Tutorial - Creating Your Own Applications

MPLAB Harmony Integrated Software Framework

Table of Contents

1 Tutorial - Creating Your Own Applications

This tutorial describes the steps necessary to create your own MPLAB Harmony applications.

Description

The Creating Your Own Applications tutorial is presented in the following topics.

1.1 Introduction

This section provides an introduction to creating your own MPLAB Harmony applications.

Description

MPLAB Harmony provides a "Configurator" MPLAB X IDE plug-in tool to help you create your own MPLAB Harmony applications (see the **MPLAB Harmony Configurator** section for details). However, you should create your first application manually, using the instructions in this tutorial so that you know what the configurator tool does for you and to gain a better understanding of the structure of a MPLAB Harmony application.

Note: The version of the MPLAB Harmony Configurator provided in this release of MPLAB Harmony is an alpha release that does not yet provide complete functionality. Please refer to the related **Release Notes** for details on the limitations in this release.

The process of creating a MPLAB Harmony application follows these basic steps:

Step 1: Create the New Project

Step 2: Identify the Required Library Modules

Step 3: Add the Library Modules to the Project

Step 4: Configure the Modules

Step 5: Initialize the System and Modules

Step 6: Call the State Machines for the System and Modules

Step 7: Develop the Application State Machine

This tutorial will walk you through this process and explain how to complete each step.

Notes:

1. Although this tutorial will perform only one pass through these steps, it is generally easier to build up an application module-by-module, making multiple passes through this process by starting over at Step 2: Identify the Required Library Modules for each new MPLAB Harmony library or module you add to your system. This allows you to build up your application piece-by-piece and test each new feature as your project develops into a complete solution.
2. If you are a Microchip Libraries for Applications (MLA) user, porting your application from the MLA TCP/IP, File System, USB Device, or Graphics libraries to the MPLAB Harmony equivalents, be sure to follow the tutorial first, and then refer to the following porting guides for additional guidance, which are prefixed by their respective MPLAB Harmony Help PDF (`<install-dir>/docs/help_harmony.pdf`) section number:

- **Section 12.3.1 Graphics Library Porting Guide**
- **Section 12.7.2 File System Service Porting Guide**
- **Section 12.8.2 TCP/IP Stack Library Porting Guide**
- **Section 12.9.2 USB Device Stack Library Porting Guide**

1.2 Prerequisites

This topic describes the prerequisites for creating your own MPLAB Harmony applications.

Description

This tutorial assumes that you have already completed these steps:

1. Installed the MPLAB X IDE (http://www.microchip.com/mplabx).
2. Installed MPLAB Harmony (http://www.microchip.com/harmony).
3. Installed the MPLAB XC32 C/C++ Compiler (http://www.microchip.com/xc32).
4. Set up a working PIC32 development platform (http://www.microchip.com/32bit).

You can download the MPLAB X IDE, MPLAB Harmony and the MPLAB XC32 C/C++ Compiler from the links provided. If you do not already have a PIC32 development platform, you can learn more about the PIC32 family and determine which hardware platform best meets your development needs by visiting the 32-bit website listed previously.

This tutorial also assumes that you have some familiarity with the MPLAB X IDE, embedded C-language programming and PIC32 microcontrollers. If you are unsure how to complete some of the steps in this tutorial, please refer to the documentation for the item on which you have questions. You may also seek assistance from your peers on the Microchip discussion forums (http://www.microchip.com/forums) or from the Microchip support staff (www.microchip.com/support).

Once you have everything installed, connected, and up and running you are ready to begin creating your own MPLAB Harmony applications.

1.3 Step 1: Create the New Project

To create a new MPLAB Harmony project, you first need to create a new MPLAB X IDE project and the basic set of source code files and functions that are necessary for a properly formed MPLAB Harmony application.

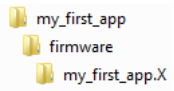
Description

Create the Project Folders

It is best to locate your first application from the `apps` folder within the MPLAB Harmony installation. By default, this will be `c:\Microchip\harmony\<version>\apps` on a Windows PC or `~/microchip/harmony/<version>/apps` on a Mac or a computer running the Linux operating system. Later, you can create projects in any location you prefer, but using the `apps` folder will keep the relative paths to library files simple and they will look similar to the paths used by the applications provided in the MPLAB Harmony installation. It will also make it easy to keep your application associated with a specific version of MPLAB Harmony and to copy it to the `apps` folder of a future release.

For this tutorial, name the top-level folder for your application `my_first_app`. This folder will hold all collateral specific to the application. Within the `my_first_app` folder, create a `firmware` sub-folder. (In more complex projects, you may add additional sub-folders to hold other related items such as web server data files, host PC driver configuration files, graphics display images, and so on.) When you create your MPLAB X IDE project, place your project folder (for example `my_first_app.X`) within the firmware folder, as follows.

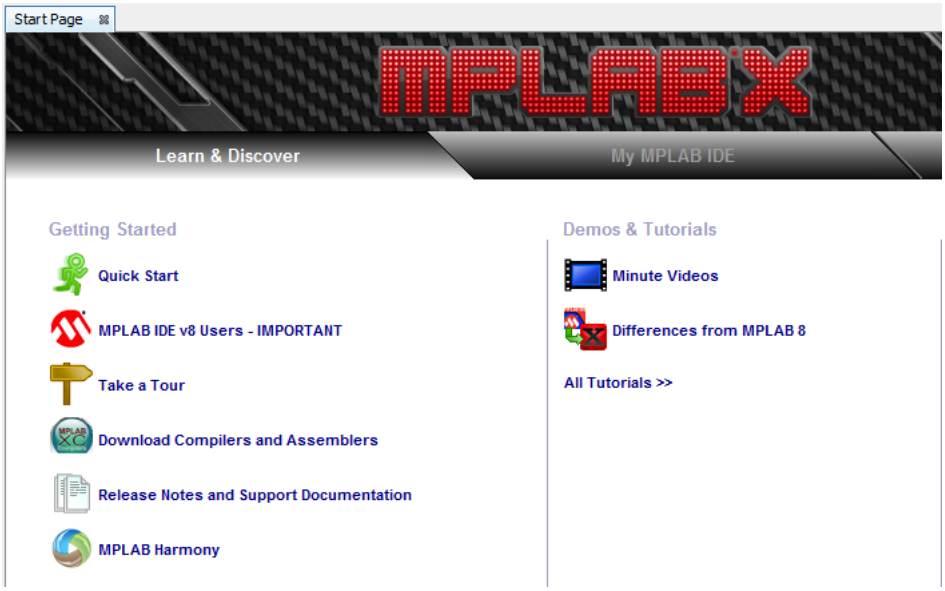
Project Folder Organization



Create a New MPLAB X IDE Project

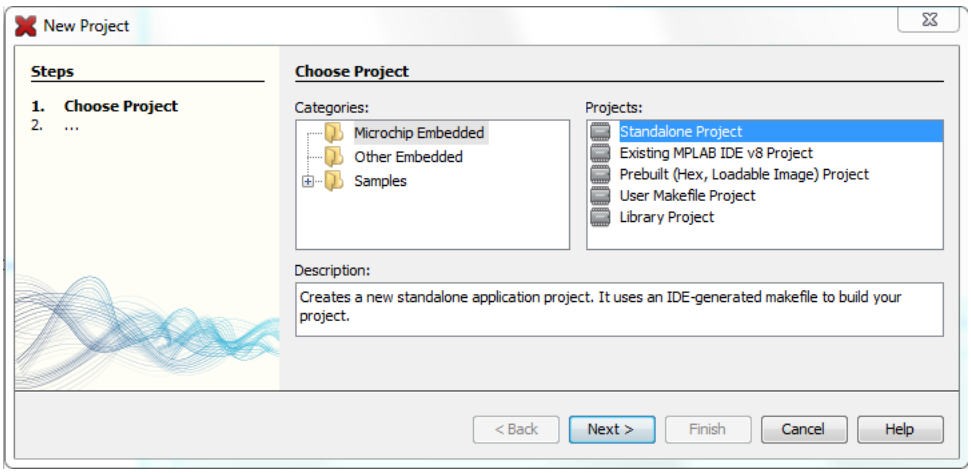
Start the MPLAB X IDE on the development computer where you installed the software listed in Prerequisites. If you are new to

MPLAB X IDE, you might want to click **Take a Tour** on the Start Page (shown in the following figure) to become more familiar with the MPLAB X IDE. Then, when you are ready to begin, click **Quick Start** and follow the instructions in the Quick Start document for detailed step-by-step instructions on how to use the New Project Wizard to create a new MPLAB X IDE project.



If you are already an experienced MPLAB X IDE user, you can create the new project the same way you normally would, by selecting *File > New Project* or by clicking the **New Project** icon ( ) in the tool bar.

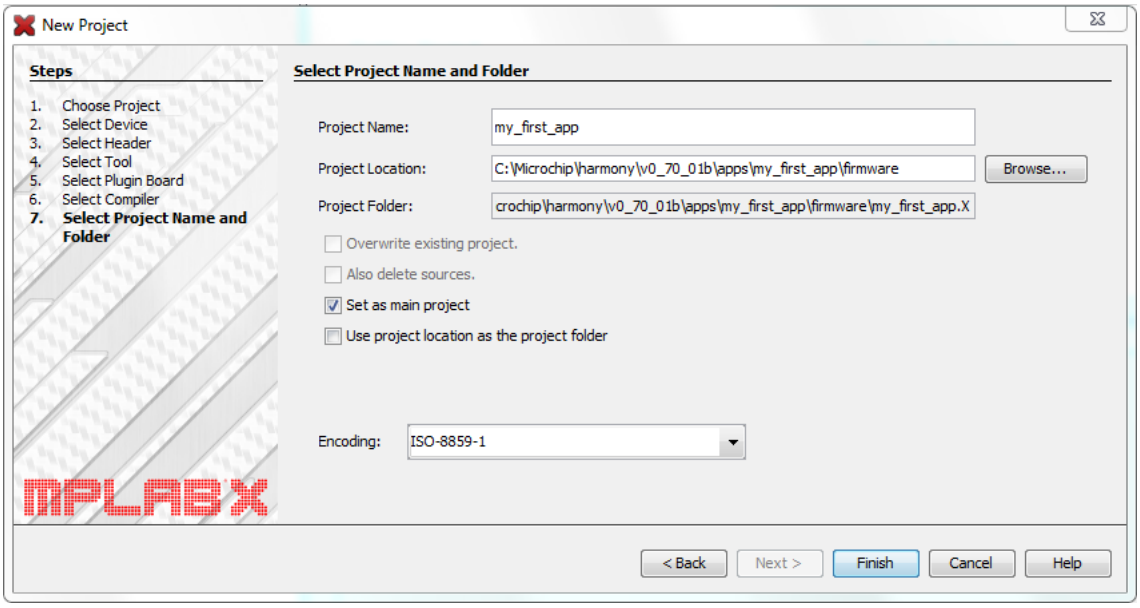
However you launch it, the New Project wizard's first dialog window (shown in the following figure) will list several categories of projects. Select the **Microchip Embedded** category if it is not already selected. Then, from the Projects list, select **Standalone Project**, and then click **Next**.



Note: If you have already installed the MPLAB Harmony Configurator plug-in tool into the MPLAB X IDE, you may see an "MPLAB Harmony" project type. That project type is the one you should select if you want to use the MPLAB Harmony Configurator wizard. However, for the purposes of this tutorial, you will create your MPLAB Harmony project manually instead of using the configurator. Therefore, you should select the "Standalone Project" type instead of the "MPLAB Harmony" project type.

Follow the remaining steps presented to complete the New Project wizard and create a new MPLAB X IDE project. Be sure to select the appropriate PIC32 device, debugger tool, and the MPLAB XC32 C/C++ Compiler for your development platform.

As shown in the next figure, as you finish the New Project wizard, you may also want to select **Set as main project** and clear **Use project location** as the project folder to simplify working with the new project and to keep the project organization similar to the organization of the demonstration and example applications.

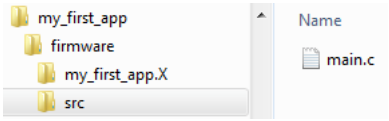


When you finish the MPLAB X IDE New Project wizard, you will have an empty MPLAB X IDE project with no source files in it. In the next steps, you will create the basic set of source files that make up a MPLAB Harmony project.

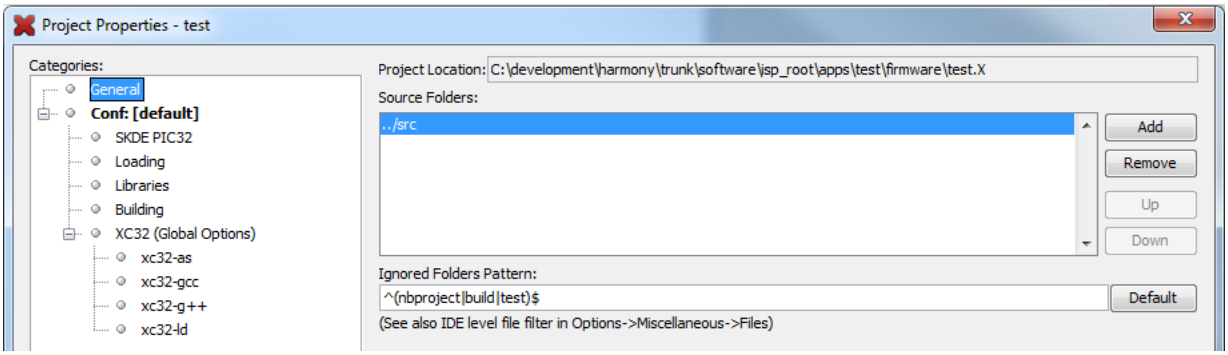
Create the Main File and Function

MPLAB Harmony applications usually follow a consistent organization for their project folders and source files. This convention makes it easy to isolate your application's source code from the "system configuration" code necessary to configure, initialize, and maintain the MPLAB Harmony "system" for a specific hardware platform or end product.

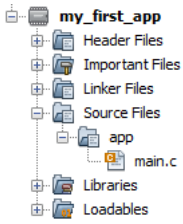
Create a new sub-folder named `src` within your new project's firmware folder and create a `main.c` file in the `src` folder.



You will need to create the folders using the operating system (with the exception of the `my_first_app.X` folder). But, once you have done so, you can add the relative path to the `src` folder to the MPLAB X IDE project properties so that you will be able to create the individual files from directly within MPLAB X IDE from that point forward. To do this, once you have created the project, Right-click the project name and select **Properties**. Select the **General** category in the Project Properties dialog window, as shown in the following figure, and add `../src` to Source Folders, and then click **OK**.



Then, create a new logical folder in the MPLAB X IDE project, rename it to `app`. The app logical folder will correspond to the `src` folder on disk. Add a new source file named `main.c` to it, as follows:



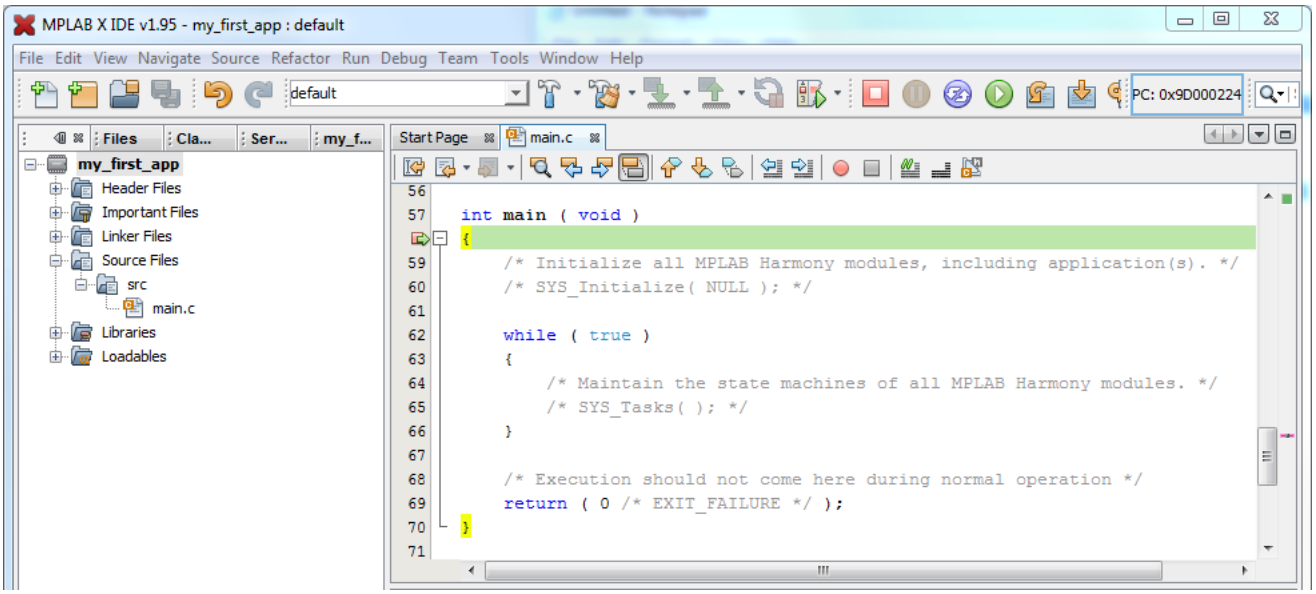
Next, implement the C-language “main” function so that it appears like the following example:

```
#include <stdbool.h>
#include <stdlib.h>
#include "system/common/sys_module.h"

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

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

    /* Execution should not come here during normal operation */
    return ( EXIT_FAILURE );
}
```

Notes:

1. You can copy starter files, with helpful "TODO" comments where you need to add your own code from the template project provided in the `<install-dir>/apps/examples/templates/basic_pic32` folder. However, please be aware that this project provides a complete set of "starter" files for creating new MPLAB Harmony projects. These additional items will be explained later in this tutorial; therefore, it is best to create your first application yourself until you have completed the tutorial.
2. The `stdbool.h` file is included to obtain the definition of “true” and the `stdlib.h` file is included so that we have a definition of “EXIT_FAILURE”. These are both extended C-language headers that are provided in the compiler installation and are already in the compiler’s default include file search path. The `sys_module.h` file contains prototypes for the SYS_Initialize and SYS_Tasks functions.

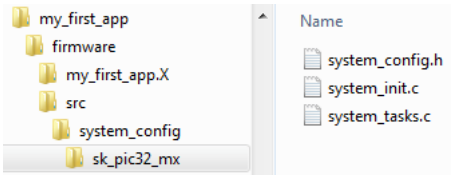
At this point, it is a good idea to verify that your compiler, debugger, and hardware platform are all installed correctly and working. To do this, “comment out” the calls to the SYS_Initialize and the SYS_Tasks functions and build the project. Then, set a breakpoint on the first line of the “main” function, build and program it to your hardware, and debug it to see if you hit the breakpoint. (You should see the IDE indicate when you have reached the breakpoint, as shown in the following figure.)



If you can debug and step through this simple “main” function, this indicates that your development environment and hardware platform are set up and working properly. If you cannot, please refer to the MPLAB X IDE help and the documentation for your development platform and resolve any setup issues before continuing.

Create System Configuration Files

After you have implemented the “main” function, you need to create the system configuration files to implement the SYS_Initialize and SYS_Tasks functions and define any build-time configuration options that may be required. By convention, these files are placed their own sub-folder that should be named in such a way that briefly describes the purpose of the configuration (for example, `sk_pic32_mx` for a PIC32MX Starter Kit configuration). Since a MPLAB Harmony application can have multiple configurations, your project could potentially have more than one such folder (each containing its own set of configuration files). To keep these files and folders organized, MPLAB Harmony application projects place the configuration-specific folders in a sub-folder of the `src` folder named `system_config`, as follows:



Create a folder hierarchy similar to the one previously shown, with an appropriate configuration name for the hardware platform you are using.

Once you have created your system configuration folder, add the following files to it.

Code: system_config.h

```
#ifndef _SYSTEM_CONFIG_H
#define _SYSTEM_CONFIG_H

/* TODO:  Define build-time Configuration Options. */

#endif /* _SYSTEM_CONFIG_H */
```

You will use the `system_config.h` file to define any necessary build-time configuration options for your MPLAB Harmony application. Most MPLAB Harmony libraries support or require some set of build-time configuration options that allow you to define various parameters such as buffer sizes and various minimums and maximums or to clients or select certain optional features like interrupt support. To obtain the definitions of these configuration options, the source files for all MPLAB Harmony libraries that support such options include the `system_config.h` file. This means that every MPLAB Harmony configuration must define this file.

Note: The `system_config.h` file must contain only `#define` macros. It must not define data types or prototypes or include

any files that do so. This is necessary to avoid potential include file loops that might cause build issues.

Code: system_init.c

```
/* TODO:  Define processor configuration bits */

/* Initialize the System */
void SYS_Initialize ( void *data )
{
    /* TODO:  Initialize all modules and the application. */
}
```

The `system_init.c` file is where you should define the SYS_Initialize function to initialize your MPLAB Harmony system. This file is also where you should define the necessary processor Configuration bits and any initialization data required by the MPLAB Harmony libraries you decide to use. Every MPLAB Harmony library module implements its own “Initialize” function and the purpose of the SYS_Initialize function is to call the initialization functions of any libraries used in the system as well as the initialization function of the application. You will add this code when you start adding libraries to the system.

PIC32MX Configuration Bits Code Example

Refer to the MPLAB XC32 C/C++ Compiler documentation for a list of the required `#pragma config` definitions supported by the compiler, and refer to the specific PIC32 device data sheet for a description of what each processor Configuration bit does.

For this tutorial example, the Configuration bit definitions provided in the following code example work for a PIC32MX795F512L device on a PIC32 USB Starter Kit II.

```
/* Processor Configuration bits */

// DEVCFG3
#pragma config FSRSSEL = PRIORITY_7     // SRS Select
#pragma config FUSBIDIO = ON            // USB USID Selection
#pragma config FVBUSONIO = ON           // USB VBUS ON Selection

// DEVCFG2
#pragma config FPLLIDIV = DIV_2         // PLL Input Divider
#pragma config FPLLMUL = MUL_20         // PLL Multiplier
#pragma config UPLLIDIV = DIV_2         // USB PLL Input Divider
#pragma config UPLLEN = OFF             // USB PLL Enable
#pragma config FPLLODIV = DIV_256       // System PLL Output Clock Divider

// DEVCFG1
#pragma config FNOSC = PRIPLL           // Oscillator Selection Bits
#pragma config FSOSCEN = OFF            // Secondary Oscillator Enable
#pragma config IESO = ON                // Internal/External Switch Over
#pragma config POSCMOD = OFF            // Primary Oscillator Configuration
#pragma config OSCIOFNC = OFF           // CLKO Output Signal Active on the OSCO Pin
#pragma config FPBDIV = DIV_1           // Peripheral Clock Divisor
#pragma config FCKSM = CSDCMD           // Clock Switching and Monitor Selection
#pragma config WDTPS = PS1048576        // Watchdog Timer Postscaler (1048576:1)
#pragma config FWDTEN = OFF             // Watchdog Timer Enable

// DEVCFG0
#pragma config DEBUG = OFF              // Background Debugger Enable
#pragma config ICESEL = ICS_PGx2        // ICE/ICD Communication Channel Select
#pragma config PWP = OFF                // Program Flash Write Protect
#pragma config BWP = OFF                // Boot Flash Write Protect bit
#pragma config CP = OFF                 // Code Protect
```

PIC32MZ Configuration Bits Code Example

The following code example shows the Configuration bit settings for a PIC32MZ2048ECH144 device on a PIC32MZ Embedded Connectivity (EC) Starter Kit.

```
/* Set up the Device Configuration */
// DEVCFG3
// USERID = No Setting
#pragma config FMIIEN   = OFF           // Ethernet RMII/MII Enable (MII Enabled)
#pragma config FETHIO   = ON            // Ethernet I/O Pin Select (Default Ethernet I/O)
#pragma config PGL1WAY  = OFF           // Permission Group Lock One Way Configuration (Allow
```

```
only one reconfiguration)
#pragma config PMDL1WAY = OFF           // Peripheral Module Disable Configuration (Allow only
one reconfiguration)
#pragma config IOL1WAY  = OFF           // Peripheral Pin Select Configuration (Allow only one
reconfiguration)
#pragma config FUSBIDIO = OFF           // USB USBID Selection (Controlled by Port Function)

// DEVCFG2
#pragma config FPLLIDIV = DIV_2         //3 System PLL Input Divider (1x Divider)
#pragma config FPLLRNG  = RANGE_8_16_MHZ // System PLL Input Range (5-10 MHz Input)
#pragma config FPLLICLK = PLL_POSC      // System PLL Input Clock Selection (POSC is input to
the System PLL)
#pragma config FPLLMULT = MUL_33        //50 System PLL Multiplier (PLL Multiply by 50)
#pragma config FPLLODIV = DIV_2
#pragma config UPLLFSEL = FREQ_24MHZ    // USB PLL Input Frequency Selection (USB PLL input is
12 MHz)
#pragma config UPLLEN   = ON            // USB PLL Enable (USB PLL is enabled)

// DEVCFG1
#pragma config FNOSC    = SPLL          // Oscillator Selection Bits (SPLL)
#pragma config DMTINTV  = WIN_127_128   //  DMT Count Window Interval (Window/Interval value
is 127/128 counter value)
#pragma config FSOSCEN  = OFF           // Secondary Oscillator Enable (Disable SOSC)
#pragma config IESO     = OFF           // Internal/External Switch Over (Disabled)
#pragma config POSCMOD  = EC            // Primary Oscillator Configuration (Primary
Oscillator enabled)
#pragma config OSCIOFNC = OFF           // CLKO Output Signal Active on the OSCO Pin (Disabled)
#pragma config FCKSM    = CSDCMD        // Clock Switching and Monitor Selection (Clock Switch
Enabled, FSCM Enabled)
#pragma config WDTPS    = PS1048576     // Watchdog Timer Postscaler (1:1048576)
#pragma config WDTSPGM  = STOP          // Watchdog Timer Stop During Flash Programming (WDT
stops during Flash programming)
#pragma config WINDIS   = NORMAL        // Watchdog Timer Window Mode (Watchdog Timer is in
non-Window mode)
#pragma config FWDTEN   = OFF           // Watchdog Timer Enable (WDT Disabled)
#pragma config FWDTWINSZ = WINSZ_25     // Watchdog Timer Window Size (Window size is 25%)
// DMTCNT = No Setting
#pragma config FDMTEN   = OFF           // Deadman Timer Enable (Deadman Timer is disabled)

/* DEVCFG0 */
#pragma config EJTAGBEN = NORMAL
#pragma config DBGPER   = PG_ALL
#pragma config FSLEEP   = OFF
#pragma config FECCCON  = OFF_UNLOCKED
#pragma config BOOTISA  = MIPS32
#pragma config TRCEN    = OFF
#pragma config ICESEL   = ICS_PGx2
#pragma config JTAGEN   = OFF
#pragma config DEBUG    = ON

// DEVCP0
#pragma config CP = OFF                 // Code Protect (Protection Disabled)
```

Hint: To get started quickly, you can look at the `system_init.c` files from any of the MPLAB Harmony example or demonstration applications that use the same processor as the one you are using in your development platform. Copy the Configuration bit definitions from one of the supplied demonstration configurations and paste them into your own `system_init.c` file in place of the `TODO: Define processor configuration bits` comment shown in the previous `system_init.c` code example and make any modifications you require.

Code: system_tasks.c

```
void SYS_Tasks ( void )
{
    /* TODO:  Call the state machines of all modules and the application. */
}
```

The `system_tasks.c` file is where MPLAB Harmony projects implement the SYS_Tasks function. In addition to a

SYS_Initialize function, almost every MPLAB Harmony library module implements one or more “Tasks” functions. A module’s tasks functions implement the state machines that perform the necessary tasks for normal operation. The purpose of the system-wide SYS_Tasks function is to call every “Tasks” function for every library and application module that executes in a polled configuration. As observed in the `main.c` file, the SYS_Tasks function is called from the top-level “super” loop (in a system that does not use an RTOS) so it will ensure that the state machines of all libraries and applications keep running.

Code: system_interrupt.c

```
#include <xc.h>
#include <sys/attribs.h>
#include “system_definitions.h”

void __ISR( _TIMER_2_VECTOR ) _InterruptHandler_TMR_2_stub( void )
{
    /* Call the timer driver's "Tasks" routine */
    DRV_TMR_Tasks( sysObject.tmrDrv );

}
```

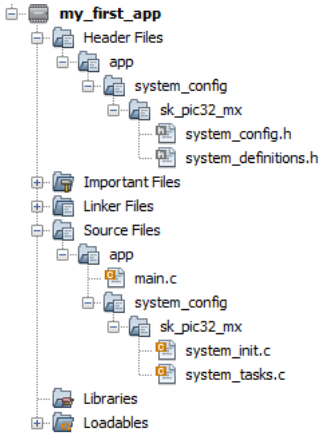
You won’t need to do it for this tutorial, but if you planned to configure some of your MPLAB Harmony modules to run interrupt-driven (instead of polled), you would need to add a `system_interrupt.c` file in which you would need to implement the interrupt vector functions. These are special functions that are called (or rather branched to) by the processor when the associated interrupt occurs. From each interrupt vector function, you would then need to call the appropriate “Tasks” function for the driver or system service library module that supports that particular interrupt and that library would need to be configured to be run as interrupt-driven. The following example shows what this might look like for the Timer2 vector on a PIC32MX device, using the timer driver. But, again, this is not necessary for this tutorial, as it runs in a polled configuration.

Code: system_definitions.h

```
#ifndef _SYSTEM_DEFINITIONS_H
#define _SYSTEM_DEFINITIONS_H

#include "system/common/sys_module.h"

#endif /* _SYSTEM_DEFINITIONS_H */
```

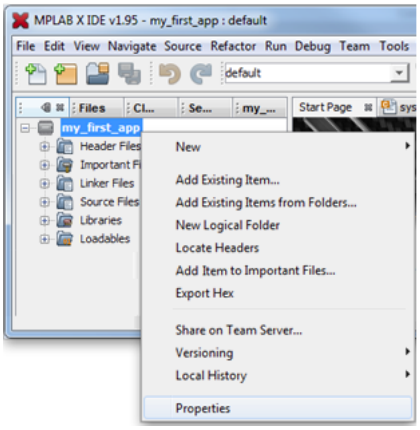
Later, you will add data type definitions to the `system_definitions.h` file that are required by the other system files. However, for now all you need to do is include the `sys_module.h` header file (which provides the prototypes for the SYS_Initialize and SYS_Tasks functions).

Add these new system configuration files to the MPLAB X IDE project and place them in the appropriate “logical” folders to keep the project organized. The end result should be similar to the following figure.

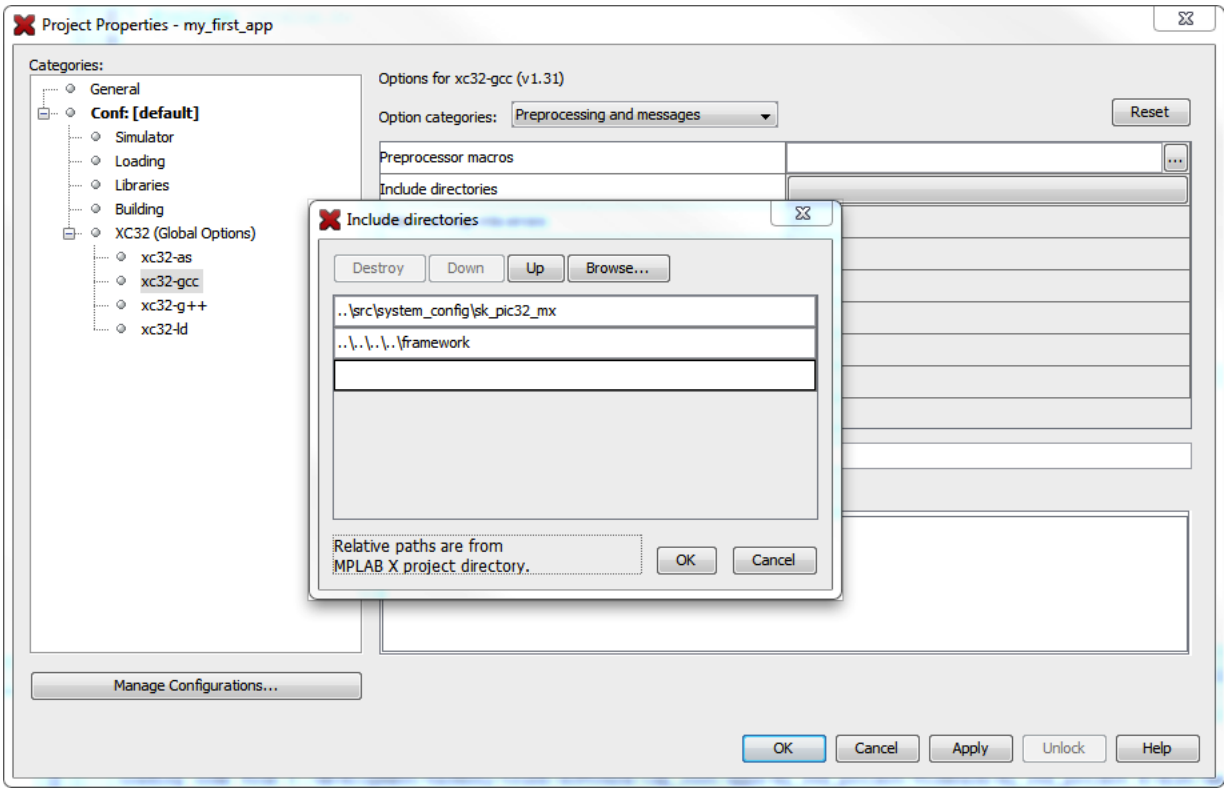


Notice that, with the exception of the `app` folder, which actually corresponds to the physical disk `src` folder, the logical folders in the project should match the physical disk folders to avoid confusion.

Then, add the system configuration folder (`../src/system_config/sk_pic32_mx`, shown in the previous examples) and the `<install-dir>/framework` folder to the include file search paths that are passed to the compiler by the MPLAB X IDE so that the libraries and other files can find the `system_config.h` header file. To do this, right-click the project name in the project pane and select **Properties**, as shown in the following figure.



As shown in the following figure, in the Project Properties dialog, select the **xc32-gcc** compiler. From the Option categories pull-down menu, select **Preprocessing and messages**. Then, click the "…" button next to the "Include directories" item. This opens a dialog where you can add the relative path to your configuration folder.



This is another good checkpoint. You should now be able to build the project and use the debugger to step through the basic outline of a MPLAB Harmony system. If you cannot step through each of the functions you have just defined, you should take time now and fix any build issues or bugs.

1.4 Step 2: Identify the Required Library Modules

This topic describes how to use the documentation to identify which library modules are required.

Description

Once you have a working MPLAB X IDE project and the basic set of files in which to configure, initialize, and maintain a MPLAB Harmony “system”, you are ready to start adding library modules and building up your system so you can develop your application. To do this, you need to find out what libraries MPLAB Harmony provides and decide which ones you want to use. If you are not already familiar with the contents of the MPLAB Harmony installation, now is a good time to take a look at a few key sections of the help documentation.

Key MPLAB Harmony Library Documentation

- **Release Contents**
- **Framework Help**, which includes:
  - **Driver Library Help**
  - **System Service Library Help**
  - **Peripheral Library Help**
- Other middleware help sections such as:
  - **TCP/IP Stack Library Help**
  - **USB Library Help**
  - **Graphics Library Help**

Release Contents

The *Release Contents* section provides tables that list every library, as well as demonstrations, utilities, and other items, that are contained in the MPLAB Harmony installation. This section is the best place to start when you need to get a quick look at what MPLAB Harmony provides for you. The demonstrations and examples provided are great resources to see what you can do with MPLAB Harmony. However, the descriptions provided in the *Release Contents* section are purposely very brief so that they fit in a convenient table format. So, you may want to review the documentation provided with each application for more information.

Framework Help

Detailed help documentation for all MPLAB Harmony libraries is provided under the *Framework Help* section. The help documentation for each individual library has a *Library Overview* topic that will give you a brief description of what the library does. You should read this topic for any library for which the brief description in the *Release Contents* table did not give you enough information to decide if you should use the library or not. If you need even more information, read the *Using the Library* topic and look at the summary tables in the *Library Interface* topic for a look at the library’s Application Program Interface (API) functions and data types. This section makes a great “programmers reference” while you are developing your application.

Directly beneath the *Framework Help* section, the documentation for libraries such as Graphics, USB, and TCP/IP, among others, stands out at the top level. Look at those libraries if you are interested in using those technologies. Other libraries are grouped together by layer. Of particular interest, while you are putting together the building blocks of your system, are the Driver, System Service, and Peripheral Library groups. Drivers usually provide the highest level of abstraction for interacting with a particular peripheral, unless use of that peripheral requires complex protocol support such as USB communication, TCP/IP networking, or graphics drawing to be effective. In these cases, the easiest way to use the associated peripheral is through the appropriate middleware library.

System Service Library Help

Some peripherals, such as general-purpose I/O ports and the Interrupt Controller, and some libraries, such as the file system library, support system-wide resources that are used by many other modules and applications. These resources are made available through MPLAB Harmony system services. Take a look at the overviews of the system service libraries in the *System Service Library Help* group. System service libraries eliminate conflicts between MPLAB Harmony drivers and middleware and they will reduce the amount of work you have to do to write your application.

Peripheral Library Help

If no middleware library, device driver, or system service provides the peripheral access or capabilities you require, the Peripheral Libraries (PLIBs) expose all features of all peripherals in the microcontroller (with very few exceptions). So, if necessary, you can directly interact with a PLIB to implement the functionality you desire. However, keep in mind that the PLIB

interface is a very low-level abstraction layer, requiring you to manage the details of the peripheral's operation and state machine. Also, PLIBs do not provide any sort of protection from interference between multiple modules in the system. Therefore, any module that directly accesses a PLIB must have complete ownership of that peripheral. No other module in the system should interact with that peripheral. Normally, that sort of protection is implemented by a device driver or system service.

Driver Library Help

Since MPLAB Harmony is a layered stack of cooperating software modules, some modules are dependent on other modules in the stack. In general, a device driver for a particular peripheral is almost always dependent on the peripheral library for that same peripheral. Device drivers are also usually dependent on certain system services (such as the ports and interrupt system services) and some drivers may even be dependent on other drivers. Also, system services can be dependent upon peripheral libraries, if they directly control one or more peripherals, or upon a device driver if they provide higher-level services such as software timers or console I/O. Almost all middleware will be dependent upon one or more device drivers and system services to interact with the hardware indirectly. These dependencies are described in the *Configuring the Library* and *Building the Library* topics (if applicable) of the help documentation for each library.

Next Steps

Once you have decided which libraries you want to use as the building blocks of your application, you can begin "stacking" them up to build your MPLAB Harmony system.

For the application implemented in this tutorial, you will need the following MPLAB Harmony libraries:

- Device Control System Service (dependent upon the Device Control Peripheral Library)
- **Ports System Service** (dependent upon the **Ports Peripheral Library**)
- **Timer System Service** (dependent upon the **Timer Driver Library**)
- **Clock System Service** (queue implementation in `system/common`)
- **Timer Driver Library** (dependent upon the **Timer Peripheral Library**)
- Peripheral Library

1.5 Step 3: Add the Library Modules to the Project

This topic describes how to add MPLAB Harmony library modules to your MPLAB X IDE project.

Description

In most cases, MPLAB Harmony libraries are provided in source form and you will need to add that source code to your application's MPLAB X IDE project to use the libraries. Source code files for MPLAB Harmony libraries are not normally copied into your application folders. They are usually left in place in the installation and added by relative path to your MPLAB X IDE project. Certainly, you can copy the source files to your application's folders if you so desire, but for purposes of this tutorial this is not necessary.

Note: If desired, you could create a separate MPLAB X IDE project (or subproject) to prebuild each library into its own binary (`.a`) file. Or, you could aggregate the individual libraries together into a single binary (`.a`) file or into any desired combination of prebuilt library files. Regardless of how you include the implementation of each library (the files under the library's `src` folder) in your project, you will need to utilize the interface headers (the `.h` files) that are provided in the library's root folder (the parent of the `src` folder).

Project Source Files

To identify which source files need to be added to your project for each library, refer to the *Building the Library* topic in the *Configuring the Library* section for each library you identified in Step 2: Identify the Required Library Modules. This help topic describes the purpose of each source file in the library's implementation. Do not attempt to just include every source file listed for a specific library in your project. Separate library features may be implemented in separate source files and you do not need to include the implementations of features that you have no intention of using in your project. Also, some features (and some entire libraries) have alternate implementations in different source files. These files implement the same functions differently for different configurations and are mutually exclusive and cannot be included in the same build project. Therefore, you must review

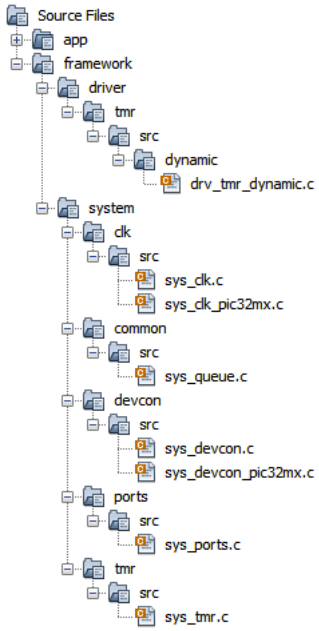
this topic to identify which files implement the features you want for the configuration you desire and add them to the MPLAB X IDE project for your MPLAB Harmony application.

For the example project in this tutorial, you should add the following source files to the project.

Tutorial Example Source (.c) Files:

- <install-dir>/framework/driver/tmr/src/dynamic/drv_tmr_dynamic.c
- <install-dir>/framework/system/clk/src/sys_clk.c
- <install-dir>/framework/system/clk/src/sys_clk_pic32mx.c
- <install-dir>/framework/system/common/src/sys_queue.c
- <install-dir>/framework/system/devcon/src/sys_devcon.c
- <install-dir>/framework/system/devcon/src/sys_devcon_pic32mx.c
- <install-dir>/framework/system/ports/src/sys_ports.c
- <install-dir>/framework/system/tmr/src/sys_tmr.c=

As mentioned previously, the logical folders in your MPLAB X IDE project should match the physical disk folders to avoid any confusion. Therefore, first add a top-level framework folder to the “Source Files” group in your MPLAB X IDE project. Then, add the necessary logical folders to match the directory hierarchy on disk and add the files to your projects. The end result should look like the following figure.



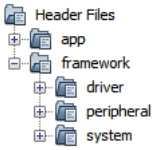
For easy reference as you are developing, you should also include the associated header (`.h`) files, although the project will build without explicitly including them in the project as long as the compiler include file search paths are correct (which we will cover shortly).

Tutorial Example Header (.h) Files:

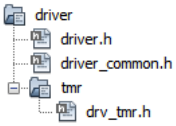
- `<install-dir>/framework/driver/driver.h`
- `<install-dir>/framework/driver/driver_common.h`
- `<install-dir>/framework/driver/tmr/drv_tmr.h`
- `<install-dir>/framework/peripheral/periheral.h`
- `<install-dir>/framework/peripheral/peripheral_common.h`

- `<install-dir>/framework/peripheral/peripheral_common_32bit.h`
- `<install-dir>/framework/peripheral/devcon/plib_devcon.h`
- `<install-dir>/framework/peripheral/ports/plib_ports.h`
- `<install-dir>/framework/peripheral/tmr/plib_tmr.h`
- `<install-dir>/framework/system/system.h`
- `<install-dir>/framework/system/clk/sys_clk.h`
- `<install-dir>/framework/system/common/sys_queue.h`
- `<install-dir>/framework/system/devcon/sys_devcon.h`
- `<install-dir>/framework/system/ports/sys_ports.h`
- `<install-dir>/framework/system/tmr/sys_tmr.h`

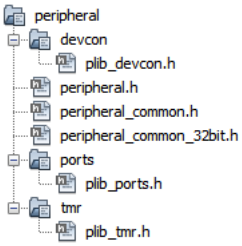
At the top level, you will have logical three logical header file folders within the framework folder, as follows:



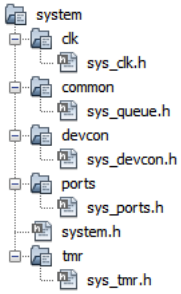
Within the `driver` header folder you will have the following:



Within the `peripheral` header folder, you will have the following:



And, within the `system` header folder, you will have the following:



Relative Path

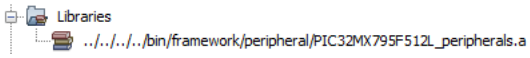
Ensure that you have added the relative path from your project folder to the MPLAB Harmony installation's framework folder to the compiler's include file search path (as you did previously in Step 1: Create the New Project for the configuration folder) so that the compiler can find all necessary header files for the MPLAB Harmony libraries. All MPLAB Harmony source files assume this folder is in the search path.

Inline Functions

MPLAB Harmony PLIBs are primarily implemented as inline functions. These inline functions are defined in headers that are included whenever a source file includes `peripheral.h` (or a specific peripheral library's interface header). You do not need to explicitly add these definition files, which are located within the processor and templates sub-folders in each specific peripheral library's folder `<install-dir>/framework/peripheral/<peripheral>` to your project, because there are a large number of them (at least one for every supported processor, plus several more). However, you do need to know that the PLIBs are also provided as a prebuilt binary (`.a`) file for each PIC32 device.

Peripheral Library Binary Files

The inclusion of the appropriate PLIB binary (`.a`) library in your project is necessary because the compiler may decide to generate an actual function call instead of generating the PLIB function's object code inline with the calling code. If this happens, the function's implementation must be available somewhere in object form so that the linker can resolve the call. To provide the function implementation, the part-specific peripheral library binary (`.a`) file needs to be added to the project within the top-level Libraries folder. These prebuilt binary peripheral library files are available within the `bin` folder in `<install-dir>/bin/framework/peripheral`. The file used for the tutorial example is shown in the next figure. For your project, add the file that contains the name of the processor that you are using.



If you attempt to build the project at this point, you will observe that it now produces several compilation errors. These errors occur because some of the libraries you have just added to the project require build-time configuration options that need to be defined before the libraries can be built. We will cover how to define those options in Step 4: Configure the Modules.

1.6 Step 4: Configure the Modules

This topic describes how to configure the MPLAB Harmony library modules.

Description

Most MPLAB Harmony libraries have some number of optional and/or required build-time configuration parameters. Optional configuration parameters have default values that are usually acceptable and thus may not need to be defined to successfully build the library. Required parameters do not have default definitions and must be defined to build the library. These items are considered critical (such as the the choice of running interrupt-driven or polled) and are made required to bring them to your attention and ensure that you make a selection.

To identify what configuration options a library supports and requires, refer to the *Configuring the Library* topic of the help documentation for the library in question. This section lists all of the library's build-time configuration options, explains the purpose of each, and describes the allowable values they can be given. Once the appropriate value has been chosen, a C-language preprocessor macro definition must be placed in the `system_config.h` header (or in a header included by `system_config.h`) and that header must be made available in the compiler's include file search path (as done previously in Step 3: Add the Library Modules to the Project).

Note: You can also look at the `<library>_config_template.h` header in the library's `config` folder (next to its `src` folder). This file lists all possible configuration options for the library. However, this file is strictly provided for documentation. Do not include the configuration template file directly in your application, as it provides example definitions of all configurations, some of which may conflict with each other.

For the tutorial example project, add the following definitions to the `system_config.h` file for now. (You will add more later.)

Code: system_config.h

```
#ifndef _SYSTEM_CONFIG_H
#define _SYSTEM_CONFIG_H

/* Prevent superfluous PLIB warnings. */
#define _PLIB_UNSUPPORTED

/* TMR Driver Build Options */
```

```
#define DRV_TMR_INSTANCES_NUMBER            1
#define DRV_TMR_CLIENTS_NUMBER              1
#define DRV_TMR_INTERRUPT_MODE              false
#define DRV_TMR_COUNT_WIDTH                 16

/* TMR System Service Build Options */
#define SYS_TMR_MAX_PERIODIC_EVENTS         1
#define SYS_TMR_MAX_DELAY_EVENTS            1

/* CLK System Service Configuration */
#define SYS_CLK_PRIMARY_CLOCK               (80000000ul)    // 80 MHz
#define SYS_CLK_CONFIG_PRIMARY_XTAL         (8000000ul)     // 8 MHz
#define SYS_CLK_CONFIG_SECONDARY_XTAL       (0ul)           // Unused
#define SYS_CLK_CONFIG_SYSPLL_INP_DIVISOR   2
#define SYS_CLK_CONFIGBIT_USBPLL_ENABLE     false
#define SYS_CLK_CONFIGBIT_USBPLL_DIVISOR    2
#define SYS_CLK_CONFIG_FREQ_ERROR_LIMIT     10

/* DEVCON System Service Configuration */
#define SYS_DEVCON_PIC32MX_MAX_PB_FREQ      SYS_CLK_PRIMARY_CLOCK

#endif /* _SYSTEM_CONFIG_H */
```

Notes:

1. Ensure that the clock system service configuration definitions match with the configuration bit definitions provided in the `system_init.c` file.
2. The PLIB build option defined above is a temporary measure, necessary to avoid superfluous warnings. The necessity for this definition will be removed in future releases.
3. For additional information on these configuration options, please refer to the help documentation as previously described.

Once you have defined these configuration options, you should be able to successfully build the project again. In Step 5: Initialize the System and Modules, you will initialize these modules.

1.7 Step 5: Initialize the System and Modules

This topic describes how to initialize the library modules.

Description

Now that you have selected the desired library modules, added the required source files to your project, and defined the necessary build-time configuration options, you need to initialize each “module” so that its state machine is placed in the correct initial state.). To understand how to initialize each library module, refer to the Help documentation for the library’s “Initialize” function and to any initialization examples given in the *Using the Library* help topic. (You can also look at example applications that use that library.)

For this tutorial, we previously identified that the ports and **Timer System Service Library** as well as the **Timer Driver Library** and the **Timer Peripheral Library** are necessary for this application. And, the Device Control System Service Library and Clock System Service Library must be properly initialized to ensure the processor runs at its maximum performance level. The Peripheral Library and the **Ports System Service Library** do not have their own active state machines, and therefore, they require no initialization. However, the Timer Driver and the Timer System Service that depends upon it do require initialization and this section will describe how to do that.

Note: In MPLAB Harmony, almost every library and application (with the notable exception of the peripheral libraries) is considered a system “module”. This normally means that it is an active element in the system that must be initialized (by calling its initialization function once at system startup and passing in the appropriate data) and that it may have one or more “Tasks” functions that must be called from a polling loop or an Interrupt Service Routine (ISR) to maintain its internal state machine.

The first thing you will normally want to do is ensure that your system is running at the desired clock rate and at the desired performance level. To do this, update your `system_definitions.h` and `system_init.c` files to initialize the clock and device control system services, as shown in the following three code examples.

Code: system_definitions.h – Clock and Device Control Headers

```
#ifndef _SYSTEM_DEFINITIONS_H
#define _SYSTEM_DEFINITIONS_H

#include "system/system.h"
#include "system/clk/sys_clk.h"
#include "system/devcon/sys_devcon.h"
#include "system/common/sys_module.h"


#endif /* _SYSTEM_DEFINITIONS_H */
```

Code: system_init.c – Included Files

```
#include <stdlib.h>
#include "system_config.h"
#include "system_definitions.h"
```

Code: system_init.c – System Initialization

```
/* Device Conrol Service Initialization Data */
const SYS_DEVCON_INIT devconInit =
{
    .moduleInit = {0},
};


/* Initialize the System */
void SYS_Initialize ( void *data )
{
    /* Set up the clock, cache and wait states for maximum performance. */
    SYS_CLK_Initialize(NULL);
    SYS_DEVCON_Initialize(SYS_DEVCON_INDEX_0, (SYS_MODULE_INIT *)&devconInit);
    SYS_DEVCON_PerformanceConfig(SYS_CLK_PRIMARY_CLOCK);

}
```

The call to SYS_CLK_Initialize (with a NULL parameter) will initialize the clock system so that it utilizes the oscillator settings provided in the processor configuration bits. After initializing the device control system service, calling the SYS_DEVCON_PerformanceConfig function will ensure that the Flash wait states and cache controller settings are set up for maximum performance at the given clock rate. Also, be sure to include the `system_definitions.h` header file in the `system_init.c`, after the inclusion of `system_config.h`.

You will also need to define and initialize an instance of a DRV_TMR_INIT structure to initialize the Timer Driver. Define this structure in the `system_init.c` file as “const” global data (so that it is located in a program Flash memory range by the linker). Then, you will need to call the DRV_TMR_Initialize function and pass it a pointer to the “init” structure, as shown in the following two code examples.

Code: system_init.c - Defining the Timer Driver Init Data

```
/* Timer Driver Initialization Data */
const DRV_TMR_INIT initDrvTmr =
{
    /* Standard Module Initialization */
    .moduleInit         = {SYS_MODULE_POWER_RUN_FULL},

    /* Identifies timer hardware module (PLIB-level) ID */
    .tmrId              = APP_TMR_ID,

    /* Clock Source select enumeration */
    .clockSource        = APP_TMR_DRV_CLOCK_SOURCE,

    /* Time Period */
    .timerPeriod = APP_TMR_DRV_PERIOD,

    /* Prescaler Selection */
    .prescale           = APP_TMR_DRV_PRESCALE,
```

```
    /* Source Edge Selection */
    .sourceEdge         = TMR_CLOCK_SOURCE_EDGE_NONE,

    /* Post Scale Selection */
    .postscale          = 0,

    /* Timer Sync Mode */
    . syncMode          = DRV_TMR_SYNC_MODE_SYNCHRONOUS_INTERNAL,

    /* Do not combine 2 16-bit timers into a single 32-bit timer */
    .combineTimers      = false,

    /* Interrupt Source */
    .interruptSource    = APP_TMR_INT_SOURCE
};
```

Code: system_init.c - Initializing the Timer Driver

```
/* Initialize the System */
void SYS_Initialize ( void *data )
{
    /* Set up the clock, cache and wait states for maximum performance. */
    SYS_CLK_Initialize(NULL);
    SYS_DEVCON_Initialize(SYS_DEVCON_INDEX_0, (SYS_MODULE_INIT *)&devconInit);
    SYS_DEVCON_PerformanceConfig(SYS_CLK_PRIMARY_CLOCK);

    /* Initialize all modules and the application. */
    sysObjects.TimerDriver = DRV_TMR_Initialize(APP_TMR_DRV_INDEX,
                                                (SYS_MODULE_INIT*)&initDrvTmr);
}
```

The values used to initialize this structure can be numeric literals, but it is better to use constants defined in the `system_config.h` header or enumeration constants defined by other libraries and the driver itself. Constants defined strictly for the application should usually start with an “APP_” prefix to easily identify them. Constants defined by a peripheral library begin with the peripheral module's abbreviation (“TMR_” in this example) and constants defined by system services or device drivers start with the prefix used by that layer (“SYS_” and “DRV_”, respectively).

Add the following application configuration definitions to the `system_config.h` header file.

Code: system_config.h - Application Configuration Options

```
/* Timer driver 0 uses Timer peripheral 2 */
#define APP_TMR_DRV_INDEX                   0
#define APP_TMR_ID                          TMR_ID_2
#define APP_TMR_INT_SOURCE                  INT_SOURCE_TIMER_2

/* 80 MHz / 256 = 312,500 cyc/sec or 625 cyc/2 ms */
#define APP_TMR_DRV_CLOCK_SOURCE            TMR_CLOCK_SOURCE_PERIPHERAL_CLOCK
#define APP_TMR_DRV_PRESCALE                TMR_PRESCALE_VALUE_256
#define APP_TMR_DRV_PERIOD                  (625)

/* Timer system service 0 runs with 2 ms resolution. */
#define APP_TMR_SYS_INDEX                   0
#define APP_TMR_SYS_PERIOD                  2

/* Port service 0, Port D, Bit 1 controls LED 2 */
#define APP_LED_PORTS_ID                    PORTS_ID_0
#define APP_LED_PORT_CHANNEL                PORT_CHANNEL_H
#define APP_LED_PIN                         PORTS_BIT_POS_1

/* App toggles LED once every 1/2 second, giving 1 Hz blink at 50% duty cycle. */
#define APP_TIMER_PERIOD                    500     // ms
```

These definitions are used in the previous Timer "init" structure and in the application itself, which follows.

Once the Timer Driver's "init" structure has been defined, you must pass its address to the DRV_TMR_Initialize function, along with an index value APP_TMR_DRV_INDEX to identify which instance of the driver you are initializing. MPLAB Harmony modules (drivers, services, middleware, and potentially even applications) can support multiple instances of themselves.

Therefore, each module's initialization function must be called once for each instance to put it in its initial state. Module instances are identified by a zero-based index number (think of it as an array index), which is always the first parameter to the module's "Initialize" function.

The "Initialize" function returns a system object "handle". Think of the object handle as a pointer to the module's state data structure instance (which is what it may actually be), but do not try to access any data in that structure directly. This object handle must be passed to all other system interface functions called by the system for that module (particularly the "Tasks" function, as you will see later) to identify the correct module instance being used. The important thing for now is to store and organize them in a convenient way for every module instance in the system so that you have them later when you need them. A good way to do this is to define them in a global data structure in `system_definitions.h` and allocate an instance of that structure in `system_init.c`, as shown in the following code example. (Be sure to include this header in your `system_init.c` file as shown in the previous code example.)

Code: system_definitions.h - The Global System Objects Structure

```
/* System Object Handles Structure */
typedef struct
{
    /* Timer driver object handle */
    SYS_MODULE_OBJ  TimerDriver;

    // More to come later

} SYSTEM_OBJECTS;


/* External forward reference for system objects structure */
extern SYSTEM_OBJECTS sysObjects;
```

Then, declare (allocate) an instance of this structure in the `system_init.c` file (as shown in the following code example), where it will later be initialized by the SYS_Tasks function (as you observed when the DRV_TMR_Initialize function was called).

Code: system_init.c - The Global System Objects Structure

```
/* Global allocation of structure to hold system objects handles */
SYSTEM_OBJECTS sysObjects;
```

Looking at the timer driver's "init" structure, another important thing to notice is that the first member is a SYS_MODULE_INIT structure. This is true for all MPLAB Harmony modules as it allows them to have a common function signature for their "Initialize" functions. That fact will be useful later for adding dynamic initialization and reinitialization of modules. However, it does require typecasting of the "init" data structure pointer in the function call.

One more key thing to notice is that the Timer Driver "init" structure has a member that identifies the specific instance of the peripheral hardware that the driver is to manage. The Timer Driver uses the Timer PLIB, so it has a member of type TMR_MODULE_ID (the second member, initialized with the APP_TMR_ID value). The available timer peripheral hardware module instance IDs for each device are defined by the Timer PLIB. Any driver or other module that uses hardware resources (such as general purpose ID ports) or other modules (such as an I2C or SPI bus driver) will also have members of their "init" structures that identify the specific instance of the resource that it should use.

You also need to define the Timer System Service's "init" structure so you can tell it which instance of the Timer Driver it should use and provide the other initialization parameters that are necessary. To do that, add the SYS TMR "init" structure to the `system_init.c` file and add a call to the SYS_TMR_Initialize function from SYS_Initialize, as shown in the following code example.

Code: system_init.c - Timer System Service "init" Structure

```
/* Timer System Service Initialization Data */
const SYS_TMR_INIT initSysTmr =
{
    /* Standard Module Initialization */
    .moduleInit         = {SYS_MODULE_POWER_RUN_FULL},

    /* Index of the driver module to use */
    .drvIndex           = APP_TMR_DRV_INDEX,

    /* Alarm period in ms (based on driver configuration) */
    .alarmPeriodMs      = APP_TMR_SYS_PERIOD
};
```

Code: system_init.c - Initialize the Timer System Service

```
void SYS_Initialize ( void *data )
{
    /* Set up the clock, cache and wait states for maximum performance. */
    SYS_CLK_Initialize(NULL);
    SYS_DEVCON_Initialize(SYS_DEVCON_INDEX_0, (SYS_MODULE_INIT *)&devconInit);
    SYS_DEVCON_PerformanceConfig(SYS_CLK_PRIMARY_CLOCK);

    /* Initialize all modules and the application. */
    sysObjects.TimerDriver  = DRV_TMR_Initialize(APP_TMR_DRV_INDEX,
                                                 (SYS_MODULE_INIT*)&initDrvTmr);
    sysObjects.TimerService = SYS_TMR_Initialize(APP_TMR_SYS_INDEX,
                                                 (SYS_MODULE_INIT*)&initSysTmr);

    /* Initialize the Application */
    APP_Initialize();
}
```

You will also need to add the "TimerService" member to the "sysObjects" structure to hold the timer system service's object handle and be sure to include any necessary library interface headers in the system_definitions.h header, as shown in the following code example.

Code: system_definitions.h - The Global System Objects Structure

```
#include "system/system.h"
#include "system/common/sys_module.h"
#include "system/clk/sys_clk.h"
#include "system/devcon/sys_devcon.h"
#include "peripheral/tmr/plib_tmr.h"
#include "driver/tmr/drv_tmr.h"
#include "system/tmr/sys_tmr.h"
#include "system/ports/sys_ports.h"
#include "peripheral/peripheral.h"


/* System Object Handles Structure */
typedef struct
{
    /* Timer driver object handle */
    SYS_MODULE_OBJ  TimerDriver;

    /* Timer system service object handle */
    SYS_MODULE_OBJ  TimerService;

} SYSTEM_OBJECTS;
```

At this point, you should be able to build your project again and step through the code to see how the modules are initialized. However, the system is not yet “running” until you call the appropriate state machine functions, which you will do in Step 6: Call the State Machines for the System and Modules.

1.8 Step 6: Call the State Machines for the System and Modules

This topic describes the steps necessary to call the module state machines.

Description

After initializing the modules, the only thing left before you can start calling their API functions from your application is to ensure that their state machines are running. Each active MPLAB Harmony module implements one or more “Tasks” functions. The module’s “Tasks” functions must be called either from a polled loop or from the appropriate ISR vector function for the specific instance of the hardware peripheral being managed by the module.

To find that information, you will need to review the help document for each library you plan to use. The “Tasks” functions are part of the “system interface”. System interface functions are called by the system configuration code, not by your application

code (or other modules). These functions are conceptually separate from the “client interface” (sometimes called the “API” or Application Program Interface). Each library will have a “System Interface” group in its *Library Interface* section that will list all of the “Tasks” functions along with the “Initialize” function and any other supported system interface functions.

Each library will also have a sub-topic within the *Using the Library* topic that explains how to call the “Tasks” functions for different configurations. The specific tasks functions to use and the locations from which they should be called may vary, depending upon the system’s configuration settings and the types of configurations supported by the library. However, all “Tasks” functions will require the object handle returned by the module’s “Initialize” function as a parameter so that they can access the correct “instance” data.

For the tutorial, you will run both the **Timer Driver Library** and the **Timer System Service Library** modules in polled modes by calling their “Tasks” functions from the SYS_Tasks function (as shown in the following code example), which is called from the top-level “super" loop (the “while” loop in “main”). Also, be sure to include the `system_definitions.h` file so that you have access to the “sysObjects” structure’s “extern” declaration.

Code: system_tasks.c – Calling Module State Machines

```
#include “system_definitions.h”


void SYS_Tasks ( void )
{
    /* Call the state machines of all modules and the application. */
    DRV_TMR_Tasks(sysObjects.TimerDriver);
    SYS_TMR_Tasks(sysObjects.TimerService);
}
```

At this point, you should be able to build, debug, and step through the Timer Driver and Timer System Service initialization. You should be able to see the Timer Driver initialize the timer hardware, watch the Timer System Service walk through its initial states where it opens the Timer Driver and sets the expected “alarm” period. Effectively, you have a running MPLAB Harmony system; however, it just doesn’t do anything yet. In Step 7: Develop the Application State Machine, you will develop your application state machine logic and make sure the system does what you want it to do.

1.9 Step 7: Develop the Application State Machine

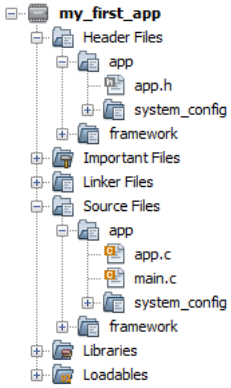
This topic describes the process to develop the application state machine.

Description

Now that you have a running MPLAB Harmony system that supports the drivers and libraries you need, you are ready to implement the application’s state machine logic. The easiest way to get started is to use a simple switch-statement based state machine. The following directions describe how to create the basic "pieces" of your application.

1. Create the application’s source file (`app.c`) in your project’s `src` folder and add it to your project’s logical `app` folder within `Source Files`.
2. Create the application’s header file (`app.h`) in your project’s `src` folder and add it to your project’s logical `app` folder within `Header Files`.
3. In the application’s source file, define outlines of the APP_Initialize and APP_Tasks functions.
4. In the application’s header file, define an enumeration to identify the states of your application’s state machine.
5. Also in the application’s header file, define a data structure to hold the state data for your application.

For a simple application such as the “blinky LED” application in this tutorial, a single pair of application files should be sufficient. When added to your project (in the logical app folder), your logical folder structure should look like the following figure.



More complex applications or projects with multiple application state machines may require multiple files. You can organize your project in any way that is convenient for you, but the layout used in this tutorial will allow you to have multiple applications and multiple configurations in a single MPLAB X IDE project and that will be very helpful as your project becomes more complex or is supported on multiple platforms.

Start out with a single state (for example, APP_STATE_INIT) in which to initialize your application's state machine and add new states, new state data, and new state transition control logic as your application grows. For this tutorial, a few simple states and state variables (see the following code example) will suffice. Define these in your application's header file.

Code: app.h - Application Definitions and Prototypes

```
#include <stdint.h>
#include <stdbool.h>
#include <stddef.h>
#include <stdlib.h>
#include "system/tmr/sys_tmr.h"


/* Application States */
typedef struct
{
    /* Application's initial state */
    APP_STATE_INIT,

    /* Register timer callback */
    APP_STATE_REGISTER_TMR,

    /* Wait for timer callback */
    APP_STATE_WAIT

} APP_STATES;


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

    /* Timer callback handle */
    SYS_TMR_HANDLE timer;

    /* Timer callback flag */
    bool           timerExpired;

} APP_DATA;
```

Define an outline version of your APP_Initialize function in your `app.c` file and place the application in its initial state. Then, as you add additional logic to your application, be sure to add any new states and required state variables to your "APP_STATES" structure and initialize them appropriately in your APP_Initialize function. For the tutorial, the state data structure declaration and initialization function should look like the following code example.

Code: app.c – Data and Initialization

```
#include "system_config.h"
#include "app.h"
#include "system_definitions.h"

APP_DATA appData;

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

Once the application's state machine has been initialized, you can implement the first state's transition logic. For many applications this will involve opening a driver or setting up a system service. Keep in mind that anything that can either block or that may take time and return an incomplete or error status should be handled in your application's state machine (not in your application's initialization function) so that your application can react appropriately without blocking the entire system. This is because in a bare-metal polled configuration, all state machines run in the same "super" loop in the main function. However, that may not be a concern in interrupt-driven or RTOS-based configurations. But, for maximum portability, you may want to manage such delays and status conditions in the state machine as shown in the following code example.

Code: app.c Continued – State Machine and Callback

```
void APP_TimerCallback ( void )
{
    appData.timerExpired = true;
}

void APP_Tasks ( void )
{
    /* Application's State Machine */
    switch ( appData.state )
    {
    case APP_STATE_INIT:
        /* Set pin direction, turn on LED, and transition to next state */
        SYS_PORTS_PinDirectionSelect(APP_LED_PORTS_ID, SYS_PORTS_DIRECTION_OUTPUT,
                                     APP_LED_PORT_CHANNEL, APP_LED_PIN);
        SYS_PORTS_PinSet(APP_LED_PORTS_ID, APP_LED_PORT_CHANNEL, APP_LED_PIN);

        appData.state = APP_STATE_REGISTER_TMR;
        break;

    case APP_STATE_REGISTER_TMR:
        /* Try to egister periodic callback with timer system service, until successful */
        appData.timer = SYS_TMR_CallbackPeriodic(APP_TIMER_PERIOD, APP_TimerCallback);
        if (SYS_TMR_HANDLE_INVALID != appData.timer)
        {
            appData.state = APP_STATE_WAIT;
        }
        break;

    case APP_STATE_WAIT:
        if (appData.timerExpired)
        {
            /* when the time period has expired, toggle the LED & clear the flag */
            SYS_PORTS_PinToggle(APP_LED_PORTS_ID, APP_LED_PORT_CHANNEL, APP_LED_PIN);
            appData.timerExpired = false;
        }
        break;

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

```
}
```

Notice that the previous example uses a separate state transition to attempt to register the timer callback. If the attempt is unsuccessful (i.e., the SYS_TMR_CallbackPeriodic function returns a value of SYS_TMR_HANDLE_INVALID), the application stays in the same state and continues trying to register the callback. The application does not advance its state machine until it is successful. This technique allows the application to manage temporary error conditions and to wait until the **Timer Driver** and the **Timer System Service** are fully initialized and ready to service requests.

The callback function, APP_TimerCallback, is implemented above the APP_Tasks function. When the “APP_TIMER_PERIOD” time has expired, the Timer System Service calls the APP_TimerCallback function to notify the application. All this function does is set a flag variable in the application’s state data structure that the state machine checks once it is in the correct state (the “APP_STATE_WAIT” state) before toggling the LED and clearing the flag. This technique is very useful in MPLAB Harmony’s environment of cooperating state machines.

Once the application’s initialization and tasks functions have been implemented, they need to be called from the appropriate location. For a bare-metal polled configuration, like the tutorial example, this is from the `system_init.c` and `system_tasks.c` files, as shown in the following two (now complete) code examples.

Code: system_init.c – Initializing the Application

```
void SYS_Initialize ( void *data )
{
    /* Set up the clock, cache and wait states for maximum performance. */
    SYS_CLK_Initialize(NULL);
    SYS_DEVCON_Initialize(SYS_DEVCON_INDEX_0, (SYS_MODULE_INIT *)&devconInit);
    SYS_DEVCON_PerformanceConfig(SYS_CLK_PRIMARY_CLOCK);

    /* Initialize all modules and the application. */
    sysObjects.TimerDriver  = DRV_TMR_Initialize(APP_TMR_DRV_INDEX,
                                                 (SYS_MODULE_INIT*)&initDrvTmr);
    sysObjects.TimerService = SYS_TMR_Initialize(APP_TMR_SYS_INDEX,
                                                 (SYS_MODULE_INIT*)&initSysTmr);

    /* Initialize the Application */
    APP_Initialize();
}
```

Code: system_tasks.c – Calling the Application’s State Machine

```
void SYS_Tasks ( void )
{
    /* Call the state machines of all modules and the application. */
    DRV_TMR_Tasks(sysObjects.TimerDriver);
    SYS_TMR_Tasks(sysObjects.TimerService);

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

Conclusion

At this point, you have a complete MPLAB Harmony application! You should be able to build, debug, and further develop it from here. To see a complete example of the application used in this tutorial, refer to the "my_first_app" project in the `<install-dir>/apps/examples` folder within your MPLAB Harmony installation.

After first creating your new project, the basic process stays the same each time you add an additional MPLAB Harmony library or module to your project, starting over at Step 2 of this tutorial.

- Step 2: Identify the Required Library Modules
- Step 3: Add the Library Modules to the Project
- Step 4: Configure the Modules
- Step 5: Initialize the System and Modules
- Step 6: Call the System and Modules State Machines
- Step 7: Develop the Application State Machine (current step)

Repeat this cycle in small steps (for each new module and starting at the bottom of each new stack), testing and debugging as you go and you will be able to very quickly and easily build up very complex applications (and multi-configuration or multi-application projects) with MPLAB Harmony and Microchip PIC32 microcontrollers.

Index