LAB Manual for 21017 DEV6

Creating Advanced PIC32 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, 21017-DEV6 class, Creating Advanced PIC32 Embedded Applications using MPLAB Harmony.

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 knowledge and understanding of MPLAB Harmony and the PIC32 MCU.

Required Development Tools:

1. DM320104 **:** Curiosity PIC32MZEF Development Board
2. MIKROE-1486 : FRAM Click Board
3. MIKROE-1978 : Weather Click Board
4. DV244005 : MPLAB® Real ICE™
5. MPLAB® X IDE v3.61.
6. MPLAB® XC32 Compiler v1.43.
7. MPLAB® Harmony Integrated Software Framework v2.03b.
8. MPLAB® Harmony Configurator (MHC) v2.0.3.5.
   (Included in MPLAB Harmony v2.03b installation.)

Prerequisites:

The lab material assumes the attendee has prior experience with:

1. MPLAB X IDE basic usage
2. Basic understanding of MPLAB Harmony
3. MPLAB based Programming/Debugging fundamentals
4. C language programming
5. Basic knowledge on PIC32 product family

Upon completion, you will:

1. Have hands-on experience with MPLAB Harmony Framework components.
2. Have hands-on experience with the MPLAB Harmony Configurator (MHC).
3. Have hands-on experience with the MPLAB Drivers and System Services
4. Have hands-on experience of using RTOS with MPLAB Harmony.
5. Have an understanding of key benefits and features of MPLAB Harmony.

LAB 1:

Create a MPLAB® Harmony application using MPLAB Harmony Drivers and System Services.

LAB 1: Create a MPLAB® Harmony application using MPLAB Harmony Drivers and System Services

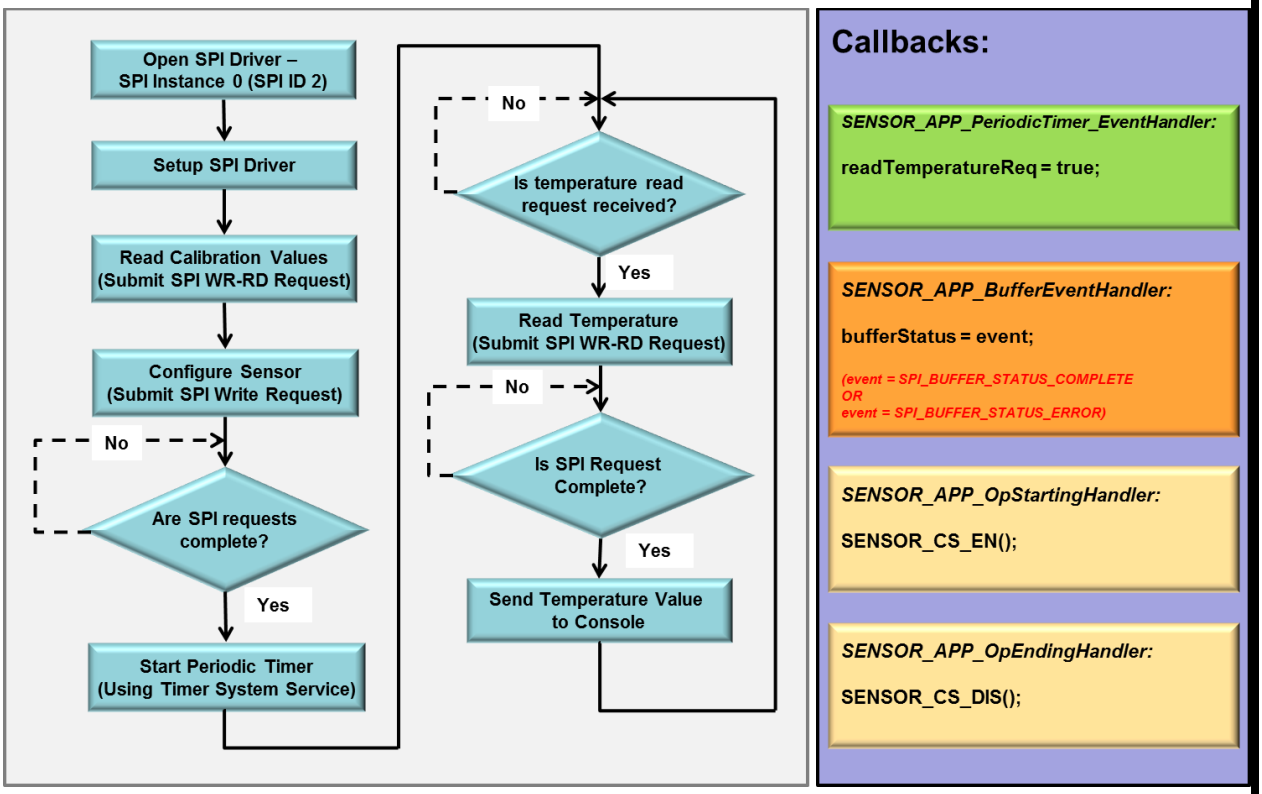
Purpose:

After completing Lab 1, you will have an understanding of how to use Harmony Drivers and System Services. You will use MPLAB Harmony Configurator (MHC) tool to configure the MPLAB Harmony project and read temperature readings at a periodic rate from a temperature sensor interfaced over SPI.

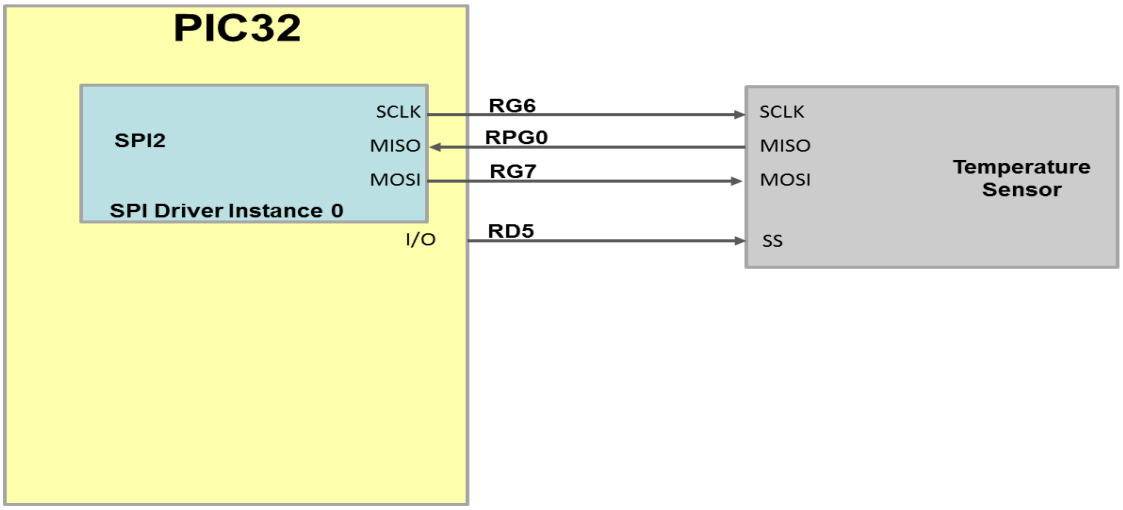
Overview:

In this lab you will create a MPLAB Harmony project using the MHC. The application you will create, will utilize a SPI Driver to read temperature from a temperature sensor, the Timer System Service to read temperature periodically and (USB CDC) Console System Service to print the temperature values on a terminal application running on a PC. In the process, the lab will also demonstrate state machine based design and the use of callbacks functions.

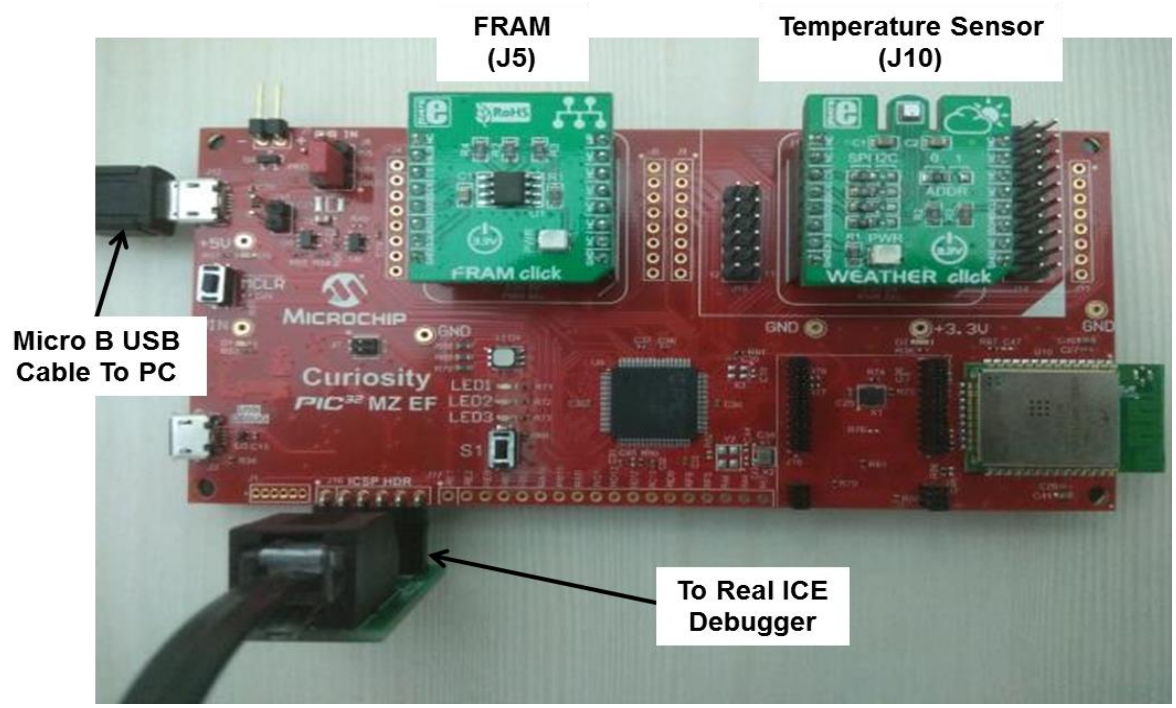
Flowchart:



Connection Diagram:



Hardware Setup:



Note: The Weather Click board must be modified by de-soldering the 0 Ohm resistors connecting to the I2C bus and re-soldering them to connect to the SPI bus. For this lab, this hardware change has already been done.

This lab is completed in the following steps:

Part 1: Create project and configure PIC32MZ

Step 1: Create MPLAB® Harmony Project using MPLAB X IDE.
Step 2: With MHC, select Board Support Package (BSP).
Step 3: With MHC, create Sensor Application.
Step 4: With MHC, configure SPI Driver and SPI pins.
Step 5: With MHC, configure Timer System Service.
Step 6: With MHC, configure (USB CDC) Console System Service
Step 7: With MHC, generate code.

Part 2: Add application code and build the application

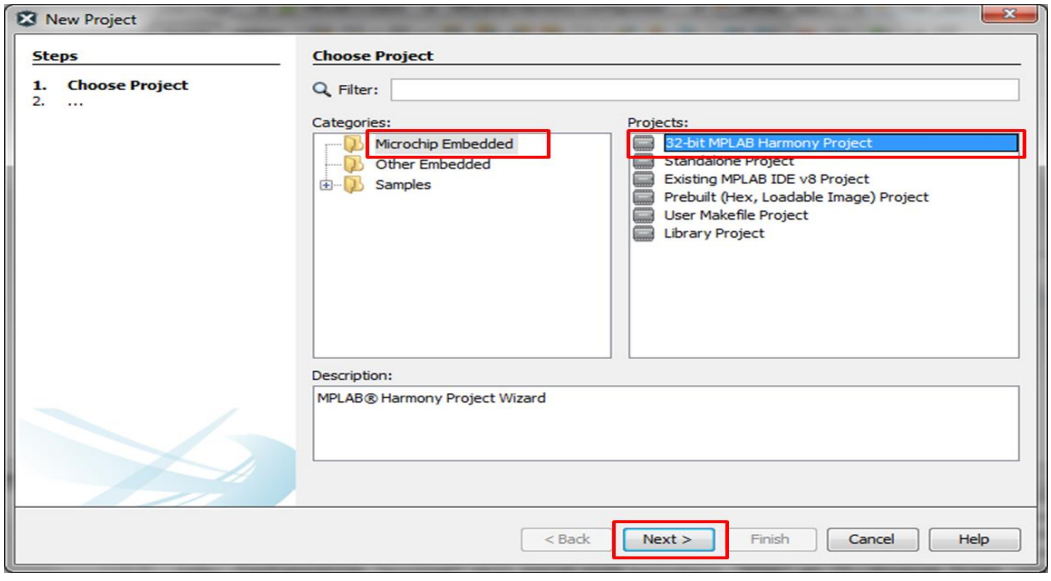
Step 8: Add application code to the Sensor Application
Step 9: Build and Run the Application

Part 1: Create project and configure PIC32MZ

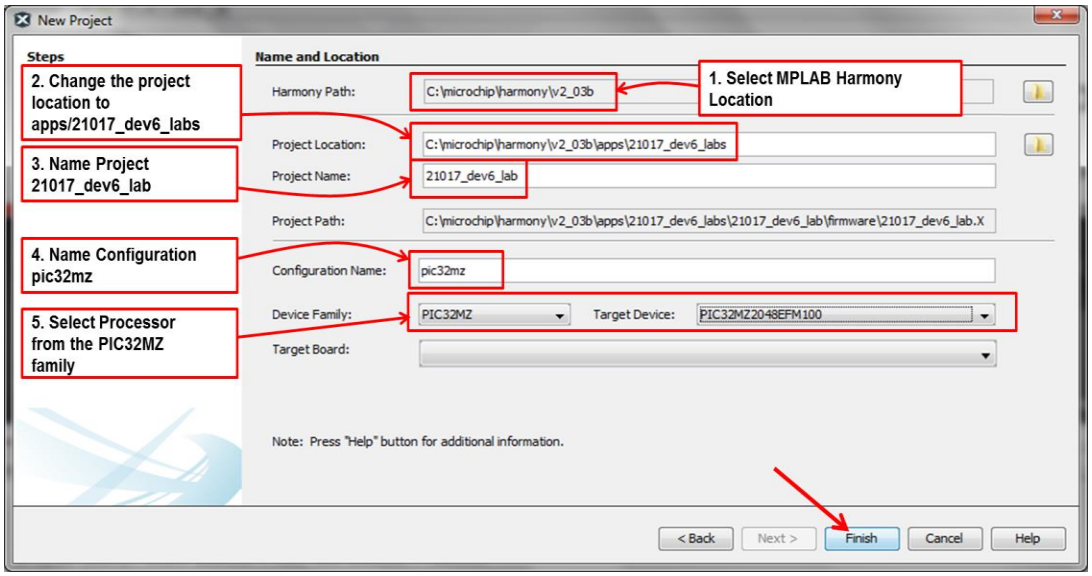
Step 1: Create MPLAB Harmony Project using MPLAB X IDE

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

2. In the Categories pane of the New Project dialog, select **Microchip Embedded**. In the Projects pane, select **32-bit MPLAB Harmony Project**, and then click **Next**.



3. Create the new project as follows:
   Harmony Path: **C:\Microchip\harmony\v2_03b**
   Project Location: C:\Microchip\harmony\v2_03b\apps**\21017_dev6_labs**
   Project Name: **21017_dev6_lab**
   Configuration Name: **pic32mz**
   Device Family: **PIC32MZ**
   Target Device: **PIC32MZ2048EFM100**

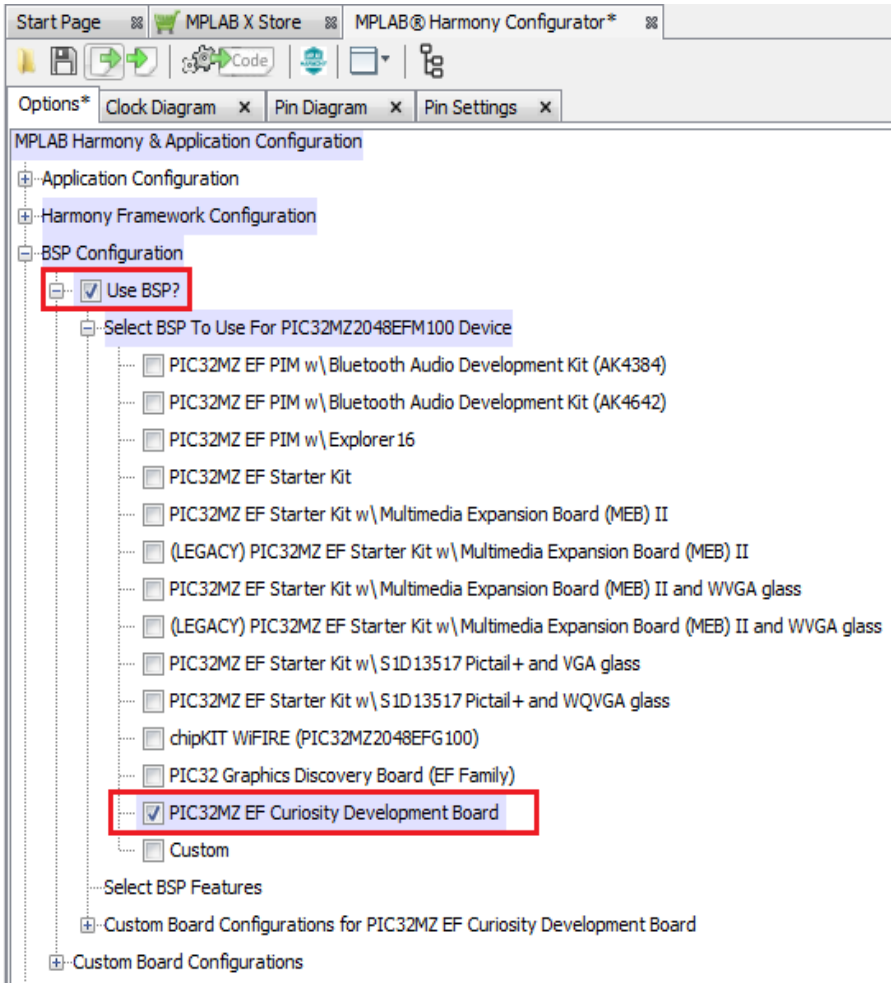


Click **Finish** when ready.

Step 2: With MHC, select Board Support Package (BSP).

1. In the MHC **Options** tab, expand **BSP Configuration** and select the **"Use BSP?"** option.

2. Expand the **Select BSP To Use For PIC32MZ2048EFM100 Device** and select the "PIC32MZ EF Curiosity Development Board".

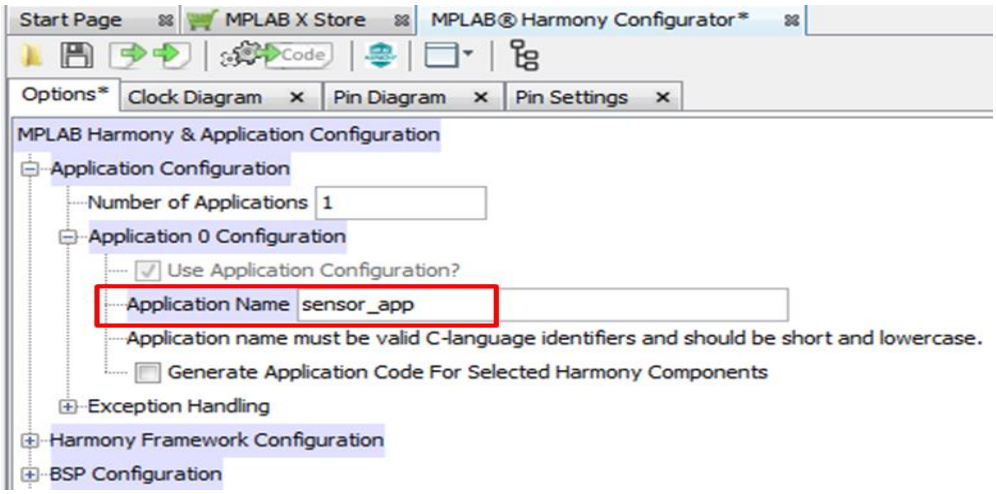


Note: The default initial target board configuration sets the processor clock at its highest rate and sets up the IO pins for switches and LEDs. Feel free to explore these settings in the Clock and Pin tabs in the MHC window.

Step 3: With MHC, create Sensor Application.

In the MHC **Options** tab, expand **Application Configuration** and **Application 0 Configuration**, as shown.

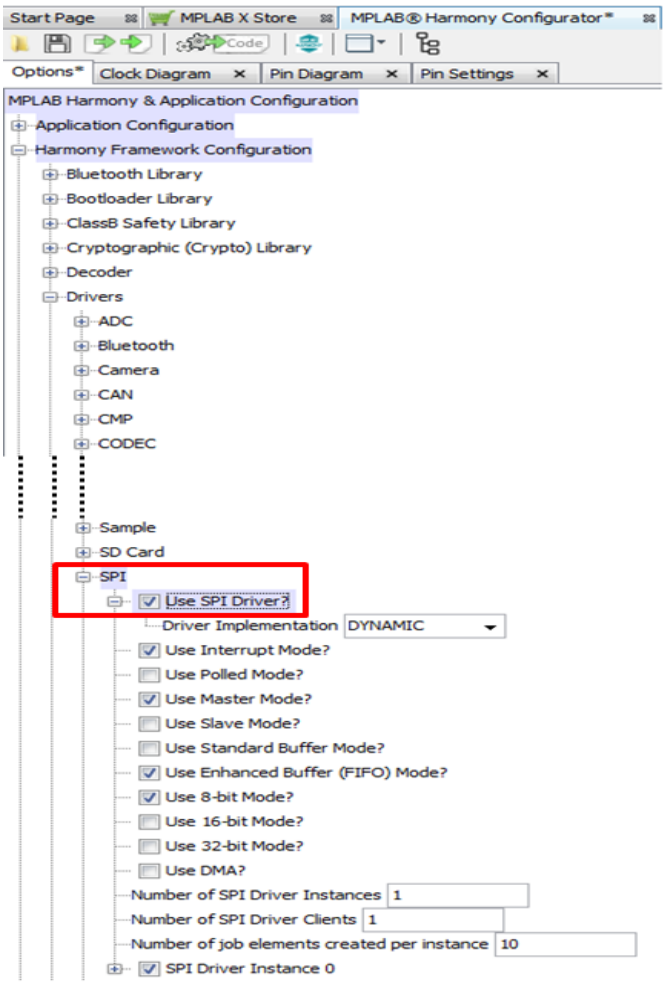
Name the application as "**sensor_app**"



Step 4: With MHC, configure SPI Driver and SPI pins.

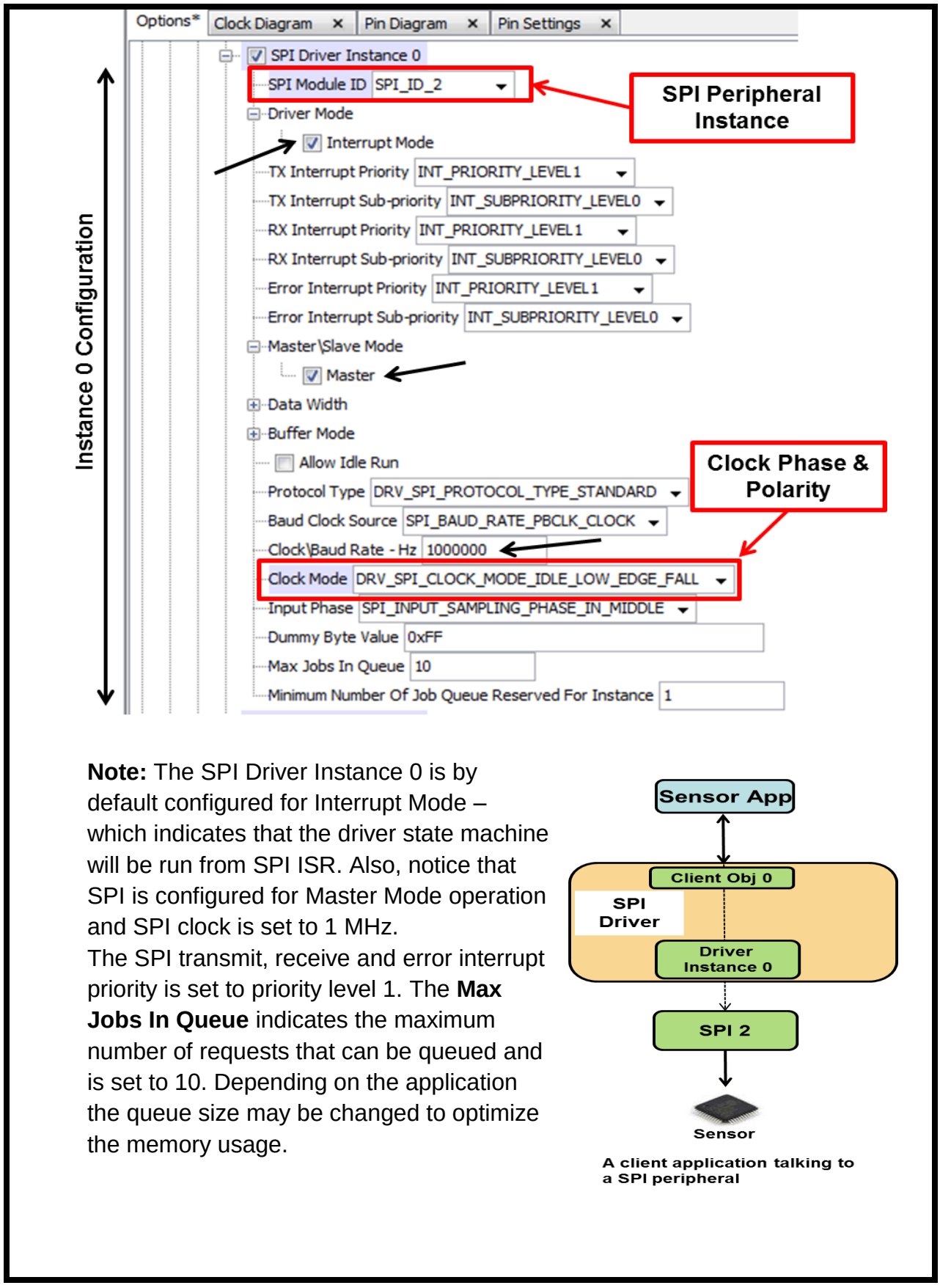
1. Expand the MPLAB **Harmony Framework Configuration > Drivers > SPI**

   a. Select the **Use SPI Driver?** check box.
   Note: When this check box is selected, the SPI Driver Instance 0 gets enabled by default. The Driver Instance number always starts from 0.



   b. The temperature sensor is interfaced to SPI Peripheral Instance 2. Expand **SPI Driver Instance 0**. Select **SPI Module ID** to SPI_ID_2.

   c. Select SPI **Clock Mode** to DRV_SPI_CLOCK_MODE_IDLE_LOW_EDGE_FALL. This sets up the SPI clock phase and polarity.



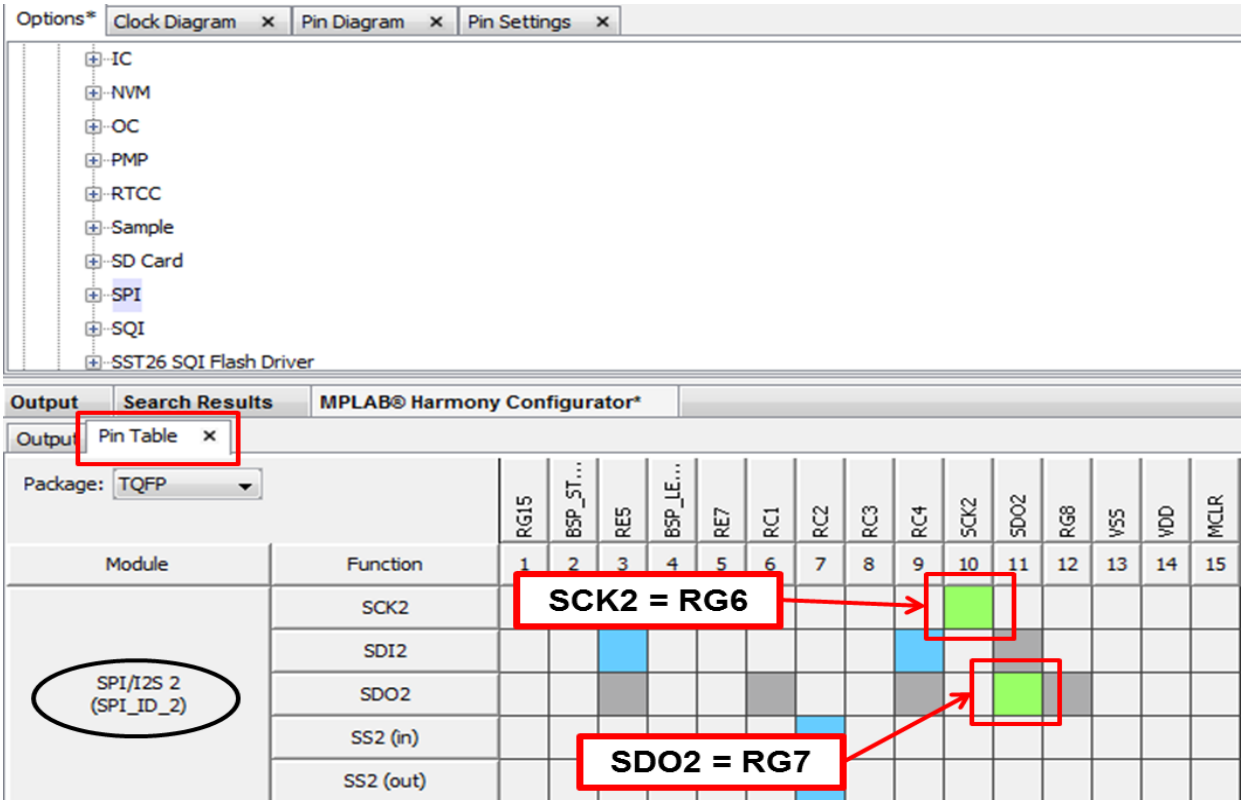
Note: The SPI Driver Instance 0 is by default configured for Interrupt Mode – which indicates that the driver state machine will be run from SPI ISR. Also, notice that SPI is configured for Master Mode operation and SPI clock is set to 1 MHz.
The SPI transmit, receive and error interrupt priority is set to priority level 1. The **Max Jobs In Queue** indicates the maximum number of requests that can be queued and is set to 10. Depending on the application the queue size may be changed to optimize the memory usage.

A client application talking to a SPI peripheral

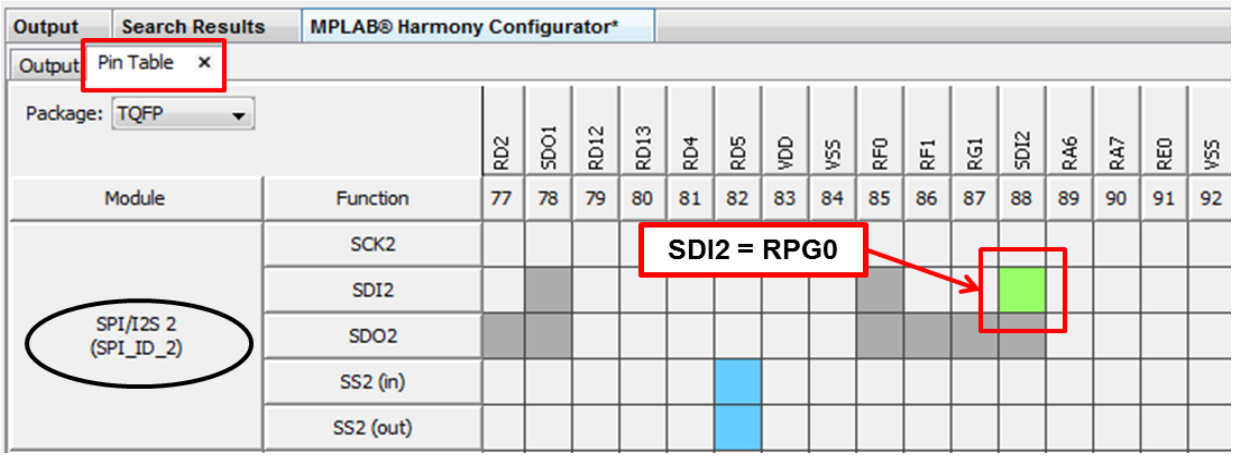
2. Select the **Pin Table** Tab in the output area of the IDE and then scroll down to the **SPI_ID_2** module as shown below.

    a. Enable SPI Clock (**SCK2**) on RG6 (Pin #10)

    b. Enable Serial Data Out (**SDO2**) on RG7 (Pin #11)

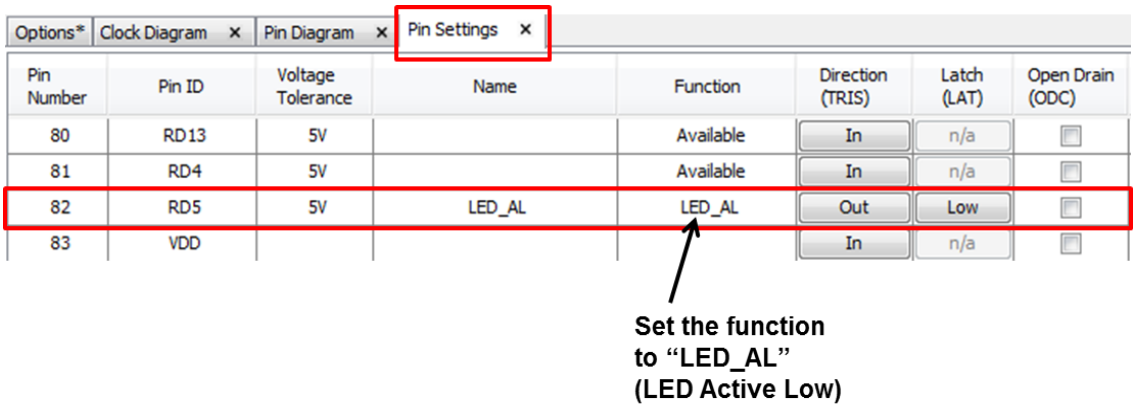


    c. Enable Serial Data In (**SDI2**) on RPG0 (Pin #88)

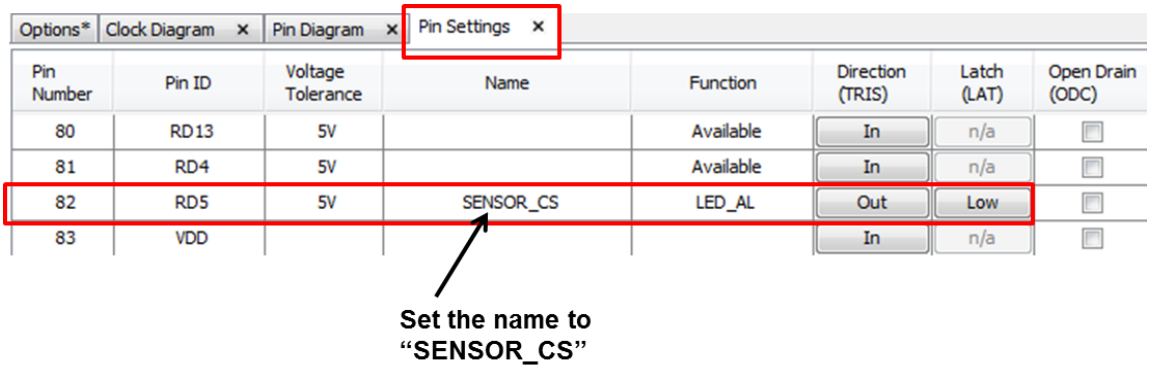


3. Select the **Pin Settings** tab as shown and go to the Pin **RD5** (Pin #82) which is mapped in the hardware to the sensor chip select pin.

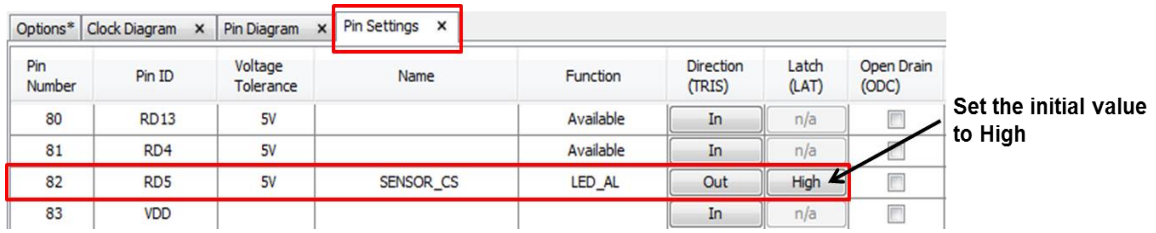
   a. As the Chip Select to the sensor is active low, set the **Function** drop-down list to LED_AL. This will set the **Name** to LED_AL, **Direction** to Out and **Latch** to the Low state.



   b. Change the **Name** to SENSOR_CS. MHC will generate the `SENSOR_CSOn(),SENSOR_CSOff(),SENSOR_CSToggle()` and `SENSOR_CSStateGet()` functions for the Chip Select pin in the system_config.h file.



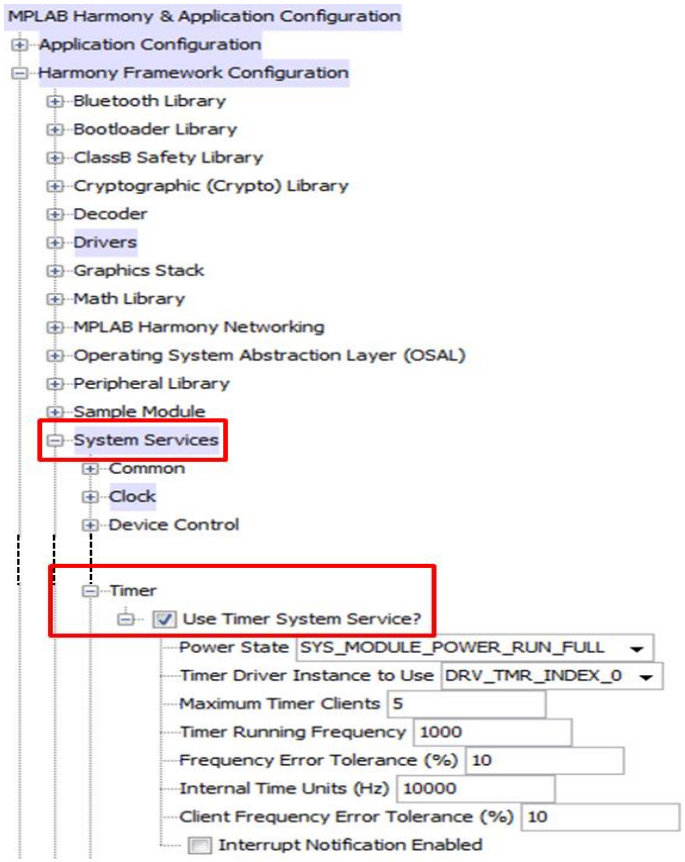
   c. Set the initial value of the Chip Select pin to the High state.



Step 5: With MHC, configure Timer System Service.

1. Go back to the **Options** tab and expand the MPLAB **Harmony Framework Configuration > System Services**

2. Expand the **Timer** system service and select the **Use Timer System Service?** check box. This will be needed later to allow the sensor application to generate an event every 500 milliseconds.



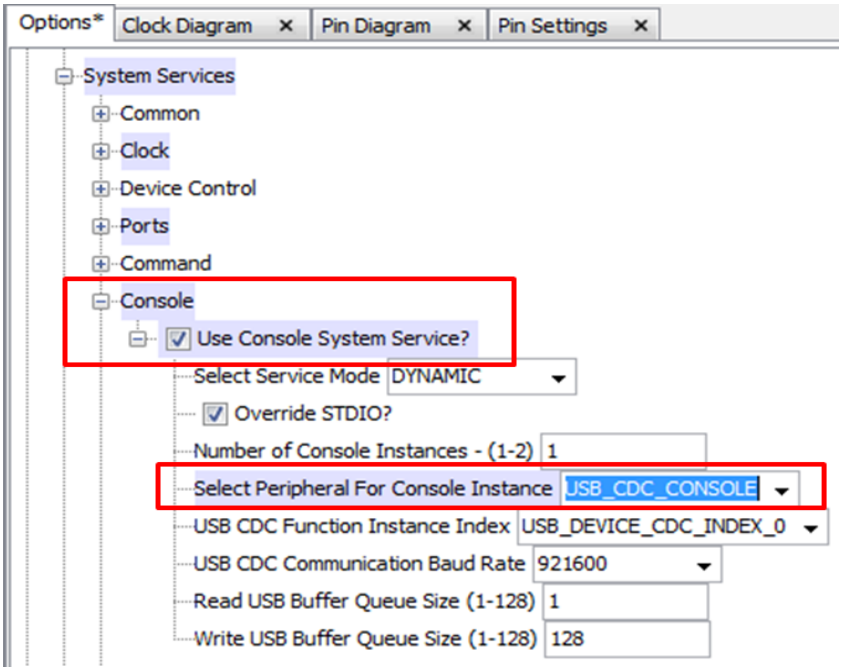
Note: The timer frequency is set to 1000 Hz. This will generate a timer interrupt every 1 millisecond and provides a timing resolution of 1 millisecond. The Timer System Service uses Timer Driver Instance 0. Upon enabling the Timer System Service, MHC will also configure the Timer Driver. Navigate to **Harmony Framework Configuration > Drivers > Timer** and observe that the MHC has configured **Timer Driver Instance 0** to use timer peripheral instance 1 - TMR_ID_1, in interrupt mode.

Step 6: With MHC, configure (USB CDC) Console System Service

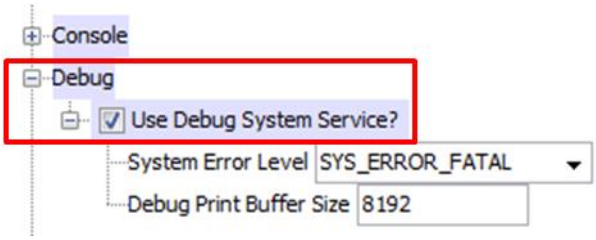
1. Expand the MPLAB **Harmony Framework Configuration > System Services**

2. Expand **Console** system service and select the **Use Console System Service?** check box.

3. The Console System Service will use USB CDC COM port (serial port) emulation to provide a console (command line) interface. Select USB_CDC_CONSOLE from the drop-down list for **Select Peripheral For Console Instance.**

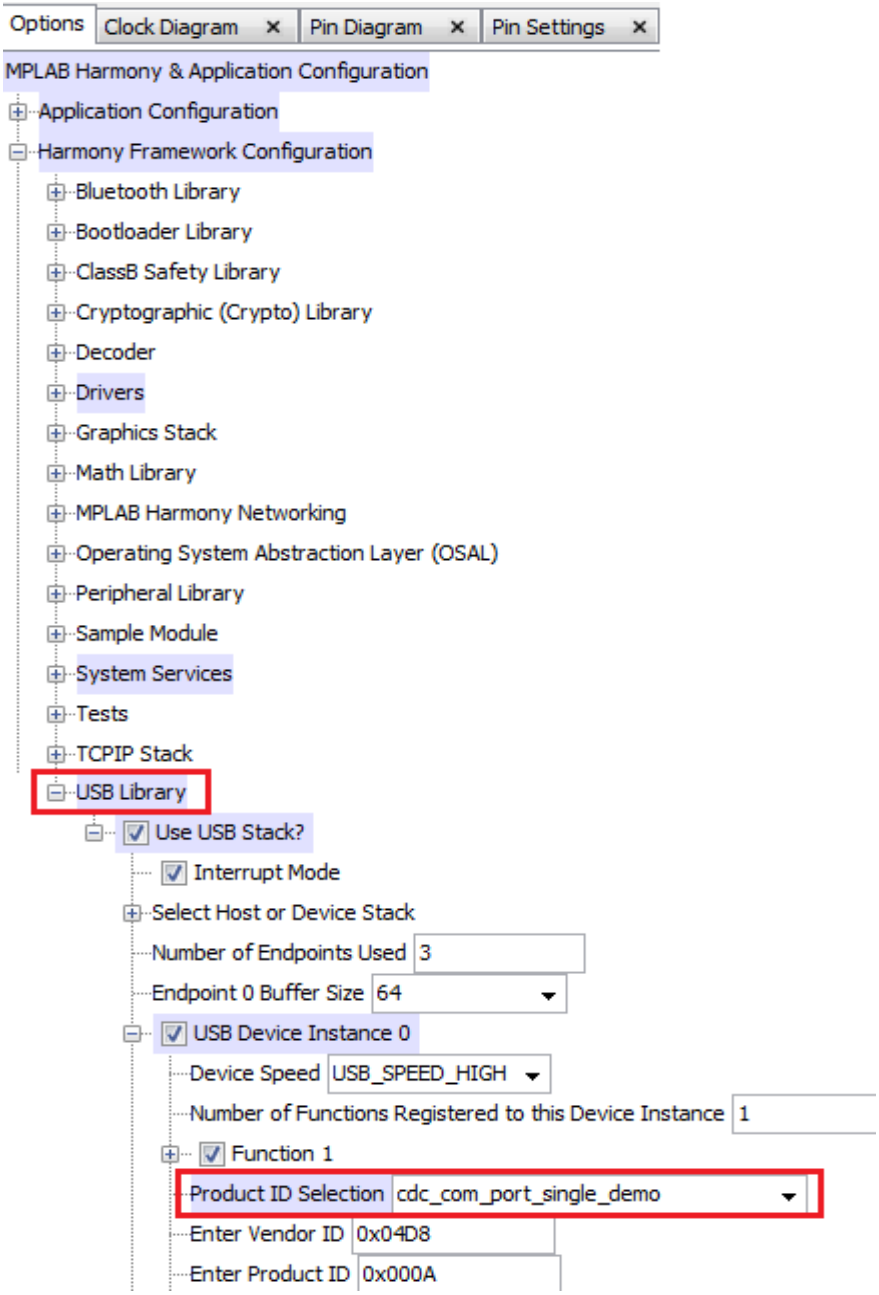


4. Expand the MPLAB **Harmony Framework Configuration > System Services > Debug** and select the **Use Debug System Service?** check box.
   The Debug System Service provides a convenient mechanism for the application developer to send formatted or unformatted messages to a system console.



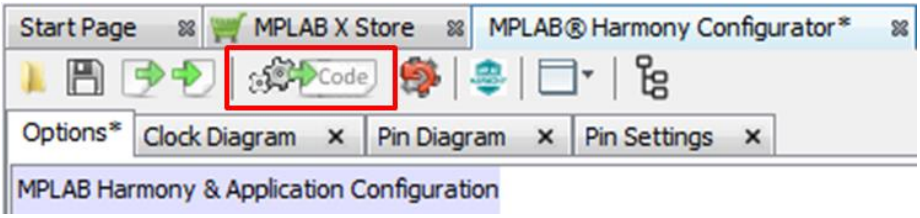
5. As the Console System Service is configured to use USB peripheral for console, MHC has proactively enabled the USB Library for you.

   Expand the MPLAB **Harmony Framework Configuration > USB Library > Use USB Stack? > USB Device Instance 0**
   Select one of the pre-defined USB configuration settings for the CDC COM port single demo - “cdc_com_port_single_demo” from the drop-down list for the **Product ID Selection**

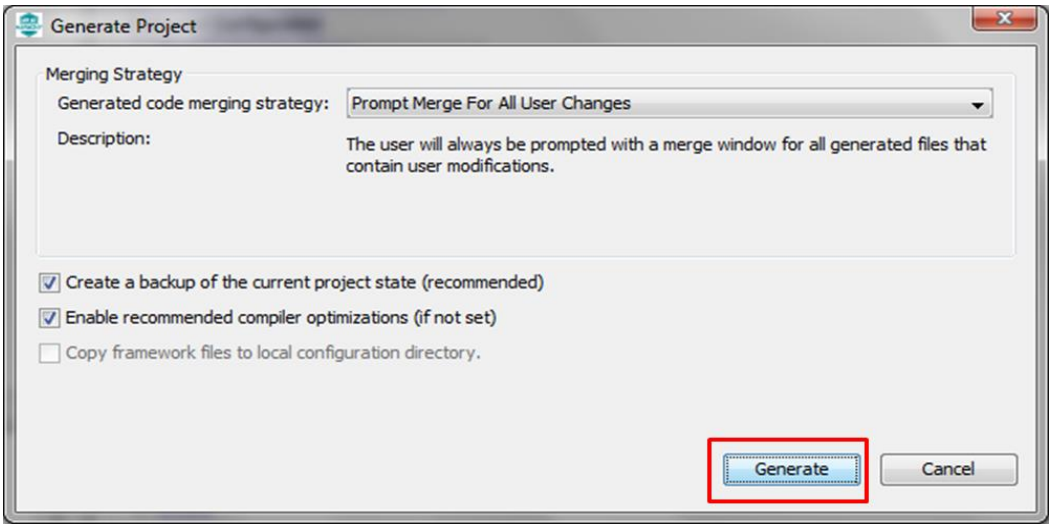


Step 7: With MHC, generate code

1. When done, generate the code as shown below.



2. Save the configuration in its default location, when prompted.
3. Click on the **Generate** button in the Generate Project window, keeping the default settings as shown below.



The MHC will include all the MPLAB® Harmony library files and generate the application starter code and the configuration code based on the MHC selections.

At this point you are ready to start implementing your application code.

Part 2: Add application code and build the application

Step 8: Add application code to the Sensor Application

Note: The application state machine for the sensor application is already developed and is available in **sensor_app.h** and **sensor_app.c** files under "**apps/21017_dev6_labs/help/lab1**" folder.

In the following steps, you will replace the contents of the **sensor_app.h** and **sensor_app.c** files in your project (which are the template application files generated by MHC), with the contents of these pre-developed files.

1. Go to the "**apps/21017_dev6_labs/help/lab1**" folder and open the pre-developed **sensor_app.h** file.

2. Copy all the contents from the pre-developed sensor_app.h file and paste (over-write) it to the sensor_app.h file of your project.

3. Similarly, go to the "**apps/21017_dev6_labs/help/lab1**" folder and open the pre-developed **sensor_app.c** file.

4. Copy all the contents from the pre-developed sensor_app.c file and paste (over-write) it to the sensor_app.c file of your project.

   Note: The **sensor_app.h** file defines the states of the sensor application. Refer to the flowchart in the beginning of this lab and relate the application states with the flowchart.

```
typedef enum
{
    /* Application's state machine's initial state. */
    SENSOR_APP_STATE_INIT=0,
    SENSOR_APP_STATE_SENSOR_INIT,
    SENSOR_APP_STATE_WAIT_INIT_COMPLETE,
    SENSOR_APP_STATE_WAIT_MEAS_REQ,
    SENSOR_APP_STATE_WAIT_MEAS_COMPLETE,
    SENSOR_APP_STATE_ERROR,

} SENSOR_APP_STATES;
```

Similarly, the **SENSOR_APP_Tasks()** function in **sensor_app.c** file implements the sensor application state machine. It also contains empty event handlers and placeholders that you will populate with necessary code in the next step.

5. Open **sensor_app.c** in MPLAB® IDE, and add application code.

> **Tip**: Search for the string **"Step #"** in **sensor_app.c** file to locate exact position where the code snippets need to be copied.

1. Add code to open the SPI driver instance 0 (SPI_ID_2) under the `SENSOR_APP_STATE_INIT` state. The `DRV_SPI_Open()` API will associate the sensor application with the SPI driver instance 0. The returned handle will be used by the sensor application in all subsequent calls to the driver.

```
case SENSOR_APP_STATE_INIT:
{
    DRV_SPI_CLIENT_DATA clientData = {0};


    /* Open SPI driver (Instance 0, SPI ID 2) here ----------> (Step #1) */
    sensor_appData.handle = DRV_SPI_Open( DRV_SPI_INDEX_0, DRV_IO_INTENT_READWRITE );
```

Search for "Step #1" and add the below code snippet.

Code snippet:

```
sensor_appData.handle = DRV_SPI_Open( DRV_SPI_INDEX_0,
DRV_IO_INTENT_READWRITE );
```

2. Add code to set the operation starting and ending event handlers in the `SENSOR_APP_STATE_INIT` state after a valid handle to the driver is obtained. These callbacks will be called by the SPI driver state machine during the SPI request execution to allow sensor application to chip select/de-select the temperature sensor.
   The `clientData` points to the application functions that handle the operation starting and the operation ending events.

```
/* Set up client event handlers for operation starting and ending events*/

//clientData.baudRate = DRV_SPI_BAUD_RATE_IDX0;
clientData.operationStarting = SENSOR_APP_OpStartingHandler;
clientData.operationEnded = SENSOR_APP_OpEndingHandler;

/* Set client configuration here ----------> (Step #2) */

DRV_SPI_ClientConfigure(sensor_appData.handle, &clientData );
```

Search for "Step #2" and add the below code snippet.
Code snippet:

```
DRV_SPI_ClientConfigure(sensor_appData.handle, &clientData );
```

3. Add code to enable sensor chip select in the operation starting event handler - `SENSOR_APP_OpStartingHandler()`. Remember that you have already configured the chip select pin in the Pin Settings tab under Step 4, based on which MHC has generated the functions to control (like turn on, turn off, toggle etc.) the chip select pin in the system_config.h file.

```
/* Application's SPI Operation Starting Callback Function */
static void SENSOR_APP_OpStartingHandler(
    DRV_SPI_BUFFER_EVENT event,
    DRV_SPI_BUFFER_HANDLE bufferHandle,
    void* context
)
{
    /* Chip Select Sensor here ----------> (Step #3) */
    SENSOR_CSOn();
}
```

Search for "Step #3" and add the below code snippet.
Code snippet:

```
SENSOR_CSOn();
```

4. Add code to start the periodic timer using Timer System Service in the `SENSOR_APP_STATE_WAIT_INIT_COMPLETE` state after the sensor initialization is complete. Set the duration to 500 milliseconds and register a callback with the Timer System Service. This callback will be called by the driver every 500 milliseconds. The `SENSOR_APP_PeriodicTimerCallback` will be implemented in the next step.

```
/* Start Periodic Timer here using Timer System Service here ----------> (Step #4) */

SYS_TMR_CallbackPeriodic (
    TEMPERATURE_READ_PERIOD,
    NULL,
    SENSOR_APP_PeriodicTimerCallback
);
```

Search for "Step #4" and add the below code snippet.

Code snippet:

```
SYS_TMR_CallbackPeriodic (
      TEMPERATURE_READ_PERIOD,
      NULL,
      SENSOR_APP_PeriodicTimerCallback
);
```

5. Add code in the periodic timer event handler `SENSOR_APP_PeriodicTimerCallback()` that was registered with the timer system service in the previous step, to set the temperature request flag to true. The sensor application state machine will check this flag in the `SENSOR_APP_STATE_WAIT_MEAS_REQ` state and submit a SPI request to read the temperature from the sensor, if the flag is set to true.

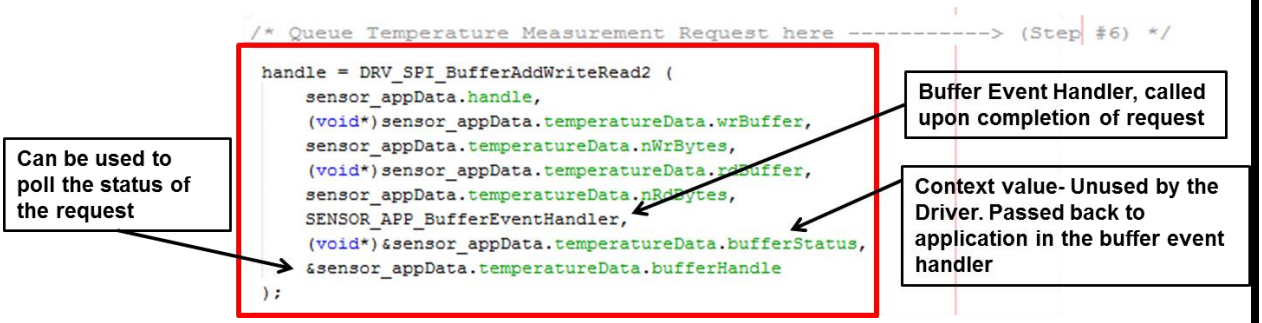
```
/* Application's Timer Callback Function */
static void SENSOR_APP_PeriodicTimerCallback(uintptr_t context, uint32_t currTick)
{
    /* Set temperature measurement request here ---------> (Step #5) */
    /* Comment the below line of code for Lab 3 ---------> (Lab 3_3) */
    sensor_appData.readTemperatureReq = true;

    /* Un-comment the below line of code for Lab 3 ---------> (Lab 3_3) */

    //xSemaphoreGive(sensor_appData.xSemaphore);
}
```

Search for "Step #5" and add the below code snippet.

Code snippet:

```
sensor_appData.readTemperatureReq = true;
```

6. Add code to queue SPI write-read request to read temperature measurement value in the `SENSOR_APP_STATE_WAIT_MEAS_REQ` state. The write-read request works by first writing the address of the register to read, and then sends dummy data to read the value of the register out.
   The `SENSOR_APP_BufferEventHandler` function will be called by the driver when this queued operation is complete.
   The `sensor_appData.temperatureData.bufferHandle` is an output parameter which will be updated by the SPI driver. This buffer handle can then be used to poll the status of the submitted request by calling the `DRV_SPI_BufferStatus()` and passing buffer handle to it.
   For this request, you will poll the status of the request by calling the `DRV_SPI_BufferStatus()` API and passing the buffer handle to it.



Search for "Step #6" and add the below code snippet.

Code snippet:

```
handle = DRV_SPI_BufferAddWriteRead2 (
    sensor_appData.handle,
    (void*)sensor_appData.temperatureData.wrBuffer,
    sensor_appData.temperatureData.nWrBytes,
    (void*)sensor_appData.temperatureData.rdBuffer,
    sensor_appData.temperatureData.nRdBytes,
    SENSOR_APP_BufferEventHandler,
   (void*)&sensor_appData.temperatureData.bufferStatus,
    &sensor_appData.temperatureData.bufferHandle
);
```

7. Add code to update the status of the submitted SPI requests in the SPI buffer event handler `SENSOR_APP_BufferEventHandler()` which was passed as an argument to the SPI request API.
   The `context` will point to the address of the application variable (for example: `sensor_appData.temperatureData.bufferStatus`) which is passed as an argument to the SPI request API. The event handler will update the value of the variable pointed by the `context`. The application can check the status of the submitted request by comparing the value of the passed variable against `SENSOR_APP_SPI_BUFFER_STATUS_COMPLETE`.

```
case DRV_SPI_BUFFER_EVENT_COMPLETE:
    if (context)
    {
        /* Copy the buffer status here ----------> (Step #7) */
        *((SENSOR_APP_SPI_BUFFER_STATUS*)context) = SENSOR_APP_SPI_BUFFER_STATUS_COMPLETE;
    }
    break;
```

Search for "Step #7" and add the below code snippet.

Code snippet:

```
*((SENSOR_APP_SPI_BUFFER_STATUS*)context) =
SENSOR_APP_SPI_BUFFER_STATUS_COMPLETE;
```

8. Add code to print the temperature value to the console using the console system service, once the temperature value is available from the sensor in the `SENSOR_APP_STATE_WAIT_MEAS_COMPLETE` state.

```
sensor_appData.temperature = SENSOR_APP_CalcTemperature(
        &sensor_appData.temperatureData.rdBuffer[1],
        sensor_appData.sensorCalibValues);

/* Print to console using System Console Service here ----------> (Step #8) */
SYS_PRINT("Temperature:%.2f\r", sensor_appData.temperature);
```

Search for "Step #8" and add the below code snippet.

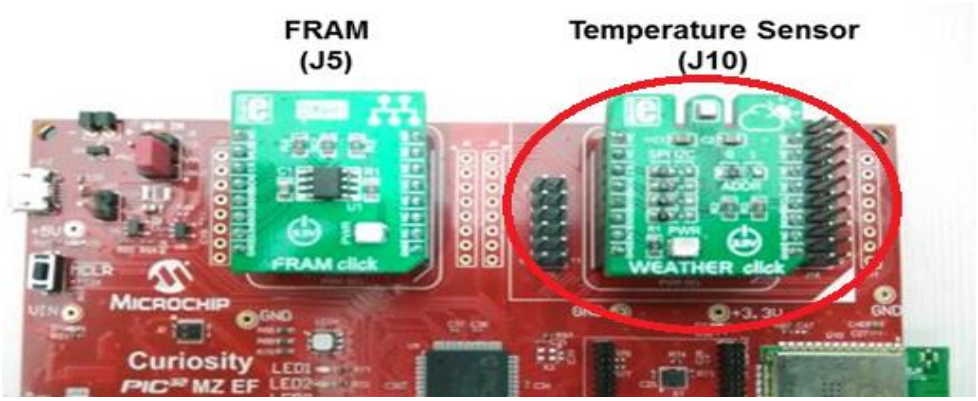
Code snippet:

```
SYS_PRINT("Temperature:%.2f\r", sensor_appData.temperature);
```

Congratulations! You're almost done. You are ready to build and run your first application.

Step 9: Build and Run the Application

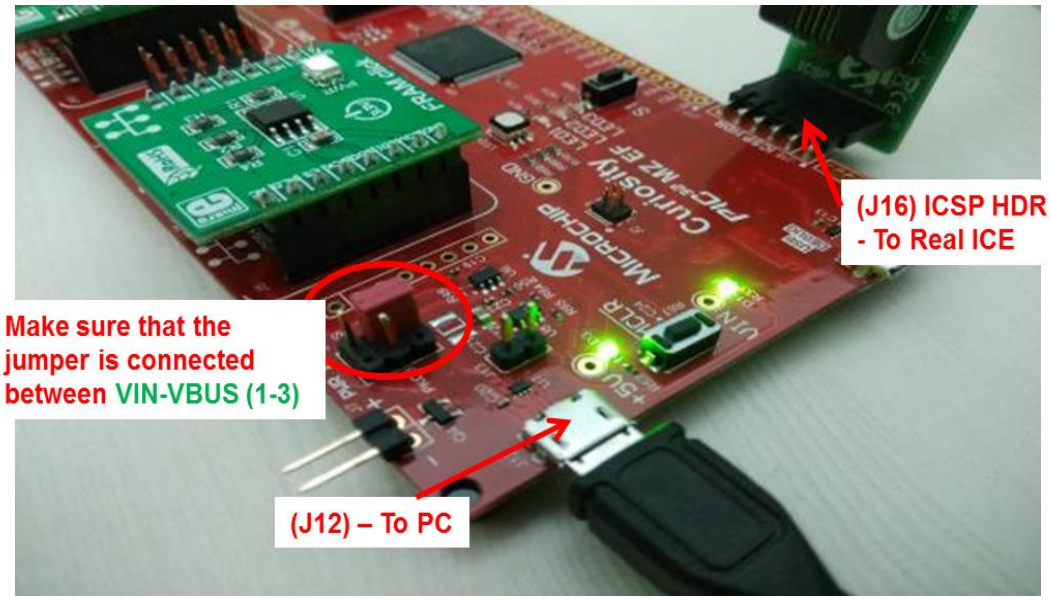
1. Verify that the temperature sensor (Weather Click Board) is connected to mikroBUS header **J10** on the **PIC32MZ EF Curiosity Development board.**



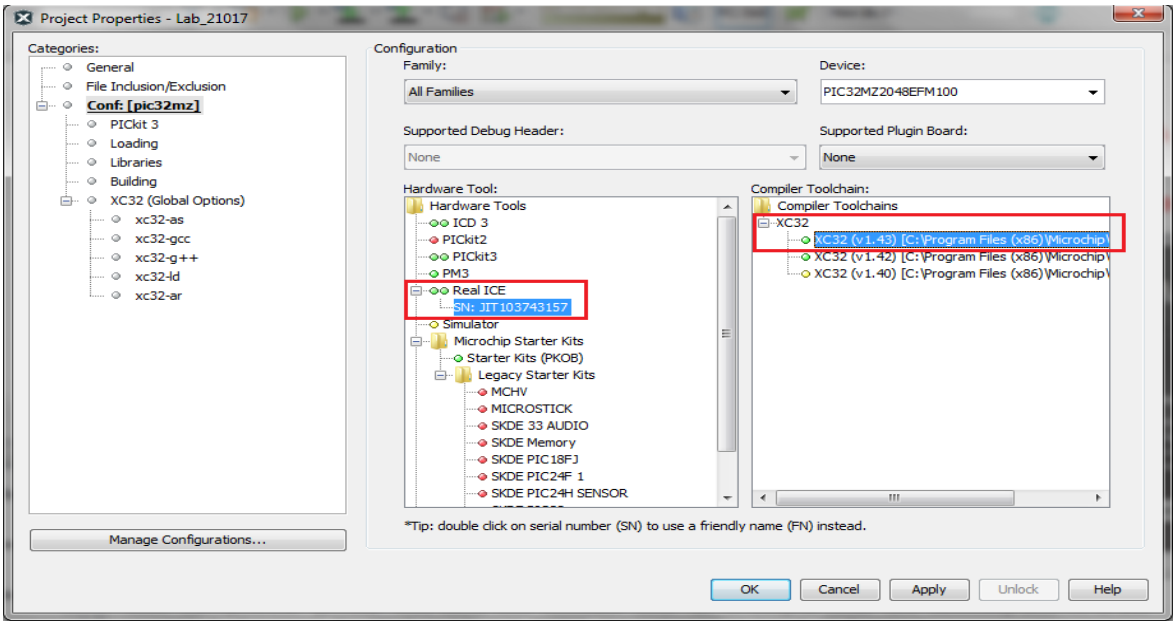
2. Make sure the **PIC32MZ EF Curiosity Development Board** is powered from a Host PC through a Type-A male to micro-B USB cable connected to micro-B port **J12.**

3. Select power source for the board. Verify that the jumper is connected between VIN-VBUS (1-3) of J8.

4. The ICSP Header on the PIC32MZ Curiosity Development board allows using external debuggers like the MPLAB® REAL ICE™. Connect the REAL ICE to J16 ICSP HDR on the board.



5. Go to **File > Project Properties** and make sure that the Real ICE is selected as the debugger under the **Hardware Tools** and XC32 (v1.43) is selected as the **Compiler Toolchain** for XC32.

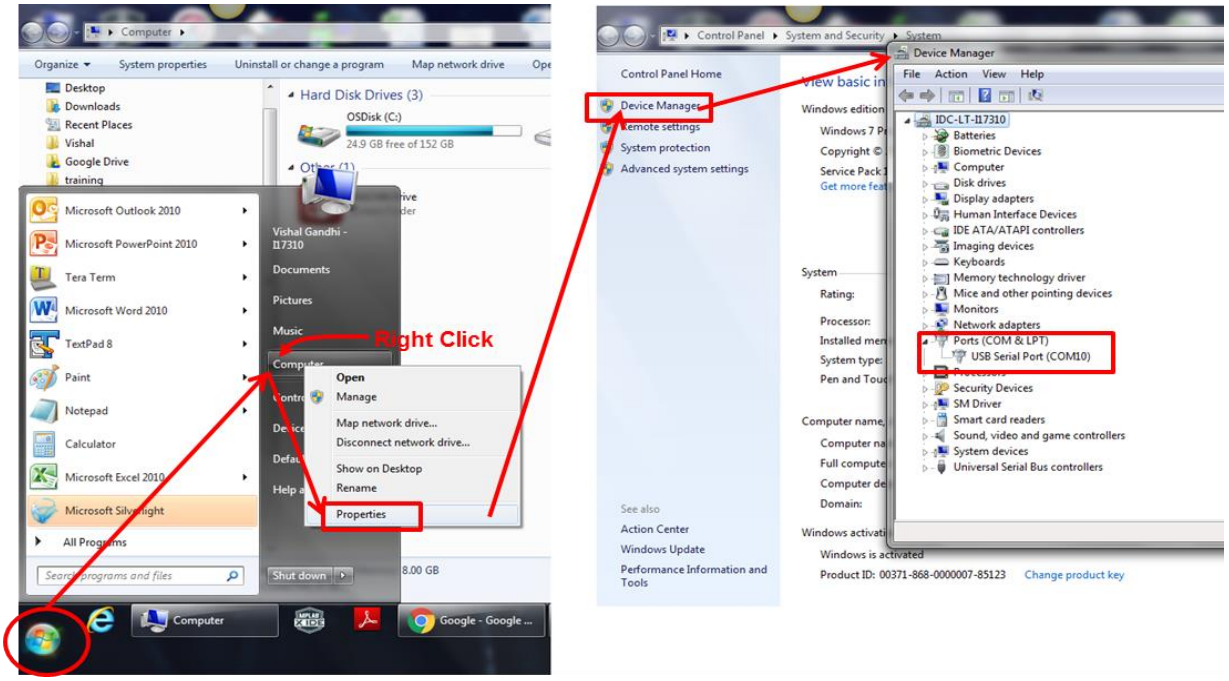


6. Build the code and program your application to the device, by clicking on the “Make and Program” button as shown below.

The Lab should build and program successfully.

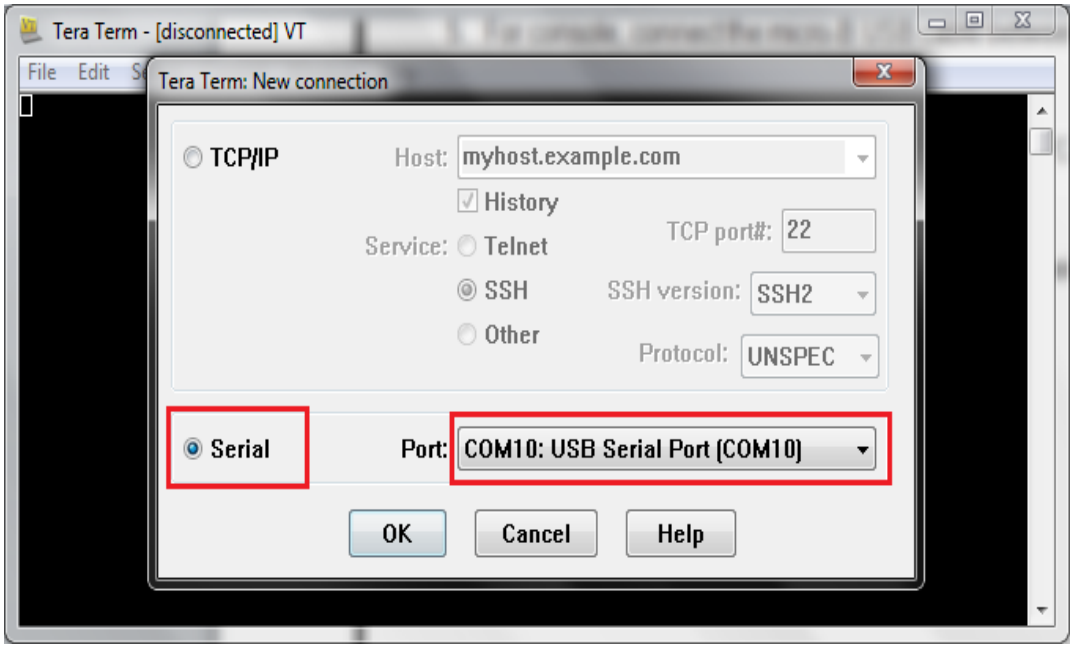
Note: To verify that the device is detected as a COM port and a COM port ID is assigned to it, click on the Windows® **Start** button and right click on **Computer**. Then, click on **Properties** and then on **Device Manager**. In the **Device Manager** window expand **Ports (COM & LPT).**

Refer to the below shown image.



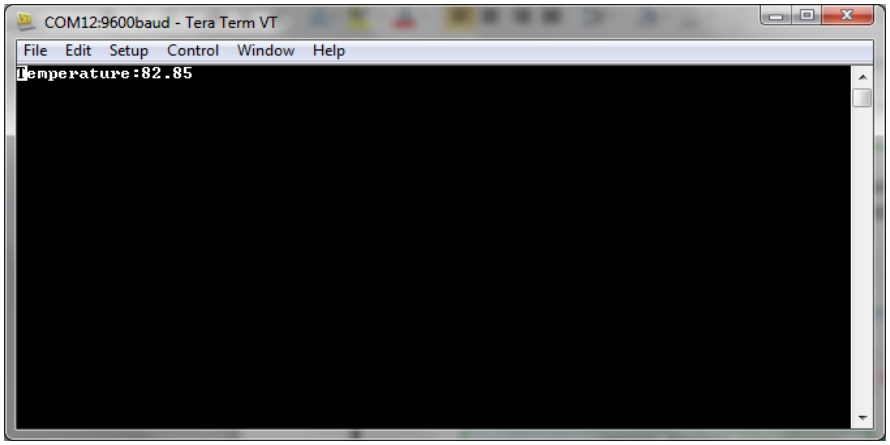
Note: The INF file for the driver is available under <harmony-install-path>\apps\usb\device\cdc_com_port_single\inf, if it is not installed on your computer already.

7. Now, open the **Tera Term** terminal application on your PC. Select the Serial Port as shown below.

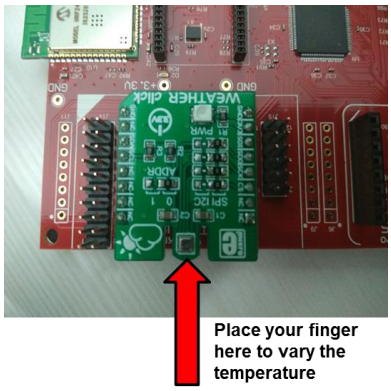


Note: Baud rate setting is not required as the data transfer between the PIC32 board and the PC takes place through a virtual serial port. The data rates depend on the USB speed and how much other traffic is there on the USB bus.

8. Once the terminal window is opened, press “Enter” key on your PC. You should be able to see the temperature values (in °F) being displayed on the terminal, as shown below.



9. You may vary the temperature by placing your finger on the temperature sensor as shown in the below figure.



Summary: Using MPLAB® Harmony, you were able to create a sensor application to read temperature from a temperature sensor.

The SPI driver was configured for interrupt mode of operation, thereby running the SPI state machine from the SPI ISR. The SPI Driver Instance 0 was configured for SPI peripheral instance 2 (SPI_ID_2) as the temperature sensor is interfaced to SPI peripheral instance 2. The SPI instance 0 was configured (by default) for master mode operation and 1 MHz baud rate.

The timer system service was configured to generate events every 500 milliseconds. MHC automatically enabled the associated timer driver used by the timer system service. The console system service was configured to use the USB COM (serial) port emulation. MHC automatically enabled the associated USB driver used by the console system service and you used a pre-defined configuration from a demo to configure the USB stack.

In the next lab, you will add another application to write the temperature readings to a memory device.

Notes:

LAB2:
Use MPLAB® Harmony Driver in Multi-Instance Configuration

LAB 2: Use MPLAB® Harmony Driver in Multi-Instance Configuration

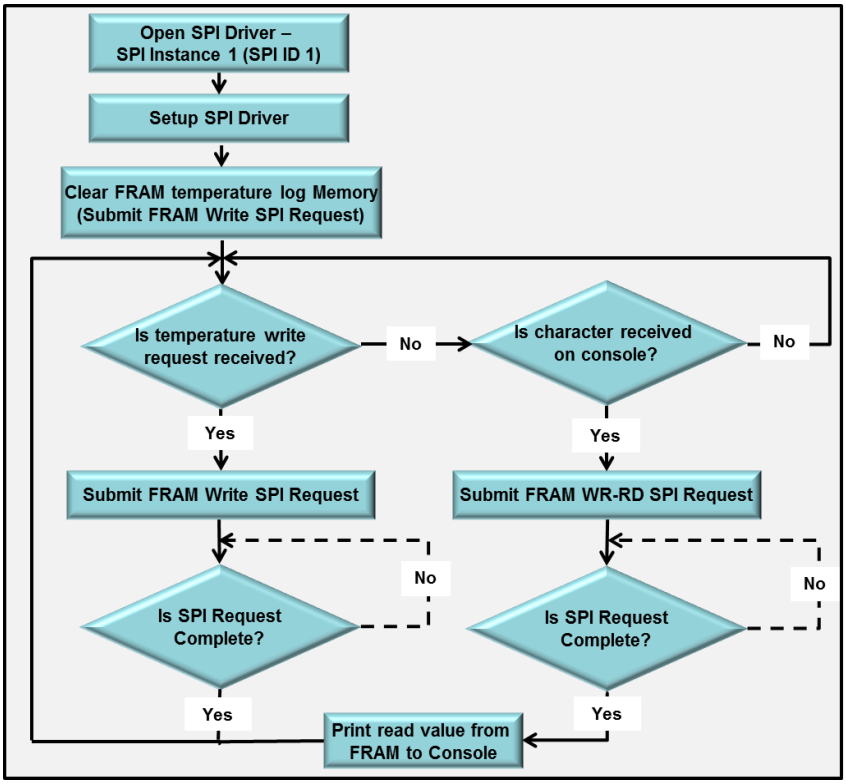
Purpose:

After completing Lab 2, you will have an understanding of how to use multiple instances of Harmony Drivers.

Overview:

In this lab you will add a second application to your project. The second application will write the temperature readings to FRAM. It will also read the temperature values written to FRAM when the user presses the “Enter” key on the console. The read value of temperature will be sent to the console using the console system service, which is already enabled in Lab1.

Flowchart:



Callbacks:

FRAM_APP_BufferEventHandler:

bufferStatus = event;

(event = SPI_BUFFER_STATUS_COMPLETE OR event = SPI_BUFFER_STATUS_ERROR)

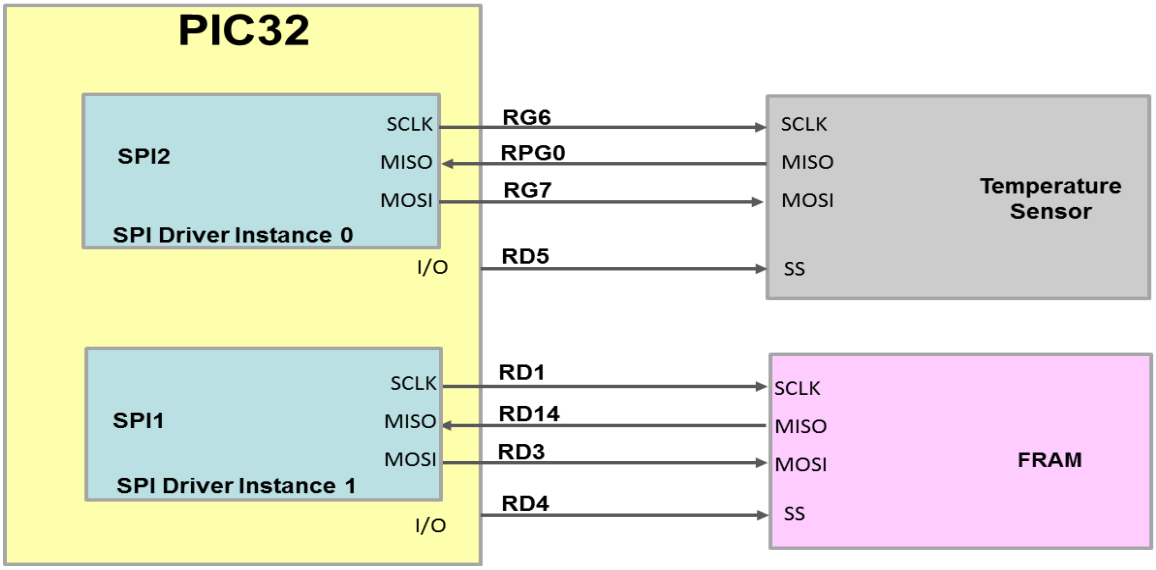
FRAM_APP_OpStartingHandler:

FRAM_CS_EN();

FRAM_APP_OpEndingHandler:

FRAM_CS_DIS();

Connection Diagram:



This lab is completed in the following steps:

Part 1: Configure second instance of SPI Driver

Step 1: With MHC, create FRAM Application.

Step 2: With MHC, configure second instance of SPI Driver and SPI pins.

Step 3: With MHC, generate code.

Part 2: Add application code and build the application

Step 4: Add application code to the FRAM Application

Step 5: Build and Run the Application

Note: You will develop Lab2 by extending the Lab1. If for some reason you were unable to complete the Lab1 successfully, then follow the below procedure:

1. Close the Lab1 project (**21017_dev6_lab**) in MPLAB® X IDE by going to **File>Close All Projects**
2. Delete the **21017_dev6_lab** folder created by you under the **apps/21017_dev6_labs** folder.
3. Copy the ready-made solution for the Lab1 available under **apps/21017_dev6_labs/solutions/Lab1** and paste the solution to the **apps/21017_dev6_labs** folder.
4. Drag and drop the MPLAB X project file (**21017_dev6_lab.X**) available under **apps/21017_dev6_labs/21017_dev6_lab/firmware/** to the MPLAB X IDE
5. In MPLAB X IDE, right click on the project and set it as the main project.
6. In MPLAB X IDE, go to **Tools > Embedded** and open the **MPLAB Harmony Configurator**.

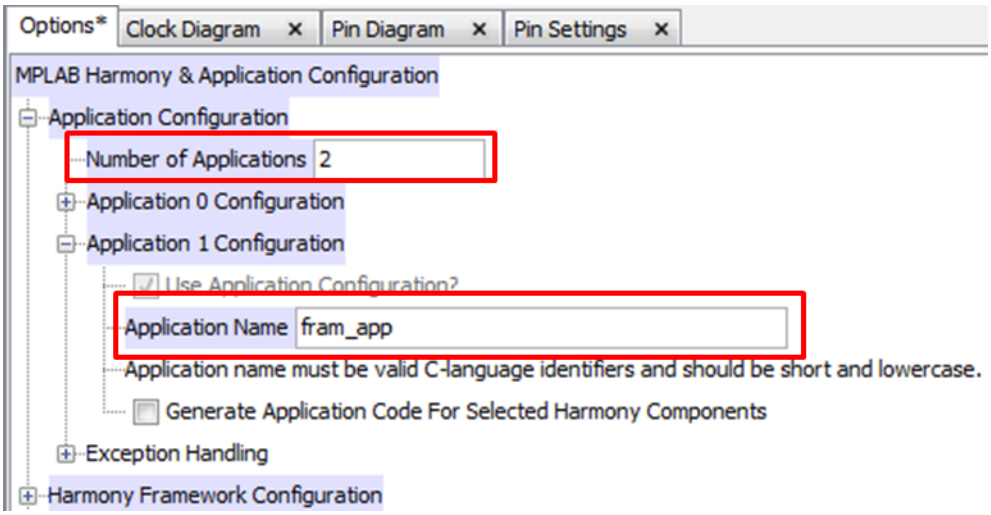
Part 1: Configure second instance of SPI Driver

Step 1: With MHC, create FRAM Application.

1. In the MHC **Options** tab, expand **Application Configuration**

2. Change the **Number of Applications** to 2

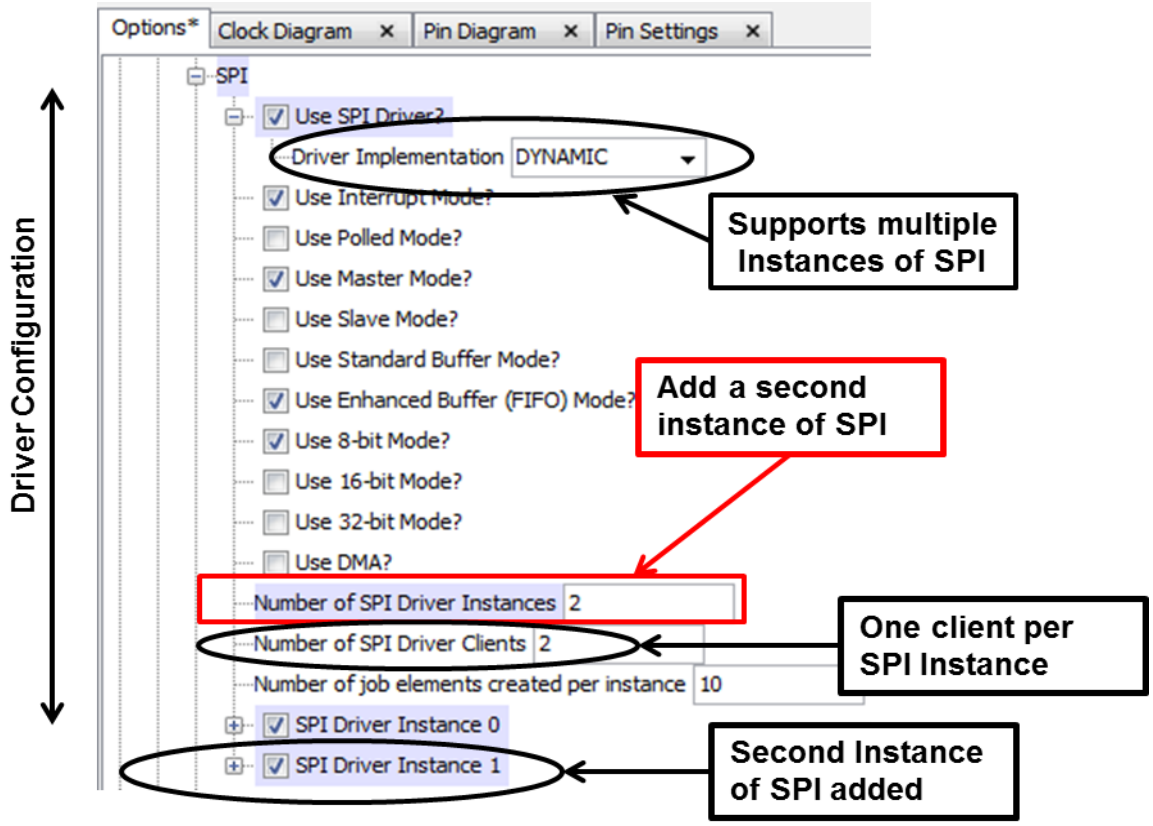
3. Expand **Application 1 Configuration** and change the **Application Name** to “fram_app” as shown below.



Step 2: With MHC, configure second instance of SPI Driver and SPI pins.

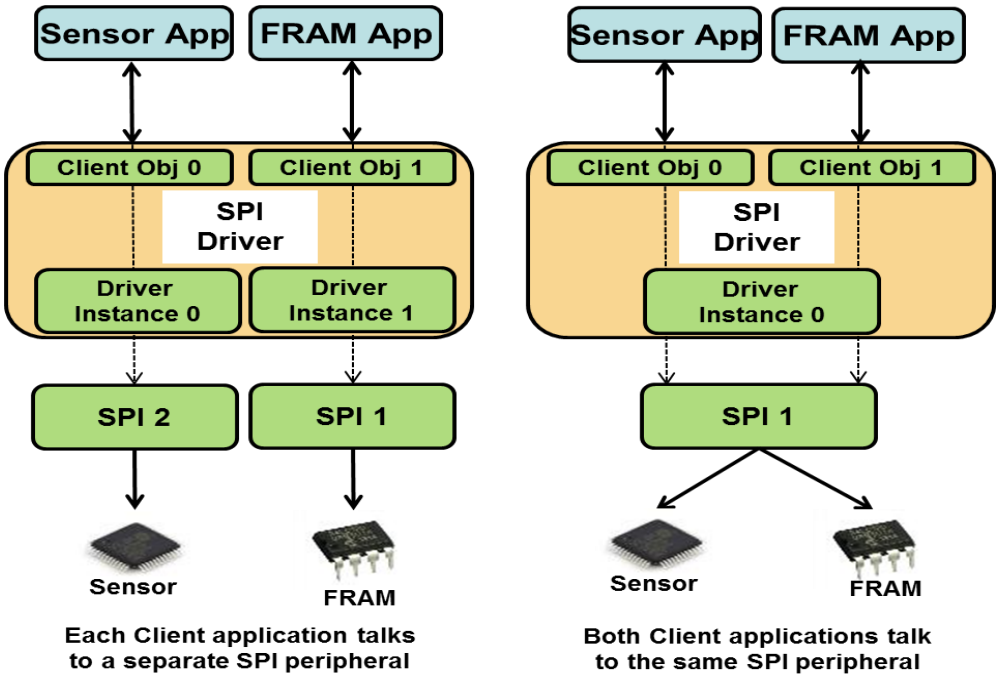
1. Expand the MPLAB **Harmony Framework Configuration > Drivers > SPI**

   a. Change the **Number of SPI Driver Instances** to 2



Note: Observe that the **Driver Implementation** is selected as DYNAMIC. This allows multiple instances of a peripheral to be handled by a single copy of the driver code.

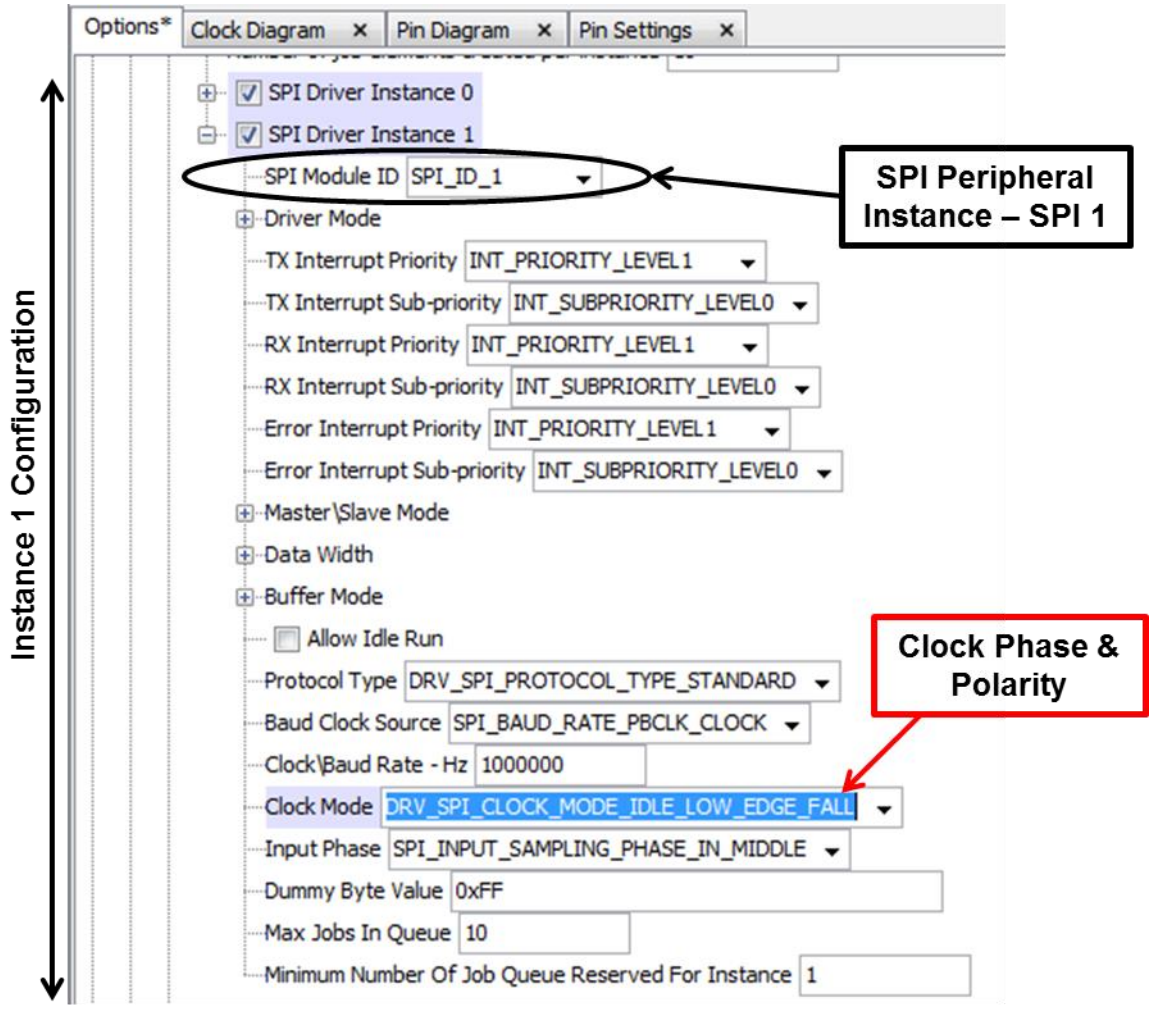
Also, notice that the **Number of SPI Driver Clients** automatically changes to 2. MHC assumes minimum one client for each instance of the driver instance enabled.
If multiple clients were to share a single instance of a SPI Driver, the **Number of SPI Driver Instances** would be selected as 1 but the **Number of SPI Driver Clients** would be changed to 2 or more.



b. Expand the **SPI Driver Instance 1**.

By default, the SPI Driver Instance 1 is associated with SPI peripheral instance 1 (SPI_ID_1). Accept the setting as it is, since FRAM is interfaced to SPI_ID_1.

c. Select SPI **Clock Mode** to DRV_SPI_CLOCK_MODE_IDLE_LOW_EDGE_FALL

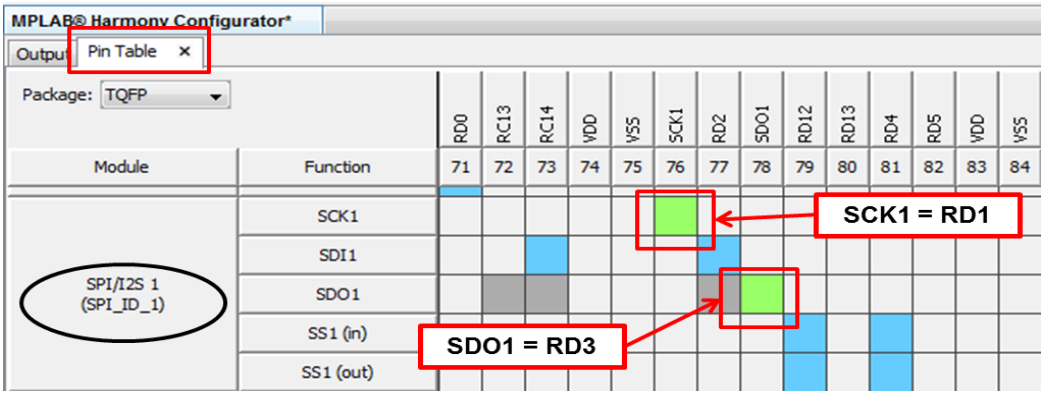


Note: The SPI Driver Instance 1 is by default configured for Interrupt Mode, SPI Master Mode and the SPI clock is set to 1 MHz. The maximum number of requests that can be queued is set to 10.

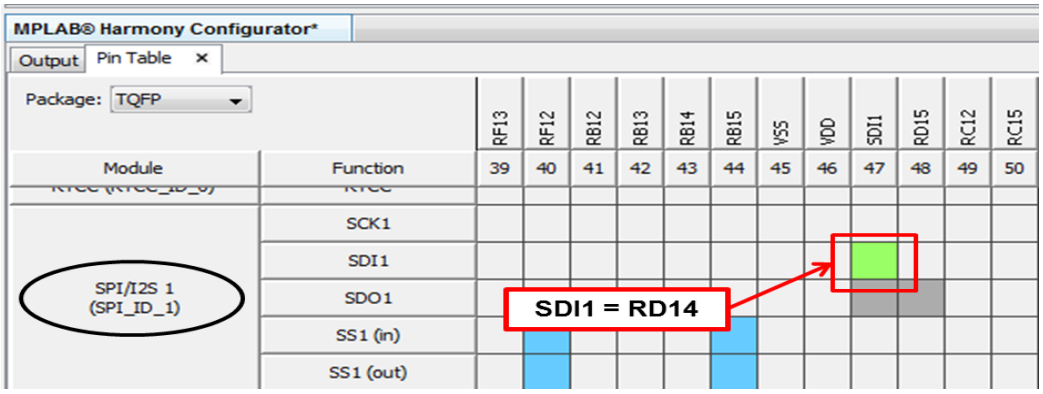
2. Select the **Pin Table** tab as shown below and then scroll down to the **SPI_ID_1** module.

   a. Enable SPI Clock (**SCK1**) on RD1 (Pin #76)

   b. Enable Serial Data Out (**SDO1**) on RD3 (Pin #78)

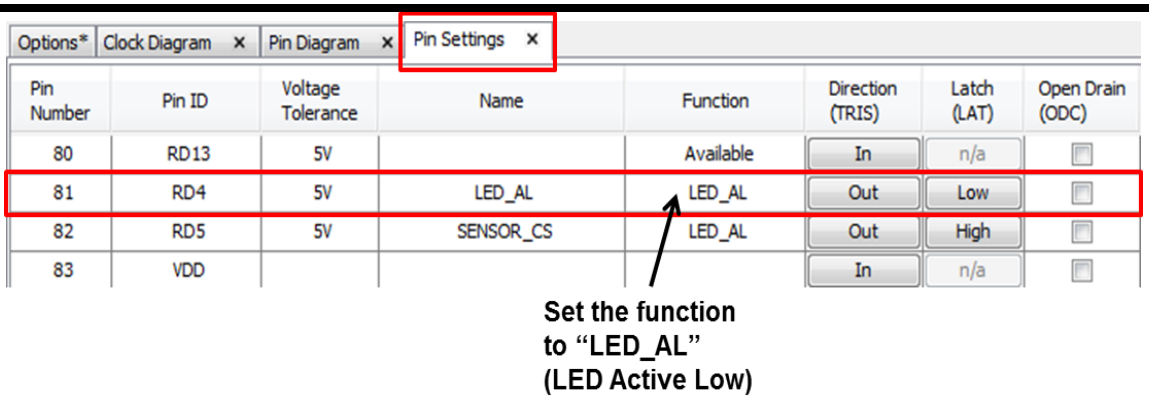


   c. Enable Serial Data In (**SDI1**) on RD14 (Pin #47)

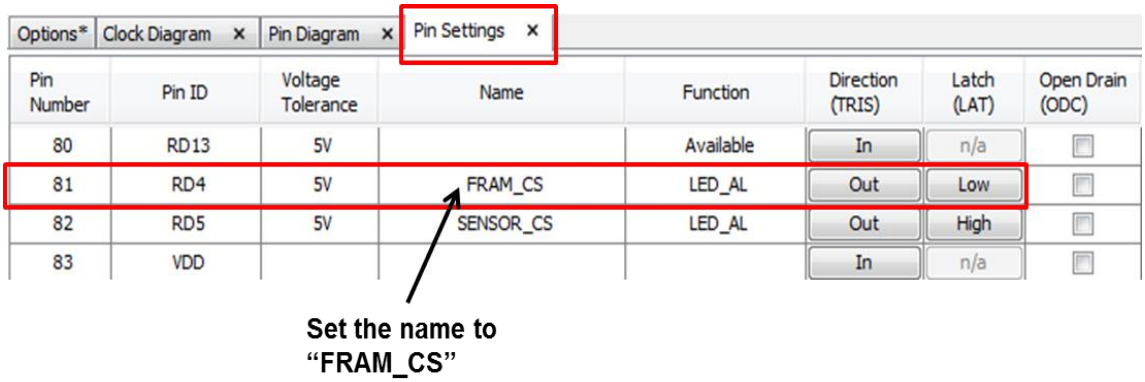


3. Select the **Pin Settings** tab as shown and go to the Pin **RD4** (Pin #81) which is mapped in the hardware to the FRAM chip select pin.

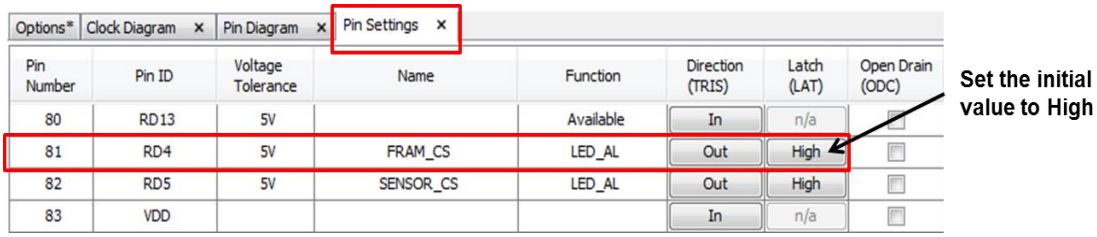
   a. As the Chip Select to the FRAM is active low, set the **Function** drop-down list to LED_AL. This will set the **Name** to LED_AL, **Direction** to Out and **Latch** to the Low state.



b. Change the **Name** to FRAM_CS. MHC will generate the `FRAM_CSOn(),FRAM_CSOff(),FRAM_CSToggle() and FRAM_CSStateGet()` functions for the Chip Select pin in the system_config.h file.

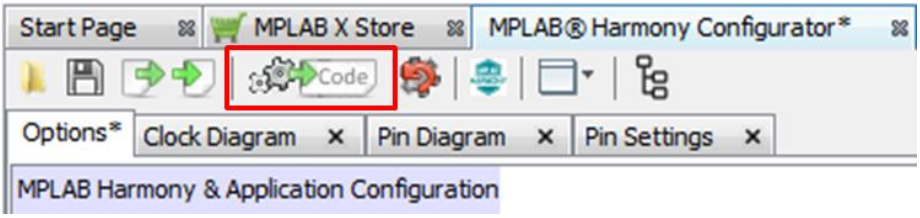


c. Set the initial value of the Chip Select pin to the High state.



Step 3: With MHC, generate code.

When done, generate the code as shown below.



The MHC will include all the MPLAB® Harmony library files and generate the application starter code and the configuration code based on the MHC selections.

At this point you are ready to start implementing your application code.

Part 2: Add application code and build the application

Step 4: Add application code to the FRAM Application

Note: Similar to the Lab1, the application state machine for the FRAM application is already developed and is available under "**apps/21017_dev6_labs/help/lab2**" folder. In the following steps, you will replace the contents of the **fram_app.h** and **fram_app.c** files in your project, with the contents of these pre-developed files.

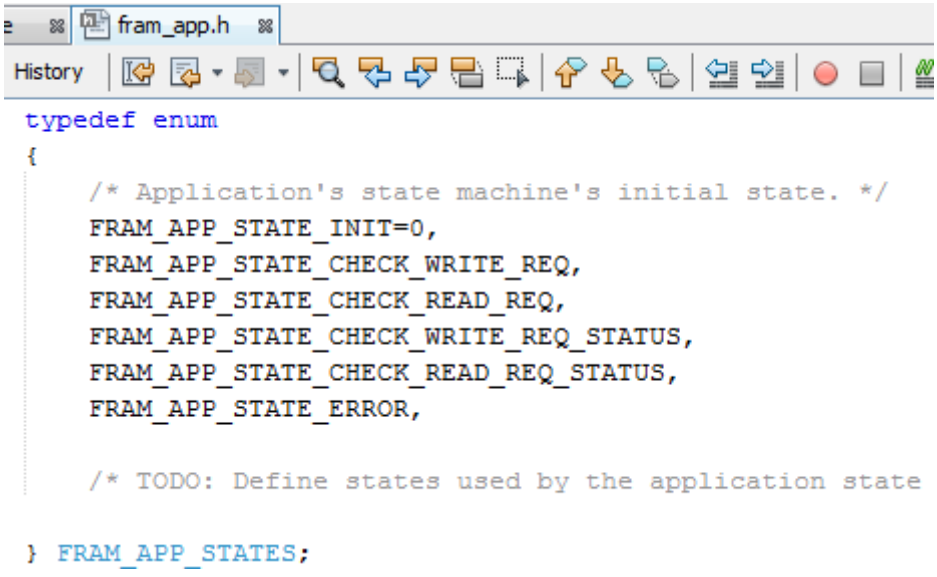
1. Go to the "**apps/21017_dev6_labs/help/lab2**" folder and open the pre-developed **fram_app.h** file.

2. Copy all the contents from the pre-developed fram_app.h file and paste (over-write) it to the fram_app.h file of your project.

3. Similarly, go to the "**apps/21017_dev6_labs/help/lab2**" folder and open the pre-developed **fram_app.c** file.

4. Copy all the contents from the pre-developed fram_app.c file and paste (over-write) it to the fram_app.c file of your project.

   Note: The **fram_app.h** file defines the states of the FRAM application. Try to relate the states with the flowchart shown in the beginning of this lab.



Note: The **`FRAM_APP_Tasks()`** function in **fram_app.c** file implements the FRAM application state machine. It also contains empty event handlers and application states that you will populate with necessary code in the next steps.

5. Open **fram_app.c** and add application code.

> **Tip**: Search for the string **"Step #"** in **fram_app.c** file to locate position where the code snippets need to be copied.

1. Add code to open the SPI driver instance 1 (SPI_ID_1) in the `FRAM_APP_STATE_INIT` state. The call to `DRV_SPI_Open()` API will associate the FRAM application (client) with the SPI driver instance 1. The returned handle will be used by the FRAM application in all the subsequent calls to the driver.

```
case FRAM_APP_STATE_INIT:
{
    DRV_SPI_CLIENT_DATA clientData = {0};


    /* Open SPI driver (Instance 1, SPI ID 1) here ----------> (Step #1) */
    fram_appData.handle = DRV_SPI_Open( DRV_SPI_INDEX_1, DRV_IO_INTENT_READWRITE);
```

Search for "Step #1" and add the below code snippet.

Code snippet:

```
fram_appData.handle = DRV_SPI_Open( DRV_SPI_INDEX_1,
DRV_IO_INTENT_READWRITE);
```

2. Add code to set the operation starting and ending event handlers in the `FRAM_APP_STATE_INIT` state after a valid handle to the driver is obtained. These callbacks will be called by the SPI driver state machine during the SPI request execution to allow FRAM application to chip select/de-select the FRAM memory.

```
//clientData.baudRate = DRV_SPI_BAUD_RATE_IDX1;
clientData.operationStarting = FRAM_APP_OpStartingHandler;
clientData.operationEnded = FRAM_APP_OpEndingHandler;

/* Set client configuration here ---------> (Step #2) */
DRV_SPI_ClientConfigure(
    fram_appData.handle,
    &clientData
);
```

Search for "Step #2" and add the below code snippet.

Code snippet:

```
DRV_SPI_ClientConfigure(
      fram_appData.handle,
      &clientData
);
```

3. Add code to enable FRAM chip select in the operation starting event handler - `FRAM_APP_OpStartingHandler()` which was registered in the previous step. Remember that you have already configured the chip select pin in the Pin Settings tab under Step 2, based on which MHC has generated the functions (like turn on, turn off, toggle etc.) in the system_config.h file to control the chip select pin.

```
/* Application's SPI Instance 1 Operation Starting Callback */
static void FRAM_APP_OpStartingHandler(
    DRV_SPI_BUFFER_EVENT event,
    DRV_SPI_BUFFER_HANDLE bufferHandle,
    void* context
)
{
    /* Chip Select FRAM here ----------> (Step #3) */
    FRAM_CSOn();
}
```

Search for "Step #3" and add the below code snippet.

Code snippet:

```
FRAM_CSOn();
```

4. Add code to get temperature value from the sensor application and write it to the FRAM in the `FRAM_APP_STATE_CHECK_WRITE_REQ` state after the sensor application has signaled the FRAM that the temperature value is available. The `FRAM_APP_Write()` is an application function that takes the memory address to write to, pointer to the data to be written and the number of bytes to write. The `FRAM_APP_Write()` function first enables the writes to FRAM by calling the `FRAM_APP_WriteEnable()` function, after which the actual write is performed.

```
if (true == fram_appData.isWriteReq)
{
    fram_appData.isWriteReq = false;

    framAddr = FRAM_APP_GetNextFramAddrToWrite();

    /* Get Temperature and write to FRAM here ---------> (Step #4)*/

    temperatureVal = SENSOR_APP_GetTemperature();

    FRAM_APP_Write(framAddr, (uint8_t*)&temperatureVal, sizeof(temperatureVal));

    fram_appData.state = FRAM_APP_STATE_CHECK_WRITE_REQ_STATUS;
}
```

Search for "Step #4" and add the below code snippet.

Code snippet:

```
temperatureVal = SENSOR_APP_GetTemperature();

FRAM_APP_Write(framAddr, (uint8_t*)&temperatureVal,
sizeof(temperatureVal));
```

5. Add code to read temperature value from FRAM in the `FRAM_APP_STATE_CHECK_READ_REQ` state after the user presses "Enter" key on the console. The `FRAM_APP_Read()` is an application function that takes the starting memory address to read the data from and the number of bytes to read. The function performs a SPI write-read operation by calling the `DRV_SPI_BufferAddWriteRead2()` API. The read value is saved in the array which is part of the `framReadData` data structure.

```
case FRAM_APP_STATE_CHECK_READ_REQ:
    if (SYS_STATUS_READY == SYS_CONSOLE_Status(sysObj.sysConsole0))
    {
        if (0x0d == fram_appData.keyValue)
        {
            /* Read back saved temperature value from FRAM here ---------> (Step #5) */

            FRAM_APP_Read(TEMPERATURE_LOG_START_ADDR, MAX_TEMPERATURE_LOGS * sizeof(float));

            fram_appData.state = FRAM_APP_STATE_CHECK_READ_REQ_STATUS;
        }
```

Search for "Step #5" and add the below code snippet.

Code snippet:

```
FRAM_APP_Read(TEMPERATURE_LOG_START_ADDR,
MAX_TEMPERATURE_LOGS * sizeof(float));
```

6. Add code to update the status of the submitted SPI request in the SPI buffer event handler `FRAM_APP_BufferEventHandler()`. This buffer event handler is registered with the driver for every SPI request and will be called by the SPI driver after each SPI request is completed.

```
case DRV_SPI_BUFFER_EVENT_COMPLETE:
    if (context)
    {
        /* Copy the buffer status here ----------> (Step #6) */
        *((FRAM_APP_SPI_BUFFER_STATUS*)context) = FRAM_APP_SPI_BUFFER_STATUS_COMPLETE;

    }
    break;
```

Search for "Step #6" and add the below code snippet.

Code snippet:

```
*((FRAM_APP_SPI_BUFFER_STATUS*)context) =
FRAM_APP_SPI_BUFFER_STATUS_COMPLETE;
```

7. Add code to print the temperature value to the console using the console system service in the `FRAM_APP_STATE_CHECK_READ_REQ_STATUS` state after the FRAM read is complete.
   The `FRAM_APP_PrintTemperatureToConsole()` is an application function that takes a pointer to the temperature values read from FRAM. The function prints the temperature values on the console with the oldest value on the top and the latest value at the last. The temperature values

read from FRAM are stored at index 3 as the first 3 bytes contain the dummy data received for the SPI write operation.

```
case FRAM_APP_STATE_CHECK_READ_REQ_STATUS:

    /* Check the buffer status which is updated in the registered callback */
    if (FRAM_APP_SPI_BUFFER_STATUS_COMPLETE == fram_appData.framReadData.bufferStatus)
    {
        /* Print read value to console here ---------> (Step #7) */

        FRAM_APP_PrintTemperatureToConsole((uint8_t*)&fram_appData.framReadData.rdBuffer[3]);

        fram_appData.state = FRAM_APP_STATE_CHECK_WRITE_REQ;
    }
```

Search for "Step #7" and add the below code snippet.
Code snippet:

```
FRAM_APP_PrintTemperatureToConsole((uint8_t*)&fram_appData.framReadData.rdBuffer[3]);
```

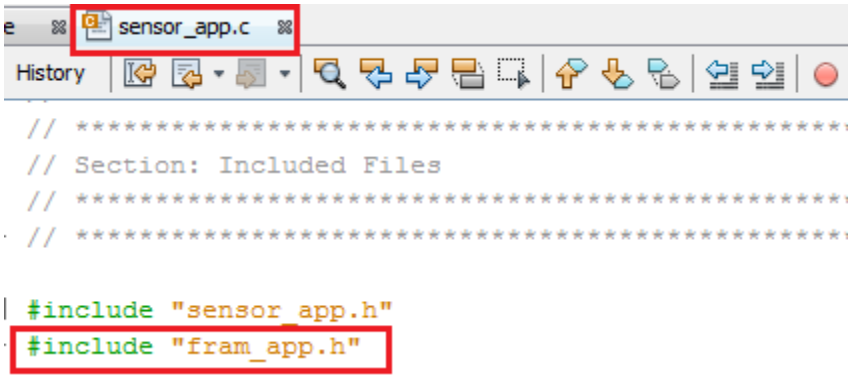
8. Now, you need to enable your sensor application to request the fram application to write temperature values to FRAM.

   Open **sensor_app.c** file and uncomment the call to function `FRAM_APP_SetWriteReq()` in the `SENSOR_APP_STATE_WAIT_MEAS_COMPLETE` state. This will signal the FRAM application to write the temperature value to FRAM.

   Search for "Lab 2" and uncomment the below line of code.

```
if (DRV_SPI_BUFFER_EVENT_COMPLETE & DRV_SPI_BufferStatus(sensor_appData.temperatureData.bufferHandle))
{
    sensor_appData.temperature = SENSOR_APP_CalcTemperature(
            &sensor_appData.temperatureData.rdBuffer[1],
            sensor_appData.sensorCalibValues);

    /* Print to console using System Console Service here ----------> (Step #8) */
    SYS_PRINT("Temperature:%.2f\r", sensor_appData.temperature);



    /* Un-comment below line for ---------> (Lab 2) */
    FRAM_APP_SetWriteReq(); Un-Comment

    sensor_appData.state = SENSOR_APP_STATE_WAIT_MEAS_REQ;
}
```

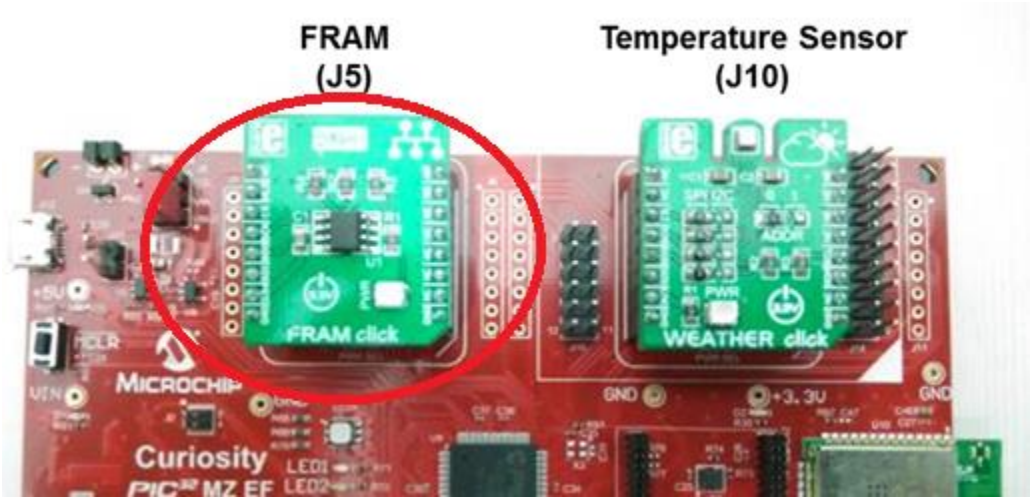
9. The `FRAM_APP_SetWriteReq` is declared in fram_app.h file. Add this declaration by including the fram_app.h file in the sensor_app.c file.



Congratulations! You are ready to build and run your second application.

Step 5: Build and Run the Application

1. Verify that the FRAM Click board is connected to the mikroBUS header **J5** of the **PIC32MZ EF Curiosity Development board**.



2. Make sure the micro-B USB cable is connected between micro-B port J12 on the board and the PC.

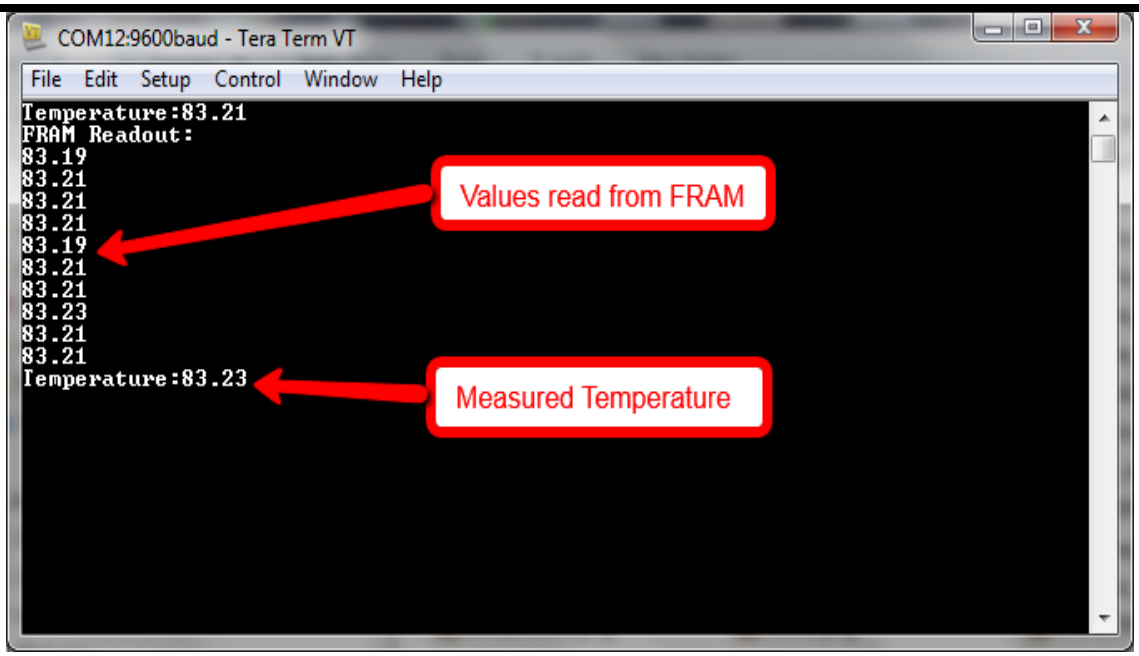
3. **If already open, then close the previous session of Tera Term.**

4. Build the code and program your application to the device, by clicking on the "Make and Program" button as shown below.

5. Open the Tera Term terminal application on your PC.

6. Once opened, press the "Enter" key on your PC to display the temperature values on the terminal.

7. Press the "Enter" key again on your PC to display the last written values to the FRAM on the terminal.



Summary: Using MPLAB® Harmony, you were able to add another application to your project. The new application writes the temperature readings to a FRAM interfaced over SPI.
The second instance of SPI (SPI Driver Instance 1) was enabled. The SPI driver instance 1 was associated with SPI peripheral instance 1 (SPI_ID_1) as the FRAM is interfaced to SPI peripheral instance 1.
In the process you learnt how to configure and use multiple instances of Harmony drivers in your project.

In the next lab, you will add RTOS to your application!

Notes:

LAB 3:

Add RTOS to the Application

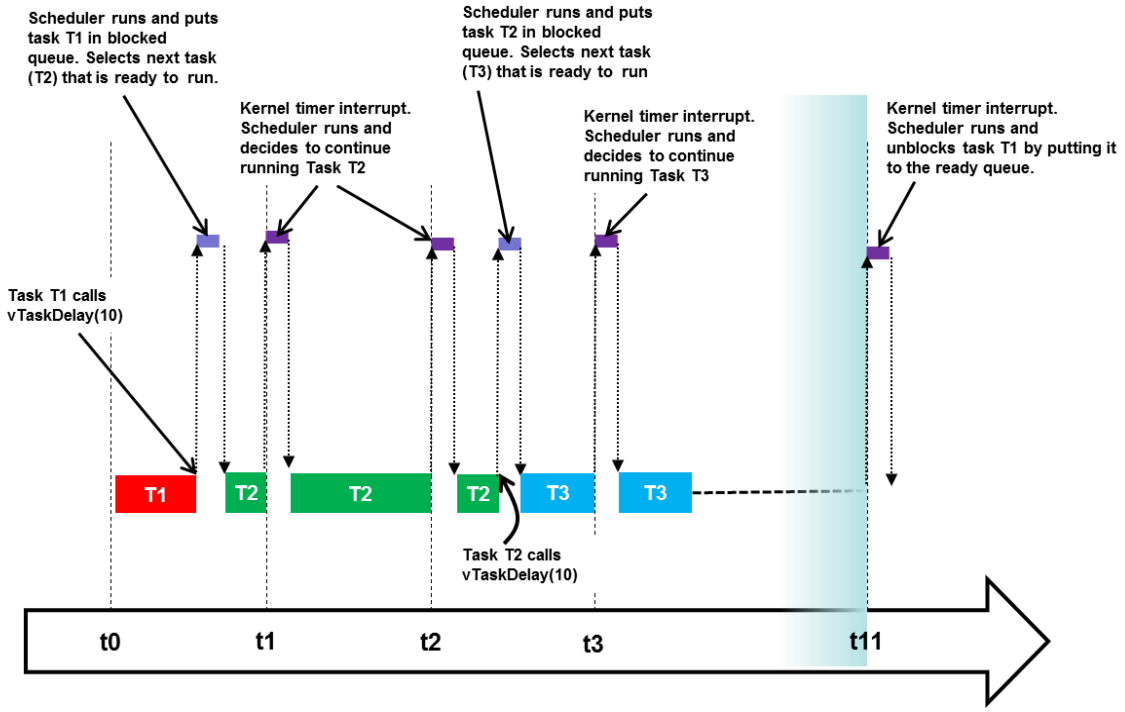
LAB 3: Add RTOS to the Application

Purpose:

After completing Lab 3, you will be able to run your application using RTOS. You will be able to configure Application Tasks, System Services and Driver Tasks to run in an RTOS environment.

Overview:

In this Lab you will add FreeRTOS to your application. You will configure and enable some of the FreeRTOS features. You will also configure your sensor application, FRAM application, Timer and Console system services, System Tasks and Drivers to run with FreeRTOS. You will verify that the output remains same when the application runs in an RTOS environment.



Task execution with tasks T1, T2 and T3 running at same priority

This lab is completed in the following steps.

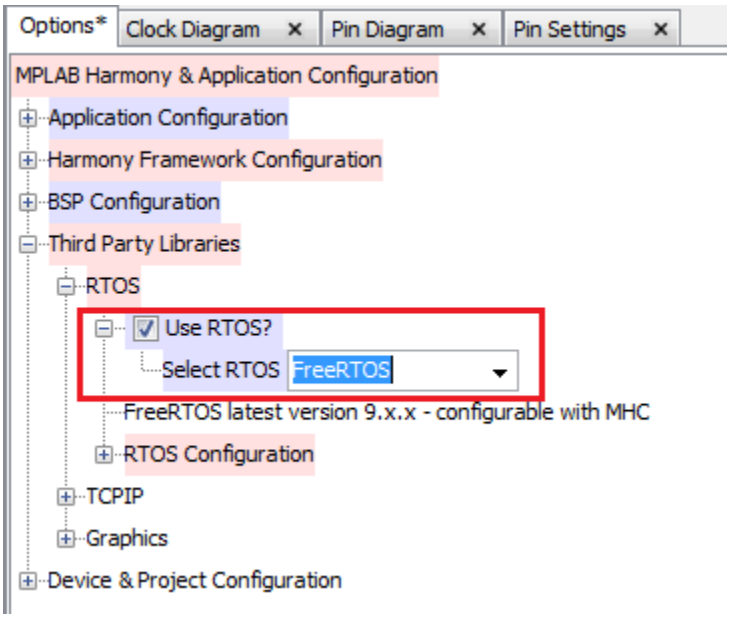
Step 1: With MHC, enable and configure FreeRTOS.
Step 2: With MHC, configure Sensor Application and FRAM Applications to run with RTOS.
Step 3: With MHC, configure System Service task to run with RTOS.
Step 4: With MHC, configure Console and Timer System Service tasks to run with RTOS
Step 5: With MHC, configure USB task to run with RTOS.
Step 6: With MHC, configure FreeRTOS heap size.
Step 7: With MHC, generate code.
Step 8: Add Application Code to Sensor Application (Optional)
Step 9: Build and Run the Application

Note: You will develop Lab3 by extending the Lab2. If for some reason you were unable to complete the Lab2 successfully, then follow the below procedure:

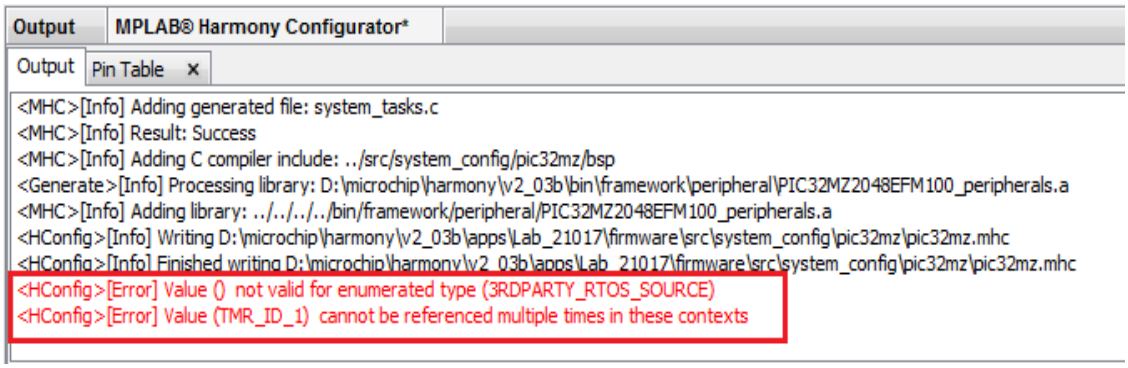
1. Close the Lab2 project (**21017_dev6_lab**) in MPLAB® X IDE by going to **File>Close All Projects**
2. Delete the **21017_dev6_lab** folder created by you under the **apps/21017_dev6_labs** folder.
3. Copy the ready-made solution for the Lab2 available under **apps/21017_dev6_labs/solutions/Lab2** and paste the solution to the **apps/21017_dev6_labs** folder.
4. Drag and drop the MPLAB X project file (**21017_dev6_lab.X**) available under **apps/21017_dev6_labs/21017_dev6_labs/firmware/** to the MPLAB X IDE
5. In MPLAB X IDE, right click on the project and set it as the main project.
6. In MPLAB X IDE, go to **Tools > Embedded** and open the **MPLAB Harmony Configurator**.

Step 1: With MHC, enable and configure FreeRTOS.

1. In MHC **Options** tab, expand **Third Party Libraries > RTOS**
2. Select the **Use RTOS**? Checkbox.
3. Select FreeRTOS in the drop-down list for **Select RTOS**

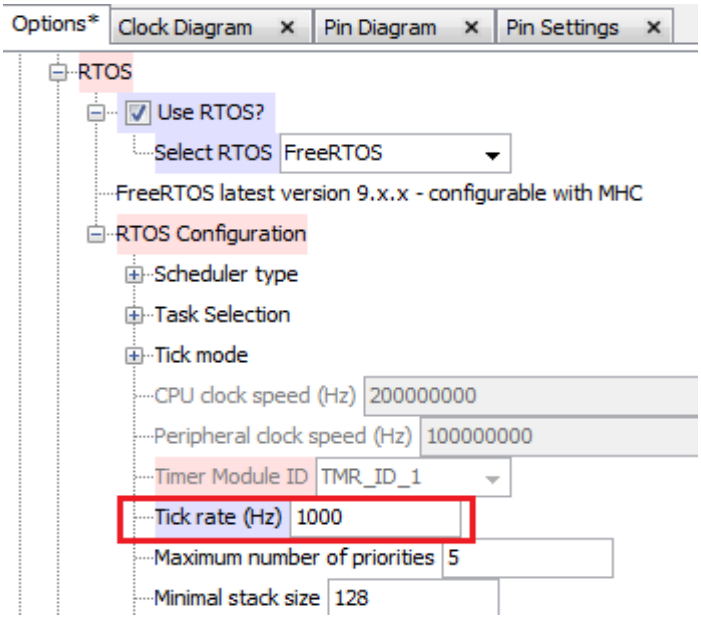


4. At this point, the MHC should show an error indicating that TMR_ID_1 cannot be referenced multiple times.

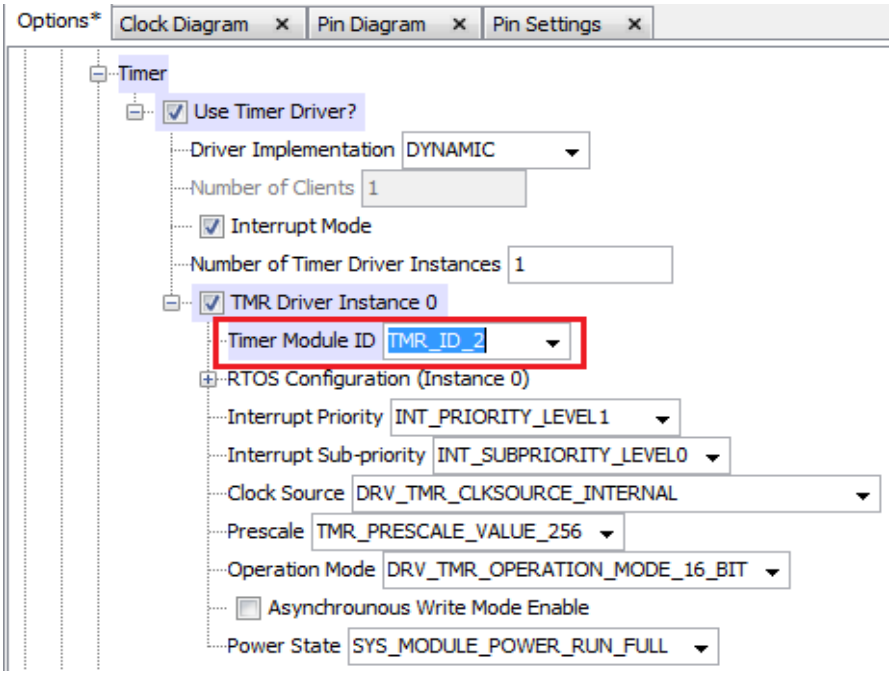


This is because, FreeRTOS uses TMR_ID_1 for generating kernel ticks and the Timer System Service also (by default) uses TMR_ID_1 for generating timer tick interrupts.
The MHC does not currently allow you to change the timer used by FreeRTOS, so you will resolve this error in the sub-step #6, by selecting a different timer peripheral for the Timer System Service.

5. Expand **RTOS > RTOS Configuration** and change the RTOS **Tick Rate (Hz)** to 1000. This will configure the kernel tick (which is generated using the TMR_ID_1) to occur every 1 millisecond.



6. Expand MPLAB **Harmony Framework Configuration > Drivers > Timer > Use Timer Driver? > TMR Driver Instance 0**.
Here, change the value of the **Timer Module ID** to TMR_ID_2. This should resolve the MHC error and also remove the red marked entries in MHC indicating an error.

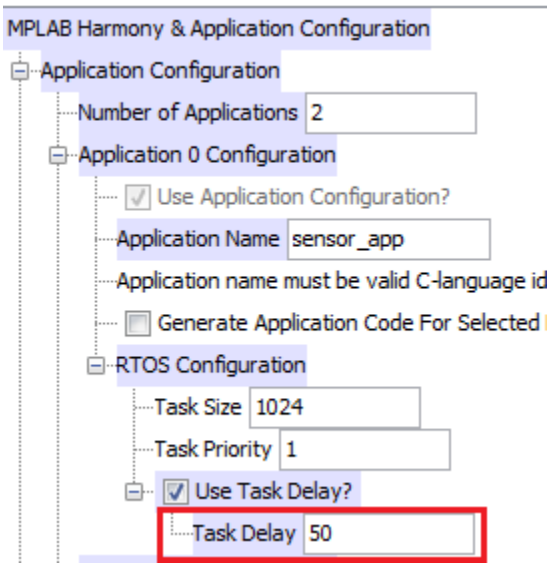


Step 2: With MHC, configure Sensor Application and FRAM Application to run with RTOS.

1. Expand the **Application Configuration > Application 0 Configuration > RTOS Configuration > Use Task Delay?**

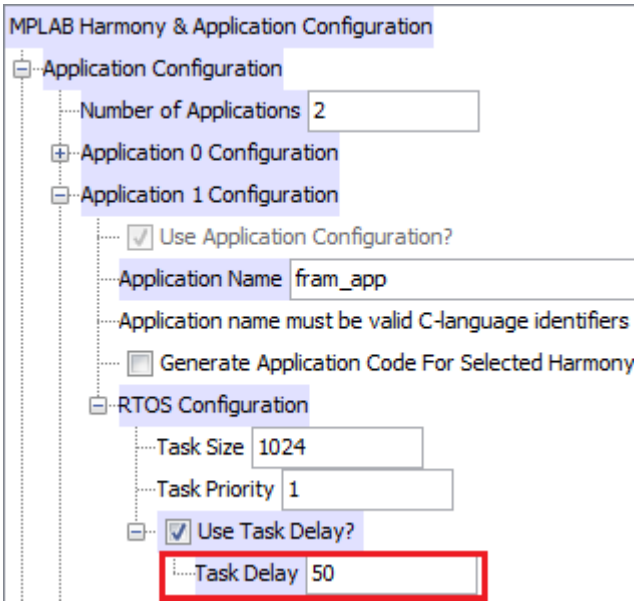
   Change the **Task Delay** to 50. The unit of the **Task Delay** is milliseconds. This will cause the Sensor Application thread to put itself in a blocked state for 50 milliseconds and thereby allow other tasks to run.

   Note: The RTOS tick rate is already configured to 1000 Hz in Step 1.
   The Task Size indicates the stack size of the task and is in word (32-bit) units. Accept the default value of 1024 words (4096 bytes). The Task Priority is by default set to 1, which can vary from 0-4 as the Maximum number of priorities is set to 5 in **RTOS > RTOS Configuration.** The configured values for the Task Size and Task Priority will be passed as arguments (by the code generated by MHC), to the FreeRTOS thread creation routines.



2. Similarly, expand **Application Configuration > Application 1 Configuration > RTOS Configuration > Use Task Delay?**

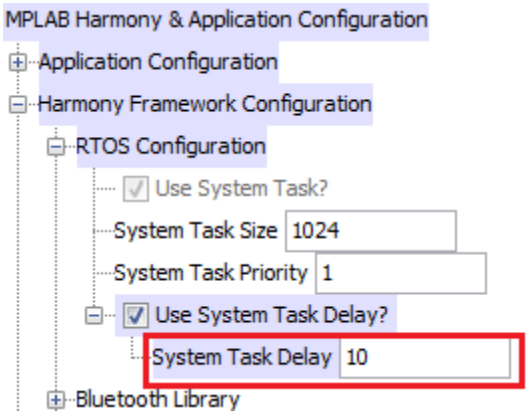
   Change the **Task Delay** to 50. This will cause the FRAM Application thread to put itself in a blocked state for 50 milliseconds and thereby allow other tasks to run.



Step 3: With MHC, configure System Service task to run with RTOS.

1. Expand the **Harmony Framework Configuration > RTOS Configuration > Use System Task Delay?**

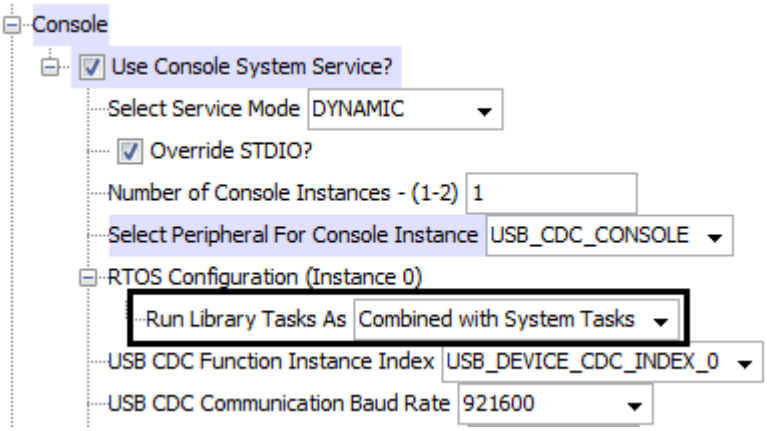
2. Change the **System Task Delay** to 10. This will put the System task thread in a blocked state for 10 milliseconds and yield the CPU for other tasks to run.



Step 4: With MHC, configure Console and Timer System Service tasks to run with RTOS

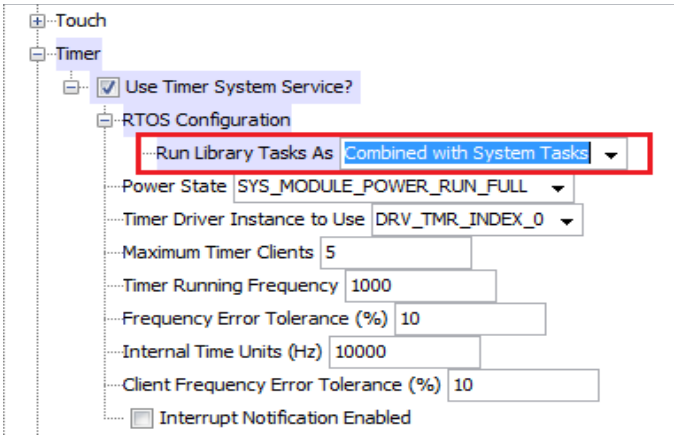
Expand the **Harmony Framework Configuration > System Services > Console > Use Console System Service? > RTOS Configuration (Instance 0)**

1. Accept the default value for **Run Library Tasks As** "Combined with System Tasks". Selecting this option will cause MHC to not generate a separate FreeRTOS thread for Console System Service. Instead, the Console System Service task will be run as part of the System Service Task, which you configured in the previous step (Step #3).



2. Expand the **Harmony Framework Configuration > System Services > Timer > Use Timer System Service? > RTOS Configuration**

3. Change the value of **Run Library Tasks As** to "Combined with System Tasks". This will cause the Timer System Service task also to be combined with the System Tasks thread.



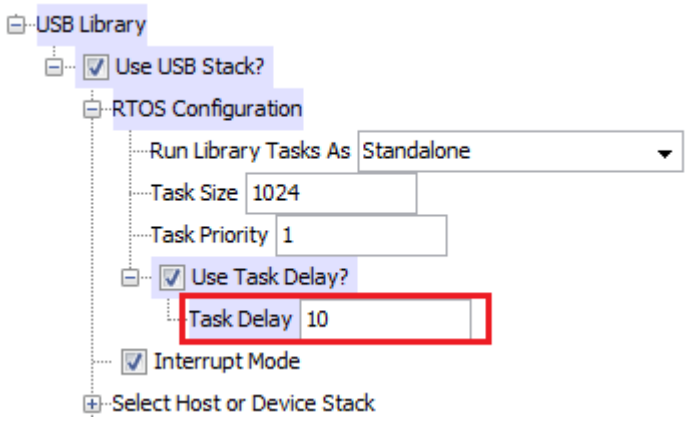
Thus, both the Console and Timer System Service tasks will be run from the common System task RTOS thread.

Note: Using individual threads allows the user to better prioritize the tasks of different libraries. However, each thread takes resources and has task-switching overhead, using up CPU bandwidth. So, it often makes sense to combine several simple libraries into a common thread as we've done here while leaving major libraries (like USB, as shown in the next step) in its own thread.

Step 5: With MHC, configure USB task to run with RTOS.

1. Expand the **Harmony Framework Configuration > USB Library > Use USB Stack? > RTOS Configuration**

2. Change the value of **Use Task Delay?** to 10. This will cause the USB task thread to put itself in a blocked state for 10 milliseconds and thereby allow other tasks to run. Also, note that the USB library is left to run from in its own thread to allow it to run independently of other apps and services.



Step 6: With MHC, configure FreeRTOS heap size.

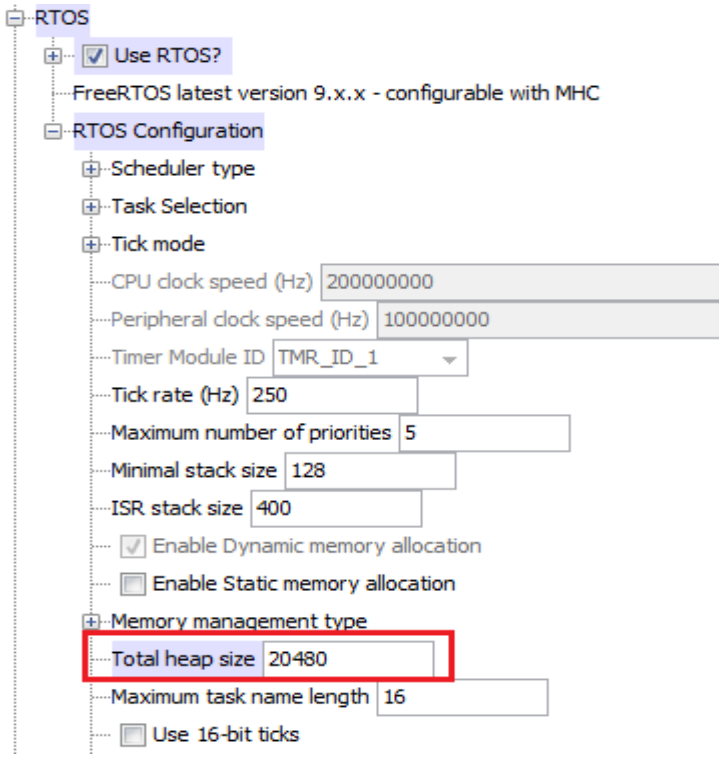
1. Expand the **Third Party Libraries > RTOS > RTOS Configuration**

2. Change the value of **Total heap size** to 20480 bytes.

Note: Calculation for FreeRTOS heap size is as follows:

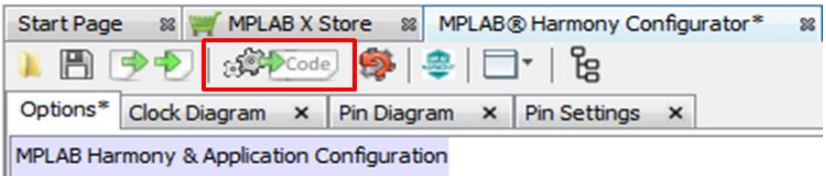
Since the default stack size of 1024 words is used for all the RTOS threads, the total stack size will be the sum of these plus the additional RAM required by the RTOS mutex and semaphores used by the Harmony Libraries and application.

Sensor Task	= 1024 words	= 4096 bytes
FRAM Task	= 1024 words	= 4096 bytes
System Task	= 1024 words	= 4096 bytes
USB Task	= 1024 words	= 4096 bytes
FreeRTOS Idle Task	= 128 words	= 512 bytes
RTOS mutex/Semaphores/queues		~ 3584 bytes (4096 – 512)
(Depends on the application. For this Lab, rounding it to 4KB)		
Total Heap Size		= 20480 bytes



Step 7: With MHC, generate code.

When done, generate the code as shown below.



Analyze the generated code:

Open **system_tasks.c** and observe how MHC simplifies application development by creating RTOS tasks in the `SYS_Tasks()` function, based on the MHC selections.

```
void SYS_Tasks ( void )
{
    /* Create OS Thread for Sys Tasks. */
    xTaskCreate((TaskFunction_t) _SYS_Tasks,
                "Sys Tasks",
                1024, NULL, 1, NULL);

    /* Create OS Thread for USB Tasks. */
    xTaskCreate((TaskFunction_t) _USB_Tasks,
                "USB Tasks",
                1024, NULL, 1, NULL);

    /* Create OS Thread for SENSOR_APP Tasks. */
    xTaskCreate((TaskFunction_t) _SENSOR_APP_Tasks,
                "SENSOR_APP Tasks",
                1024, NULL, 1, NULL);

    /* Create OS Thread for FRAM_APP Tasks. */
    xTaskCreate((TaskFunction_t) _FRAM_APP_Tasks,
                "FRAM_APP Tasks",
                1024, NULL, 1, NULL);


    /***************
     * Start RTOS *
     ***************/
    vTaskStartScheduler(); /* This function never returns. */
}
```

```
static void _SYS_Tasks ( void)
{
    while(1)
    {
        /* Maintain system services */
        SYS_CONSOLE_Tasks(sysObj.sysConsole0);
        /* SYS_TMR Device layer tasks routine */
        SYS_TMR_Tasks(sysObj.sysTmr);

        /* Maintain Device Drivers */

        /* Maintain Middleware */


        /* Task Delay */
        vTaskDelay(10 / portTICK_PERIOD_MS);
    }
}
static void _SENSOR_APP_Tasks(void)
{
    while(1)
    {
        SENSOR_APP_Tasks();
        vTaskDelay(50 / portTICK_PERIOD_MS);
    }
}
```

Console and Timer System Service combined under _SYS_Tasks()

Observe that each of the application and the driver state machines are now run in their individual RTOS tasks. Also, notice how the Timer System Service and the Console System Service are combined to run from a single `_SYS_Tasks()` RTOS task, based on MHC selections made in Step #4.

At this point you are ready to start implementing your application code.

Step 8: Add Application Code to Sensor Application (Optional)

Instead of setting a flag from periodic timer handler, you will use FreeRTOS semaphore to signal the sensor application to read temperature from the sensor.

(Note: This step is optional and can be skipped to go to Step 9)

1. Open **sensor_app.h** file and uncomment the following line of code that holds a handle to the semaphore.
   Search for "Lab 3_1" and un-comment the below line of code.

```
typedef struct
{
    /* The application's current state */
    SENSOR_APP_STATES            state;
    DRV_HANDLE                   handle;
    SENSOR_APP_SPI_WR_REQ        sensorControlData;
    SENSOR_APP_SPI_WR_RD_REQ     calibrationData;
    SENSOR_APP_SPI_WR_RD_REQ     temperatureData;
    uint8_t                      sensorCalibValues[6];
    volatile bool                readTemperatureReq;
    float                        temperature;


    /* Un-comment below line of code for Lab 3 ---------> (Lab 3_1) */
    SemaphoreHandle_t    xSemaphore;   Un-Comment

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

} SENSOR_APP_DATA;
```

2. Open **sensor_app.c** file. Create a binary semaphore by uncommenting the following lines of code.

   Search for "Lab 3_2" and add un-comment the below line of code.

```
void SENSOR_APP_Initialize ( void )
{
    /* Place the App state machine in its initial state. */
    sensor_appData.state = SENSOR_APP_STATE_INIT;

    sensor_appData.readTemperatureReq = false;

    /* Un-comment the below lines of code for Lab 3 ---------> (Lab 3_2) */
                                                    Un-Comment
    sensor_appData.xSemaphore = xSemaphoreCreateBinary();

    if (NULL == sensor_appData.xSemaphore)
    {
        sensor_appData.state = SENSOR_APP_STATE_ERROR;
    }

}
```

3. Update the periodic timer handler to signal the sensor application using a semaphore, instead of setting a flag, by commenting and un-commenting the following lines of code.
   Search for "Lab 3_3" and add comment/un-comment the below lines of code.

```
/* Application's Timer Callback Function */
static void SENSOR_APP_PeriodicTimerCallback(uintptr_t context, uint32_t currTick)
{
    /* Set temperature measurement request here ---------> (Step #5) */
    /* Comment the below line of code for Lab 3 ---------> (Lab 3_3) */
    //sensor_appData.readTemperatureReq = true;          Comment

    /* Un-comment the below line of code for Lab 3 ---------> (Lab 3_3) */

    xSemaphoreGive(sensor_appData.xSemaphore);          Un-Comment
}
```

4. Modify the sensor application to wait on a semaphore instead of waiting on a flag, by commenting and un-commenting the following lines of code. The RTOS scheduler will put the sensor task in a blocked state if the semaphore is not available. This will allow other tasks to run. When the semaphore is released by a call to `xSemaphoreGive()` in the periodic timer handler, the RTOS scheduler will unblock the sensor task.

   Search for "Lab 3_4" and comment/un-comment the below lines of code.

```
case SENSOR_APP_STATE_WAIT_MEAS_REQ:
{
    DRV_SPI_BUFFER_HANDLE handle;

    /* Comment the below line of code for Lab 3 ---------> (Lab 3_4) */
    //if (true == sensor_appData.readTemperatureReq)    Comment

    /* Un-comment the below line of code for Lab 3 ---------> (Lab 3_4) */
    xSemaphoreTake( sensor_appData.xSemaphore, portMAX_DELAY);   Un-Comment
    {
        sensor_appData.temperatureData.wrBuffer[0]    = RD(BME280_TEMP_REG_ADDR);
        sensor_appData.temperatureData.nWrBytes       = 1;
        sensor_appData.temperatureData.nRdBytes       = 1+3;
        sensor_appData.temperatureData.bufferStatus   = SENSOR_APP_SPI_BUFFER_STATUS_INVALID;
```

   Congratulations! You have added FreeRTOS to your application. You are now ready to build and run your RTOS based application!

Step 9: Build and Run the Application

1. Make sure the micro-B USB cable connected between micro-B port J12 on the board and the PC

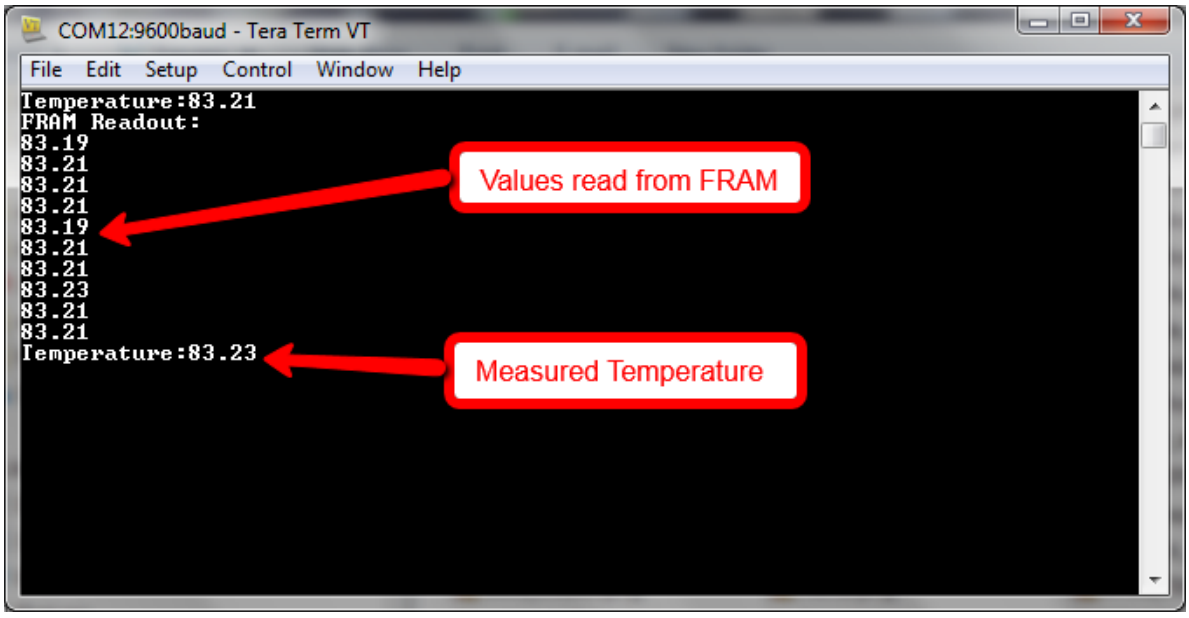
2. **If already open, then close the previous session of Tera Term.**

3. Click on the Make and Program icon to build and program the application.

4. Open the Tera Term terminal application on your PC.

5. Press the “Enter” key on your PC to display the temperature values on the terminal.

6. Press the “Enter” key again on your PC to display the last written value to the FRAM on the terminal.



Summary: Using MPLAB® Harmony, you were able to add FreeRTOS to your application. In the process you gained an understanding on how to configure RTOS parameters using MHC. You also learnt how to configure your application and system services to run in an RTOS environment.

Notes: