure:

With the new names given to the LEDs your simple custom BSP is ready. Save the custom BSP to use in any other application which you write for this board.

1. Click on the "*Save Configuration"* Button on MHC toolbar and select the "*Board Support Package*" tab in the pop-up.



(LEGACY) PIC32MZ EF Starter Kit w\ Multimedia Expansion
PIC32MZ EF Starter Kit w\ S1D13517 Pictail+ and VGA glass
PIC32MZ EF Starter Kit w\ S1D13517 Pictail+ and WQVGA
chipKIT WiFIRE (PIC32MZ2048EFG100)

2. Make the following changes in "Board Support Package" tab,

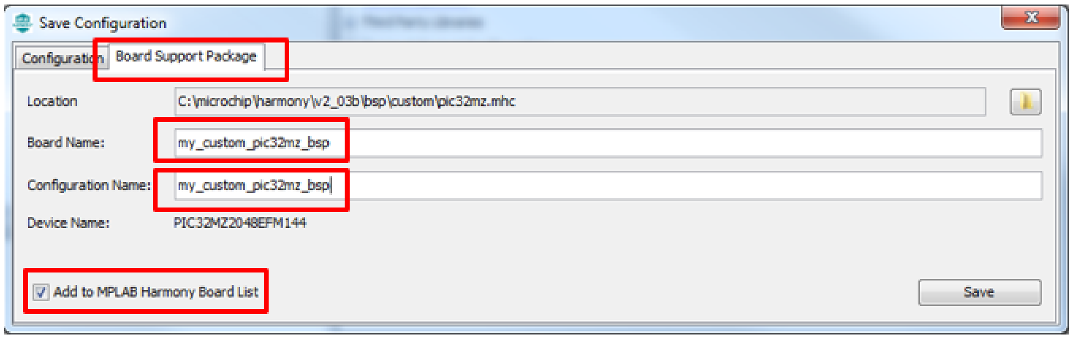
   a. Change the Board name to "**my_custom_pic32mz_bsp**".

   b. Change the Configuration name to "**my_custom_pic32mz_bsp**".

   c. Accept the default location.

   d. Ensure that "*Add to the MPLAB® Harmony Board List"* is selected.

   e. Click on "*Save"*.



Location C:\microchip\harmony\v2_03b\bsp\custom\pic32mz.mhc
Board Name: my_custom_pic32mz_bsp
Configuration Name: my_custom_pic32mz_bsp
Device Name: PIC32MZ2048EFM144
[✓] Add to MPLAB Harmony Board List

Step 3: Use new LED names in the existing application

Procedure:

1. Save and Generate code for the project to use new BSP names.

2. In **`app.c`** file, replace **`BSP_LED_<index>`** with new names that you have configured for the custom BSP. Search and find (Ctrl+F) the existing LED names and replace them with new names.

Existing LED name	New LED name
BSP_LED_1	BSP_LED_RED
BSP_LED_2	BSP_LED_YELLOW
BSP_LED_3	BSP_LED_GREEN

Step 4: Run the Application.

Procedure:

Click the "*Program*" button to build, program and run the application. YELLOW LED should be lit by default to show system is ON. RED LED and GREEN LED should be toggling at 0.5 and 2 second intervals as configured.

Observe that you are now using the customized LED names instead of default ones!

******************** **Congratulations! You're done with Lab 4** ********************
You have gained experience on how to create a custom BSP using MHC and how to use it in the Harmony application.
****

Extra Credit 1 – Create a new project, import your custom BSP and use it!

Procedure:

1. Use your learning from Lab 1 and Lab 3 to create a new project.

2. While selecting the target board, observe that the custom BSP "**my_custom_pic32mz_bsp**" is available in the list, select it and use the custom BSP in your application!

Extra Credit 2 – Change system clock frequency and see if the application is affected.

Procedure:

1. Use your learning from Lab 1 on how to configure system clock for a new frequency (you can use auto-calculate) in the clock manager.

2. Select 100MHz instead of default 200MHz and generate code (you could do this to save power consumption).

3. See if your application behavior is changed: You would still notice that the LEDs are blinking at the same intervals!

4. Harmony framework does take care of this through abstraction.

   ****Welcome to Scalable, Flexible and Portable Application development****

Welcome to Harmony!