Harmony House Call

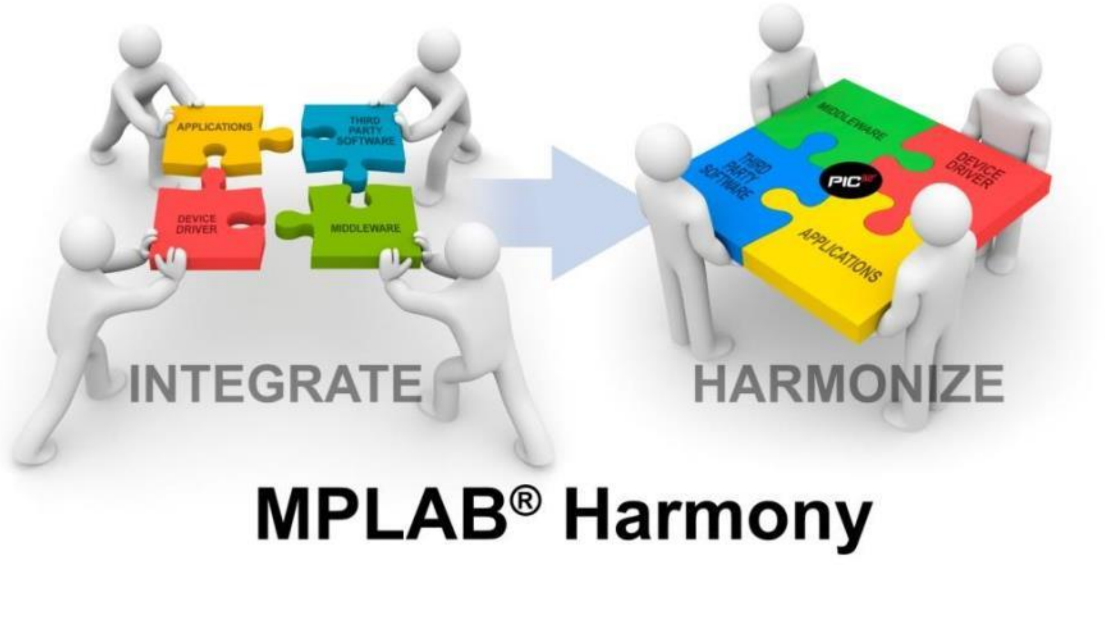
Lab Manual

(for Harmony v1.07.01)

.

April, 2016

Table of Contents



Lab Introduction:

Please carefully complete each step in the labs. Due to the limited time allotted for each lab, the details as to why each configuration is requested are not provided. Upon completion of the class labs, you are encouraged to further your knowledge and understanding on the PIC32 MCU setup and configuration options.

Note: The hardware used in this workshop is the Arduino compatible chipKIT™ WF32 development board from Digilent Inc.

Lab Prerequisites:

This lab material assumes you have prior experience with:

1. MPLAB® X IDE
2. MPLAB® based Programming/Debugging
3. ‘C’ language programming
4. Basic PIC32 product family knowledge

Lab Objectives:

1. Create an MPLAB® X IDE Harmony project for PIC32 from scratch.
2. Use the MPLAB® Harmony Configurator (MHC) to configure and generate Harmony code.
3. Create new application states and variables for Harmony projects.
4. Identify the proper place to add your own application code to the Harmony project.
5. Demonstrate the use of Harmony libraries to control a PIC32 peripheral (ADC).
6. Apply the Harmony Debug System Service (print ADC results to a UART).
7. Use MHC to configure and generate a USB bootloader application.

Hardware / software installation:

PC installation

1. MPLAB® X IDE: **v3.25** (or later…Instructor is using MPLAB v3.25)
2. MPLAB® XC32 compiler: **v1.40** (or later)
   - Lab 2b requires a Standard or PRO (not free) XC32 compiler version to generate 16-bit (MIPS16) code. Note the free version includes a 60-day PRO evaluation that can be started at any time.
3. MPLAB® Harmony **v1.07.01**
4. MPLAB® Harmony Configurator (MHC) plugin for MPLAB® X IDE

C:\microchip\harmony\v1_07_01\utilities\mhc\com-microchip-mplab-modules-mhc.nbm

5. Windows® or Mac® (Lab screenshots use Windows®)
6. TeraTerm, CoolTerm or other terminal emulator program

Hardware equipment

1. chipKIT™ WF32 board
   - Digilent Inc. part #: 410-273P-KIT
   - Microchip part #: TDGL021
2. Microchip MPLAB® ICD3 or PICKIT3 with required cables and 6 pin header
3. USB Flash Drive formatted with FAT, FAT16 or FAT32
4. mini USB-to-USB cable

LAB 1a

Read a POT using the ADC

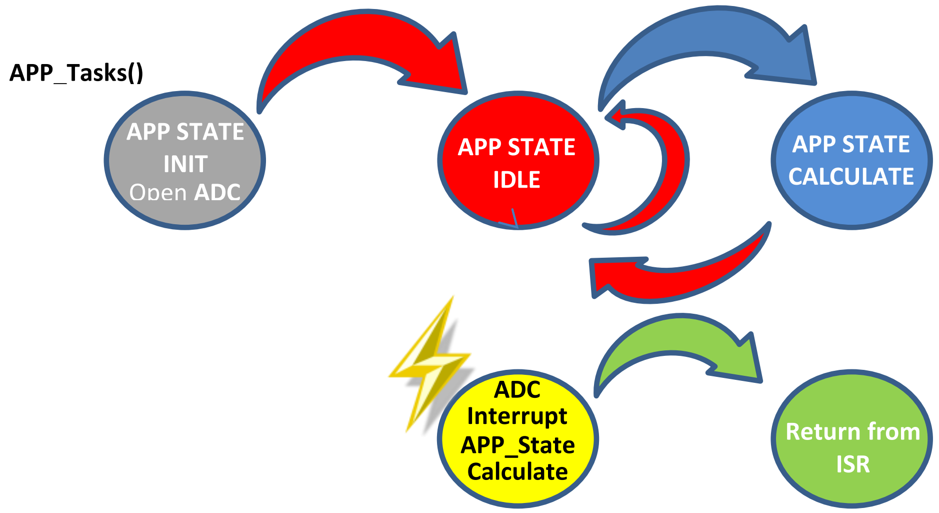
LAB 1a: Read a POT using the ADC

Purpose:

After completing LAB 1a, you will have an understanding of the fundamental elements, layout, and execution model of an MPLAB® Harmony project. You will also learn how to add features and functionality to your MPLAB® Harmony project by using the **MPLAB® Harmony Configurator (MHC).**

Overview:

In this lab you will create a simple MPLAB® Harmony project from scratch. You will use MHC to configure the ADC to read the potentiometer value on the Digilent WF32 board. The lab will demonstrate basic system initialization and polled state machine design.



Procedure:

1. All steps must be completed **before you will be ready to build, download, and run the application**.

Part 1: Create project and configure the PIC32MX

- Step 1 – Create an MPLAB® Harmony project in the MPLAB® X IDE
- Step 2 – With MHC, Select BSP for Board (Or configure manually)
- Step 3 – With MHC, Verify Config Bits are correct
- Step 4 – With MHC, Verify Oscillator Settings
- Step 5 – With MHC, configure I/O pins using the Graphical Pin Manager
- Step 6 – With MHC, Configure ADC
- Step 7 – Generate Code

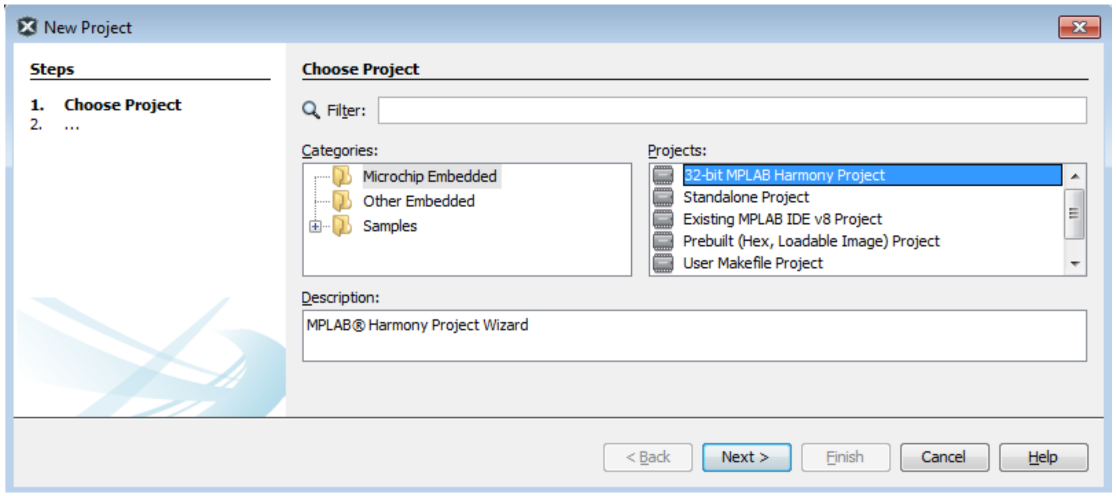
Part 2: Add Application code

- Step 8 – Add required variables and states to app.h
- Step 9 – Add required code to APP.C to Start ADC, and calculate average of 16 samples
- Step 10 – In ADC interrupt handler, read ADC results and update application state
- Step 11 – Debug your new application!

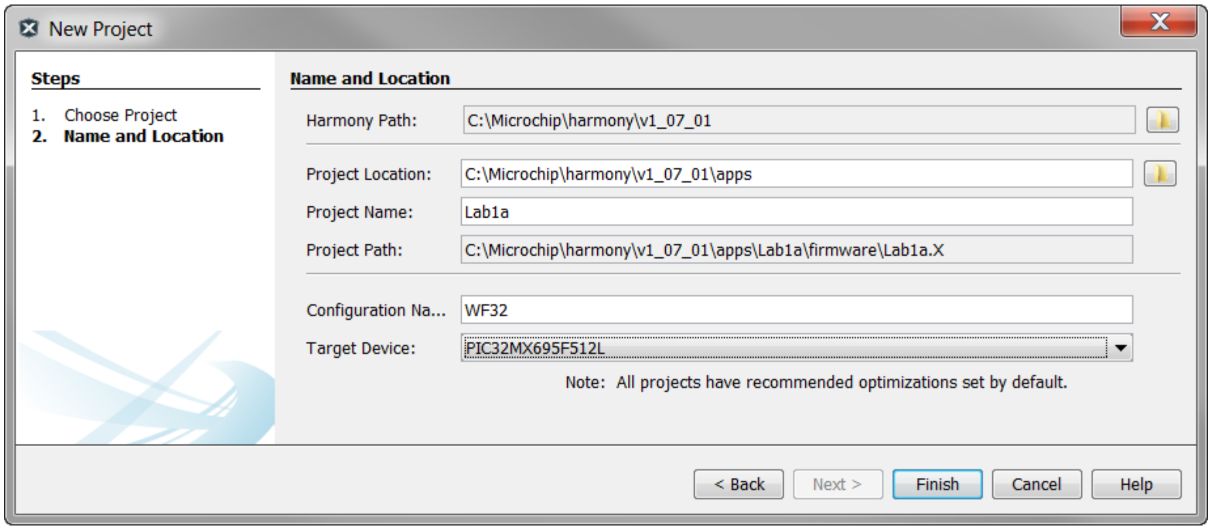
Please follow the steps described below. You can refer to the solutions if needed.

Step 1: Create an MPLAB® Harmony Project in the MPLAB® X IDE

1. Start the MPLAB® X IDE and create a New Project by selecting ***File*** > ***New Project***.
2. In the Categories pane of the New Project dialog, select **Microchip Embedded**.
3. In the Projects pane, select **32-bit MPLAB Harmony Project**, and then click **Next**.



4. Specify the following in the New Project dialog (see below):
   - **Harmony Path** (path to the folder containing the MPLAB® Harmony framework)
   - **Project Location** (navigate where you want to put your project)
   - **Enter Project Name : Lab1a**
   - **Configuration Name : WF32** (this is optional)
   - **Target Device : PIC32MX695F512L**



Harmony Path: C:\Microchip\harmony\v1_07_01

Project Location: C:\Microchip\harmony\v1_07_01\apps

Project Name: Lab1a

Project Path: C:\Microchip\harmony\v1_07_01\apps\Lab1a\firmware\Lab1a.X

Configuration Na... WF32

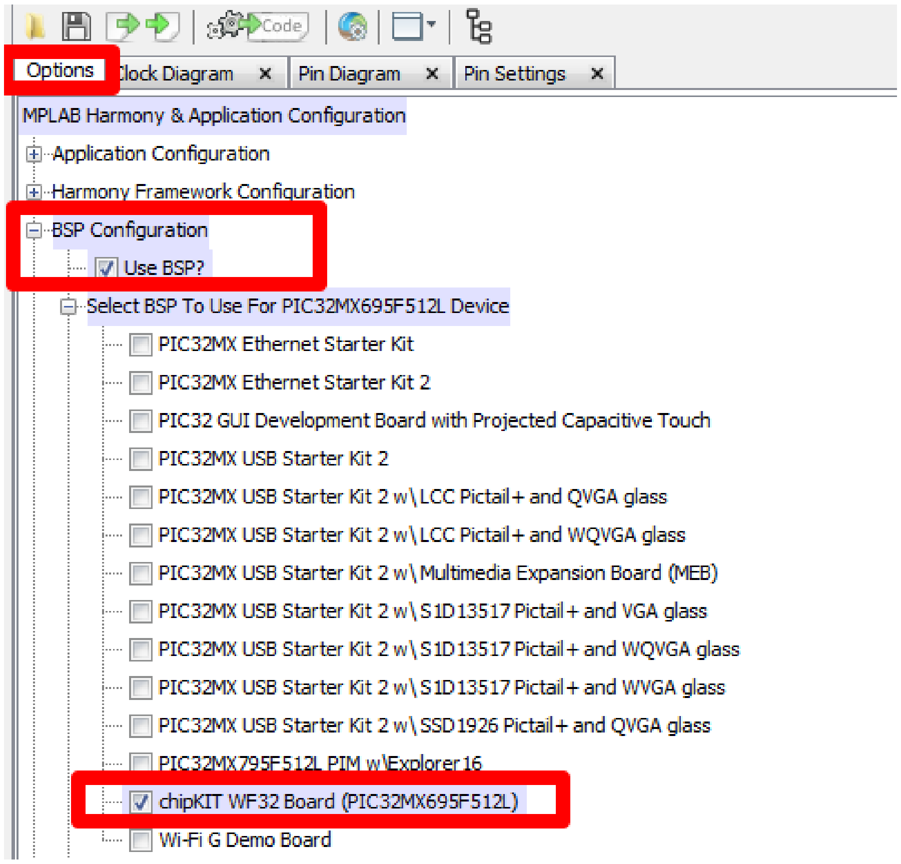
Target Device: PIC32MX695F512L

Note: All projects have recommended optimizations set by default.

5. After clicking the **Finish** button, the project will be created and opened. You will see the MPLAB® Harmony Configurator (MHC) window along with the integrated Harmony Help file.

Step 2: Select Board Support Package (BSP)

1. In the central window under the **MPLAB® Harmony Configurator** tab, click on the **Options** sub-tab, to view the MPLAB Harmony & Application Configuration tree selections.
2. Expand the **BSP Configuration** tree, then select **chipKit WF32 Board (PIC32MX695F512L)**



MPLAB Harmony & Application Configuration
- Application Configuration
- Harmony Framework Configuration
- BSP Configuration
  - [✓] Use BSP?
  - Select BSP To Use For PIC32MX695F512L Device
    - [ ] PIC32MX Ethernet Starter Kit
    - [ ] PIC32MX Ethernet Starter Kit 2
    - [ ] PIC32 GUI Development Board with Projected Capacitive Touch
    - [ ] PIC32MX USB Starter Kit 2
    - [ ] PIC32MX USB Starter Kit 2 w\LCC Pictail+ and QVGA glass
    - [ ] PIC32MX USB Starter Kit 2 w\LCC Pictail+ and WQVGA glass
    - [ ] PIC32MX USB Starter Kit 2 w\Multimedia Expansion Board (MEB)
    - [ ] PIC32MX USB Starter Kit 2 w\S1D13517 Pictail+ and VGA glass
    - [ ] PIC32MX USB Starter Kit 2 w\S1D13517 Pictail+ and WQVGA glass
    - [ ] PIC32MX USB Starter Kit 2 w\S1D13517 Pictail+ and WVGA glass
    - [ ] PIC32MX USB Starter Kit 2 w\SSD1926 Pictail+ and QVGA glass
    - [ ] PIC32MX795F512L PIM w\Explorer16
    - [✓] chipKIT WF32 Board (PIC32MX695F512L)
    - [ ] Wi-Fi G Demo Board

Note: If a Board Support Package exists for your development board you will want to use it. Choosing a BSP lets the MPLAB Harmony Configurator (MHC) know about the hardware you will use for the project. With this knowledge, MHC can automatically control the following settings for you:

- PIC32 core configuration (watchdog timer, debugger channel)
- PIC32 oscillator configuration (including external clock/crystal)
- PIC32 I/O Port pin connections to LEDs and switches

In addition to configuring hardware options for you, the BSP comes with a small group of library functions that allow you to more easily interface with LEDs and switches.

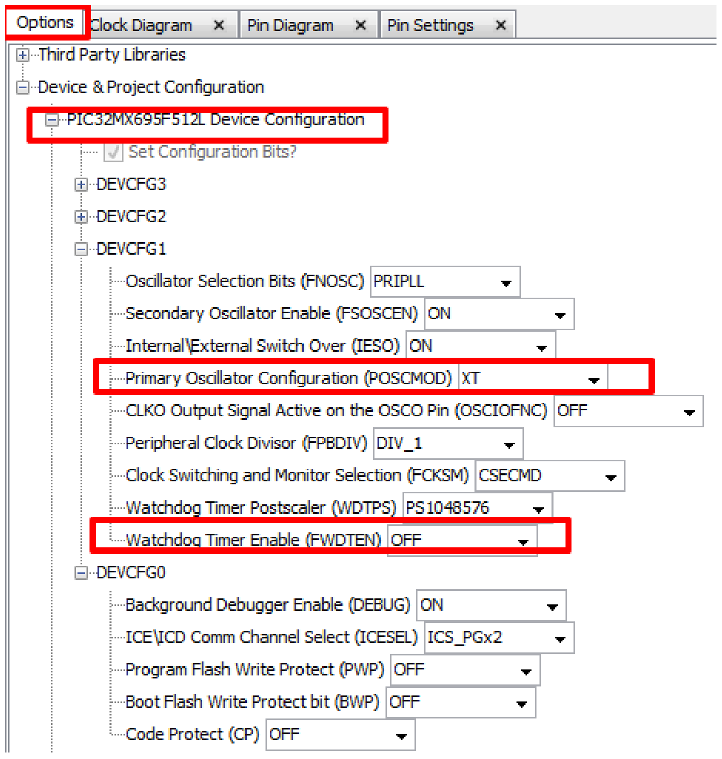
In this lab, you will observe the selections the BSP makes for you. This will show you how to make these selections manually in case a BSP does not exist for the board you want to use.

Step 3: Verify PIC32 Core Configuration Settings are Correct

1. In the central window under the **MPLAB® Harmony Configurator** tab, click on the **Options** sub-tab, to view the MPLAB Harmony & Application Configuration tree selections.

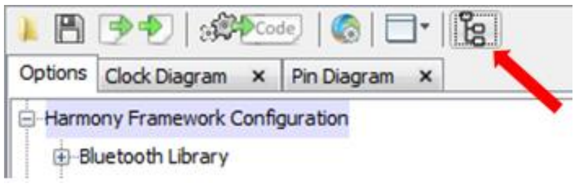
2. Expand the **Device & Project Configuration** tree then expand the PIC32MX695F512L Device Configuration.

   Note: These options should be correct since we selected a BSP
   - DEVCFG3 and DEVCFG2 – No change
   - DEVCFG1
     i. Verify Watchdog Timer Enable (FWDTEN)…… **OFF**
   - DEVCFG0 – No Change



```
Oscillator Selection Bits (FNOSC)                 PRIPLL
Secondary Oscillator Enable (FSOSCEN)             ON
Internal\External Switch Over (IESO)              ON
Primary Oscillator Configuration (POSCMOD)        XT
CLKO Output Signal Active on the OSCO Pin (OSCIOFNC)  OFF
Peripheral Clock Divisor (FPBDIV)                 DIV_1
Clock Switching and Monitor Selection (FCKSM)     CSECMD
Watchdog Timer Postscaler (WDTPS)                 PS1048576
Watchdog Timer Enable (FWDTEN)                    OFF
Background Debugger Enable (DEBUG)                ON
ICE\ICD Comm Channel Select (ICESEL)              ICS_PGx2
Program Flash Write Protect (PWP)                 OFF
Boot Flash Write Protect bit (BWP)                OFF
Code Protect (CP)                                 OFF
```

HINT! You can click this to exclusively show all changes you make in MHC!

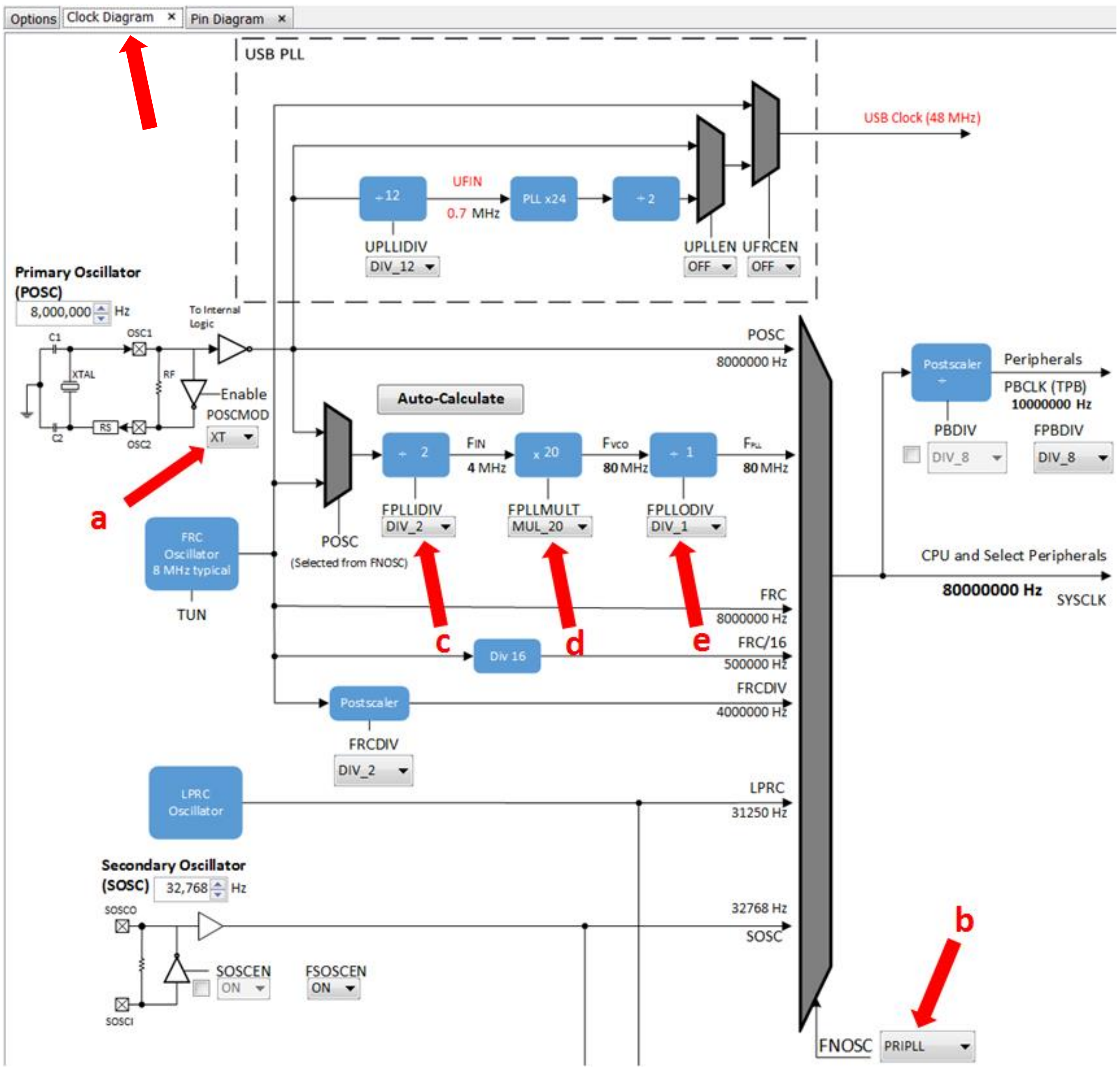


Before moving to step4, you may want to collapse the **Device & Project Configuration** tree

Step 4: Verify Settings for the PIC32 oscillator module

Select the **Clock Diagram** tab to display the Clock Configurator window.

1. Verify the following clock parameters: (These were set by BSP but this is here to show how to configure clocks for a board without a BSP)
    a. Select **XT** for POSCMOD
    b. Select **PRIPLL** for FNOSC
    c. Select **DIV_2** for FPLLIDIV
    d. Select **MUL_20** for FPLLMULT
    e. Select **DIV_1** for FPLLODIV



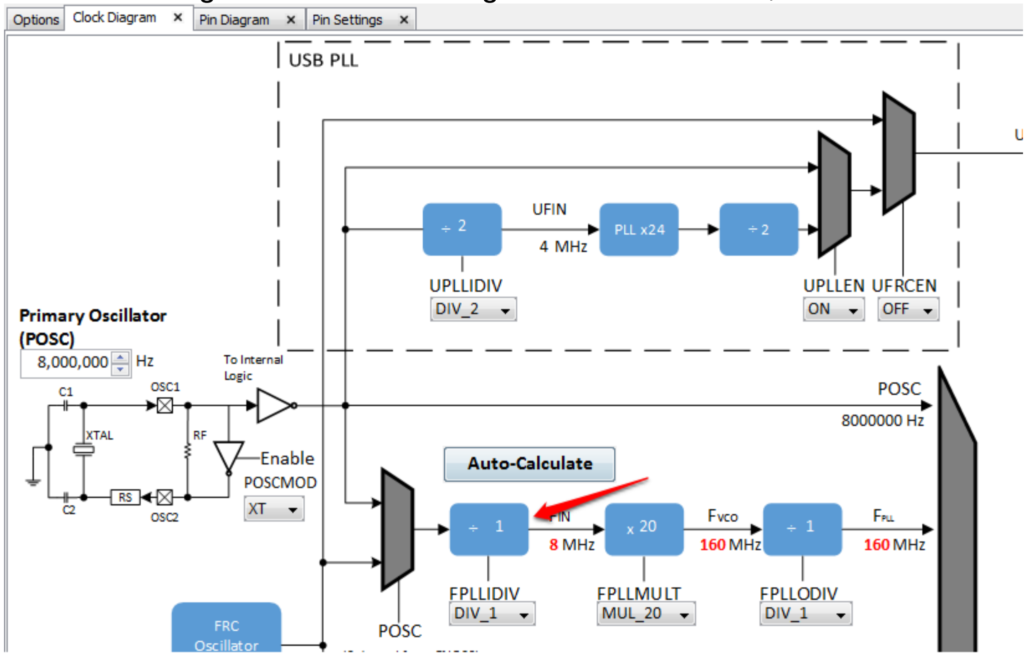
Experiment with other clock settings. Notice how some selections produce red values? These red values indicate a bad clock configuration. Hover you mouse over the red values to see what the problem is.

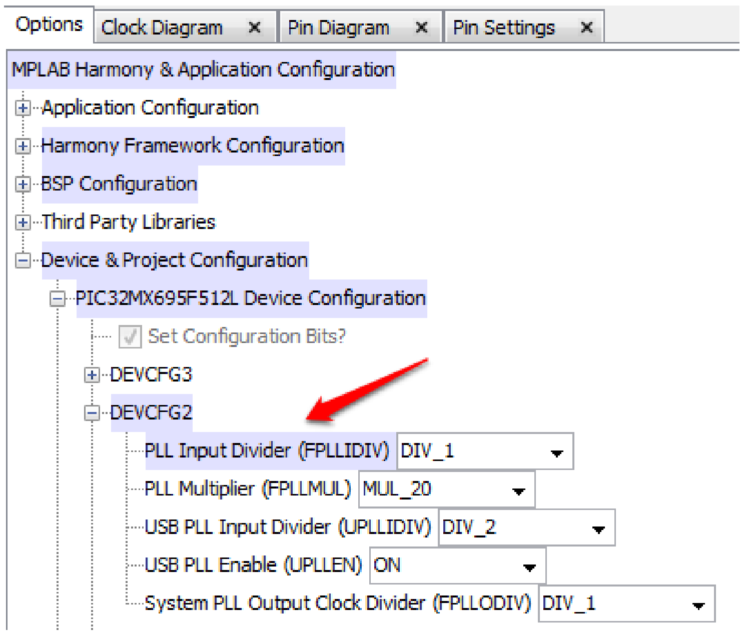
Note: The PIC32 is connected to an 8MHz crystal on the chipKIT™ WF32 board. We are not using the internal PIC32 oscillator. NO ADDITIONAL STEPS ARE REQUIRED HERE.

When you change the configured clocks away from the default values on the graphical interface, they are reflected as a shaded field in the Options Config Bits. Since the BSP selected the default values, there should be no shading.

To illustrate this, notice when the **FPLL DIV** is changed to a Divide By 1 instead of 2 in the CLOCK DIAGRAM graphical interface, the **OPTIONS** tab will reflect the changes in DEVCFG1 with Shading.

You can configure the clocks using the tree selections, but it is much easier to do graphically!

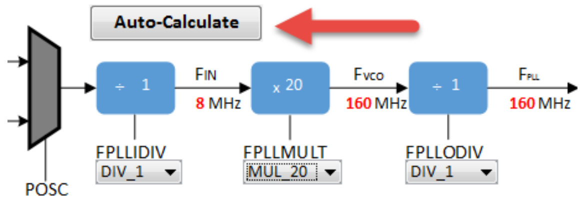


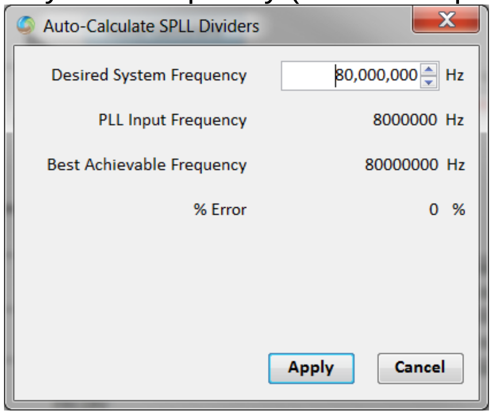

Options | Clock Diagram | Pin Diagram | Pin Settings

- MPLAB Harmony & Application Configuration
  - Application Configuration
  - Harmony Framework Configuration
  - BSP Configuration
  - Third Party Libraries
  - Device & Project Configuration
    - PIC32MX695F512L Device Configuration
      - ✓ Set Configuration Bits?
      - DEVCFG3
      - DEVCFG2
        - PLL Input Divider (FPLLIDIV) DIV_1
        - PLL Multiplier (FPLLMUL) MUL_20
        - USB PLL Input Divider (UPLLIDIV) DIV_2
        - USB PLL Enable (UPLLEN) ON
        - System PLL Output Clock Divider (FPLLODIV) DIV_1

The BSP has already configured the PLL using the selections for the WF32 board. For Custom boards without a BSP, you can use the PLL's "**Auto Calculate**" feature to determine and set the PLL multiply and divide values (FPLLIDIV, FPLLMULT, FPLLODIV). You can see how this works by going back to the Clock Configurator window (Clock Diagram tab).

1. Change the FPLLIDIV value to DIV_1. Note how this setting violates the PLL specs (see red values).
2. Select "Auto Calculate" and enter the desired system frequency (80MHz output)



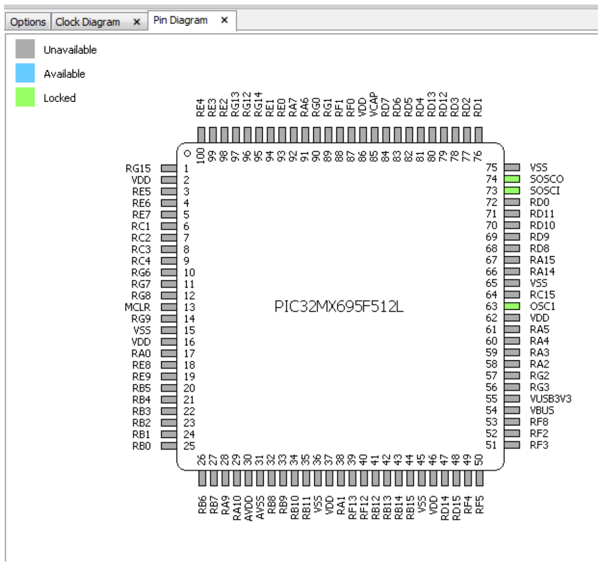


Desired System Frequency 80,000,000 Hz
PLL Input Frequency 8000000 Hz
Best Achievable Frequency 80000000 Hz
% Error 0 %

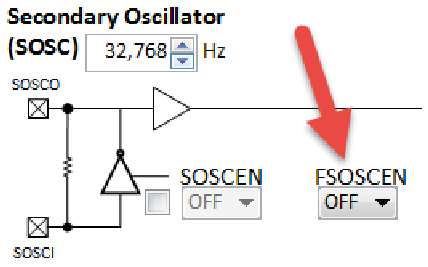
3. Click "Apply" to calculate and configure the PLL.

Step 5: Configure the I/O Pins using the Graphical Pin Manager

1. Open the Graphical Pin Manager by clicking on the “Pin Diagram” tab.
   a. Wait! I see pins 74 and 73 locked to the secondary oscillator (SOSCO and SOSCI). We don’t use this (and it’s not populated on the board), so let’s turn it off.

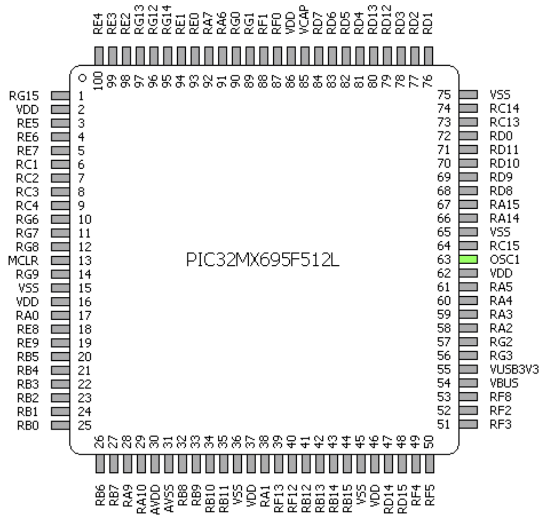

Options
Clock Diagram
Pin Diagram
Unavailable
Available
Locked
PIC32MX695F512L
RE4 RE3 RE2 RG13 RG12 RG14 RE1 RE0 RA7 RA6 RG0 RG1 RF1 RF0 VDD VCAP RD7 RD6 RD5 RD4 RD13 RD12 RD3 RD2 RD1
RG15 VDD RE5 RE6 RE7 RC1 RC2 RC3 RC4 RG6 RG7 RG8 MCLR RG9 VSS VDD RA0 RE8 RE9 RB5 RB4 RB3 RB2 RB1 RB0
VSS SOSCO SOSCI RD0 RD11 RD10 RD9 RD8 RA15 RA14 VSS RC15 OSC1 VDD RA5 RA4 RA3 RA2 RG2 RG3 VUSB3V3 VBUS RF8 RF2 RF3
RB6 RB7 RA9 RA10 AVDD AVSS RB8 RB9 RB10 RB11 VSS VDD RA1 RF13 RF12 RB12 RB13 RB14 RB15 VSS VDD RD14 RD15 RF4 RF5

   b. Go back to the Clock Configurator (Clock Diagram tab) and find the secondary oscillator enable setting (FSOSCEN, lower left corner) and turn it off.



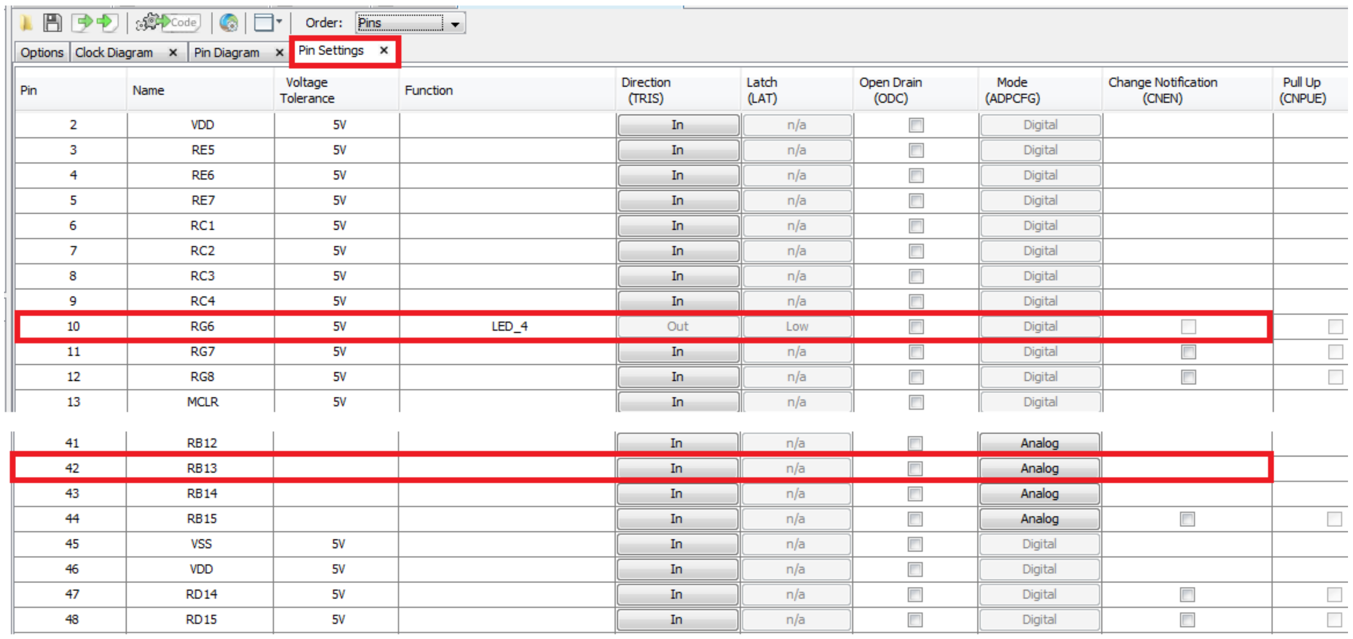
   c. Go back to the Graphical Pin Diagram window to verify these pins are now unlocked, and are available to be used as general purpose I/O.

      Side note:
      The second Primary oscillator pin (OSC2 pin 64) should also be displayed as locked but it is not. This bug will be fixed in a future Harmony revision.


PIC32MX695F512L
RE4 RE3 RE2 RG13 RG12 RG14 RE1 RE0 RA7 RA6 RG0 RG1 RF1 RF0 VDD VCAP RD7 RD6 RD5 RD4 RD13 RD12 RD3 RD2 RD1
RG15 VDD RE5 RE6 RE7 RC1 RC2 RC3 RC4 RG6 RG7 RG8 MCLR RG9 VSS VDD RA0 RE8 RE9 RB5 RB4 RB3 RB2 RB1 RB0
VSS RC14 RC13 RD0 RD11 RD10 RD9 RD8 RA15 RA14 VSS RC15 OSC1 VDD RA5 RA4 RA3 RA2 RG2 RG3 VUSB3V3 VBUS RF8 RF2 RF3
RB6 RB7 RA9 RA10 AVDD AVSS RB8 RB9 RB10 RB11 VSS VDD RA1 RF13 RF12 RB12 RB13 RB14 RB15 VSS VDD RD14 RD15 RF4 RF5

2. We will be using one ADC channel, and one GPIO. Let’s check the Pin configurations Select the “Pin Settings” tab

   - Notice that PIN 10 already has an LED selected on it. This is from the BSP.



Order: Pins

Options | Clock Diagram | Pin Diagram | Pin Settings

Pin	Name	Voltage Tolerance	Function	Direction (TRIS)	Latch (LAT)	Open Drain (ODC)	Mode (ADPCFG)	Change Notification (CNEN)	Pull Up (CNPUE)
2	VDD	5V		In	n/a	☐	Digital		
3	RE5	5V		In	n/a	☐	Digital		
4	RE6	5V		In	n/a	☐	Digital		
5	RE7	5V		In	n/a	☐	Digital		
6	RC1	5V		In	n/a	☐	Digital		
7	RC2	5V		In	n/a	☐	Digital		
8	RC3	5V		In	n/a	☐	Digital		
9	RC4	5V		In	n/a	☐	Digital		
10	RG6	5V	LED_4	Out	Low	☐	Digital	☐	☐
11	RG7	5V		In	n/a	☐	Digital	☐	☐
12	RG8	5V		In	n/a	☐	Digital	☐	☐
13	MCLR	5V		In	n/a	☐	Digital		
41	RB12			In	n/a	☐	Analog		
42	RB13			In	n/a	☐	Analog		
43	RB14			In	n/a	☐	Analog		
44	RB15			In	n/a	☐	Analog	☐	☐
45	VSS	5V		In	n/a	☐	Digital		
46	VDD	5V		In	n/a	☐	Digital		
47	RD14	5V		In	n/a	☐	Digital	☐	☐
48	RD15	5V		In	n/a	☐	Digital	☐	☐

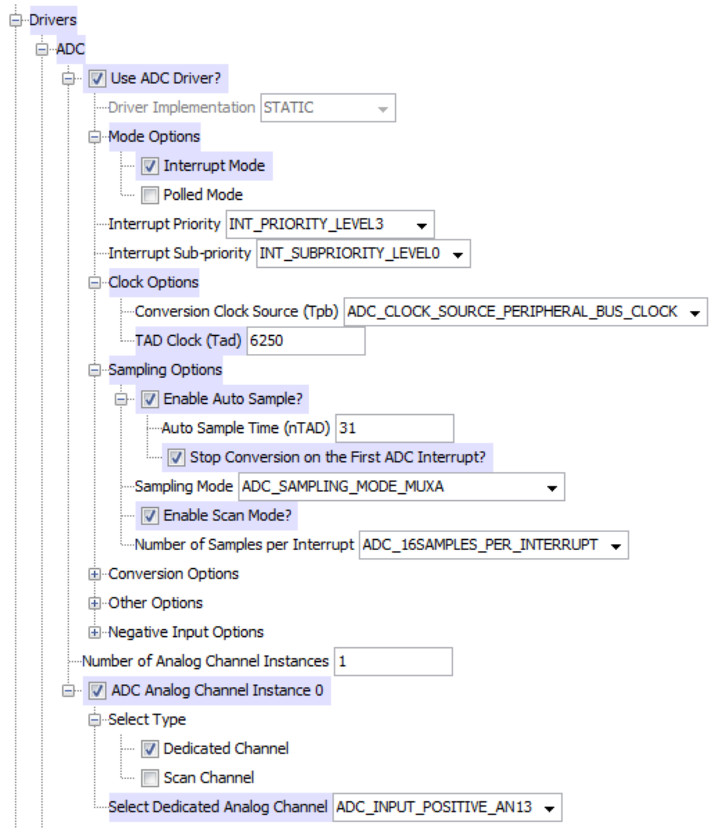
   - Verify PIN42 is set to INPUT and ANALOG. This is where the POTENTIOMETER is on the board. The PIC32’s Inputs are by default set to Analog. Additionally, the BSP should have already defined this as a “POTENTIOMETER”

Step 6: Configure the ADC Peripheral with MHC

1. Select the **MPLAB Harmony Configurator** tab and **Options** sub-tab.

   Note: If you ever "lose" the MHC Tool, you can find it here: click on "Tools, Embedded, MPLAB Harmony Configurator"

2. Expand the **Harmony Framework Configuration** tree, expand **Drivers**, and expand **ADC** tree.
   a. Check the **Use ADC Driver?** box
   b. Under **Mode Options** Check the **Interrupt Mode** box
   c. Expand **Clock Options** and enter **6250** in **TAD Clock** box, press Enter! (see note below)
   d. Expand **Sampling Options**, and check **Enable Auto Sample,** and **Stop Conversion on First ADC Interrupt?**
   e. Select and Expand **ADC Analog Channel Instance 0** tree option. Select **Dedicated Channel**, and **ADC_INPUT_POSITIVE_A13**
   f. Check **Enable Scan Mode** and assure **16 Samples per Interrupt** is selected

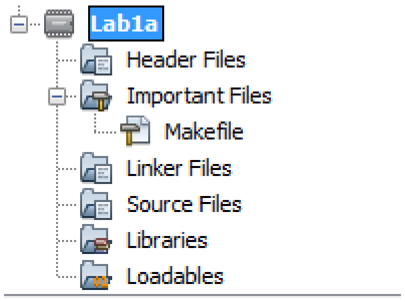


```
Drivers
  ADC
    [✓] Use ADC Driver?
      Driver Implementation  STATIC
      Mode Options
        [✓] Interrupt Mode
        [ ] Polled Mode
      Interrupt Priority  INT_PRIORITY_LEVEL3
      Interrupt Sub-priority  INT_SUBPRIORITY_LEVEL0
      Clock Options
        Conversion Clock Source (Tpb)  ADC_CLOCK_SOURCE_PERIPHERAL_BUS_CLOCK
        TAD Clock (Tad)  6250
      Sampling Options
        [✓] Enable Auto Sample?
          Auto Sample Time (nTAD)  31
          [✓] Stop Conversion on the First ADC Interrupt?
        Sampling Mode  ADC_SAMPLING_MODE_MUXA
        [✓] Enable Scan Mode?
        Number of Samples per Interrupt  ADC_16SAMPLES_PER_INTERRUPT
      Conversion Options
      Other Options
      Negative Input Options
    Number of Analog Channel Instances  1
    [✓] ADC Analog Channel Instance 0
      Select Type
        [✓] Dedicated Channel
        [ ] Scan Channel
      Select Dedicated Analog Channel  ADC_INPUT_POSITIVE_AN13
```

Note: The ADC requires one ADC clock cycle (TAD) to convert each bit of the result. There is a minimum amount of time required to convert each bit. The number associated with TAD clock allows you to scale the ADC input clock by some amount to be sure you meet this minimum time. Please see the device data sheet for more information.

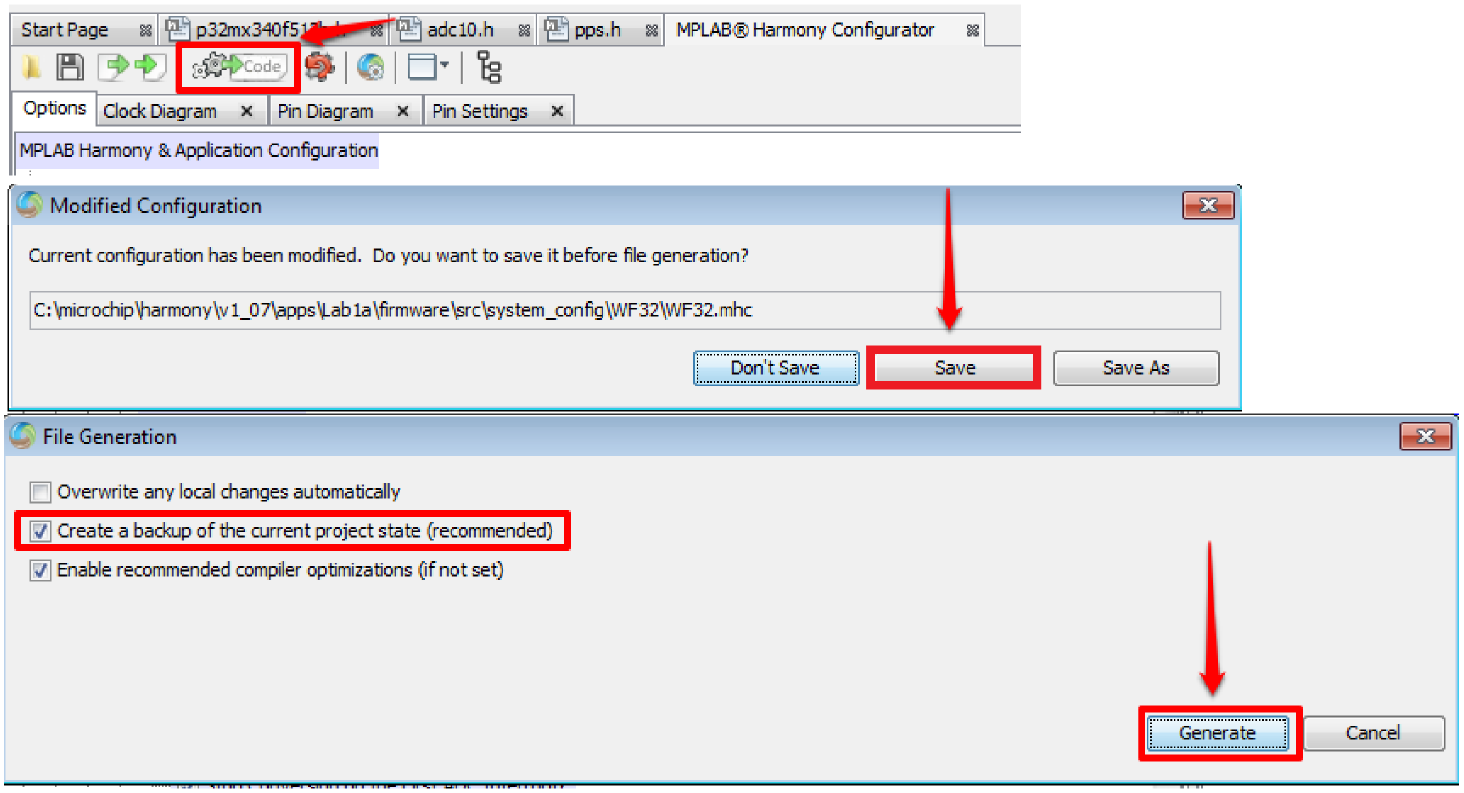
Step 7: Generate Harmony code and build project

1. Expand LAB 1a. Notice how there is no source files at all. Just an empty project with the correct harmony structure



2. Generate and save the configuration as shown below :

3. Notice the “Backup” option added in Harmony 1.07

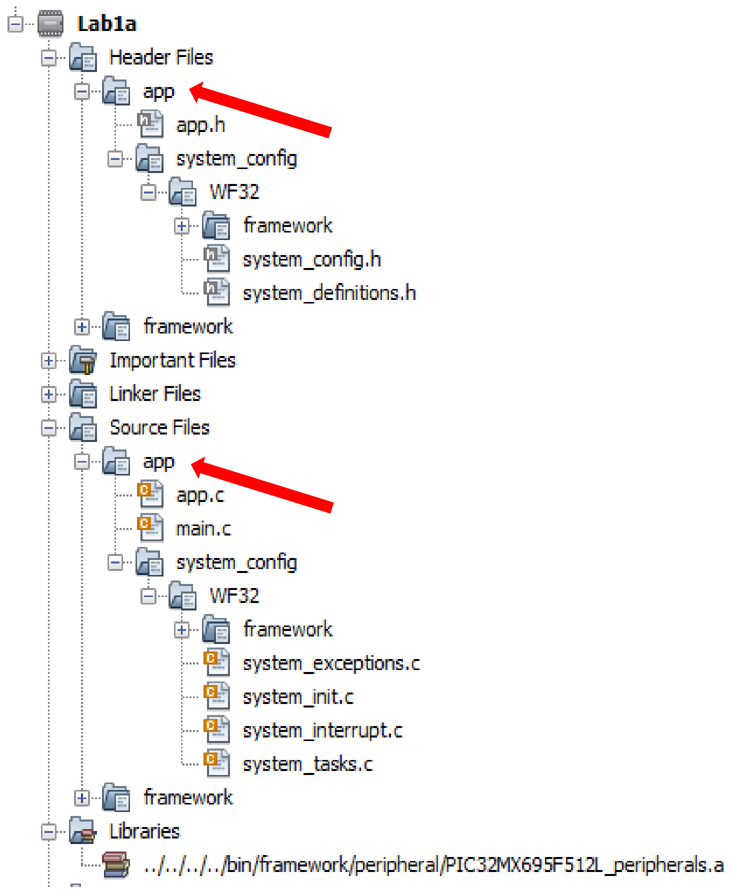


Current configuration has been modified. Do you want to save it before file generation?

C:\microchip\harmony\v1_07\apps\Lab1a\firmware\src\system_config\WF32\WF32.mhc

- [ ] Overwrite any local changes automatically
- [✓] Create a backup of the current project state (recommended)
- [✓] Enable recommended compiler optimizations (if not set)

4. Let’s examine what was done after “Generating Code”



```
Lab1a
  Header Files
    app
      app.h
      system_config
        WF32
          framework
          system_config.h
          system_definitions.h
    framework
  Important Files
  Linker Files
  Source Files
    app
      app.c
      main.c
      system_config
        WF32
          framework
          system_exceptions.c
          system_init.c
          system_interrupt.c
          system_tasks.c
    framework
  Libraries
    ../../../../bin/framework/peripheral/PIC32MX695F512L_peripherals.a
```

The “**app**” folder contains files related to your specific application. In the following steps, you will add your application code to app.c, system_interrupt.c, and app.h.

The “**framework**” folder under the “app” folder contains Harmony Framework files that have been generated by the MHC (based on your MHC selections). We will see the ADC folder get added to this when we add the ADC driver.

5. Compile the project to verify that the generated project compiles properly without generating errors. Click on

Note :
At this point, you should be able to debug and step through the application. Effectively, you have a running MPLAB® Harmony system. **However, it is not yet ready to do anything**. Next, you will develop your application state machine logic and make sure the system does what you want it to do. You’re ready to start implementing the application now.

In Steps 8, 9 and 10 you will:

1. Implement constants of “*enumerated” types.*
    a. Set of named integer constants.
2. Declare, define and access elements of “*struct”* type.
    a. Do you remember structures?
3. Add declarations in the *app.h* file.
4. Add code in the *app.c* and the *system_interrupt.c* files.

C LANGUAGE REFRESHER: enumerated types and structures

Remember this refresher slide on “emum” and “struct” types for future reference in following labs. You may need it. It may come in handy. ☺

```
typedef enum _app_states
{
    APP_STATE_INIT = 0,
    APP_STATE_EVENT,
    APP_STATE_IDLE

} APP_STATES;

typedef struct _app_data
{
    APP_STATES state;
    int        count;
    char       command;

} APP_DATA;

APP_DATA app1Data, app2Data;

app1Data.state = APP_STATE_INIT;
app2Data.state = APP_STATE_INIT;
```

“enum” Types

- Used to give names to constants
- Better than “#define”
    - Groups related names together
    - Associates names with a type
    - Values auto-generated
    - Improves type checking
    - Improves debugging

“struct” Types

- Groups related variables
- Better than separate variables
    - Access member/field with ‘.’
    - Avoids name collisions

`app1Data.state` different from `app2Data.state`

- Reference struct by one pointer:

```
APP_DATA *pApp = &app1Data;
pApp->state = APP_STATE_IDLE;
```

MPLAB® X IDE EDITOR TIPS

1) Use the **auto-complete** function after typing a few characters : type **CTRL+SPACE**
2) To navigate labels, functions, structures use hyperlink navigation : **CTRL+”click”**

Step 8: Add Application States and Variables for App

(Note: the solutions for this step can be found on the next page)

In this step we will add three named constants within an *enum*. These constants are implemented in switch case statements as well as in an interrupt handler to determine the current application state as well as set the next application state for processing.

In other words, you are adding states to service in your application.

1. If not already opened or visible, open the Projects window (Window > Projects)
2. If not expanded, expand the Lab1a project tree
3. If not expanded, expand Lab1a header files
4. Open the file app.h: (located under Header Files, in app folder)
   a. In **typedef enum** "APP_STATES", add three constants under /* TODO: Define states used by the application state machine. */,:

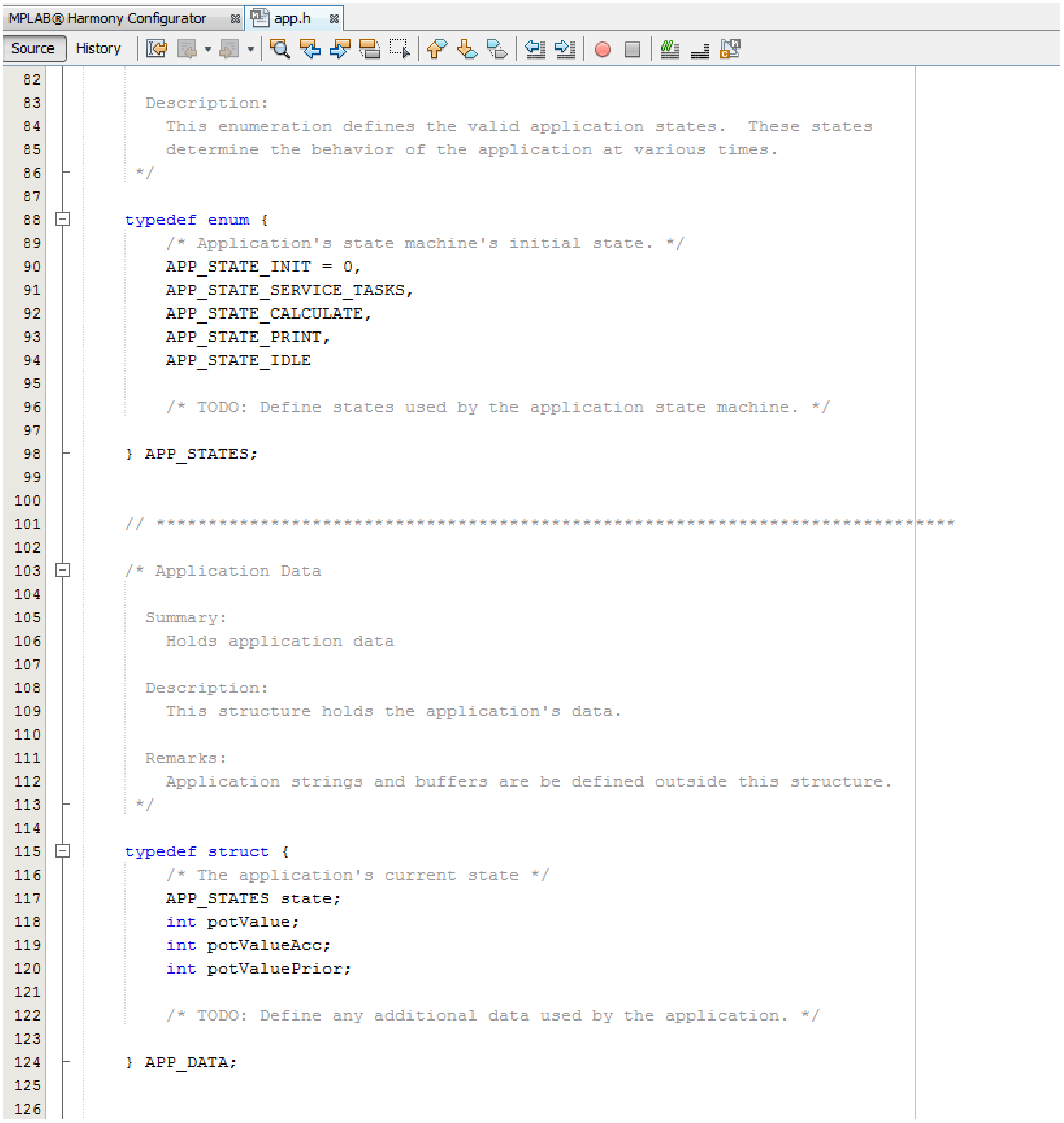
   ```
   APP_STATE_CALCULATE,
   APP_STATE_PRINT,
   APP_STATE_IDLE
   ```

   b. In **typedef struct** "APP_DATA", add the required variables for our app. Notice how "APP_STATES state;" is already added for you?
   /* TODO: Define any additional data used by the application. */

   ```
   int potValue;
   int potValueAcc;
   int potValuePrior;
   ```

Step 8: Solution

1. Open file app.h
2. Add three named constants (application states) as shown.
3. Add the three int type variables potValue, potValueAcc and potValuePrior
4. Save the file, before closing.



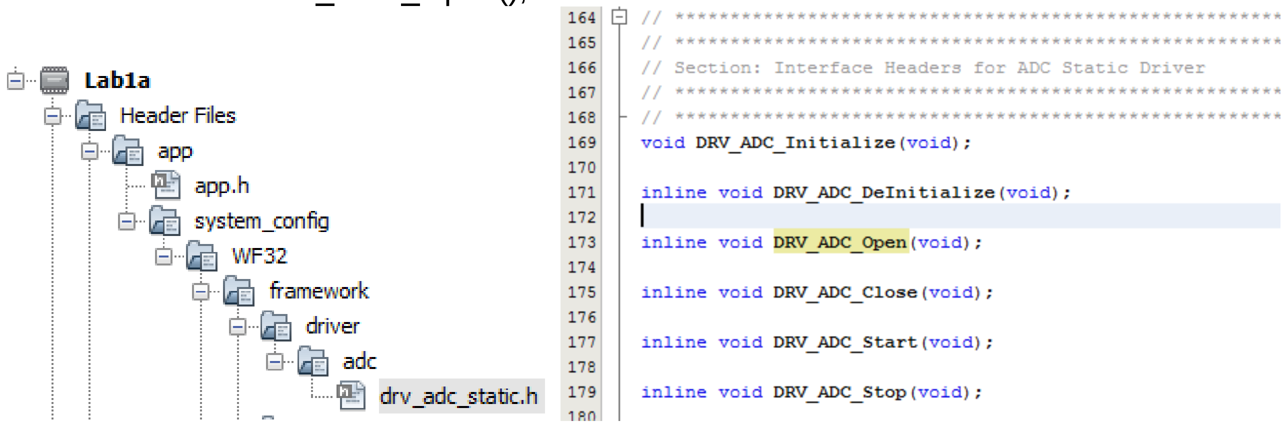
```
      Description:
        This enumeration defines the valid application states.  These states
        determine the behavior of the application at various times.
     */

    typedef enum {
        /* Application's state machine's initial state. */
        APP_STATE_INIT = 0,
        APP_STATE_SERVICE_TASKS,
        APP_STATE_CALCULATE,
        APP_STATE_PRINT,
        APP_STATE_IDLE

        /* TODO: Define states used by the application state machine. */

    } APP_STATES;


    // *****************************************************************************

    /* Application Data

      Summary:
        Holds application data

      Description:
        This structure holds the application's data.

      Remarks:
        Application strings and buffers are be defined outside this structure.
     */

    typedef struct {
        /* The application's current state */
        APP_STATES state;
        int potValue;
        int potValueAcc;
        int potValuePrior;

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

    } APP_DATA;

```

Step 9: Start the ADC Driver and calculate an average of 16 samples

(Note: the solutions for this step can be found on the next page)

Open file app.c (located under Source Files in the app folder). In the function ***APP_Tasks(),*** you will add code in three switch case statements:

1. In the first (existing) case statement: **APP_STATE_INIT**
   a. Add code to open the ADC Driver (all ADC driver function prototypes can be found in the drv_adc_static.h file):
      DRV_ADC_Open();



```
// *****************************************************************
// *****************************************************************
// Section: Interface Headers for ADC Static Driver
// *****************************************************************
// *****************************************************************
void DRV_ADC_Initialize(void);

inline void DRV_ADC_DeInitialize(void);

inline void DRV_ADC_Open(void);

inline void DRV_ADC_Close(void);

inline void DRV_ADC_Start(void);

inline void DRV_ADC_Stop(void);
```

   b. Change the existing code to move to APP_STATE_IDLE instead of APP_STATE_SERVICE_TASKS;
   c. Rename the APP_STATE_SERVICE_TASKS case statement to APP_STATE_IDLE

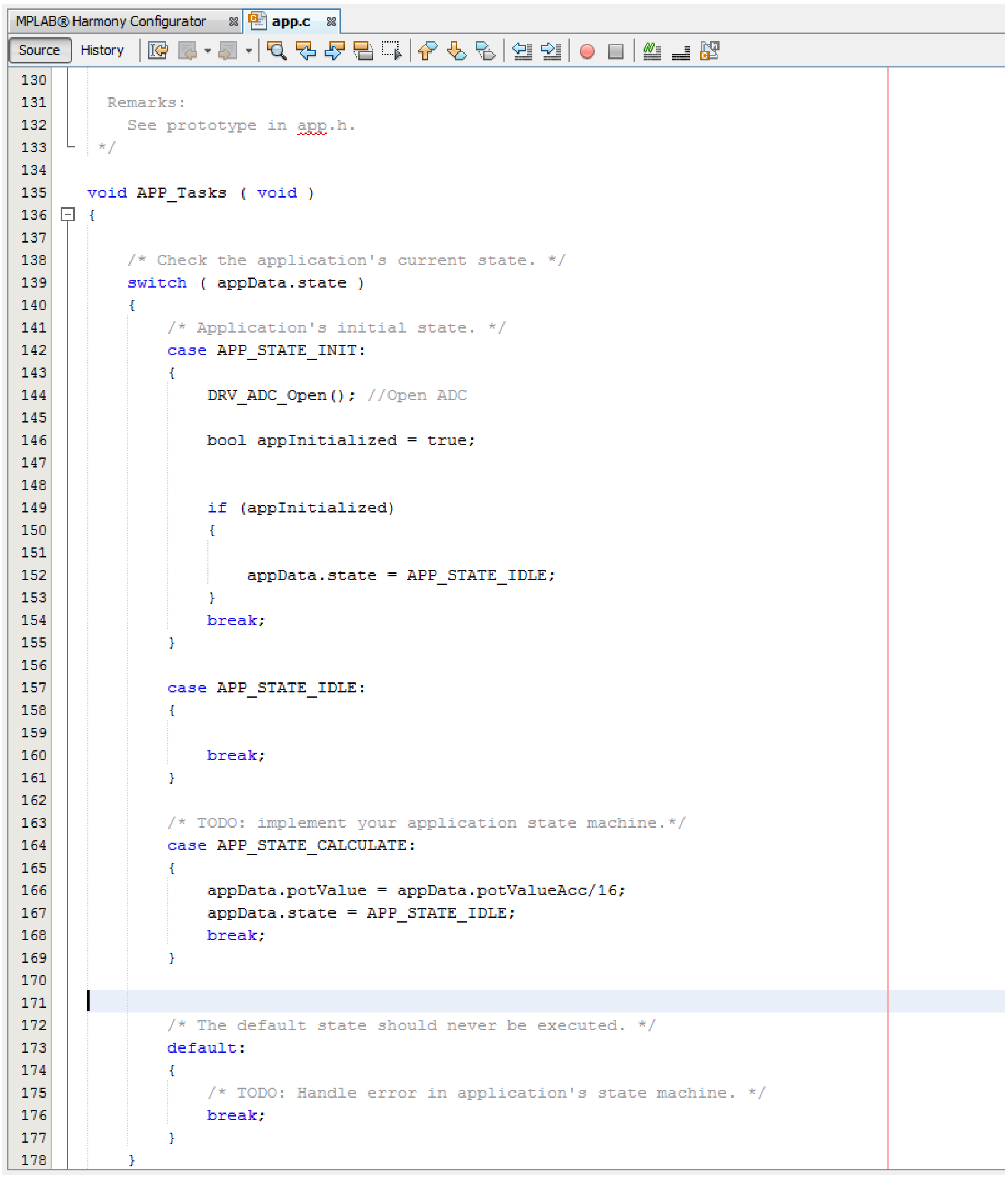
2. Create a 2rd case statement under /* TODO: implement your application state machine.*/,
   a. Create a new state:
      case APP_STATE_CALCULATE:

   b. Average the 16 samples from the ADC:
      appData.potValue = appData.potValueAcc / 16;

   c. Set appData.state back to idle.
      appData.state = APP_STATE_IDLE;

   d. Do not forget to also add a break; at the end of this 2nd case

Step 9: Solution



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

void APP_Tasks ( void )
{

    /* Check the application's current state. */
    switch ( appData.state )
    {
        /* Application's initial state. */
        case APP_STATE_INIT:
        {
            DRV_ADC_Open(); //Open ADC

            bool appInitialized = true;


            if (appInitialized)
            {

                appData.state = APP_STATE_IDLE;
            }
            break;
        }

        case APP_STATE_IDLE:
        {

            break;
        }

        /* TODO: implement your application state machine.*/
        case APP_STATE_CALCULATE:
        {
            appData.potValue = appData.potValueAcc/16;
            appData.state = APP_STATE_IDLE;
            break;
        }


        /* The default state should never be executed. */
        default:
        {
            /* TODO: Handle error in application's state machine. */
            break;
        }
    }
```

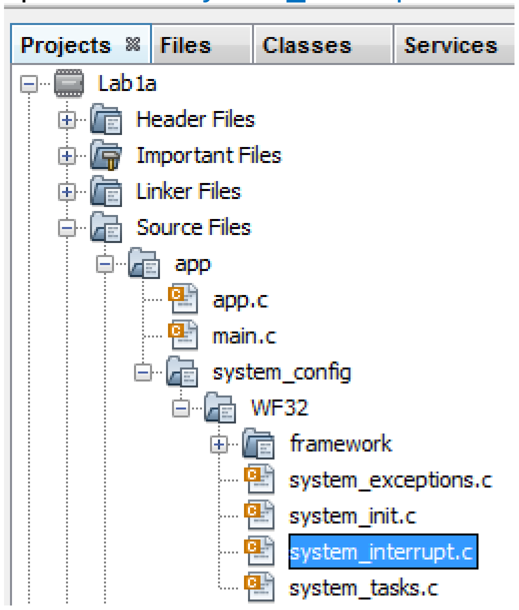
Step 10: Service ISR: read ADC results and update application state

(Note: the solutions for this step can be found on the next page)

This last step has some instructions that may not seem intuitive. We use two Harmony Peripheral Library (PLIB) functions that may make you say "How did you know to do that?".

Harmony provides lots of code examples and documentation that demonstrate how to use these library functions. You can find them in the Harmony Help file: c:/microchip/harmony/v1_07/doc directory. Many fully functional example projects are also included with the Harmony installation. Find them in this directory: c:/microchip/harmony/v1_07/apps. The context sensitive Harmony Help window and, and the "Find in Projects" (Ctrl + Shift + F) search tool can also be helpful.

Open the file system_interrupt.c:



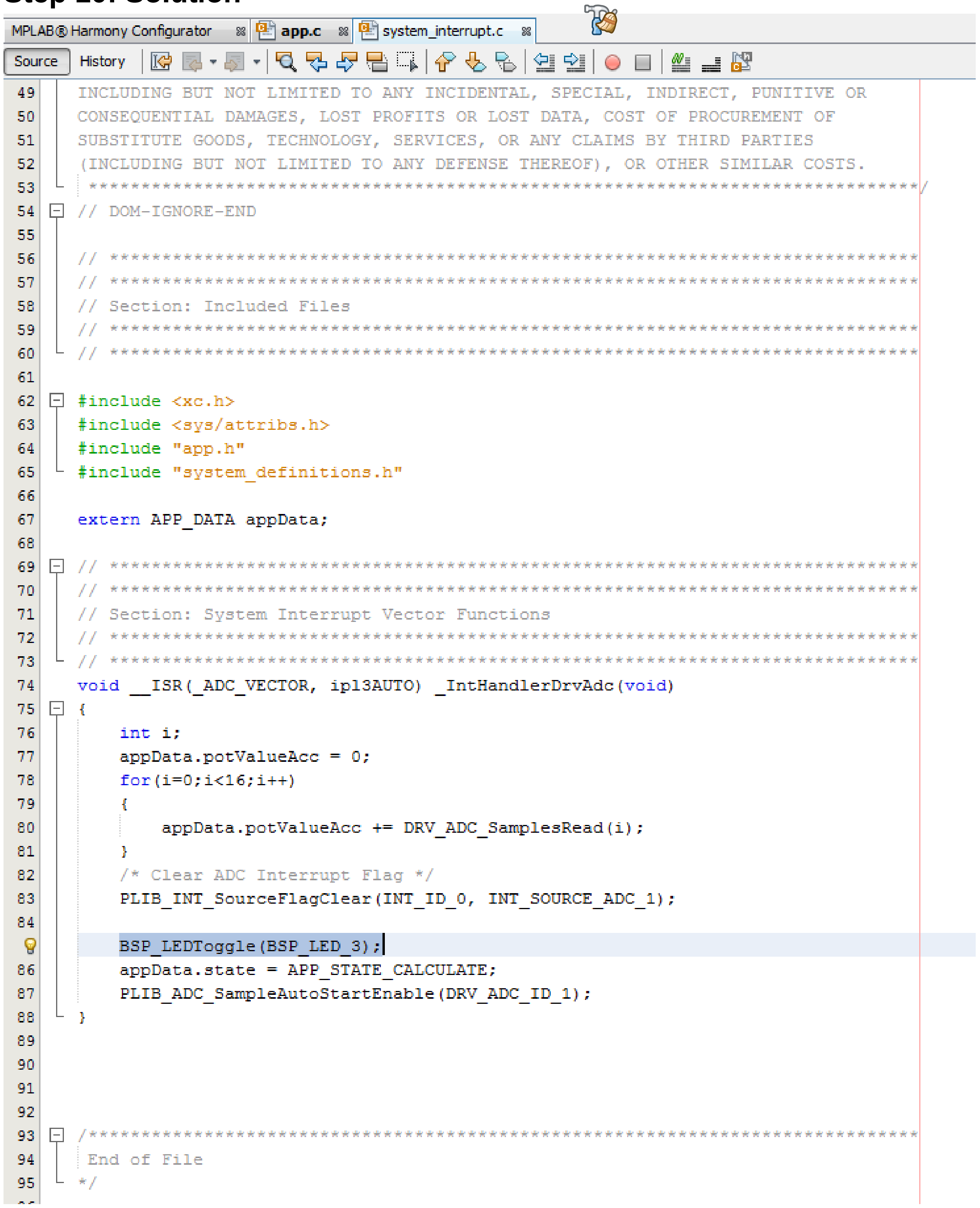
1. Between the header file "#include" statements and the start of the interrupt service routine (ISR) handler:
    a. Declare and provide external reference for the "appData" variable (declared in app.c file):
    ```
    extern APP_DATA appData;
    ```

2. In the ADC interrupt function (_IntHandlerDrvAdc()), **before** the existing code that clears the ADC interrupt flag:
    a. Create a for loop variable (i). This for loop will read a buffer 16 times:
    ```
    int i;
    ```
    b. Initialize appData.potValue to zero:
    ```
    appData.potValueAcc = 0;
    ```
    c. Create a "for" loop to read 16 values from the ADC (use the PLIB_ADC_ResultGetByIndex() Harmony library function). After each read, add the result to appData.potValue.
    ```
    for (i = 0; i < 16; i++)
    {
    appData.potValueAcc += PLIB_ADC_ResultGetByIndex(ADC_ID_1, i);
    }
    ```
    Note: Be sure to put this code BEFORE the code that was auto generated to clear the ADC interrupt request. This is because the ADC interrupt on the PIC32 is PERSISTANT (it won't go away until you read all the results).

3. In the ADC interrupt function (_IntHandlerDrvAdc()), **after** the existing code that clears the ADC interrupt flag:
    a. Use the following BSP function to invert the LED state after 16 samples.
    BSP_LEDToggle(BSP_LED_3);

    THIS IS NOT A STEP HERE: (JUST FYI) Before the BSP was available for the WF32 board, a PLIB call was necessary!
    The LED is connected to the PIC32's RG6 I/O pin.
    ```
    PLIB_PORTS_Toggle(PORTS_ID_0, PORT_CHANNEL_G, 1<<PORTS_BIT_POS_6);
    ```
    b. Move the state machine to the calculate state:
    ```
    appData.state = APP_STATE_CALCULATE;
    ```
    c. Finally, since we configured the ADC to stop sampling on the first interrupt request, start the next sample using this PLIB:
    ```
    PLIB_ADC_SampleAutoStartEnable(ADC_ID_1);
    ```

Step 10: Solution

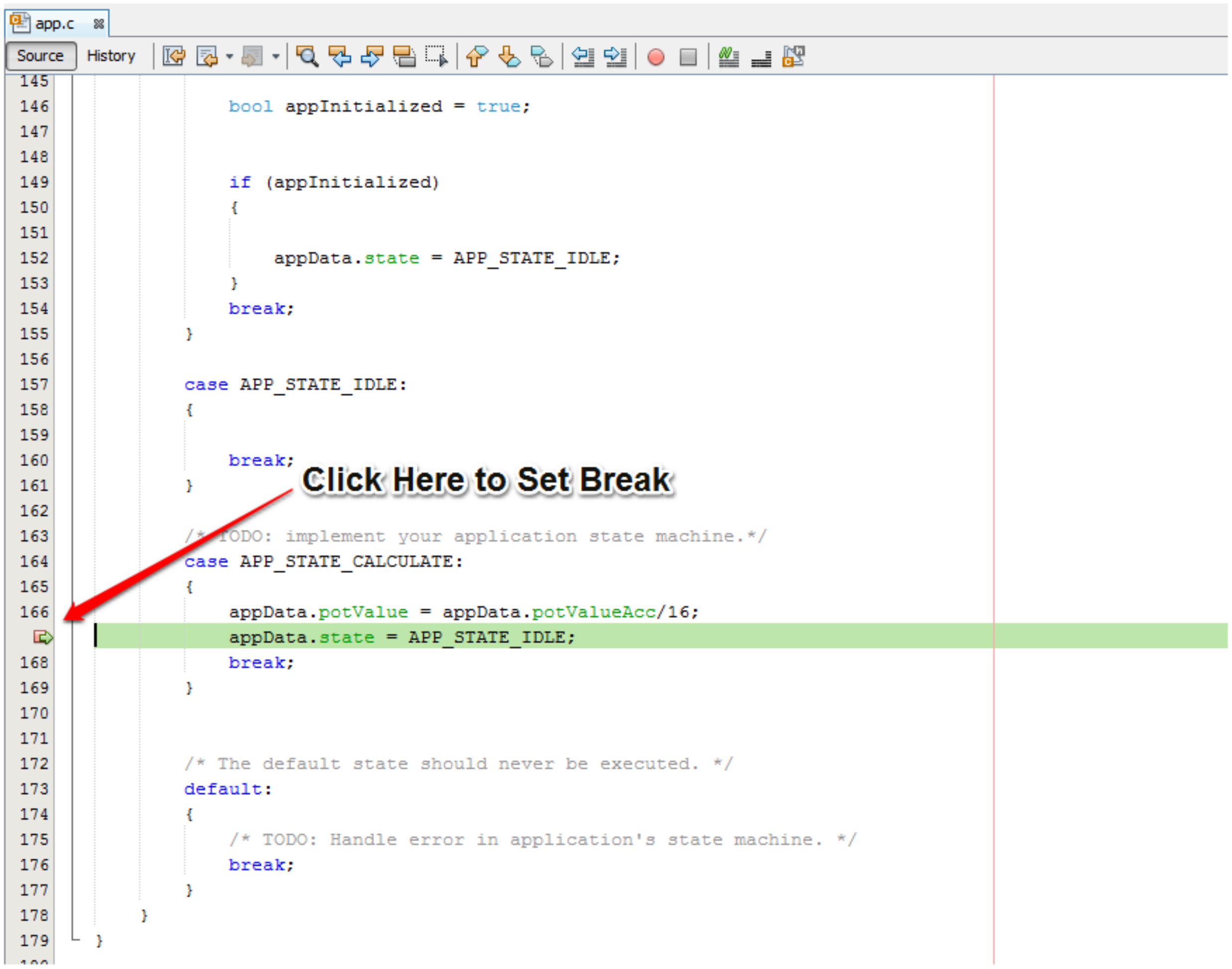


```
INCLUDING BUT NOT LIMITED TO ANY INCIDENTAL, SPECIAL, INDIRECT, PUNITIVE OR
CONSEQUENTIAL DAMAGES, LOST PROFITS OR LOST DATA, COST OF PROCUREMENT OF
SUBSTITUTE GOODS, TECHNOLOGY, SERVICES, OR ANY CLAIMS BY THIRD PARTIES
(INCLUDING BUT NOT LIMITED TO ANY DEFENSE THEREOF), OR OTHER SIMILAR COSTS.
 *******************************************************************************/
// DOM-IGNORE-END

// *****************************************************************************
// *****************************************************************************
// Section: Included Files
// *****************************************************************************
// *****************************************************************************

#include <xc.h>
#include <sys/attribs.h>
#include "app.h"
#include "system_definitions.h"

extern APP_DATA appData;

// *****************************************************************************
// *****************************************************************************
// Section: System Interrupt Vector Functions
// *****************************************************************************
// *****************************************************************************
void __ISR(_ADC_VECTOR, ipl3AUTO) _IntHandlerDrvAdc(void)
{
    int i;
    appData.potValueAcc = 0;
    for(i=0;i<16;i++)
    {
        appData.potValueAcc += DRV_ADC_SamplesRead(i);
    }
    /* Clear ADC Interrupt Flag */
    PLIB_INT_SourceFlagClear(INT_ID_0, INT_SOURCE_ADC_1);

    BSP_LEDToggle(BSP_LED_3);
    appData.state = APP_STATE_CALCULATE;
    PLIB_ADC_SampleAutoStartEnable(DRV_ADC_ID_1);
}




/*******************************************************************************
 End of File
*/
```

Step 11: Debug, and ENJOY your new application!

Congratulations! You're done. I assure you, the hard part is over! You are now able to **build, program, and DEBUG** your first application!

1. Before you start the debugger, set a breakpoint in the APP_STATE_CALCULATE case in APP.C. This breakpoint will allow us to observe the ADC result after every interrupt.



```
            bool appInitialized = true;


            if (appInitialized)
            {

                appData.state = APP_STATE_IDLE;
            }
            break;
        }

        case APP_STATE_IDLE:
        {

            break;
        }

        /* TODO: implement your application state machine.*/
        case APP_STATE_CALCULATE:
        {
            appData.potValue = appData.potValueAcc/16;
            appData.state = APP_STATE_IDLE;
            break;
        }


        /* The default state should never be executed. */
        default:
        {
            /* TODO: Handle error in application's state machine. */
            break;
        }
    }
}
```

2. **DEBUG** your first application! Click the "Debug Main Project" icon:
3. Press Continue a few times, to let the ADC value stabilize
4. Hover your mouse over the potValue variable in the editor. A small window will open to display the current value of that variable. Move the POT on the chipKIT™ WF32 board, and step a few more times. See it change?

You are now an MPLAB® Harmony developer! 🙂

Feel free to debug it and step through the code to see how it works. As you continue stepping through the project, you will eventually arrive back at the super loop in the main function. Fundamentally, this is how polled state machines execute within an MPLAB® Harmony system.

Note: Don't be surprised if you cannot step into the PLIB function calls. They are implemented as inline functions and source code is provided in the MPLAB® Harmony installation. It is possible to rebuild the PLIB, converting them to actual function calls, without optimizations and with debug symbols enabled, so that you can step through them. But, they are generally very simple functions that provide access to peripheral Special Function Registers (SFRs) through an abstracted C-language function call interface that remains the same on all parts on which they are supported.

Time permitting; explore the Lab1a project files and folders. Observe that the project consists of several "logical" folders within the MPLAB® X IDE and several physical folders on disk. This organization is used consistently by MPLAB® Harmony applications to keep the files well organized. The MPLAB® X IDE separates header (**`.h`**) files from source (**`.c`**) files, so the logical folder structure mirrors (with some small differences) the physical folder structure on disk under both the **`Header Files`** and **`Source Files`** top-level logical folders. In most cases, the folders on disk correspond directly to the logical folders in the MPLAB® X IDE, but the header files and source files are not separated on disk as they are in the MPLAB® X IDE logical folder tree. Also, the **`app`** logical folder corresponds to the **`src`** folder on disk.

The application's **`src`** folder contains the **`main.c`** file as well as the **`app.c`** and **`app.h`** files that implement the application. (Note: By convention, the main.c file is identical for all MPLAB® Harmony applications.) The **`system_config`** folder normally holds one or more configuration-specific folders (**`system_config\WF32`** in this case) that each contains a complete set of configuration files for an MPLAB® Harmony application.

===

LAB 1b

Use Debug System Service to write ADC Result to UART

LAB 1b: Use Debug System Service to write ADC result to UART

Purpose:

After completing LAB 1b, you will understand how to add the “DEBUG” system service to a project. This section should really start to show you the power of Harmony!
NOTE: There are other ways to write values to the UART.

Overview:

In this lab, you will add the DEBUG system service to your project. The lab result will allow you to start communicating with the real world. Instead of setting a breakpoint to show you the ADC result, we will now print it to the UART. Future labs will show the ease of sending this out the USB port of the PIC32 with minimal changes. The lab will demonstrate how easy it is to both enable and configure new PIC32 features using MHC.

Procedure:

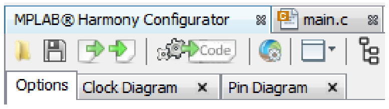
1. If not active already, we need to enable MHC : **Tools > Embedded > MPLAB Harmony Configurator**
2. This lab has a lot less steps than the last one:

   Part 1: With MHC, add the “Debug System Service”
   - Step 1 – With MHC, add Debug System Service
   - Step 2 – Configure “Console” system service to use the UART
   - Step 3 – With MHC, Generate Code

   Part 2: Add Application code
   - Step 4 – Add code to send ADC result to the console

   Part 3: Run the program
   - Step 5 – Observe the ADC results using the UART console (9600 baud)

Step 1: Enable the Debug System Service

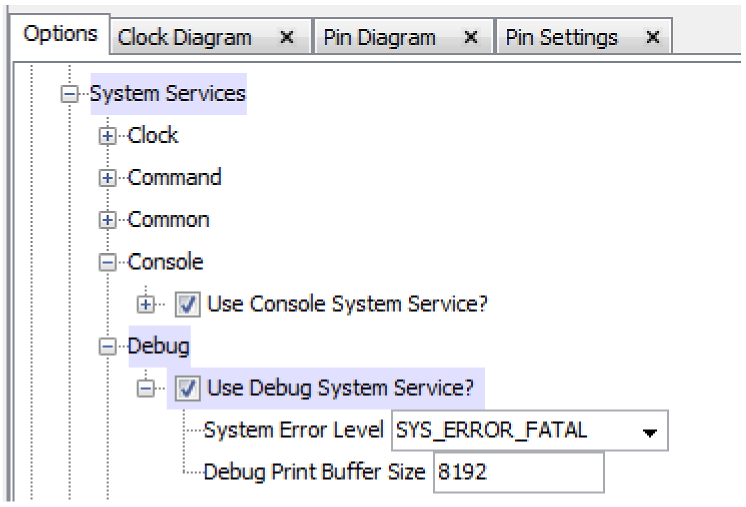
1. Select the **MPLAB® Harmony Configurator** tab 

2. Expand the **Harmony Framework Configuration** tree, expand **System Services**, expand **DEBUG** tree.
   a. Check the **Use Debug System Service?** box

==

Step 1: Solution

1. Select and Configure Debug System Service as shown.



[✓] Use Console System Service?

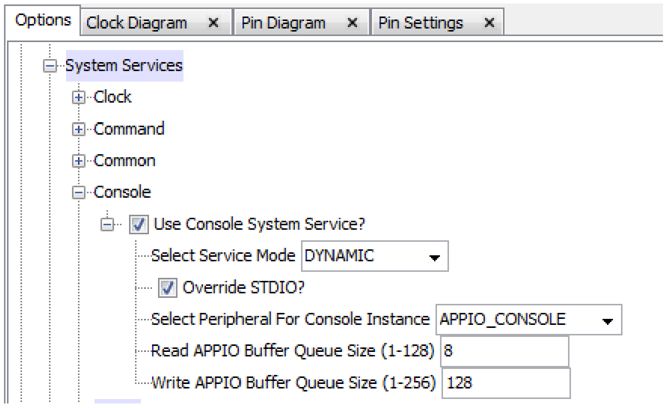
[✓] Use Debug System Service?

System Error Level: SYS_ERROR_FATAL

Debug Print Buffer Size: 8192

Step 2: Configure the Console System Service to use the UART

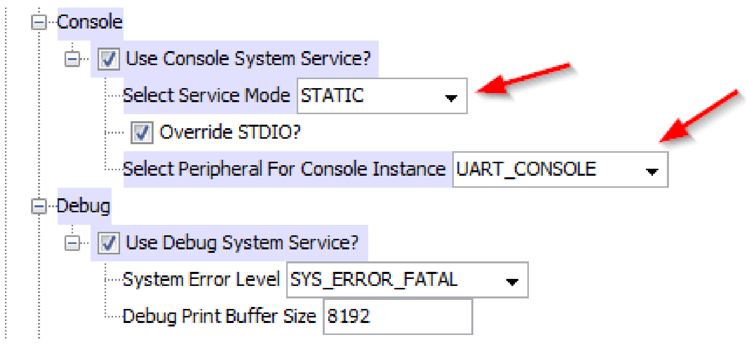
1. In the prior step, you selected “Debug System Service”. Notice what happened: “Console” was also automatically selected. Let’s expand that:



System Services
- Clock
- Command
- Common
- Console
  - Use Console System Service? ✓
    - Select Service Mode: DYNAMIC
    - Override STDIO? ✓
    - Select Peripheral For Console Instance: APPIO_CONSOLE
    - Read APPIO Buffer Queue Size (1-128): 8
    - Write APPIO Buffer Queue Size (1-256): 128

2. Let’s change **“APPIO_CONSOLE”** to “**UART_CONSOLE**”. APPIO_CONSOLE is a setting to allow you to get output directly in the MPLAB$^{®}$ X IDE via a debug channel.

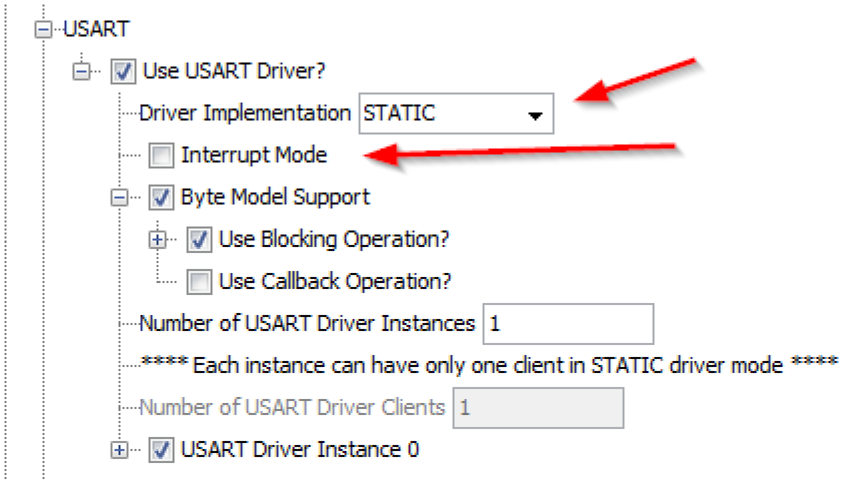
   Also, change Select Service Mode to “STATIC”



Console
- Use Console System Service? ✓
  - Select Service Mode: STATIC
  - Override STDIO? ✓
  - Select Peripheral For Console Instance: UART_CONSOLE

Debug
- Use Debug System Service? ✓
  - System Error Level: SYS_ERROR_FATAL
  - Debug Print Buffer Size: 8192

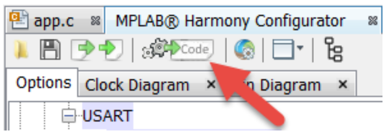
In response to the “UART_CONSOLE” selection above, MHC included the UART driver! Expand the “Harmony Framework Configuration > Drivers > USART” tree. Select “**STATIC**” mode under Driver Implementation



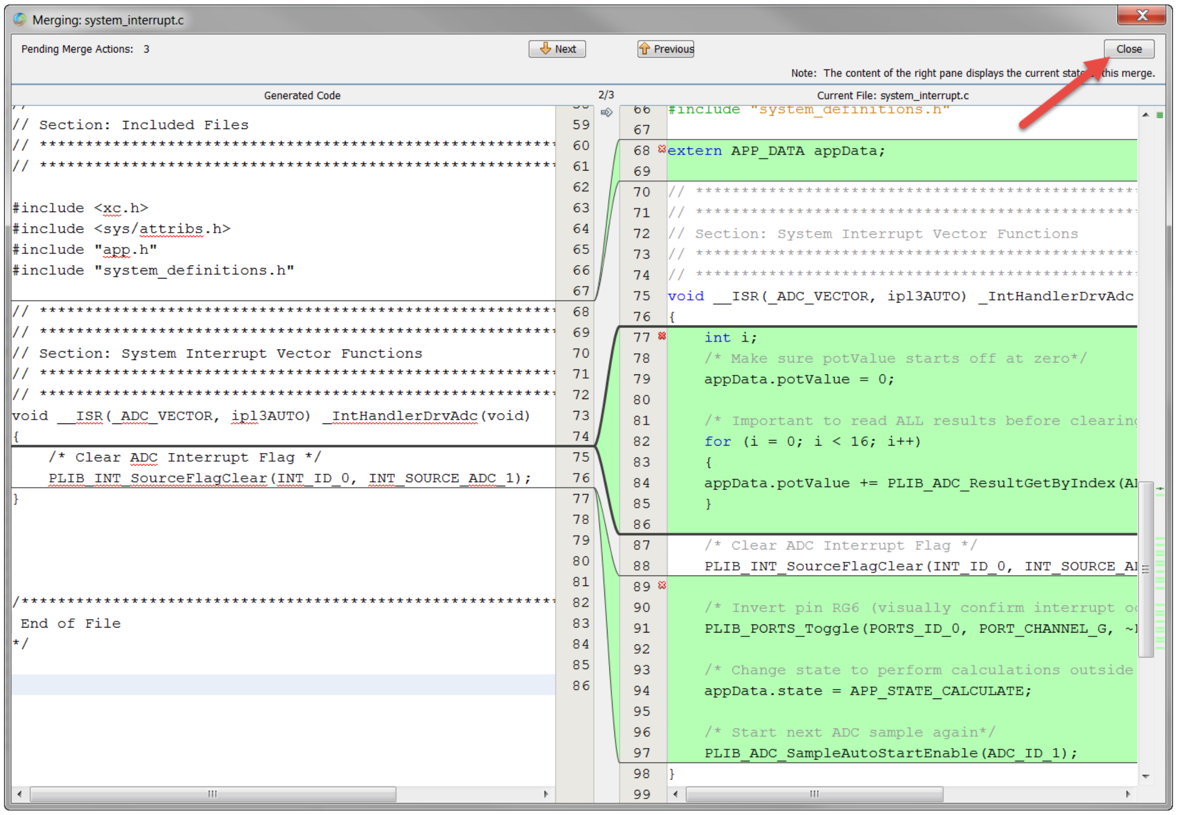
USART
- Use USART Driver? ✓
  - Driver Implementation: STATIC
  - Interrupt Mode ☐
  - Byte Model Support ✓
    - Use Blocking Operation? ✓
    - Use Callback Operation? ☐
  - Number of USART Driver Instances: 1
  - **** Each instance can have only one client in STATIC driver mode ****
  - Number of USART Driver Clients: 1
  - USART Driver Instance 0 ✓

Step 3: Generate the newly configured code

Click the “Gear” → Code icon



MHC will not blindly modify your existing code (unless you tell it to). When MHC generates code for a file that is different from the existing code in that file, it will open a window showing the differences between the generated code (left side) and the existing code (right side). In this case, MHC isn’t creating any new code for this file, so there is nothing to merge. Just click the close button.


Merging: system_interrupt.c
Pending Merge Actions: 3
Next
Previous
Close
Note: The content of the right pane displays the current state this merge.
Generated Code
2/3
Current File: system_interrupt.c

```
// Section: Included Files
// ************************************************************
// ************************************************************

#include <xc.h>
#include <sys/attribs.h>
#include "app.h"
#include "system_definitions.h"

// ************************************************************
// ************************************************************
// Section: System Interrupt Vector Functions
// ************************************************************
// ************************************************************
void __ISR(_ADC_VECTOR, ipl3AUTO) _IntHandlerDrvAdc(void)
{
    /* Clear ADC Interrupt Flag */
    PLIB_INT_SourceFlagClear(INT_ID_0, INT_SOURCE_ADC_1);
}



/*************************************************************
 End of File
*/
```

```
#include "system_definitions.h"

extern APP_DATA appData;

// ************************************************
// ************************************************
// Section: System Interrupt Vector Functions
// ************************************************
// ************************************************
void __ISR(_ADC_VECTOR, ipl3AUTO) _IntHandlerDrvAdc
{
    int i;
    /* Make sure potValue starts off at zero*/
    appData.potValue = 0;

    /* Important to read ALL results before clearin
    for (i = 0; i < 16; i++)
    {
    appData.potValue += PLIB_ADC_ResultGetByIndex(AI
    }

    /* Clear ADC Interrupt Flag */
    PLIB_INT_SourceFlagClear(INT_ID_0, INT_SOURCE_AI

    /* Invert pin RG6 (visually confirm interrupt oc
    PLIB_PORTS_Toggle(PORTS_ID_0, PORT_CHANNEL_G, ~I

    /* Change state to perform calculations outside
    appData.state = APP_STATE_CALCULATE;

    /* Start next ADC sample again*/
    PLIB_ADC_SampleAutoStartEnable(ADC_ID_1);
}
```

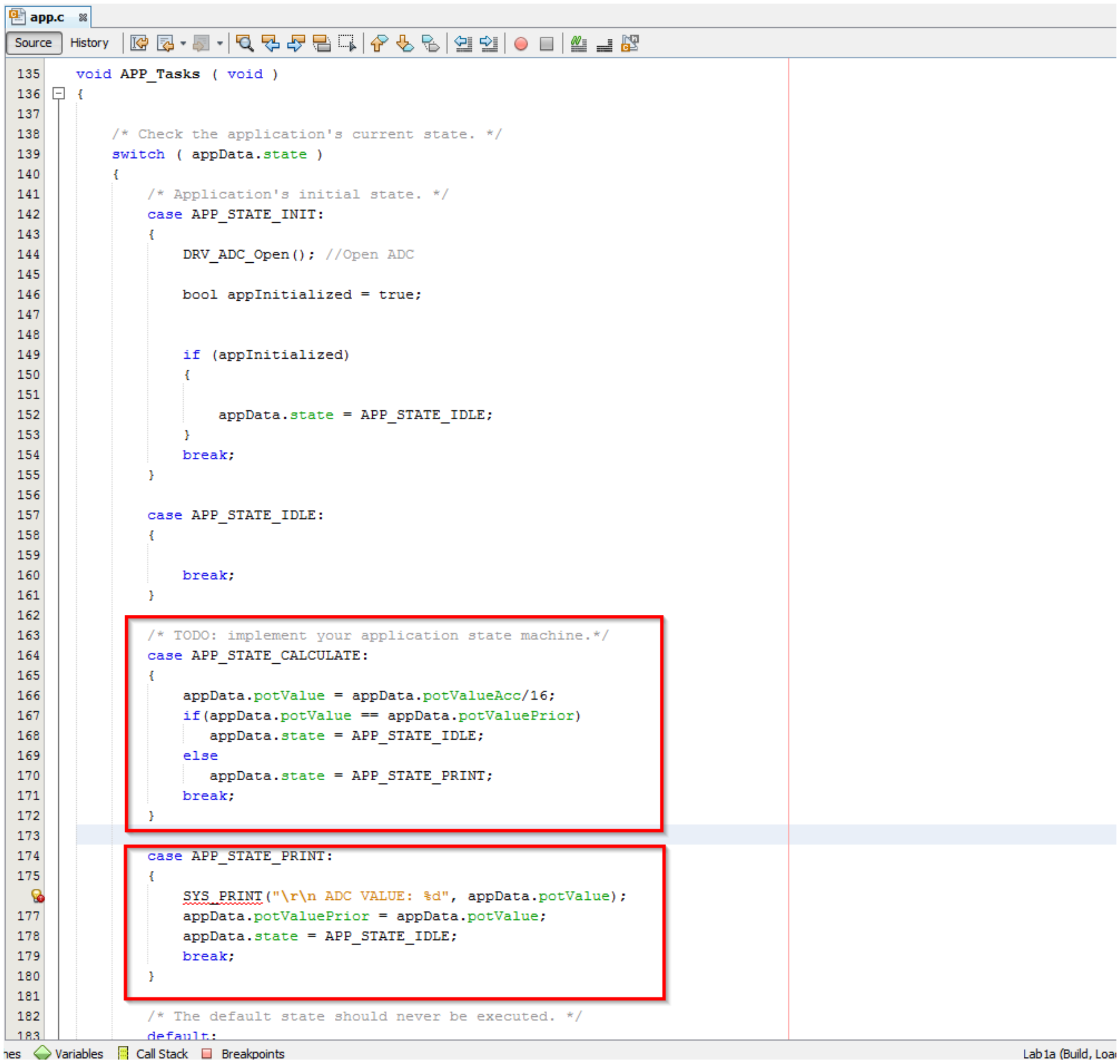
Step 4: Add code to print ADC result to UART

(Note: the solutions for this step can be found on the next page)

Open file app.c (located under Source Files in the app folder). In the function ***APP_Tasks(),*** you will add a new state to print values to the UART. You will also add code to integrate this new state into the current state machine.

1. In the "APP_TASKS()" function (app.c), add a new state (case statement) called "APP_STATE_PRINT". This state will print the ADC result to the UART.
   a. Create a new state:
      ```
      case APP_STATE_PRINT:
      ```
   b. Print the ADC result (appData.potValue) using the System Library function "SYS_PRINT()":
      ```
      SYS_PRINT("\r\n ADC VALUE: %d ", appData.potValue);
      ```
      Note: SYS_PRINT uses the same formatting as PRINTF. Of course, this comes at the expense of Data and Program memory! Print out the appData.potValue variable.
   c. Save the last result sent to the UART. We will only print a value to the UART if there is a change from the previous value.
      ```
      appData.potValuePrior = appData.potValue;
      ```
   d. Advance the state machine to "idle":
      ```
      appData.state = APP_STATE_IDLE;
      ```
   e. Don't forget to add the break at the end of the case:
      ```
      break;
      ```
2. In the "APP_TASKS()" function (app.c), inside the "calculate" state, add code to print the new ADC value to the UART, *only if the value is different from the previous one*.
   a. Modify the "calculate" state to print a new ADC value if there is a change:
      ```
      if (appData.potValue == appData.potValuePrior)
         appData.state = APP_STATE_IDLE;
      else
         appData.state = APP_STATE_PRINT;
      ```

Step 4: Solution

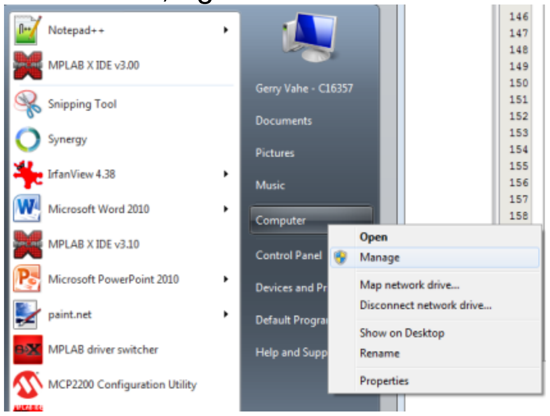


```c
void APP_Tasks ( void )
{

    /* Check the application's current state. */
    switch ( appData.state )
    {
        /* Application's initial state. */
        case APP_STATE_INIT:
        {
            DRV_ADC_Open(); //Open ADC

            bool appInitialized = true;


            if (appInitialized)
            {

                appData.state = APP_STATE_IDLE;
            }
            break;
        }

        case APP_STATE_IDLE:
        {

            break;
        }

        /* TODO: implement your application state machine.*/
        case APP_STATE_CALCULATE:
        {
            appData.potValue = appData.potValueAcc/16;
            if(appData.potValue == appData.potValuePrior)
               appData.state = APP_STATE_IDLE;
            else
               appData.state = APP_STATE_PRINT;
            break;
        }

        case APP_STATE_PRINT:
        {
            SYS_PRINT("\r\n ADC VALUE: %d", appData.potValue);
            appData.potValuePrior = appData.potValue;
            appData.state = APP_STATE_IDLE;
            break;
        }

        /* The default state should never be executed. */
        default:
```

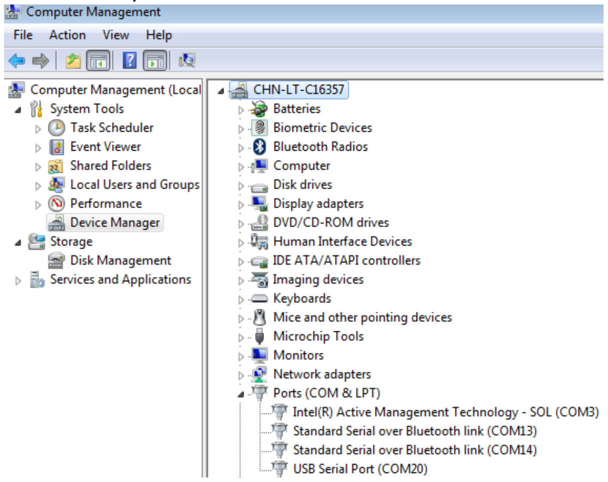
Step 5: Enjoy the results!

You are now able to **build, program, and run** your project. Click on this icon:

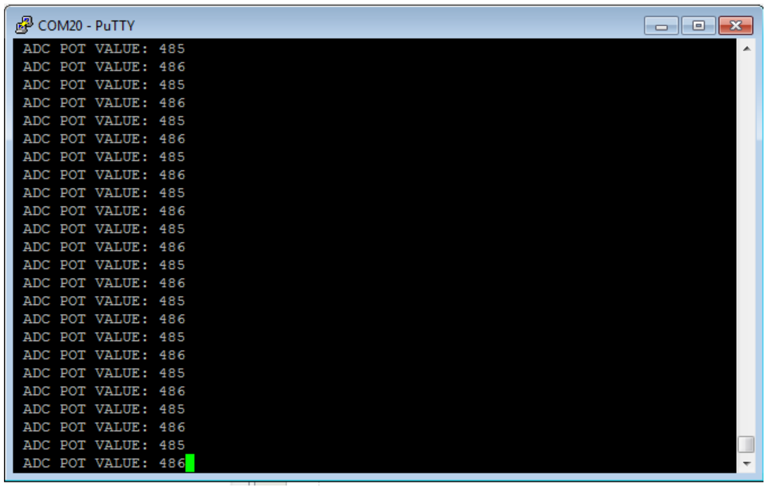
Determine which COM port the FTDI UART to USB chip on the WF32 enumerated to: On Start menu, right click on “COMPUTER” and click “Manage”



Click “Device Manager” and expand “Ports” to determine what com port the WF32 is on. In this case, it’s on COM20



Open your favorite terminal program, and select the com port determined in step 2 above, @ 9600 Baud and “ENJOY!”



```
ADC POT VALUE: 485
ADC POT VALUE: 486
ADC POT VALUE: 485
ADC POT VALUE: 486
ADC POT VALUE: 485
ADC POT VALUE: 486
ADC POT VALUE: 485
ADC POT VALUE: 486
ADC POT VALUE: 485
ADC POT VALUE: 486
ADC POT VALUE: 485
ADC POT VALUE: 486
ADC POT VALUE: 485
ADC POT VALUE: 486
ADC POT VALUE: 485
ADC POT VALUE: 486
ADC POT VALUE: 485
ADC POT VALUE: 486
ADC POT VALUE: 485
ADC POT VALUE: 486
ADC POT VALUE: 485
ADC POT VALUE: 486
ADC POT VALUE: 485
ADC POT VALUE: 486
```

LAB 2a
Create a USB flash drive bootloader

LAB 2a: Create a USB Flash Drive Bootloader

Purpose:

After completing LAB 2a, you will know how to configure the bootloader library. This section shows how a complex task can be accomplished efficiently in Harmony!
NOTE: There are other bootloader configurations we will not cover in this lab. However, it is highly encouraged to experiment with the other configurations

Overview:

In this lab, you will add the Harmony Bootloader Library to your project. The lab result will demonstrate how quickly a practical, real world feature can be set up. When finished with this lab, you will be able to put the HEX file of the project you created in LAB 1 on a USB Flash Drive. Then, with the bootloader we create in LAB 2a, the file will be read off the Flash Drive and programmed in the PIC32 flash, and executed.

This lab requires a Standard or PRO (not free) XC32 compiler version to generate 16-bit (MIPS16) code. Note the free version includes a 60-day PRO evaluation that can be started at any time.

Procedure:

Create new project, configure the device, and enable the bootloader library

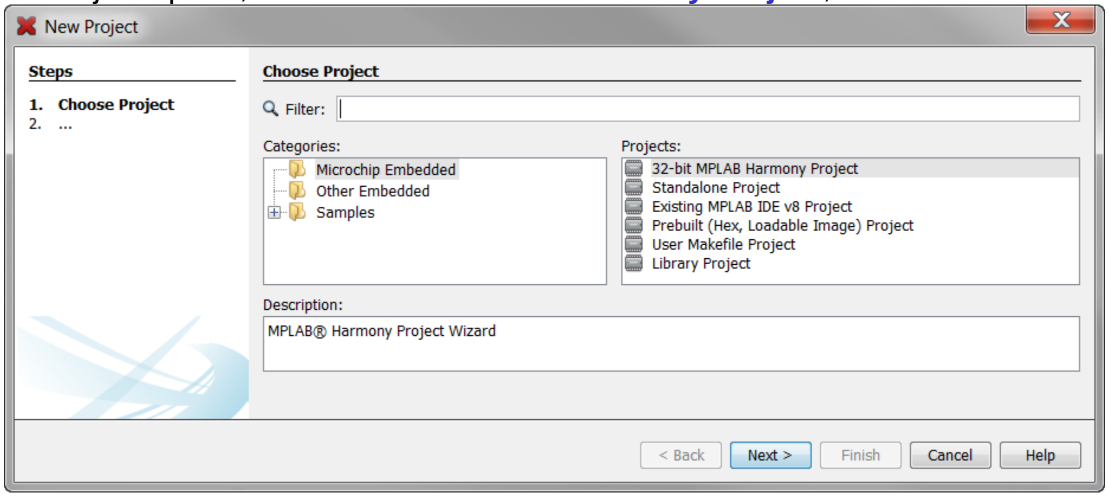
- Step 1 – Create an MPLAB® Harmony project in the MPLAB® X IDE
- Step 2 – With MHC, Select BSP for WF32 board.
- Step 3 – With MHC, set up the Clocks (NOT REQUIRED FOR HARMONY 1.07 with WF32 BSP!)
- Step 4 – With MHC, select and configure the Bootloader Library

Configure the Hardware and program the board

- Step 5 – Configure WF32 Jumpers to properly operate with USB HOST
- Step 6 – Load bootloader application to target

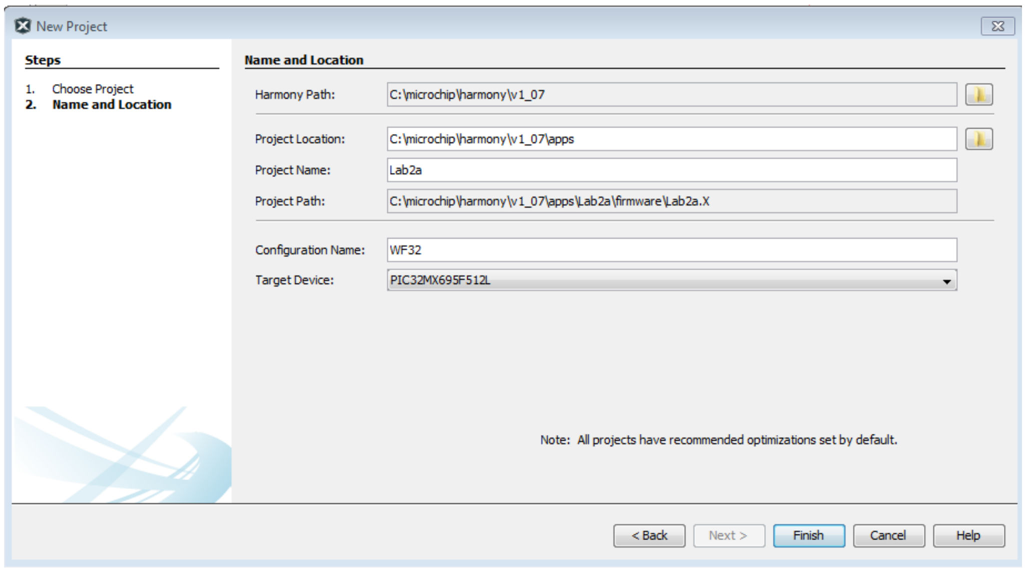
Step 1: Create an MPLAB® Harmony Project for USB Bootloader

1. These steps closely resemble the beginning of LAB 1a. Create a New Project by selecting ***File*** > ***New Project***.
2. In the Categories pane of the New Project dialog, select **Microchip Embedded**.
3. In the Projects pane, select **32-bit MPLAB Harmony Project**, and then click **Next**.



Description: MPLAB® Harmony Project Wizard

4. Specify the following in the New Project dialog (see below):
   - **Harmony Path** (path to the folder containing the MPLAB® Harmony framework)
   - **Project Location** (navigate where you want to put your project)
   - **Enter Project Name : Lab2a**
   - **Configuration Name : WF32** (this is optional)
   - **Target Device : PIC32MX695F512L**



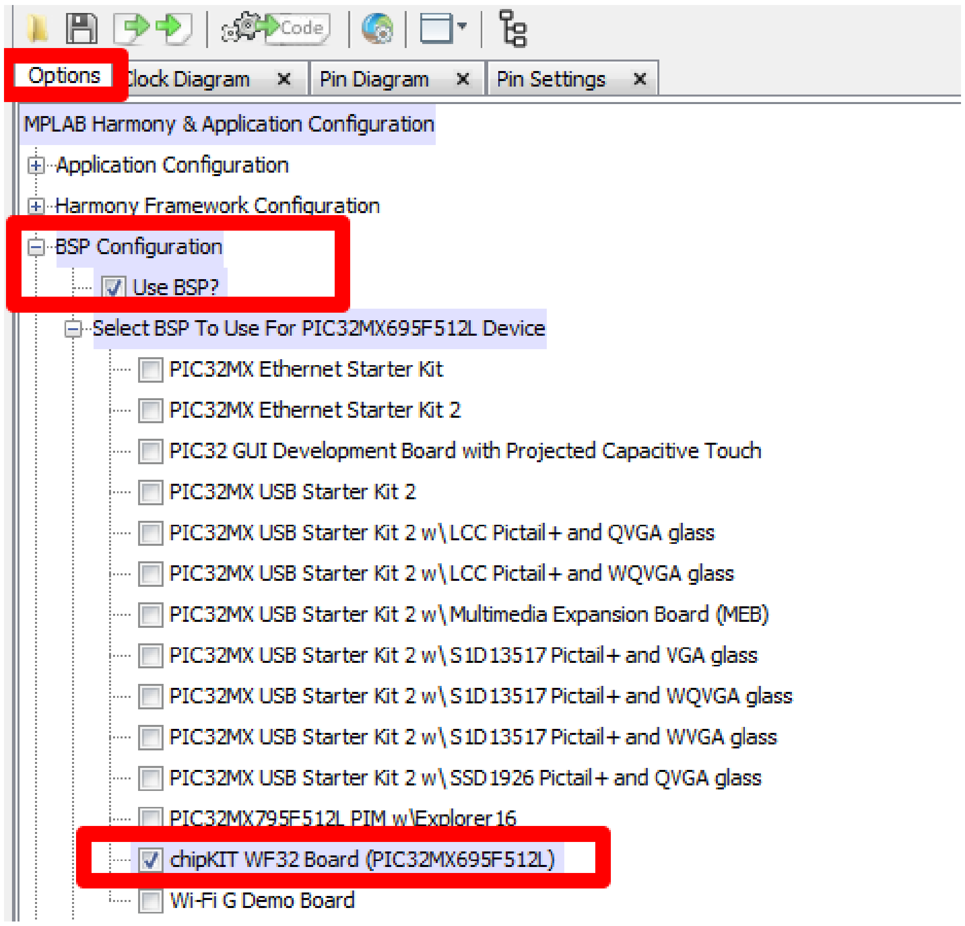
Harmony Path: C:\microchip\harmony\v1_07
Project Location: C:\microchip\harmony\v1_07\apps
Project Name: Lab2a
Project Path: C:\microchip\harmony\v1_07\apps\Lab2a\firmware\Lab2a.X
Configuration Name: WF32
Target Device: PIC32MX695F512L

Note: All projects have recommended optimizations set by default.

5. After clicking the **Finish** button, the project will be created and opened. You will see the MPLAB® Harmony Configurator (MHC) window along with the integrated Harmony Help file.

Step 2: Select Board Support Package (BSP)

1. In the central window under the **MPLAB® Harmony Configurator** tab, click on the **Options** sub-tab, to view the MPLAB Harmony & Application Configuration tree selections.
2. Expand the **BSP Configuration** tree, then select **chipKit WF32 Board (PIC32MX695F512L)**



MPLAB Harmony & Application Configuration
- Application Configuration
- Harmony Framework Configuration
- BSP Configuration
  - [✓] Use BSP?
  - Select BSP To Use For PIC32MX695F512L Device
    - [ ] PIC32MX Ethernet Starter Kit
    - [ ] PIC32MX Ethernet Starter Kit 2
    - [ ] PIC32 GUI Development Board with Projected Capacitive Touch
    - [ ] PIC32MX USB Starter Kit 2
    - [ ] PIC32MX USB Starter Kit 2 w\LCC Pictail+ and QVGA glass
    - [ ] PIC32MX USB Starter Kit 2 w\LCC Pictail+ and WQVGA glass
    - [ ] PIC32MX USB Starter Kit 2 w\Multimedia Expansion Board (MEB)
    - [ ] PIC32MX USB Starter Kit 2 w\S1D13517 Pictail+ and VGA glass
    - [ ] PIC32MX USB Starter Kit 2 w\S1D13517 Pictail+ and WQVGA glass
    - [ ] PIC32MX USB Starter Kit 2 w\S1D13517 Pictail+ and WVGA glass
    - [ ] PIC32MX USB Starter Kit 2 w\SSD1926 Pictail+ and QVGA glass
    - [ ] PIC32MX795F512L PIM w\Explorer16
    - [✓] chipKIT WF32 Board (PIC32MX695F512L)
    - [ ] Wi-Fi G Demo Board

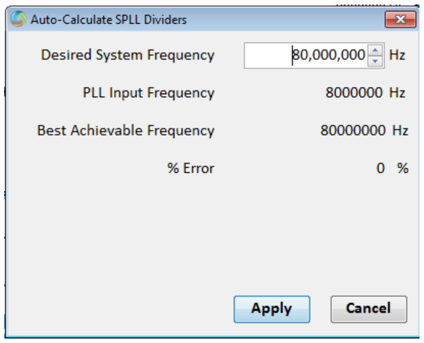
Step 3: Configure the PIC32 oscillator module

NOTE!! STEP 3 IS NOT REQUIRED!! We added a BSP for the chipKIT WF32 board, and STEP 2 completes all of step 3 for you!!!

If you didn't use a BSP, the following is what you would have had to do:

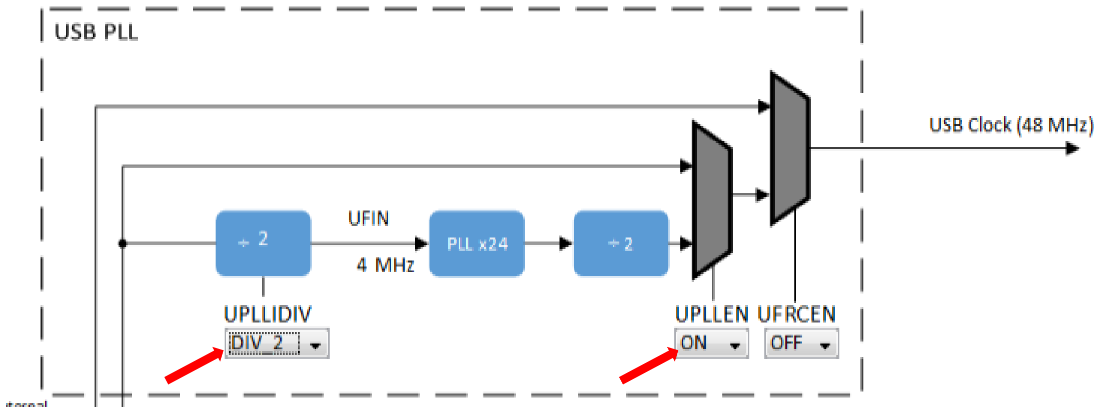
Configure the USB clock (needed for the bootloader). Select the **Clock Diagram** tab.

1. Use the following clock parameters:
   a. Select **XT** for POSCMOD
   b. Select **PRIPLL** for FNOSC
   c. Select "**Auto Calculate**" and enter your desired output clock frequency (80 MHz).



Desired System Frequency	80,000,000	Hz
PLL Input Frequency	8000000	Hz
Best Achievable Frequency	80000000	Hz
% Error	0	%

   d. There is no auto calculate for the USB section. We need to Divide the 8MHz by 2, and enable the USB PLL to get 48MHz out.
      - Select **DIV_2** for UPLLIDIV
      - Select **On** for UPLLEN
      - Set UFRCEN to **OFF**



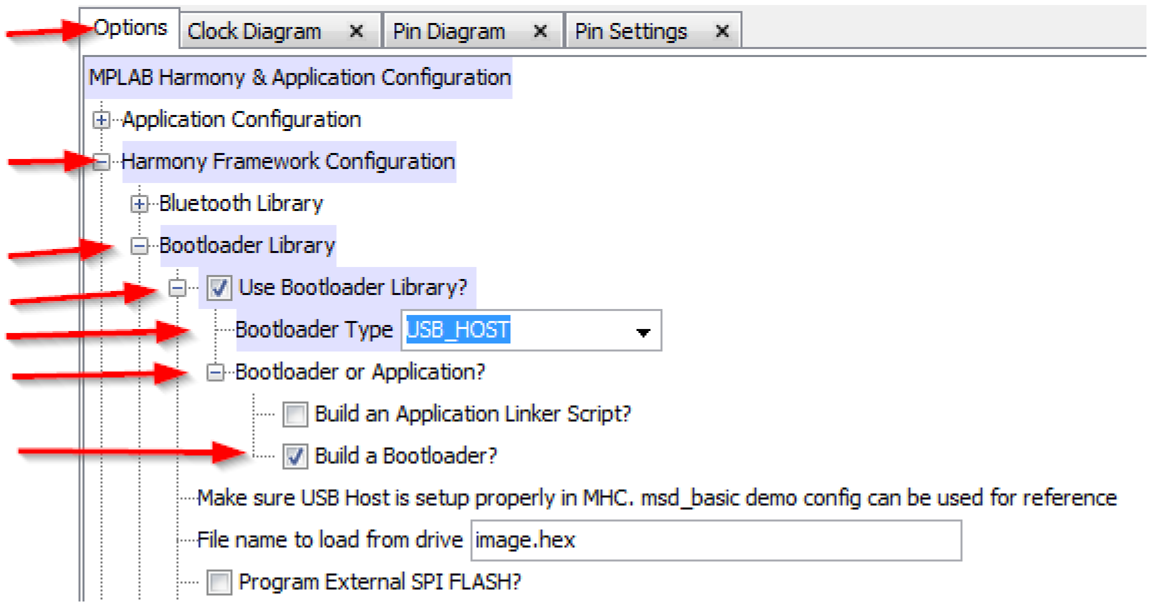
Step 4: Configure Bootloader Library for USB MSD

1. Go back to the “Options” tab in MHC. Expand **Harmony Framework Configuration,** and then **Bootloader Library.**

2. Check the **Use Bootloader Library?** box.

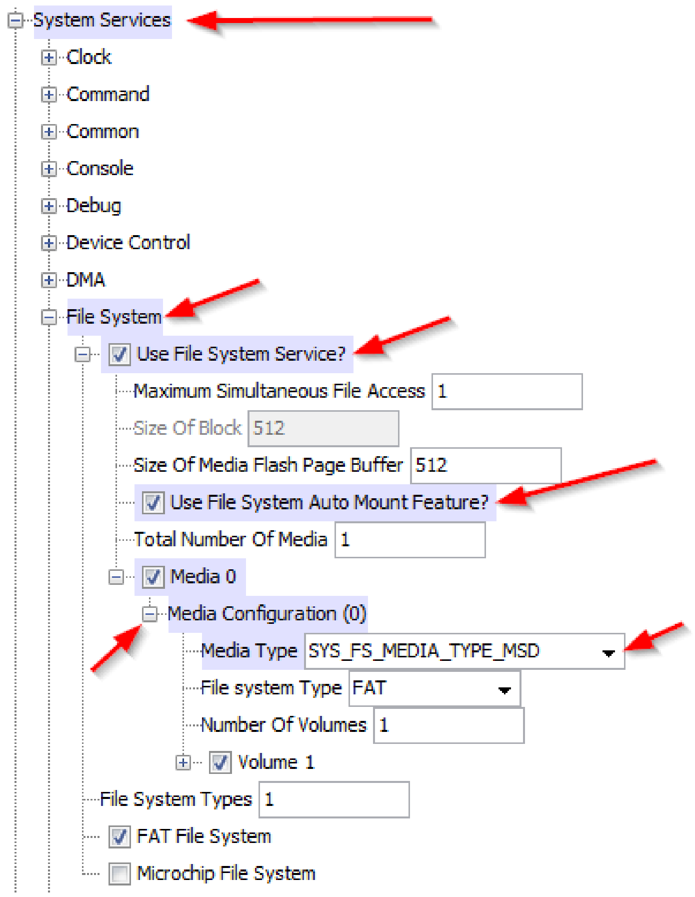
3. Expand **Bootloader or Application?** and ensure **Build a Bootloader?** is checked.

4. Under **Bootloader Type**, select **USB_HOST**.


Options
Clock Diagram
Pin Diagram
Pin Settings
MPLAB Harmony & Application Configuration
Application Configuration
Harmony Framework Configuration
Bluetooth Library
Bootloader Library

Use Bootloader Library? [✓]
Bootloader Type: USB_HOST
Bootloader or Application?
Build an Application Linker Script? [ ]
Build a Bootloader? [✓]
Make sure USB Host is setup properly in MHC. msd_basic demo config can be used for reference
File name to load from drive: image.hex
Program External SPI FLASH? [ ]

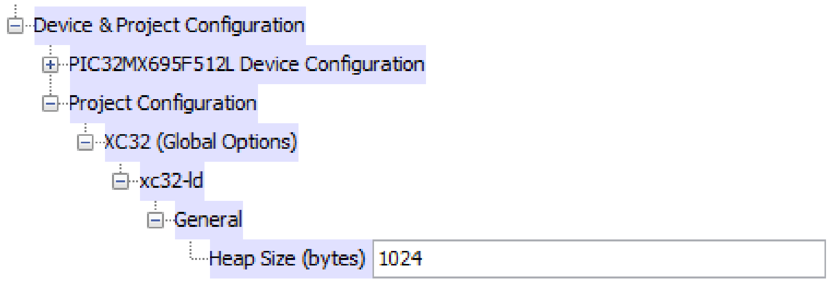
5. Under **Harmony Framework Configuration - System Services - File System,** ensure **"Use File System Service?"** is already checked.
6. Expand **Use File System Service?** and check **Use File System Auto Mount Feature?**
7. Expand **Media 0 – Media Configuration (0)** , and then change **Media Type** to **SYS_FS_MEDIA_TYPE_MSD**



```
System Services
  Clock
  Command
  Common
  Console
  Debug
  Device Control
  DMA
  File System
    [✓] Use File System Service?
      Maximum Simultaneous File Access  1
      Size Of Block  512
      Size Of Media Flash Page Buffer  512
      [✓] Use File System Auto Mount Feature?
      Total Number Of Media  1
      [✓] Media 0
        Media Configuration (0)
          Media Type  SYS_FS_MEDIA_TYPE_MSD
          File system Type  FAT
          Number Of Volumes  1
          [✓] Volume 1
    File System Types  1
    [✓] FAT File System
    [ ] Microchip File System
```

8. Under **Device and Project Config**, **Project Configuration**, **XC32 (Global Options)**, **xc32-ld**, **General, Heap Size**……Enter 1024 (make sure to press Enter).

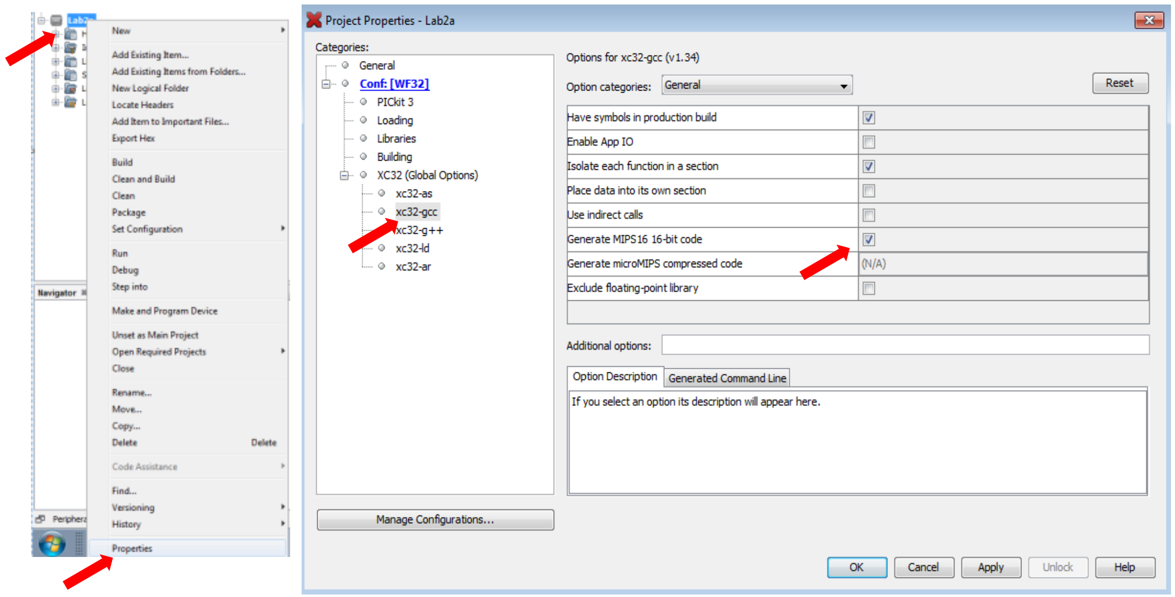
   You can enter this into the compiler settings instead. However, there is a CAUTION. Every time you "Generate Code" in MHC, it will override the setting you enter. By default, this value is 0, and the bootloader won't work!



```
Device & Project Configuration
  PIC32MX695F512L Device Configuration
  Project Configuration
    XC32 (Global Options)
      xc32-ld
        General
          Heap Size (bytes)  1024
```

9. One more subtle change. We need to select “**Generate MIPS 16-bit code**” in the compiler settings. The simpler bootloaders place all the code in the “Boot” section of flash. Although the USB HOST bootloader takes up considerably more space than a simple bootloader, the project was designed to fit as much as could be fit in the BOOT section of flash.

   Right Click on your Project name (Lab2a), and then Properties. Under the **XC32-GCC** settings, check the **Generate MIPS16 16-bit code** option



Options for xc32-gcc (v1.34)

Option categories: General

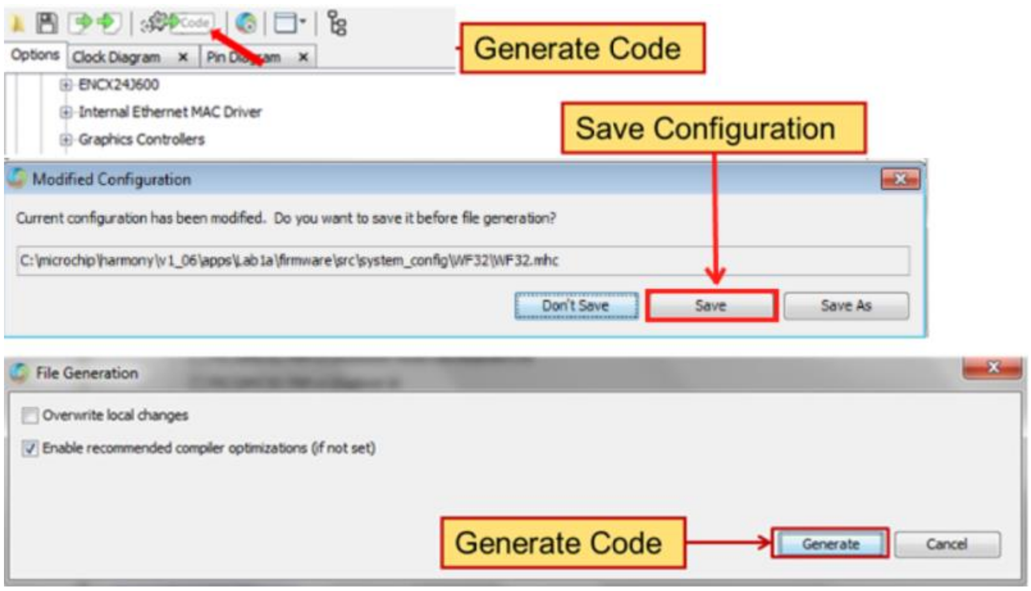
Have symbols in production build	✓
Enable App IO	
Isolate each function in a section	✓
Place data into its own section	
Use indirect calls	
Generate MIPS16 16-bit code	✓
Generate microMIPS compressed code	(N/A)
Exclude floating-point library	

Additional options:

Option Description | Generated Command Line

If you select an option its description will appear here.

10. That seems like a lot of steps, but reflect on what you have just accomplished! You now have a USB Mass Storage Device Bootloader! Generate the code for all your hard work!



Current configuration has been modified. Do you want to save it before file generation?

C:\microchip\harmony\v1_06\apps\Lab1a\firmware\src\system_config\WF32\WF32.mhc

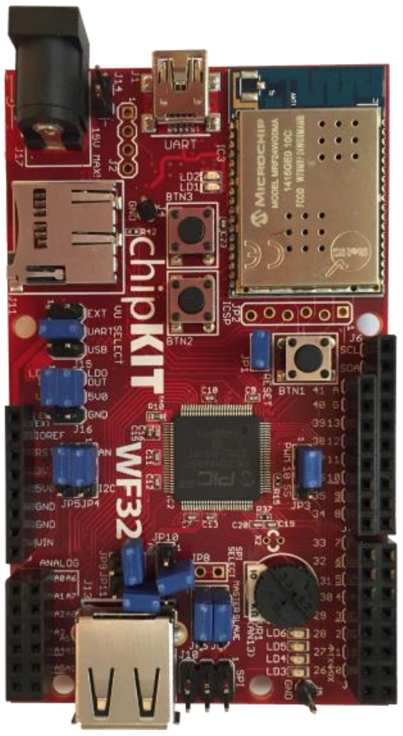
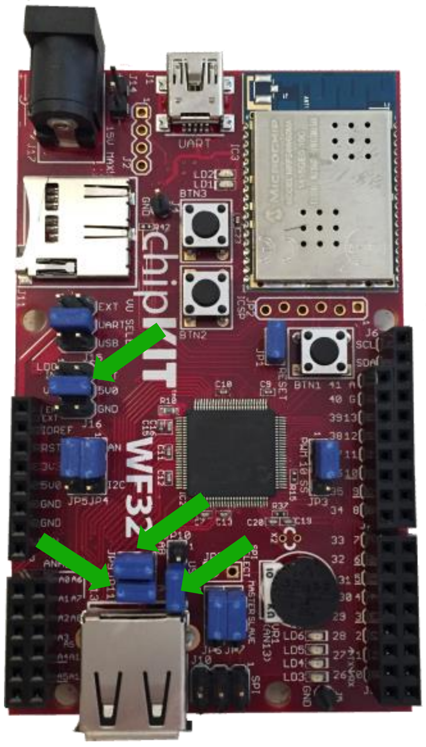
Overwrite local changes [ ]

Enable recommended compiler optimizations (if not set) [✓]

Step 5: Configure WF32 Jumpers for USB HOST

We need to change the default JUMPERS on the WF32, in order to provide power to the USB DRIVE. Change J16, JP9, JP10, JP11 marked by Green Arrows

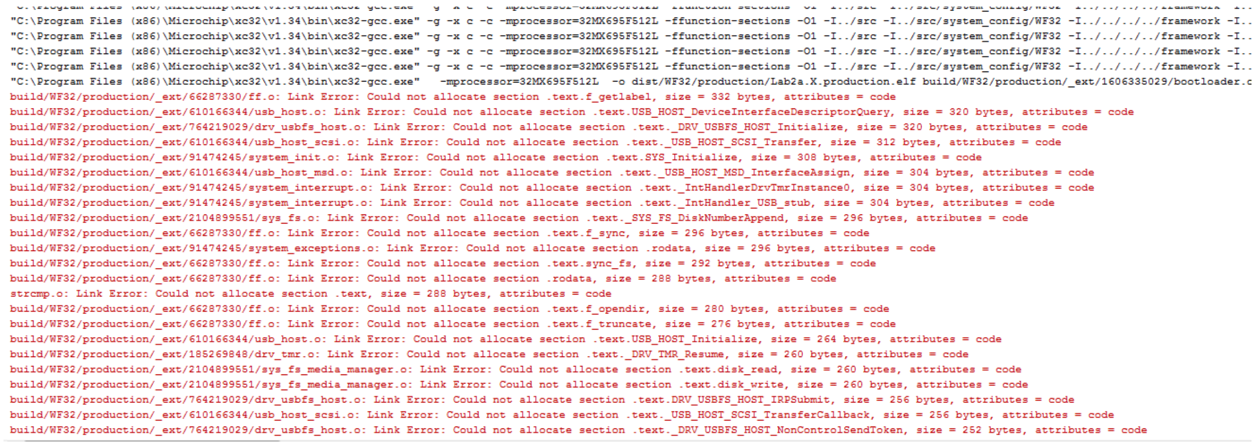
- **<u>DEFAULT JUMPER POSITIONS</u>** **<u>CHANGE TO THIS</u>**

Step 6: Load Bootloader Application to Target

1. Right click LAB 2a, and select **RUN**. The code should compile with no issues, and download to the target.

2. It is very easy to forget the previous “Generate MIPS16 16-Bit Code” step. If you do, you will get a slew of RED linker errors:



```
"C:\Program Files (x86)\Microchip\xc32\v1.34\bin\xc32-gcc.exe" -g -x c -c -mprocessor=32MX695F512L -ffunction-sections -O1 -I../src -I../src/system_config/WF32 -I../../../../framework -I..
"C:\Program Files (x86)\Microchip\xc32\v1.34\bin\xc32-gcc.exe" -g -x c -c -mprocessor=32MX695F512L -ffunction-sections -O1 -I../src -I../src/system_config/WF32 -I../../../../framework -I..
"C:\Program Files (x86)\Microchip\xc32\v1.34\bin\xc32-gcc.exe" -g -x c -c -mprocessor=32MX695F512L -ffunction-sections -O1 -I../src -I../src/system_config/WF32 -I../../../../framework -I..
"C:\Program Files (x86)\Microchip\xc32\v1.34\bin\xc32-gcc.exe" -g -x c -c -mprocessor=32MX695F512L -ffunction-sections -O1 -I../src -I../src/system_config/WF32 -I../../../../framework -I..
"C:\Program Files (x86)\Microchip\xc32\v1.34\bin\xc32-gcc.exe"   -mprocessor=32MX695F512L  -o dist/WF32/production/Lab2a.X.production.elf build/WF32/production/_ext/1606335029/bootloader.c
build/WF32/production/_ext/66287330/ff.o: Link Error: Could not allocate section .text.f_getlabel, size = 332 bytes, attributes = code
build/WF32/production/_ext/610166344/usb_host.o: Link Error: Could not allocate section .text.USB_HOST_DeviceInterfaceDescriptorQuery, size = 320 bytes, attributes = code
build/WF32/production/_ext/764219029/drv_usbfs_host.o: Link Error: Could not allocate section .text._DRV_USBFS_HOST_Initialize, size = 320 bytes, attributes = code
build/WF32/production/_ext/610166344/usb_host_scsi.o: Link Error: Could not allocate section .text._USB_HOST_SCSI_Transfer, size = 312 bytes, attributes = code
build/WF32/production/_ext/91474245/system_init.o: Link Error: Could not allocate section .text.SYS_Initialize, size = 308 bytes, attributes = code
build/WF32/production/_ext/610166344/usb_host_msd.o: Link Error: Could not allocate section .text._USB_HOST_MSD_InterfaceAssign, size = 304 bytes, attributes = code
build/WF32/production/_ext/91474245/system_interrupt.o: Link Error: Could not allocate section .text._IntHandlerDrvTmrInstance0, size = 304 bytes, attributes = code
build/WF32/production/_ext/91474245/system_interrupt.o: Link Error: Could not allocate section .text._IntHandler_USB_stub, size = 304 bytes, attributes = code
build/WF32/production/_ext/2104899551/sys_fs.o: Link Error: Could not allocate section .text._SYS_FS_DiskNumberAppend, size = 296 bytes, attributes = code
build/WF32/production/_ext/66287330/ff.o: Link Error: Could not allocate section .text.f_sync, size = 296 bytes, attributes = code
build/WF32/production/_ext/91474245/system_exceptions.o: Link Error: Could not allocate section .rodata, size = 296 bytes, attributes = code
build/WF32/production/_ext/66287330/ff.o: Link Error: Could not allocate section .text.sync_fs, size = 292 bytes, attributes = code
build/WF32/production/_ext/66287330/ff.o: Link Error: Could not allocate section .rodata, size = 288 bytes, attributes = code
strcmp.o: Link Error: Could not allocate section .text, size = 288 bytes, attributes = code
build/WF32/production/_ext/66287330/ff.o: Link Error: Could not allocate section .text.f_opendir, size = 280 bytes, attributes = code
build/WF32/production/_ext/66287330/ff.o: Link Error: Could not allocate section .text.f_truncate, size = 276 bytes, attributes = code
build/WF32/production/_ext/610166344/usb_host.o: Link Error: Could not allocate section .text.USB_HOST_Initialize, size = 264 bytes, attributes = code
build/WF32/production/_ext/185269848/drv_tmr.o: Link Error: Could not allocate section .text._DRV_TMR_Resume, size = 260 bytes, attributes = code
build/WF32/production/_ext/2104899551/sys_fs_media_manager.o: Link Error: Could not allocate section .text.disk_read, size = 260 bytes, attributes = code
build/WF32/production/_ext/2104899551/sys_fs_media_manager.o: Link Error: Could not allocate section .text.disk_write, size = 260 bytes, attributes = code
build/WF32/production/_ext/764219029/drv_usbfs_host.o: Link Error: Could not allocate section .text.DRV_USBFS_HOST_IRPSubmit, size = 256 bytes, attributes = code
build/WF32/production/_ext/610166344/usb_host_scsi.o: Link Error: Could not allocate section .text._USB_HOST_SCSI_TransferCallback, size = 256 bytes, attributes = code
build/WF32/production/_ext/764219029/drv_usbfs_host.o: Link Error: Could not allocate section .text._DRV_USBFS_HOST_NonControlSendToken, size = 252 bytes, attributes = code
```

LAB 2b
Configure LAB 1 to be loaded from USB flash drive

LAB 2b: Configure LAB 1 to be loaded from USB flash drive

Purpose:

After completing LAB 2b, you will know how to reconfigure a Harmony Project so it can be loaded by the Harmony Bootloader.

Overview:

In this lab, you will reconfigure a Harmony Project using MHC so it can be loaded by the Bootloader we created in LAB 2a. The lab result will demonstrate how a complex task such as modifying a linker script can be easily accomplished with Harmony. When finished with this lab, you will create a HEX file of the project you created in LAB 1, and will move it to a USB Flash Drive. Then, with the bootloader we create in LAB 2a, the file will be read off the Flash Drive, programmed in the PIC32MX flash, and executed.

Procedure:

Re-Build LAB 1 with "Bootloader linker script"

- Step 1 – In MHC, build an application linker script for the LAB 1 project

Move HEX file to USB Flash Drive,& plug into chipKIT™ WF32 to upload program

- Step 2 – Build new project, and move the HEX file to USB drive
- Step 3 – Configure WF32 for bootloading, plug in USB drive, and watch it load and run!

Step 1: Configure Bootloader Linker Script

We are ALMOST there!!!

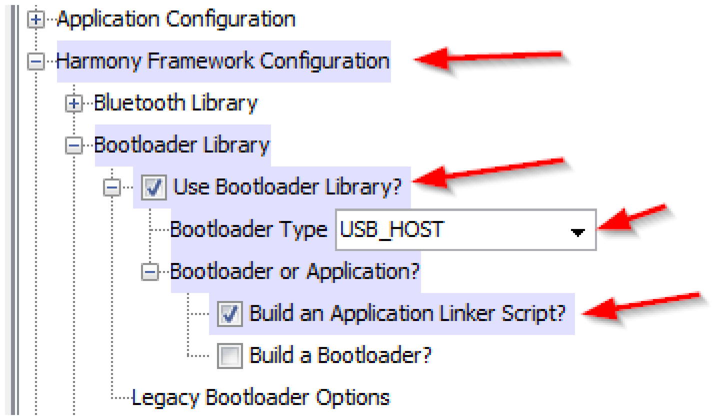
1. Go back to the original project we created in LAB 1. Right Click on the LAB 1a project name, and select **Set as Main Project**

2. If MHC (Microchip Harmony Configurator) is not open, go to **TOOLS, EMBEDDED, MPLAB HARMONY CONFIGURATOR**

3. Select the **Options** tab and expand **Harmony Framework Configuration**, **Bootloader Library**

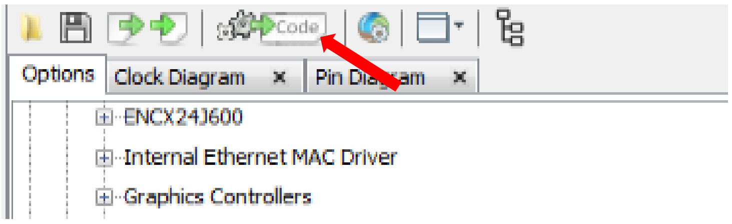
4. Select **Use Bootloader Library?**

5. Expand **Bootloader or Application?** And select **Build an Application Linker Script?**
6. Ensure **"Bootloader Type"** is set to **USB_HOST** to match what was selected in the bootloader built in prior step.

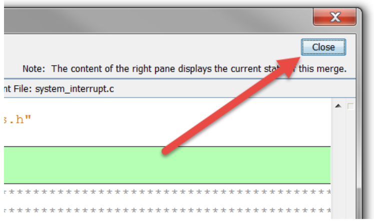


- Application Configuration
- Harmony Framework Configuration
  - Bluetooth Library
  - Bootloader Library
    - [✓] Use Bootloader Library?
      - Bootloader Type: USB_HOST
      - Bootloader or Application?
        - [✓] Build an Application Linker Script?
        - [ ] Build a Bootloader?
    - Legacy Bootloader Options

7. Generate the code by clicking this icon.



8. Click the Close button if the merge window opens. No changes need to be made to the previous system_interrupt.c file.



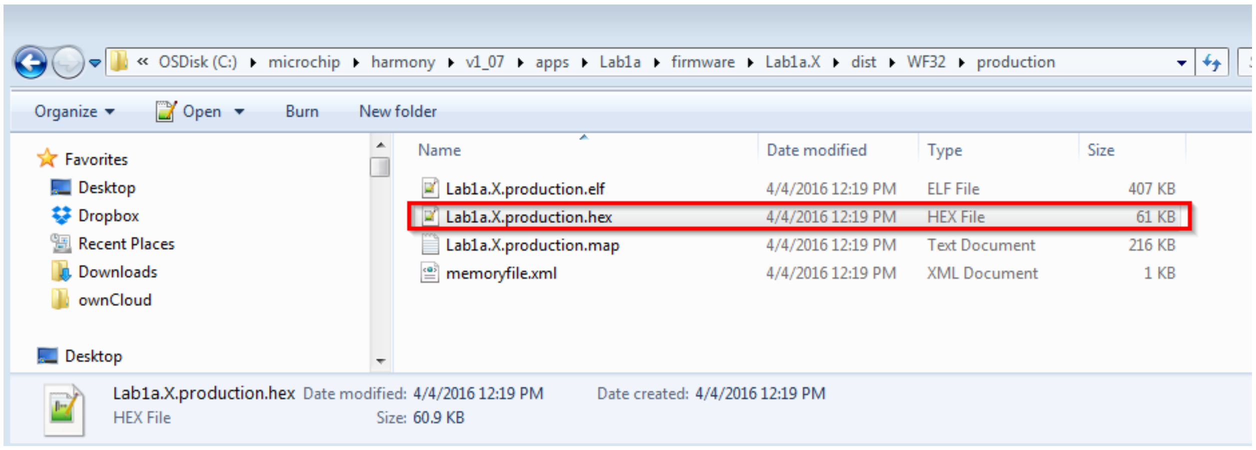
Note: The content of the right pane displays the current sta this merge.

nt File: system_interrupt.c

Step 2: Build new image and move to USB flash drive

1. Build the project by right clicking on LAB 1a, and selecting **BUILD**

2. Go to the following directory, and rename the .hex file to **image.hex.**


« OSDisk (C:) ▸ microchip ▸ harmony ▸ v1_07 ▸ apps ▸ Lab1a ▸ firmware ▸ Lab1a.X ▸ dist ▸ WF32 ▸ production
Organize
Open
Burn
New folder
Favorites
Desktop
Dropbox
Recent Places
Downloads
ownCloud
Desktop

Name	Date modified	Type	Size
Lab1a.X.production.elf	4/4/2016 12:19 PM	ELF File	407 KB
Lab1a.X.production.hex	4/4/2016 12:19 PM	HEX File	61 KB
Lab1a.X.production.map	4/4/2016 12:19 PM	Text Document	216 KB
memoryfile.xml	4/4/2016 12:19 PM	XML Document	1 KB

Lab1a.X.production.hex Date modified: 4/4/2016 12:19 PM Date created: 4/4/2016 12:19 PM
HEX File Size: 60.9 KB

“C:\microchip\harmony\v1_07_01\apps\Lab1a\firmware\Lab1a.X\dist\WF32\production

3. Copy the image.hex file to the USB Flash Drive

Step 3: Configure WF32 for bootloading, and watch it boot!

1. With target powered off, Insert the USB flash drive to the target. Power the device with the USB Mini Cable. If your USB Drive has an indicator light, you should see it blink several times. Eventually, you will see the target start to run the LAB we created in LAB 1!   You will probably need to reconnect your Terminal Program to see the output. Even if you don’t, you should see LED 3 Glow, as in the Interrupt Service routine for the ADC, we “Toggle” LED 3

Congratulations! You have finished all of the labs!