MPLAB® Harmony Help - MPLAB Harmony Tutorial: Creating an Application

MPLAB Harmony Integrated Software Framework v1.11

Creating Your First Project

This tutorial guides you through the process of using the MPLAB Harmony Configurator (MHC) and MPLAB Harmony libraries to develop your first MPLAB Harmony project.

Overview

Lists the basic steps necessary to create a MPLAB Harmony application using the MHC.

Description

MPLAB Harmony provides a convenient MPLAB X IDE plug-in configuration utility, the MPLAB Harmony Configurator (MHC), which you can use to easily create MPLAB Harmony-based projects. This tutorial will show you how to use the MHC to quickly create your first MPLAB Harmony application using the following steps:

- Step 1: Create a New Project
- Step 2: Configure the Processor Clock
- Step 3: Configure Key Device Settings
- Step 4: Configure the I/O Pins
- Step 5: Add and Configure Libraries
- Step 6: Generate the Project's Starter Files
- Step 7: Develop the Application

Your first application should be extremely simple. The example used in this tutorial blinks an LED on the selected platform to provide an application heartbeat health indicator. For guidance on using MPLAB Harmony to develop more capable applications, please refer the Help documentation listed in the Summary section.

Prerequisites and Assumptions

Describes the prerequisites for starting this tutorial and the assumptions made when it was written.

Description

Before beginning this tutorial, ensure that you have installed the MPLAB X IDE and necessary language tools as described in Volume I: Getting Started With MPLAB Harmony > Prerequisites. In addition, an appropriate hardware platform is required.

The example project in this tutorial utilizes the PIC32MZ Embedded Connectivity (EC) Starter Kit. In the event you do not have this development hardware, refer to the Supported Development Boards section for a list of available development boards that you could use to complete this tutorial.

To complete this tutorial you will need to know the following about your selected hardware platform:

- The name of the PIC32 device it uses
- The Primary Oscillator input clock frequency and desired processor clock frequency
- The debugger interface settings
- The I/O Port and Pin that connect to the desired indicator LED
- The input clock source and frequency for the timer selected to blink the indicator LED

Please refer to the documentation for the selected hardware platform and PIC32 device for this information, as described in *Volume I: Getting Started With MPLAB Harmony* > *Guided Tour* > Documentation.

Finally, this tutorial assumes that you have some familiarity with the following; however, you should be able to follow the directions in this tutorial with very little experience:

- MPLAB X IDE development and debugging fundamentals
- C language programming
- PIC32 product family and supported development boards

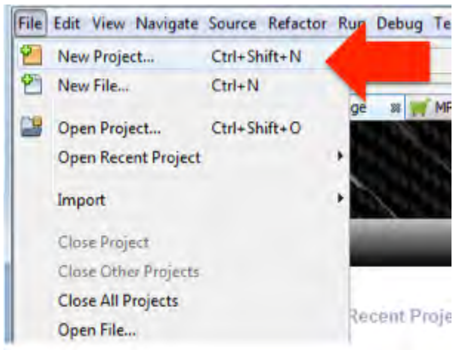
Step 1: Create a New Project

Provides directions for creating a new empty MPLAB Harmony project.

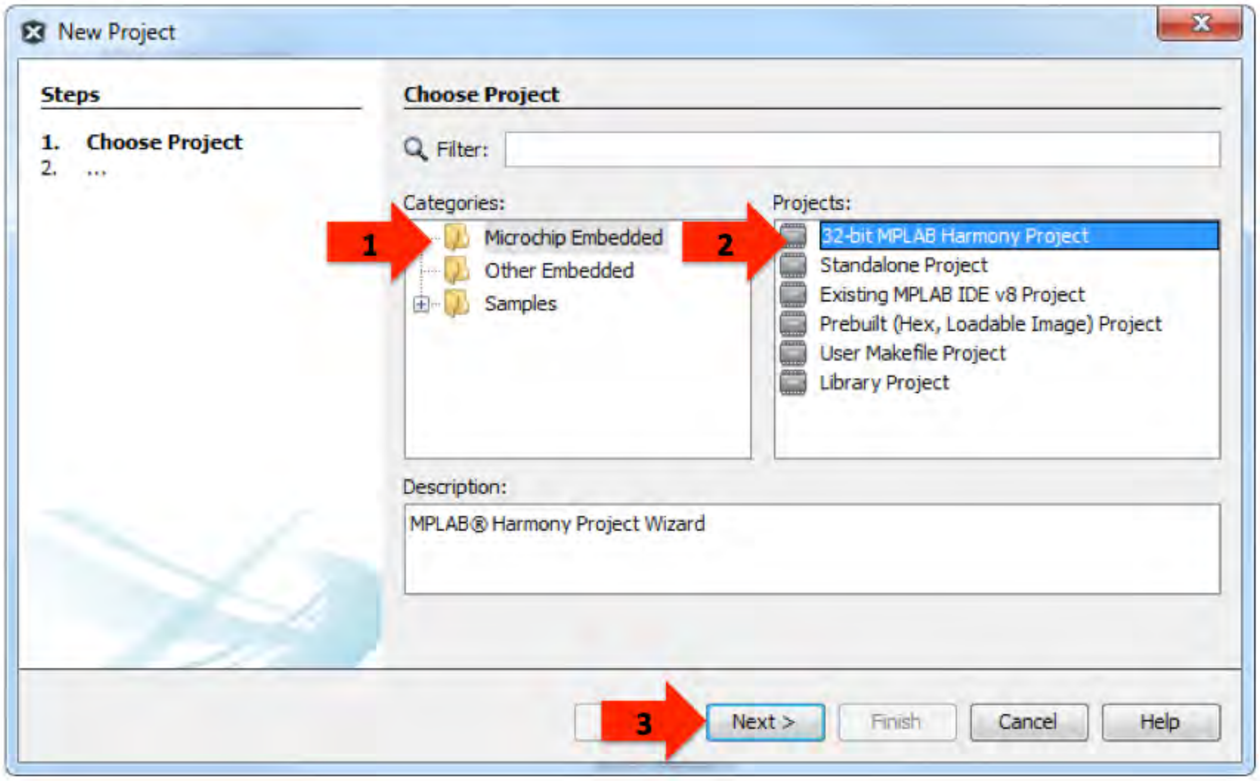
Description

After launching MPLAB X IDE and ensuring that the MHC plug-in is properly installed, you can create a new MPLAB Harmony project using the following the directions.

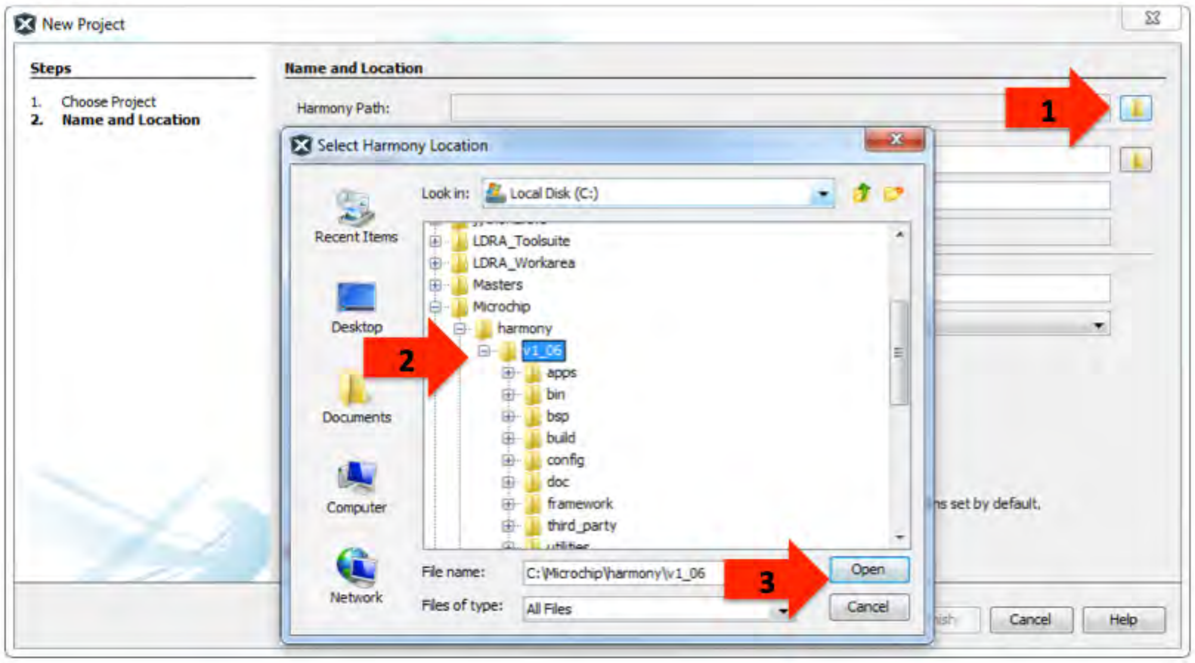
1. Select New Project from the MPLAB X IDE File menu (shown by the red arrow in the following figure), which opens the New Project dialog and launches the New Project wizard.



2. In the Choose Project pane of the New Project window, 1) select **Microchip Embedded**, 2) select **32-bit MPLAB Harmony** Project, 3) and then click **Next**, which opens the Name and Location pane in the New Project dialog window.



3. Select the desired MPLAB Harmony installation by 1) clicking the folder icon next to the Harmony Path display box, 2) navigating to and selecting the folder for the desired MPLAB Harmony distribution (by default, this is the folder matching the installation's version number), and 3) clicking **Open**, as shown in the following figure. The path to the desired MPLAB Harmony installation should now be displayed.

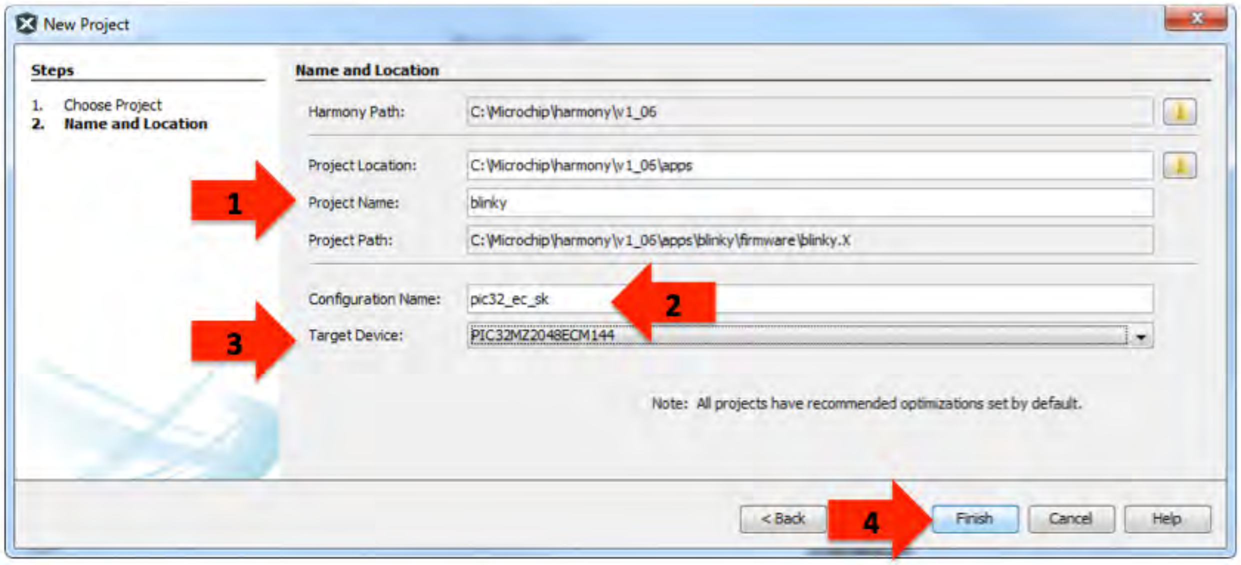


4. As shown in the following figure, choose your project location, name, and initial configuration name by 1) entering the new project name, “blinky” in this example (by default, the MHC will locate this in the `apps` folder of the selected MPLAB Harmony installation), 2) entering a new configuration name, `pic32_ec_sk` in this example, as shown in the following figure (a configuration is commonly named after the board it uses, but you can use any name that meaningfully identifies the configuration), 3) selecting the target device on the PIC32MZ EC Starter Kit in use (see the following **Note**), and 4) click **Finish**.

Note: There are two versions of the PIC32MZ EC Starter Kit, which use different microcontrollers:

- DM320006-C (with Crypto Engine) uses the PIC32MZ2048ECM144
- DM320006 (without Crypto Engine) uses the PIC32MZ2048ECH144

Depending on the starter kit in use, you will need to select the appropriate device.



Harmony Path: C:\Microchip\harmony\v1_06
Project Location: C:\Microchip\harmony\v1_06\apps
Project Name: blinky
Project Path: C:\Microchip\harmony\v1_06\apps\blinky\firmware\blinky.X
Configuration Name: pic32_ec_sk
Target Device: PIC32MZ2048ECM144
Note: All projects have recommended optimizations set by default.

After clicking Finish, the New Project wizard will create several new folders on disk within the project location folder and an empty MPLAB X IDE project (i.e., no source files exist within the folder).

Generated Folders:

```
<project-location>/<project-name>
   firmware
     <project-name>.X
     src
       system_config
         <configuration-name>
```

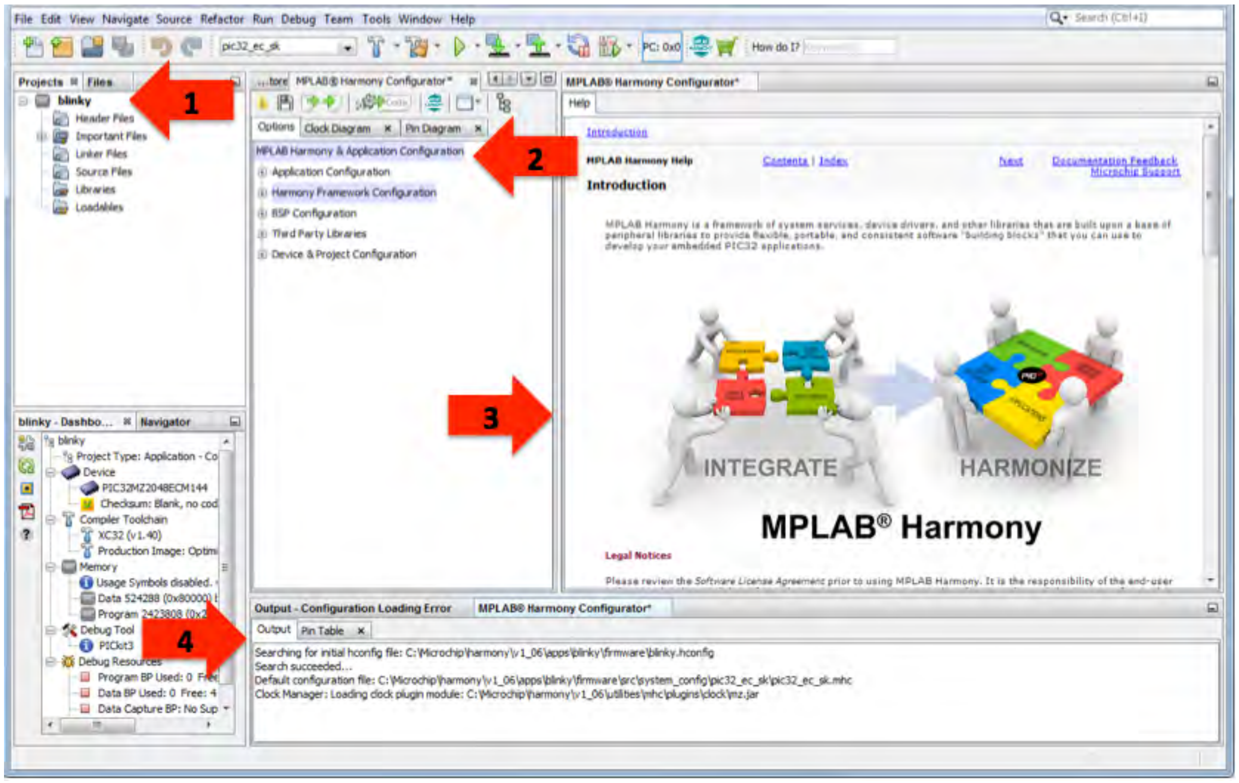
Where `<project-location>`, `<project-name>`, and `<configuration-name>` match the values provided in the New Project Name and Location window (in this example, `C:\microchip\harmony\v1_06`, `blinky`, and `pic32_ec_sk`).

After creating the empty project, the New Project wizard will launch the MHC plug-in so that you may configure the project.

From within MPLAB X IDE, depending on the current layout of the MPLAB X IDE panes, you should see something similar to the following figure, which should display the following items:

1. The empty project (with no files).
2. The MPLAB Harmony configuration tree.
3. The MPLAB Harmony Help system.
4. The MPLAB Harmony Configurator (MHC) output window.

Note: The MPLAB Harmony new project wizard will automatically set the new project as the *main* project in MPLAB X IDE. This is required because the MHC will not launch if there is no currently selected main project open within the MPLAB X IDE.



Introduction

MPLAB Harmony is a framework of system services, device drivers, and other libraries that are built upon a base of peripheral libraries to provide flexible, portable, and consistent software "building blocks" that you can use to develop your embedded PIC32 applications.

Please review the *Software License Agreement* prior to using MPLAB Harmony. It is the responsibility of the end-user

```
Searching for initial hconfig file: C:\Microchip\harmony\v1_06\apps\blinky\firmware\blinky.hconfig
Search succeeded...
Default configuration file: C:\Microchip\harmony\v1_06\apps\blinky\firmware\src\system_config\pic32_ec_sk\pic32_ec_sk.mhc
Clock Manager: Loading clock plugin module: C:\Microchip\harmony\v1_06\utilities\mhc\plugins\clock\mz.jar
```

The project you have just created (see arrow 1, in the previous figure) is empty, meaning it does not yet contain any source files. In later steps, you will use the MHC to generate an initial set of source files for the initial configuration of your project. You will use the MPLAB Harmony configuration tree (see arrow 2) to make the selections for your initial configuration. As you select items in the configuration tree, you will see help for that item appear in the help window (see arrow 3). As you use the MHC, it will display activity, warning, and possibly error messages in the output window (see arrow 4), depending on the level of output for which it is set. By default, it only displays key activity and error messages.

Step 2: Configure the Processor Clock

Provides instructions on how to use the MHC to configure the processor clock.

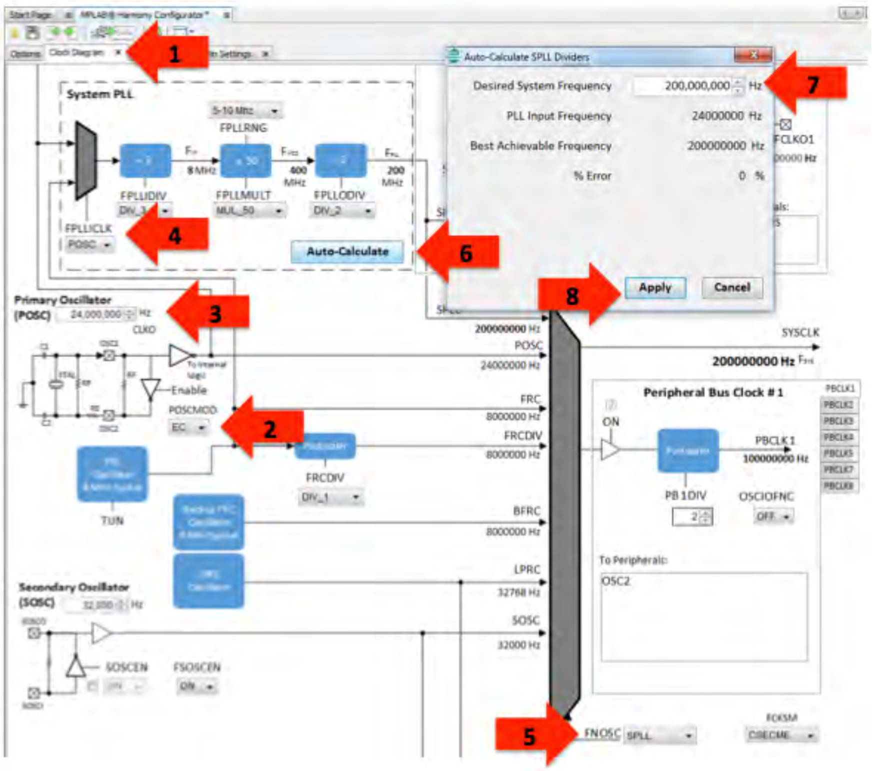
Description

After creating a new blank MPLAB Harmony project, the next step is to use the interactive clock diagram in the MHC to configure the processor clock. To do this, use the following process. The numbered red arrows in the following clock diagram show the location of each step in the process.

1. Select the clock diagram in the MPLAB Harmony Configurator pane.
2. Select the External Clock (EC) mode for the primary oscillator.
3. Enter the primary oscillator input clock frequency (24 MHz for the PIC32MZ EC Starter Kit).
4. Select the primary oscillator (POSC) as the system PLL input clock.

Note: Values that are not supported on the selected device appear in red and must be changed to a supported value

5. Select the system PLL (SPLL) from the oscillator multiplexer (FNOSC).
6. Click **Auto-Calculate**.
7. Enter the desired system frequency (200 MHz for the PIC32MZ EC Starter Kit).
8. Click **Apply**.



Auto-Calculate SPLL Dividers

Desired System Frequency 200,000,000 Hz
PLL Input Frequency 24000000 Hz
Best Achievable Frequency 200000000 Hz
% Error 0 %

The exact settings will vary for different Microchip development boards. Most Microchip PIC32 development boards will utilize a primary oscillator input at a frequency that allows running the processor at its highest rate. Refer to the documentation for the development board in use for information on the crystal or oscillator used. You may also find the settings used by MPLAB Harmony demonstration applications helpful as examples.

Use of a Board Support Package (BSP), such as the ones provided for Microchip development boards in the `<install-dir>/bsp` folder, will automatically set the clock configuration for maximum performance. BSPs are used by most demonstration applications provided in the MPLAB Harmony installation, but they are not required for your own applications. Also, do not worry if you decide to change frequencies later. It is easy to return to the clock diagram at any time and change the processor's clock settings.

Step 3: Configure Key Device Settings

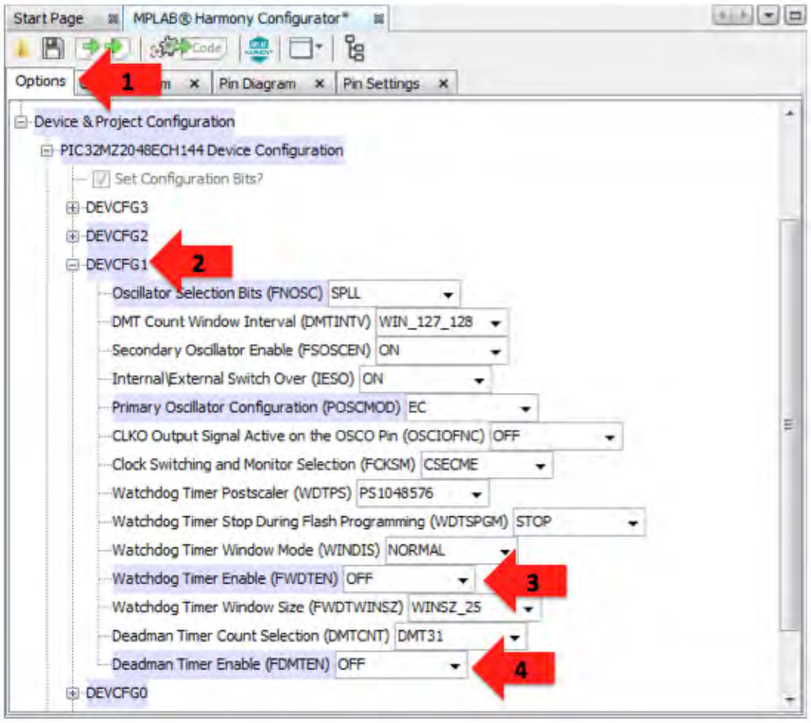
Explains how to configure key device settings to program, debug, and run firmware.

Description

There are a few key device configuration settings for most processors that must be correct to effectively run and debug the firmware. Exactly which settings are important depends on which processor is in use and what you are currently doing.

To get started, it is usually necessary to disable all Watchdog and Deadman Timers to avoid having the timer expire and reset the processor before the firmware has been implemented to support that functionality. To do this for the PIC32MZ2048ECM144 device used by this example, perform the following steps, which are also shown by numbered red arrows in the next figure:

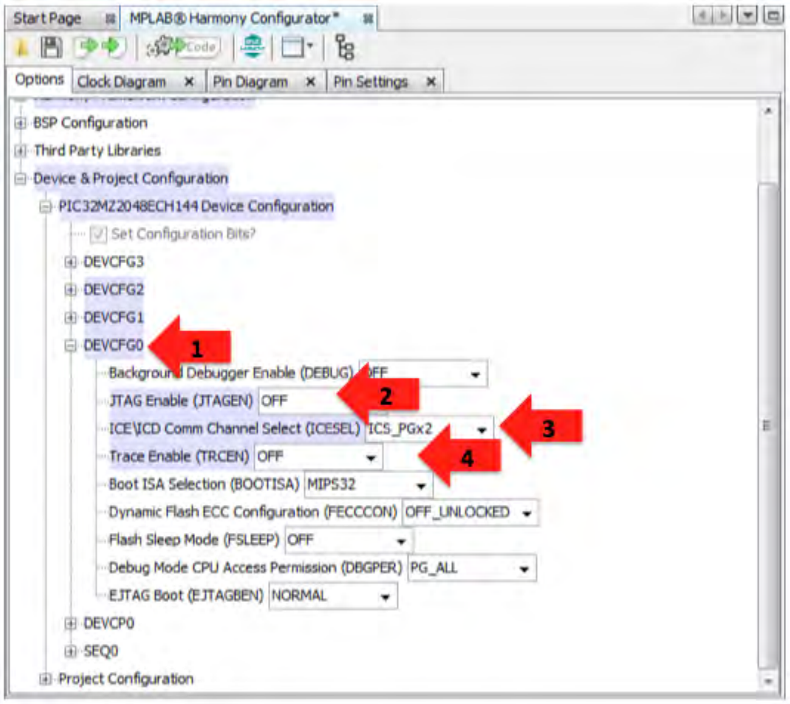
1. Select the Options pane in the MPLAB Harmony Configurator Window.
2. Expand the Device Configuration 1 (DEVCFG1) register settings tree.
3. Change the Watchdog Timer Enable (FWDTEN) flag to OFF.
4. Change the Deadman Timer Enable (FDMTEN) flag to OFF.



```
Device & Project Configuration
  PIC32MZ2048ECH144 Device Configuration
    Set Configuration Bits?
    DEVCFG3
    DEVCFG2
    DEVCFG1
      Oscillator Selection Bits (FNOSC) SPLL
      DMT Count Window Interval (DMTINTV) WIN_127_128
      Secondary Oscillator Enable (FSOSCEN) ON
      Internal\External Switch Over (IESO) ON
      Primary Oscillator Configuration (POSCMOD) EC
      CLKO Output Signal Active on the OSCO Pin (OSCIOFNC) OFF
      Clock Switching and Monitor Selection (FCKSM) CSECME
      Watchdog Timer Postscaler (WDTPS) PS1048576
      Watchdog Timer Stop During Flash Programming (WDTSPGM) STOP
      Watchdog Timer Window Mode (WINDIS) NORMAL
      Watchdog Timer Enable (FWDTEN) OFF
      Watchdog Timer Window Size (FWDTWINSZ) WINSZ_25
      Deadman Timer Count Selection (DMTCNT) DMT31
      Deadman Timer Enable (FDMTEN) OFF
    DEVCFG0
```

Additionally, it is often necessary to ensure that the in-circuit emulator interface and other debugging capabilities like instruction trace (for devices that support it) are properly configured. Most PIC32 starter kits do not utilize JTAG, so it will need to be disabled, and instruction trace is not required for this tutorial and will need to be disabled. To do this for the PIC32MZ2048ECM144 device used by this demonstration, perform the following steps.

1. Expand the Device Configuration 0 (DEVCFG0) register settings tree.
2. Change the JTAG Enable (JTAGEN) setting to OFF.
3. Change the In-circuit Emulator Communication Channel Select (ICESEL) setting to ICSxPGx2. The ICSxPG setting must match the debug channel used on the physical hardware.
4. Change the Trace Enable (TRCEN) setting to OFF.



BSP Configuration
Third Party Libraries
Device & Project Configuration
PIC32MZ2048ECH144 Device Configuration
Set Configuration Bits?
DEVCFG3
DEVCFG2
DEVCFG1
DEVCFG0
Background Debugger Enable (DEBUG) OFF
JTAG Enable (JTAGEN) OFF
ICE\ICD Comm Channel Select (ICESEL) ICS_PGx2
Trace Enable (TRCEN) OFF
Boot ISA Selection (BOOTISA) MIPS32
Dynamic Flash ECC Configuration (FECCCON) OFF_UNLOCKED
Flash Sleep Mode (FSLEEP) OFF
Debug Mode CPU Access Permission (DBGPER) PG_ALL
EJTAG Boot (EJTAGBEN) NORMAL
DEVCP0
SEQ0
Project Configuration

When you select the <*device name*> Device Configuration item within the MHC Options tree, the MHC Help window will show information describing the device configuration settings for the processor selected for the main project's currently selected configuration. You may also reference information about the device's configuration settings in the compiler's user guide and in the device's data sheet. If your hardware and debugger connections are correct and you are unable to program, debug, or run code on the device, it is very likely that one of these key device configuration settings is incorrect. You must ensure that these settings are correct before continuing.

Step 4: Configure the I/O Pins

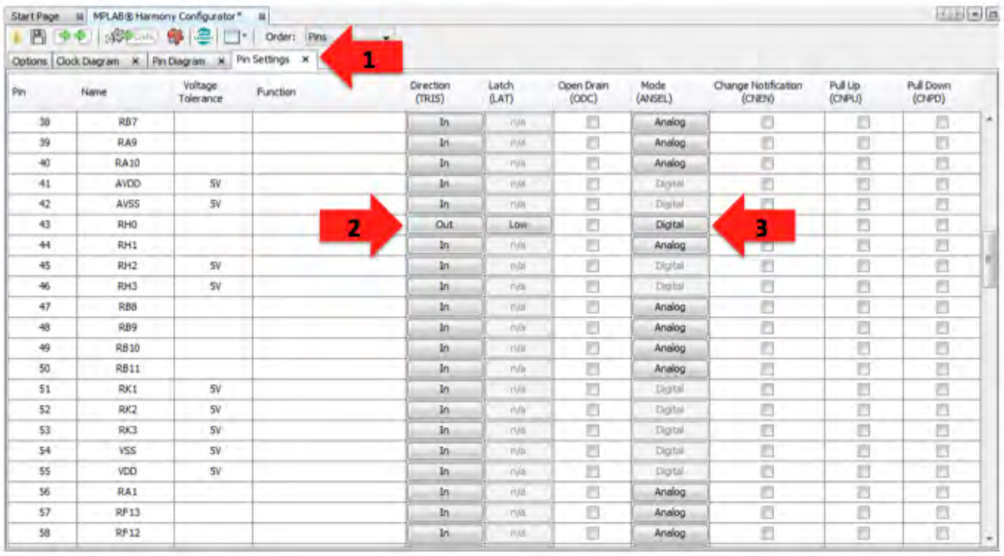
Provides an example of how to use the MHC to configure I/O pins.

Description

After configuring the processor clock and key settings, configure any general purpose I/O pins required by your project using the interactive Pin Diagram and Pin Settings panes provided by the MHC.

For example, the PIC32MZ EC Starter Kit has LED1 connected to port pin RH0. Therefore, that pin will need to be set as an output to blink the heartbeat indicator LED, by performing the following actions.

1. Select the Pin Settings tab in the MHC pane.
2. Scroll to pin 43, RH0 and select it as a digital output pin by clicking the **Out** Direction (TRIS) button. (You can accept the default Low initial output level.)
3. Select **Digital** mode.



Pin	Name	Voltage Tolerance	Function	Direction (TRIS)	Latch (LAT)	Open Drain (ODC)	Mode (ANSEL)	Change Notification (CNEN)	Pull Up (CNPU)	Pull Down (CNPD)
38	RB7			In	n/a	☐	Analog	☐	☐	☐
39	RA9			In	n/a	☐	Analog	☐	☐	☐
40	RA10			In	n/a	☐	Analog	☐	☐	☐
41	AVDD	5V		In	n/a	☐	Digital	☐	☐	☐
42	AVSS	5V		In	n/a	☐	Digital	☐	☐	☐
43	RH0			Out	Low	☐	Digital	☐	☐	☐
44	RH1			In	n/a	☐	Analog	☐	☐	☐
45	RH2	5V		In	n/a	☐	Digital	☐	☐	☐
46	RH3	5V		In	n/a	☐	Digital	☐	☐	☐
47	RB8			In	n/a	☐	Analog	☐	☐	☐
48	RB9			In	n/a	☐	Analog	☐	☐	☐
49	RB10			In	n/a	☐	Analog	☐	☐	☐
50	RB11			In	n/a	☐	Analog	☐	☐	☐
51	RK1	5V		In	n/a	☐	Digital	☐	☐	☐
52	RK2	5V		In	n/a	☐	Digital	☐	☐	☐
53	RK3	5V		In	n/a	☐	Digital	☐	☐	☐
54	VSS	5V		In	n/a	☐	Digital	☐	☐	☐
55	VDD	5V		In	n/a	☐	Digital	☐	☐	☐
56	RA1			In	n/a	☐	Analog	☐	☐	☐
57	RF13			In	n/a	☐	Analog	☐	☐	☐
58	RF12			In	n/a	☐	Analog	☐	☐	☐

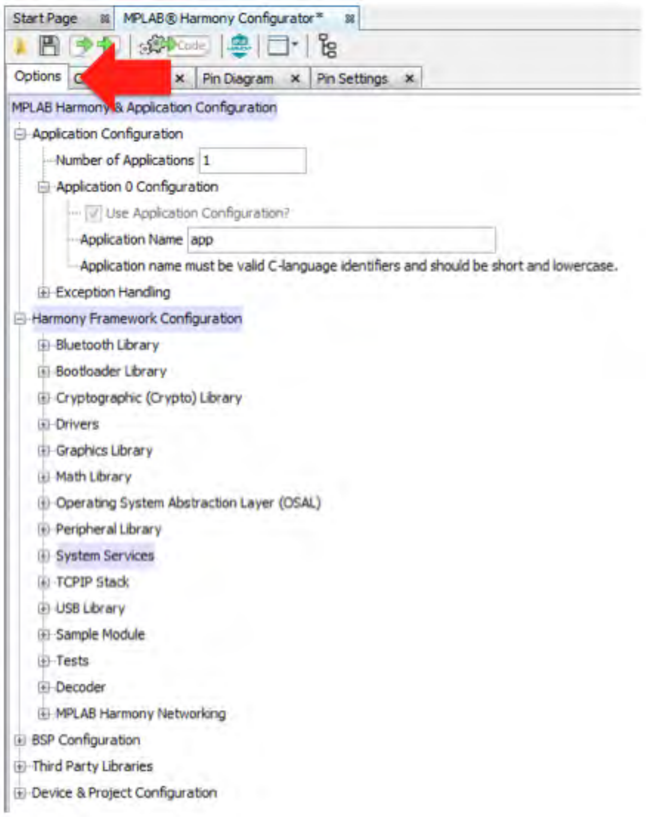
Often, boards will have switches and LEDs and it is a good idea to provide some sort of *heartbeat* indicator to show that the application is running, particularly during development. Therefore, you will normally configure a few general purpose I/O pins when you first create a project. However, do not worry about configuring all of your I/O pins when you first create a new project. Predefined BSPs will automatically configure the I/O pins for LEDs and switches for supported Microchip development boards. (Refer to Board Support Packages Help for more information on BSPs.) And, you can return to the Pin Settings and Pin Diagram tabs in the MHC at any time to update your I/O port settings.

Step 5: Add and Configure Libraries

Demonstrates how to add and configure a library required by an application.

Description

The MHC allows you to configure basic application features and select and configure the desired MPLAB Harmony libraries. Click the Options tab to select this pane, as shown by the red arrow in the following figure.

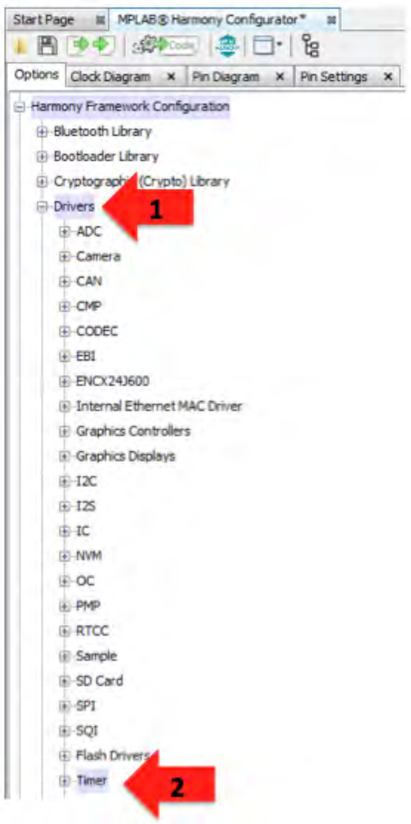


MPLAB Harmony & Application Configuration

- Application Configuration
  - Number of Applications 1
  - Application 0 Configuration
    - ☑ Use Application Configuration?
    - Application Name app
    - Application name must be valid C-language identifiers and should be short and lowercase.
  - Exception Handling
- Harmony Framework Configuration
  - Bluetooth Library
  - Bootloader Library
  - Cryptographic (Crypto) Library
  - Drivers
  - Graphics Library
  - Math Library
  - Operating System Abstraction Layer (OSAL)
  - Peripheral Library
  - System Services
  - TCPIP Stack
  - USB Library
  - Sample Module
  - Tests
  - Decoder
  - MPLAB Harmony Networking
- BSP Configuration
- Third Party Libraries
- Device & Project Configuration

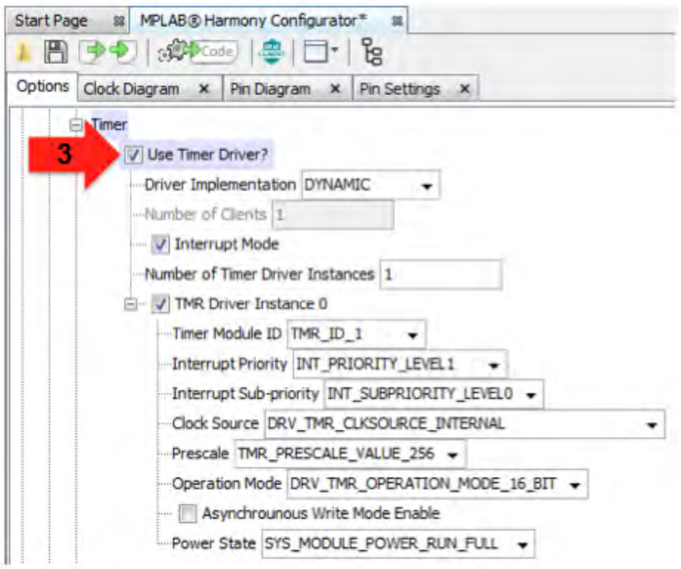
In this example, you can accept the default application configuration settings, but you will need to utilize the Timer Driver. To access the Timer Driver configuration options, you must expand the Harmony Framework configuration tree by clicking the plus (+) icons next to Drivers and Timer.

1. Expand the Drivers tree.
2. Expand the Timer driver's configuration tree.



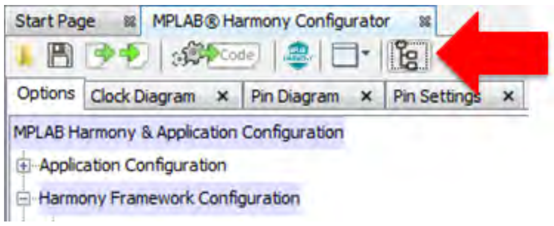
Harmony Framework Configuration
- Bluetooth Library
- Bootloader Library
- Cryptographic (Crypto) Library
- Drivers
  - ADC
  - Camera
  - CAN
  - CMP
  - CODEC
  - EBI
  - ENCX24J600
  - Internal Ethernet MAC Driver
  - Graphics Controllers
  - Graphics Displays
  - I2C
  - I2S
  - IC
  - NVM
  - OC
  - PMP
  - RTCC
  - Sample
  - SD Card
  - SPI
  - SQI
  - Flash Drivers
  - Timer

3. Enable use of the Timer Driver, by selecting the check box next to *Use Timer Driver*, as shown in the following figure. This will expand the Timer Driver configuration options. For this example, you can accept all of the default options. This will configure the Timer Driver to use a single interrupt-driven dynamic implementation as instance 0 (DRV_TMR_INDEX_0). That instance will utilize hardware Timer Peripheral 1 (TMR_ID_1) in 16-bit mode with a maximum prescale divisor of 256, using the internal peripheral bus clock as the timer's clock source.



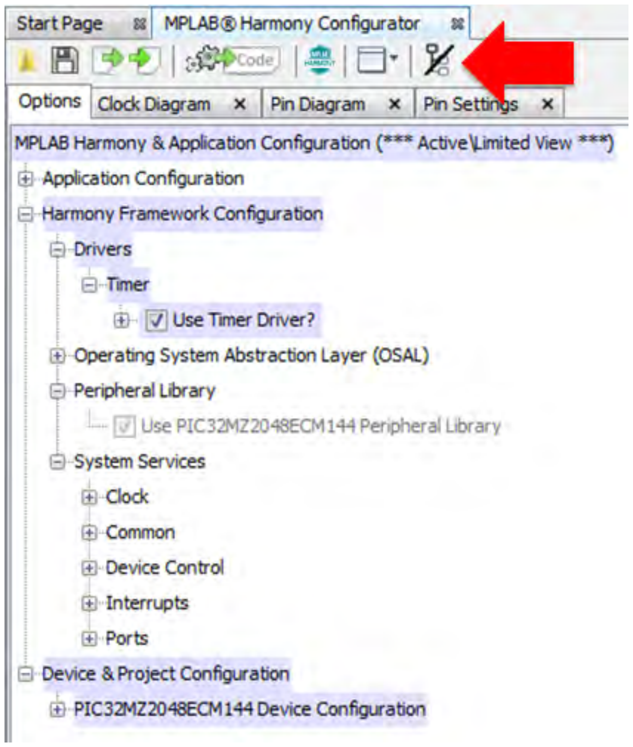
Timer
- [✓] Use Timer Driver?
- Driver Implementation: DYNAMIC
- Number of Clients: 1
- [✓] Interrupt Mode
- Number of Timer Driver Instances: 1
- [✓] TMR Driver Instance 0
  - Timer Module ID: TMR_ID_1
  - Interrupt Priority: INT_PRIORITY_LEVEL1
  - Interrupt Sub-priority: INT_SUBPRIORITY_LEVEL0
  - Clock Source: DRV_TMR_CLKSOURCE_INTERNAL
  - Prescale: TMR_PRESCALE_VALUE_256
  - Operation Mode: DRV_TMR_OPERATION_MODE_16_BIT
  - [ ] Asynchronous Write Mode Enable
  - Power State: SYS_MODULE_POWER_RUN_FULL

The blinky example application used in this tutorial will also utilize the Ports System Service (SYS_PORTS). However, that service is enabled by default and you have already configured it previously, using the Pin Settings pane. Also, the SYS CLOCK service (previously configured using the interactive clock diagram) is enabled by default. To see which libraries are enabled and how they're configured, select the Options Tree icon to see the limited view of the options tree that shows only the active libraries, as shown in the following figure.



MPLAB Harmony & Application Configuration
- Application Configuration
- Harmony Framework Configuration

When viewing the limited Options Tree and showing only the active libraries, the Tree View icon shows a black slash through it and only the currently enabled libraries are shown in the MHC configuration tree, as shown in the following figure. Click the Tree View icon again to return to the full view.



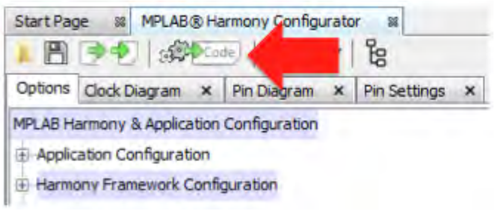
MPLAB Harmony & Application Configuration (*** Active\Limited View ***)
- Application Configuration
- Harmony Framework Configuration
  - Drivers
    - Timer
      - [✓] Use Timer Driver?
  - Operating System Abstraction Layer (OSAL)
  - Peripheral Library
    - [✓] Use PIC32MZ2048ECM144 Peripheral Library
  - System Services
    - Clock
    - Common
    - Device Control
    - Interrupts
    - Ports
- Device & Project Configuration
  - PIC32MZ2048ECM144 Device Configuration

Step 6: Generate the Project's Starter Files

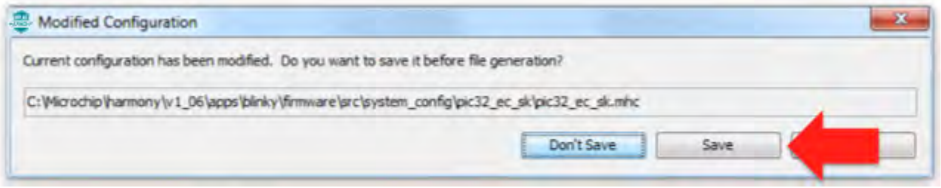
Describes how to generate the project’s starter files.

Description

Once the initial configuration options have been selected in the MHC, click the Generate Code icon, as shown in the following figure to generate the initial set of source files for your application.


Start Page
MPLAB® Harmony Configurator
Code
Options
Clock Diagram
Pin Diagram
Pin Settings
MPLAB Harmony & Application Configuration
Application Configuration
Harmony Framework Configuration

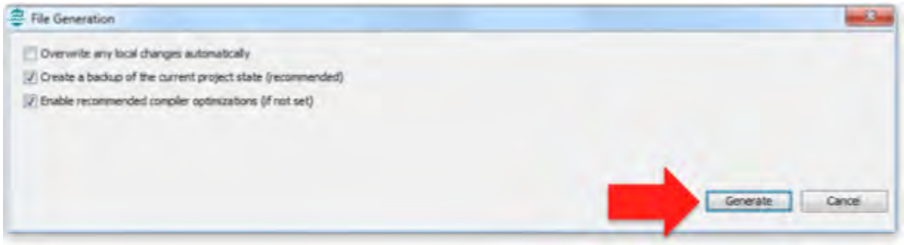
Accept the default location for the configuration setting's (`.mhc`) file and click **Save** in the Modified Configuration dialog (see the following figure), unless you have previously saved the current configuration settings. This file stores the selections made in the MHC window.



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

C:\Microchip\harmony\v1_06\apps\blinky\firmware\src\system_config\pic32_ec_sk\pic32_ec_sk.mhc

Accept the default settings and click Generate in the File Generation dialog, as shown in the next figure.



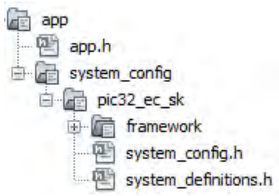
- [ ] Overwrite any local changes automatically
- [✓] Create a backup of the current project state (recommended)
- [✓] Enable recommended compiler optimizations (if not set)

Notes:

1. Selecting 'Overwrite any local changes automatically' will cause the MHC to overwrite the existing configuration fields with the newly generated versions. If this option is not selected, the MHC will identify when configuration files have been changed and when it does it will open a diff/merge utility so that you can choose which changes to keep. By default, this option is not selected.
2. If the 'Create a backup of the current project state' option is selected (enabled by default), the MHC will generate a back-up of the current configuration settings before updating them. Refer to the MPLAB Harmony Configurator User's Guide for details on using and restoring configurations.
3. If the 'Enable recommended compiler optimizations' option is selected (enabled by default), the MHC will automatically enable a minimal level of optimization (equivalent to `-O1` on the XC32 command line) in the project settings. To change this setting, refer to the MPLAB X IDE project Properties window.

This will cause the MHC to generate an initial set of project source files, based on your current configuration selections. This set of files will include the following application files, as shown in the MPLAB X IDE Project window.

Generated Header Files



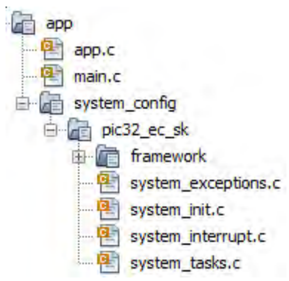
The `app.h` header file contains definitions and prototypes required by the application or by other modules that use the application.

The `system_config.h` header file defines build options and is included by all other files that require any configuration options.

The `system_definitions.h` header file provides definitions required by the system configuration files.

These files implement a configuration of an MPLAB Harmony firmware system. They are generated in a configuration-specific folder (`pic32_ec_sk` folder in this example, as follows).

Generated Source Files



The `main.c` source file contains a standard MPLAB Harmony C-language main function.

The `app.c` source file contains the application logic state machine.

The system files (`system_*.c`) implement the (`pic32_ec_sk`) configuration of the system.

The `system_init.c` file contains the standard MPLAB Harmony SYS_Initialize function (and supporting code) that is called by main to initialize all modules in the system.

The `system_tasks.c` file implements the standard MPLAB Harmony SYS_Tasks function that is called in a loop by main to keep the state machines of all non-interrupt-driven modules running in the system.

The `system_interrupt.c` file implements the Interrupt Service Routine (ISR) functions for any interrupt-driven modules in the system.

Finally, the `system_exceptions.c` file implements any exception-handling functions required by the system.

The MHC generates some MPLAB Harmony libraries, either wholly or in part, using knowledge of configuration selections made by the user, which makes them configuration specific. The `app/system_config/<configuration>/framework` folder contains these configuration-specific files that are conceptually part of the MPLAB Harmony framework, not the application.

Refer to the Project Layout section for additional information on the files generated by the MHC.

Notes:

1. The `app` folder, as displayed in MPLAB X IDE, corresponds to the `src` folder on disk.

2. The top-level (non-configuration-specific) `<install-dir>/framework` folder corresponds to the top-level `framework` folder in the MPLAB Harmony installation directory. Files in this folder are referenced and used by your project, but are not copied into the project-specific folders.

Step 7: Develop the Application

Describes how to develop MPLAB Harmony application logic, using the Blinky Heartbeat Indicator project as an example.

Description

Once the initial set of application and configuration files have been generated, you're ready to begin developing your application logic. To do this, you will modify some of the generated files, adding your own custom code. Some of these files, such as `app.h` and `app.c`, are only generated once, when the project is initially created. These are starter files where you are intended to do most of your development. Other files (like the system configuration files) are under control of the MHC. You can, and will occasionally need to modify these files; however, you need not worry about losing your changes if you regenerate the project files. The MHC will open a *diff* tool when it detects that you have modified a file and allow you to choose which changes you want to keep and which ones you want to update.

Note: A default *diff* tool is included in MPLAB X IDE. However, you can replace this tool with one of your own preference. Refer to the MPLAB X IDE documentation for details on how to use a custom *diff* tool.

Heartbeat Indicator

For this example, you will develop a heartbeat indicator. The heartbeat indicator is an LED that blinks at a regular pace (a rate of about one-half Hz) as long as your application's main tasks function (APP_Tasks) is running properly. If your APP_Tasks function is called too slowly or becomes blocked, the heartbeat indicator will slow down or stop blinking. This functionality is a good thing to have in most projects (particularly during development), so this example is a good place to start most new applications.

The state variables that belong to an application are usually managed in a data structure called APP_DATA. To implement the heartbeat indicator, add the following variables to this structure in `app.h`, as follows.

Heartbeat Variables:

- heartbeatTimer – Heartbeat timer driver handle
- heartbeatCount – Count of heartbeat timer driver alarms
- heartbeatToggle – A flag indicating when to toggle the heartbeat LED

Updated APP_DATA Structure in app.h

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

    /* Heartbeat driver timer handle. */
    DRV_HANDLE      heartbeatTimer;

    /* Heartbeat timer timeout count. */
    unsigned int    heartbeatCount;

    /* Heartbeat LED toggle flag. */
    bool            heartbeatToggle;

} APP_DATA;
```

You will also need to add an idle state to the application's state machine so that the application has a state to which it can transition after its initial state has started the heartbeat timer running. To do this, add the APP_STATE_IDLE entry to the APP_STATES enumeration, also defined in the following `app.h` header file

Updated APP_STATES Enumeration in app.h

```
typedef enum
{
    /* Application's state machine's initial state. */
    APP_STATE_INIT=0,

    /* Application state machine's idle state. */
    APP_STATE_IDLE,
    APP_STATE_SERVICE_TASKS,

} APP_STATES;
```

While you're editing the `app.h` file, this would be a good time to add include statements for the Timer Driver and Ports System Service header files (`drv_tmr.h` and `sys_ports.h`, respectively) to `app.h` to provide the prototypes that the application's heartbeat code will require, as follows.

Included Files in app.h

```
#include <stdint.h>
#include <stdbool.h>
#include <stddef.h>
#include <stdlib.h>
```

```
#include "system_config.h"
#include "system_definitions.h"
#include "driver/tmr/drv_tmr.h"
#include "system/ports/sys_ports.h"
```

There are several application-specific configuration constants that are used in place of hard-coded values. There will also be a few other constants needed to identify things such as the indicator LED port and pin numbers, the timer alarm period, and maxim alarm counts, as follows.

Application-Specific Configuration Constants

- APP_HEARTBEAT_TMR
- APP_HEARTBEAT_TMR_IS_PERIODIC
- APP_HEARTBEAT_TMR_PERIOD
- APP_HEARTBEAT_COUNT_MAX
- APP_HEARTBEAT_PORT
- APP_HEARTBEAT_PIN

These macros are best defined in the `system_config.h` header file. Defining them as application-specific configuration options in the `system_config.h` header allows the heartbeat logic to be easily changed to use a different timer instance or a different indicator port pin in different configurations of the application, if desired.

Application-Specific Configuration Macros in system_config.h

```
#define APP_HEARTBEAT_TMR              DRV_TMR_INDEX_0
#define APP_HEARTBEAT_TMR_IS_PERIODIC  true
#define APP_HEARTBEAT_TMR_PERIOD       0xFE51
#define APP_HEARTBEAT_COUNT_MAX        6
#define APP_HEARTBEAT_PORT             PORT_CHANNEL_H
#define APP_HEARTBEAT_PIN              PORTS_BIT_POS_0
```

The usage of these items will be discussed in more detail in the following sections, but first, the application should initialize these new variables to ensure that they will have appropriate initial values when the application's state machine function (APP_Tasks) is called at a later time. To do this, update the APP_Initialize function in the `app.c` file, as follows.

Updated APP_Initialize Funciton in app.c

```
void APP_Initialize ( void )
{
    /* Place the App state machine in its initial state. */
    appData.state           = APP_STATE_INIT;
    appData.heartbeatTimer  = DRV_HANDLE_INVALID;
    appData.heartbeatCount  = 0;
    appData.heartbeatToggle = false;
}
```

To use the Timer Driver, it must first be opened and a handle to it obtained from the driver's *Open* function (DRV_TMR_Open). This must be done in the application's state machine (in the APP_STATE_INIT state) before the application can call any other Timer Driver functions. If a valid handle is returned from the driver's *Open* function, an alarm callback (to a function to be defined later by the application) can be registered with the driver by calling DRV_TMR_AlarmRegister and the timer can be started by calling DRV_TMR_Start function.

If the handle returned from the driver's *Open* function is invalid, the application should stay in its initial state (APP_STATE_INIT) and try again next time its state-machine function is called. Once the application has successfully opened the Timer Driver, registered a callback, and started the timer, it can move to the next state (APP_STATE_IDLE) by assigning a new value to the appData.state variable.

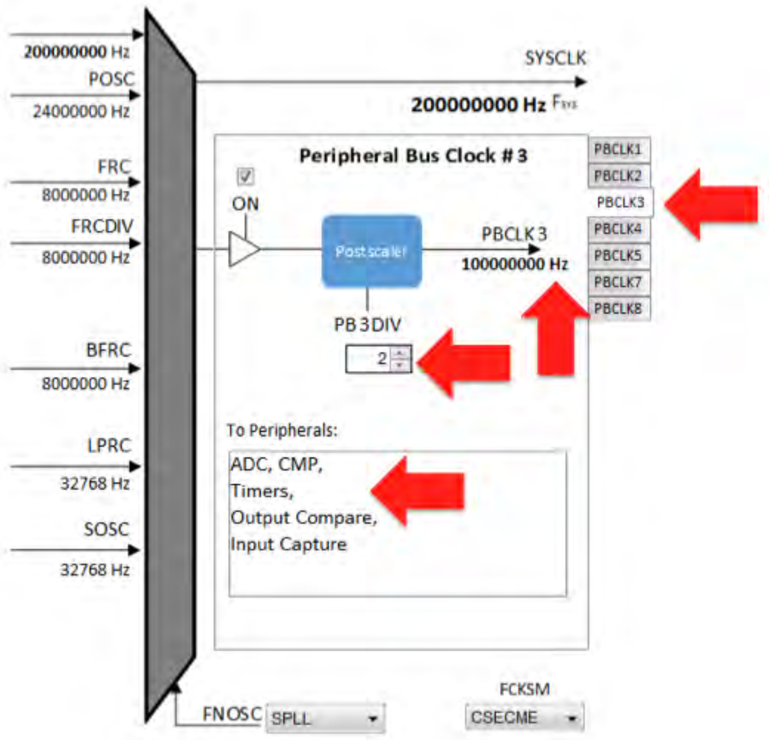
The following updated code shows how to implement the logic previously described.

Updated APP_Tasks Switch Statement in app.c

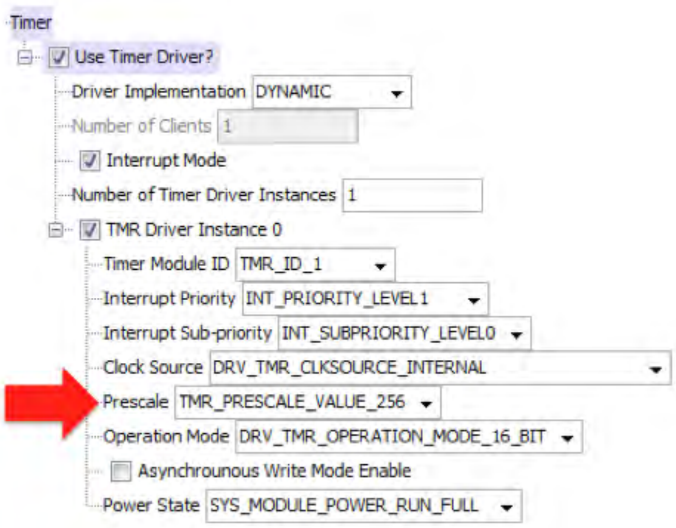
```
switch ( appData.state )
{
    case APP_STATE_INIT:
    {
        appData.heartbeatTimer = DRV_TMR_Open( APP_HEARTBEAT_TMR,
            DRV_IO_INTENT_EXCLUSIVE);
        if ( DRV_HANDLE_INVALID != appData.heartbeatTimer )
        {
            DRV_TMR_AlarmRegister(appData.heartbeatTimer,
                APP_HEARTBEAT_TMR_PERIOD,
                APP_HEARTBEAT_TMR_IS_PERIODIC,
                (uintptr_t)&appData,
                APP_TimerCallback);
            DRV_TMR_Start(appData.heartbeatTimer);
            appData.state = APP_STATE_IDLE;
        }
        break;
    }

    case APP_STATE_IDLE:
    {
        break;
    }

    default:
    {
        break;
    }
}
```

The APP_HEARTBEAT_TMR macro identifies which instance of the Timer Driver was configured for its use. The APP_HEARTBEAT_TMR_IS_PERIODIC macro is mostly provided as documentation (heartbeat timer would always have to be periodic, so that value could be hard-coded), but it does help make the call to DRV_TMR_AlarmRegister easier to read and understand. The APP_HEARTBEAT_PORT and APP_HEARTBEAT_PIN macros define the I/O port pin that will be used to toggle the heartbeat indicator LED.

The APP_HEARTBEAT_TMR_PERIOD macro defines the counter period used to divide down the timer *alarm* (a.k.a., interrupt or match) period (which must be 16 bits to match the timer) and the APP_HEARTBEAT_COUNT_MAX macro defines how many times the heartbeat logic (which you will implement shortly) will count that alarm occurrence before it toggles the indicator LED. These macros are used, in combination with the system clock and timer prescale divisor settings, to define the indicator blink rate. The timers in the PIC32MZ2048ECM144 device receive their input clocks from peripheral bus clock 3 (PBCLK3). You can see this from the MHC's interactive clock diagram by looking at the PBCLK3 tab in the following diagram.

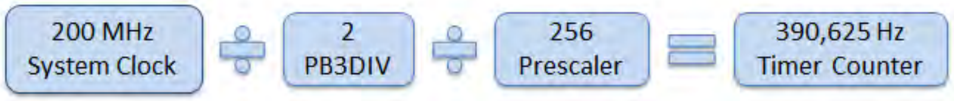


PBCLK3 was configured (by default) with a divisor of 2 from the system clock, and providing a 100 MHz input clock to the timers (based on the previously selected clock configuration settings). Next, observe that the timer prescale divisor was configured with a value of 256, as selected (again by default) in the Timer Driver instance 0 configuration settings, as shown in the following diagram.

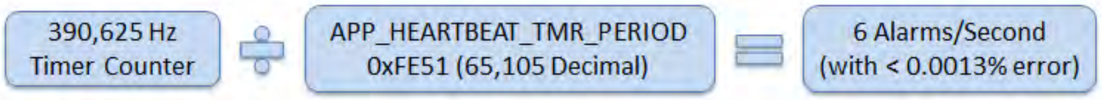


Timer
- [✓] Use Timer Driver?
  - Driver Implementation: DYNAMIC
  - Number of Clients: 1
  - [✓] Interrupt Mode
  - Number of Timer Driver Instances: 1
  - [✓] TMR Driver Instance 0
    - Timer Module ID: TMR_ID_1
    - Interrupt Priority: INT_PRIORITY_LEVEL1
    - Interrupt Sub-priority: INT_SUBPRIORITY_LEVEL0
    - Clock Source: DRV_TMR_CLKSOURCE_INTERNAL
    - Prescale: TMR_PRESCALE_VALUE_256
    - Operation Mode: DRV_TMR_OPERATION_MODE_16_BIT
    - [ ] Asynchrounous Write Mode Enable
    - Power State: SYS_MODULE_POWER_RUN_FULL

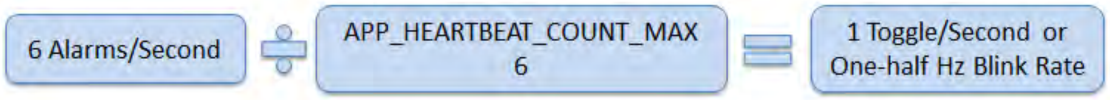
This allows you to calculate the timer counter rate as shown in the next diagram.



With this information and the knowledge that the alarm period must be 16 bits (to match the timer counter) as required by the parameter data type for the DRV_TMR_AlarmRegister function, you can determine (with a little trial and error) a large timer alarm period to minimize the number of alarm interrupts required while still giving a very small error, as shown in the following diagram.


390,625 Hz Timer Counter
÷
APP_HEARTBEAT_TMR_PERIOD 0xFE51 (65,105 Decimal)
=
6 Alarms/Second (with < 0.0013% error)

With six alarms per second, you simply need to count alarm occurrences before toggling the indicator LED to give a nice leisurely heartbeat indicator blink rate.


6 Alarms/Second
÷
APP_HEARTBEAT_COUNT_MAX 6
=
1 Toggle/Second or One-half Hz Blink Rate

The logic to count the heartbeat alarms must be implemented in the `app.c` file in the APP_TimerCallback function (as follows) because that is the function that was registered with the Timer Driver by the DRV_TMR_AlarmRegister function.

Timer Callback Function in app.c

```
static void APP_TimerCallback (  uintptr_t context, uint32_t  alarmCount )
{
    appData.heartbeatCount++;
    if (appData.heartbeatCount >= APP_HEARTBEAT_COUNT_MAX)
    {
        appData.heartbeatCount  = 0;
        appData.heartbeatToggle = true;
    }
}
```

In an interrupt-driven configuration of the Timer Driver (as was previously selected in the Timer Driver configuration options), this function will be called from an ISR context. Therefore, it must do a minimal amount of work and it must have no possibility of corrupting any data shared by the rest of the application. For this reason, the appData.HeartBeatCount variable (once initialized) must only be updated by this callback function. So, to signal that the application needs to toggle the heartbeat indicator LED, the callback function sets a Boolean flag (appData.heartbeatToggle) using an atomic (single-instruction) assignment statement, allowing the count variable to be ignored by the rest of the application logic.

With the heartbeat indicator's toggle rate managed independently by the timer driver, the APPS_Tasks function only needs to check the appData.heartbeatToggle flag and use the PORTS system service SYS_PORTS_PinToggle function to toggle the indicator LED pin (identified by the APP_HEARTBEAT_PORT and APP_HEARTBEAT_PIN macros).

Once the appData.heartbeatToggle flag is set, this logic will toggle the indicator pin and reset the flag (also done with an atomic single-instruction assignment). This could be done within the application's main switch statement, but that would complicate it unnecessarily once other states were added and the heartbeat indicator should run independently as a parallel (if extremely simple) state machine. Therefore, it is far better to implement this logic outside of the switch statement, but still within the APP_Tasks function, as shown in the following complete APP_Tasks function implementation.

Complete APP_Tasks Implementation in app.c

```
void APP_Tasks ( void )
{
    /* Signal the application's heartbeat. */
    if (appData.heartbeatToggle == true)
    {
        SYS_PORTS_PinToggle(PORTS_ID_0, APP_HEARTBEAT_PORT,
                            APP_HEARTBEAT_PIN);
        appData.heartbeatToggle = false;
    }

    /* Check the application's current state. */
    switch ( appData.state )
    {
        /* Application's initial state. */
        case APP_STATE_INIT:
        {
            appData.heartbeatTimer = DRV_TMR_Open( APP_HEARTBEAT_TMR,
                                                   DRV_IO_INTENT_EXCLUSIVE);
            if ( DRV_HANDLE_INVALID != appData.heartbeatTimer )
            {
                DRV_TMR_AlarmRegister(appData.heartbeatTimer,
                                      APP_HEARTBEAT_TMR_PERIOD,
                                      APP_HEARTBEAT_TMR_IS_PERIODIC,
                                      (uintptr_t)&appData,
                                      APP_TimerCallback);
                DRV_TMR_Start(appData.heartbeatTimer);

                appData.state = APP_STATE_IDLE;
            }
            break;
        }

        case APP_STATE_IDLE:
        {
            break;
        }

        default:
        {
            break;
        }
    }
}
```

This implementation ensures that the heartbeat indicator blinks correctly when the APP_Tasks function is being called at a sufficient rate. However, if the main loop becomes blocked and the function is not called often enough, the heartbeat indicator will blink more slowly, change its duty cycle, or stop entirely even though the timer interrupts may continue to occur at the normal rate. This behavior makes it useful as an indicator of the health of the application's state machine.

As you develop the application logic, build, run, and test the project each time you add a new bit of functionality. If it does not behave as you expect, use the debugging capabilities supported by the MPLAB X IDE and the debugger interface you have to determine what the logic actually does, and then you can fix any incorrect behavior. Once you have a working heartbeat indicator, you have successfully created your first MPLAB Harmony project.

Summary

Summarizes the teachings of this tutorial and describes where to find additional help.

Description

This tutorial has demonstrated how to create a new MPLAB Harmony project, use the MHC to configure the processor clocks, ports, and other settings, and to enable and configure MPLAB Harmony libraries. It has also provided a good starting point for your own custom applications by providing an application health *heartbeat* indicator.

Although this tutorial performs only one pass through these steps, it is generally best to build up an application module-by-module, making multiple passes through this process as needed for each new library, module, or feature you add to your system. This allows you to build up your application one piece at a time and to test each new feature as your project develops into a complete solution.

Additional Information

You are now ready to begin developing your own MPLAB Harmony applications. Please refer to the following MPLAB Harmony Help volumes for additional information:

- Volume II: MPLAB Harmony Configurator (MHC) - For details on how to use and develop the MHC
- Volume III: MPLAB Harmony Development - For a better understanding of key concepts of MPLAB Harmony and additional information on how to develop your own libraries, drivers and other items for MPLAB Harmony
- Volume IV: MPLAB Harmony Framework Reference - For interface details and information on how to use the libraries provided in the MPLAB Harmony installation
- Volume V: Third-Party Products - For information on third party products provided with the MPLAB Harmony installation
- Volume VI: Utilities - For information on utilities (used during development and testing) that are provided with the MPLAB Harmony installation

Enjoy!

Using MHC Application Templates to Create an Application

This tutorial guides you through the process of using the MPLAB Harmony Configurator (MHC) application templates to develop a MPLAB Harmony application.

Description

The MHC provides Application Templates that can be used to create a baseline working project (i.e., application).

In this tutorial, the project will be minimal, but will be able to perform a high-level function (e.g., transmit a string with a fully configured USART) without a lot of peripheral configuration and connecting code together.

The software module connections are made by the precoded application templates. These templates generate one (or more) specific applications based on the module(s) chosen (i.e., USART, SPI).

This tutorial will describe the process for using MHC to choose ‘Application Templates’, choose a function (e.g., USART transmit) and generate the code in less than a minute and be running ‘Hello World’.

- Step 1: Select Your Application Template
- Step 2: Save the Configuration
- Step 3: Generate the Code
- Step 4: Build the Code
- Step 5: Building on the Application Template
- Step 6: Integrating Multiple Modules
- Step 7: Creating Multiple Applications
- Step 8: Application Peripheral Integrity

Once you have a working application, you can modify it further to provide a more specific application to suit your needs.

Prerequisites and Assumptions

Describes the prerequisites for starting this tutorial and the assumptions made when it was written.

Description

Before beginning this tutorial, ensure that you have installed the MPLAB X IDE and necessary language tools as described in *Volume I: Getting Started With MPLAB Harmony > Prerequisites*. In addition, an appropriate hardware platform is required.

The example project in this tutorial utilizes the PIC32MZ Embedded Connectivity (EC) Starter Kit. In the event you do not have this development hardware, refer to the *Volume I: Getting Started With MPLAB Harmony > Supported Development Boards* section for a list of available development boards that you could use to complete this tutorial.

To complete this tutorial you will need to know the following about your selected hardware platform:

- The name of the PIC32 device it uses
- The Primary Oscillator input clock frequency and desired processor clock frequency
- The debugger interface settings
- The input clock source and frequency for the timer selected to blink the indicator LED

Please refer to the documentation for the selected hardware platform and PIC32 device for this information, as described in *Volume I: Getting Started With MPLAB Harmony > Guided Tour > Documentation*.

Finally, this tutorial assumes that you have some familiarity with the following; however, you should be able to follow the directions in this tutorial with very little experience:

- MPLAB X IDE development and debugging fundamentals
- C language programming
- PIC32 product family and supported development boards

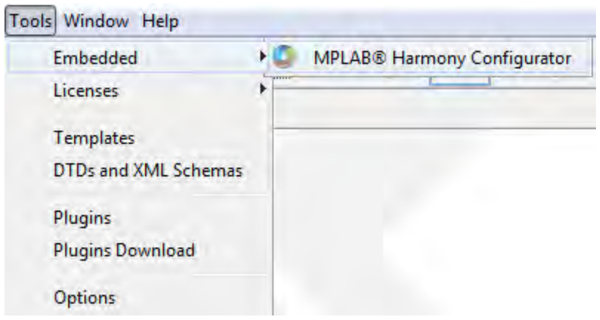
Step 1: Select Your Application Template

Describes opening MHC and selecting your application.

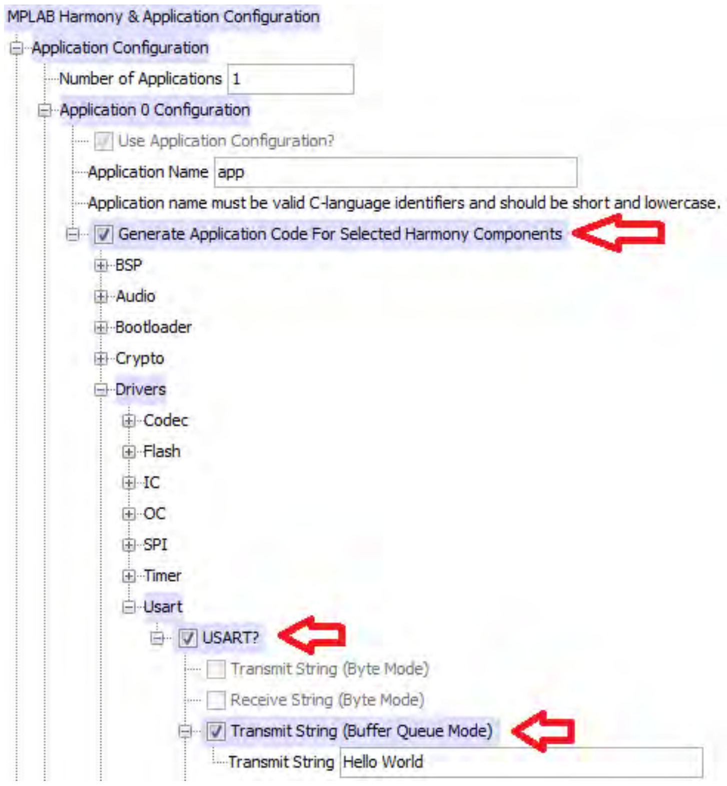
Description

Perform the following steps to select your application template:

1. In MPLAB X IDE, start the MPLAB Harmony Configurator (MHC) by selecting *Tools* > *Embedded* > *MPLAB Harmony Configurator*.



2. Within MPLAB Harmony & Application Configuration, expand *Application Configuration* > *Application 0 Configuration* and make the selections shown in the following figure.



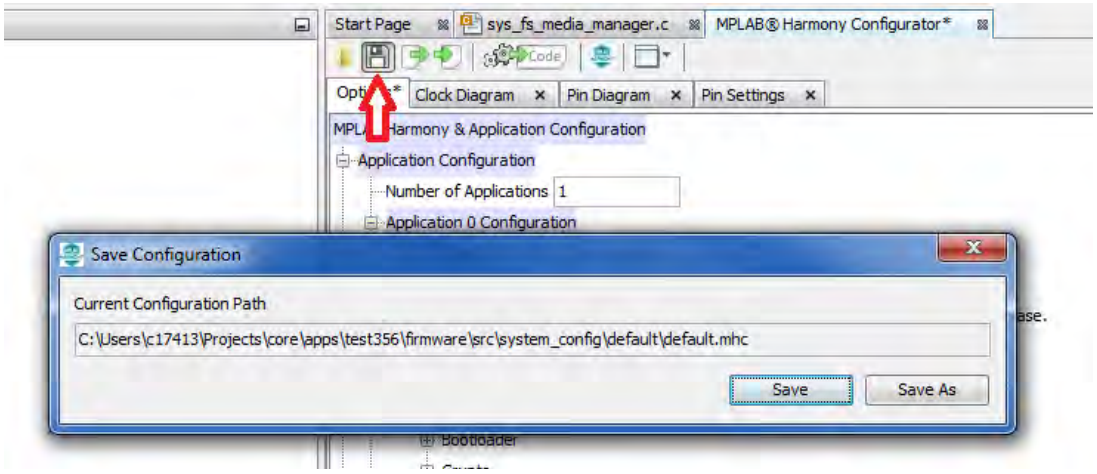
MPLAB Harmony & Application Configuration
- Application Configuration
  - Number of Applications: 1
  - Application 0 Configuration
    - [✓] Use Application Configuration?
    - Application Name: app
    - Application name must be valid C-language identifiers and should be short and lowercase.
    - [✓] Generate Application Code For Selected Harmony Components
      - BSP
      - Audio
      - Bootloader
      - Crypto
      - Drivers
        - Codec
        - Flash
        - IC
        - OC
        - SPI
        - Timer
        - Usart
          - [✓] USART?
            - [ ] Transmit String (Byte Mode)
            - [ ] Receive String (Byte Mode)
            - [✓] Transmit String (Buffer Queue Mode)
              - Transmit String: Hello World

Step 2: Save the Configuration

Describes saving the application.

Description

After making the initial selections, save the configuration by clicking the Save icon, and then clicking **Save**, as shown in the following figure.



Current Configuration Path

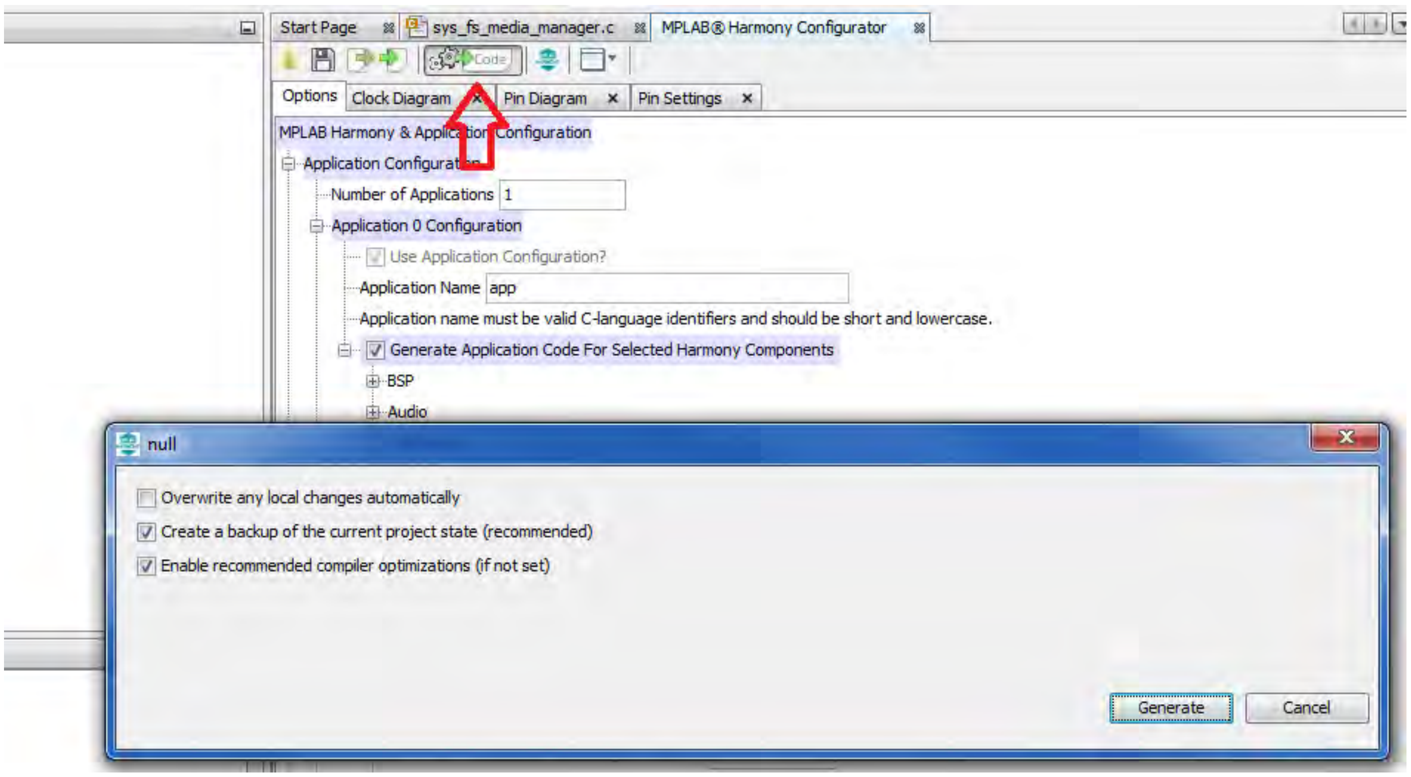
C:\Users\c17413\Projects\core\apps\test356\firmware\src\system_config\default\default.mhc

Step 3: Generate the Code

Describes how to generate the code for the new application.

Description

Next, generate the application code by clicking the Generate Code icon, and then clicking **Generate**, as shown in the following figure. Code generation should complete without errors. You now have a code base that is ready to be built.



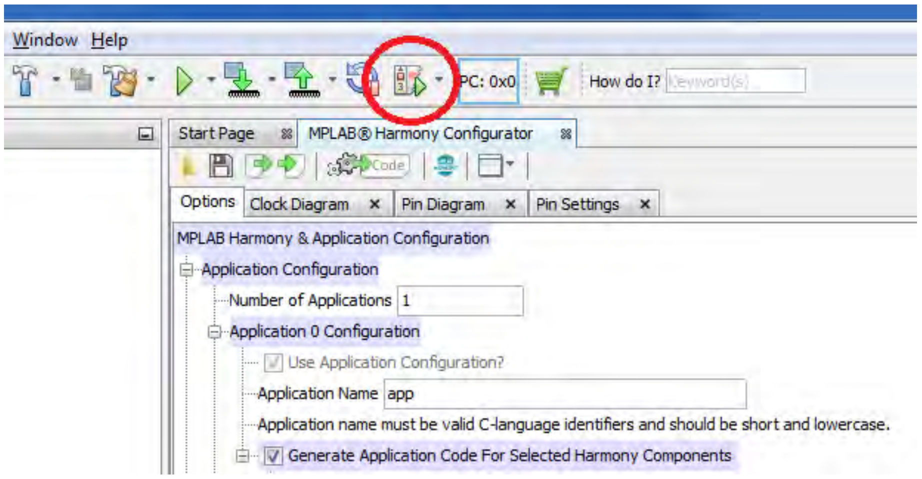
Start Page
sys_fs_media_manager.c
MPLAB® Harmony Configurator
Code
Options
Clock Diagram
Pin Diagram
Pin Settings
MPLAB Harmony & Application Configuration
Application Configuration
Number of Applications 1
Application 0 Configuration
Use Application Configuration?
Application Name app
Application name must be valid C-language identifiers and should be short and lowercase.
Generate Application Code For Selected Harmony Components
BSP
Audio
null
Overwrite any local changes automatically
Create a backup of the current project state (recommended)
Enable recommended compiler optimizations (if not set)
Generate
Cancel

Step 4: Build the Code

Describes how to build and debug the application code.

Description

The next step is the build and start a debug session by clicking the Debug icon, as shown in the following figure. Once the application is running, refer to the Help for the specific application template to see the results.



MPLAB Harmony & Application Configuration
Application Configuration
Number of Applications 1
Application 0 Configuration
Use Application Configuration?
Application Name app
Application name must be valid C-language identifiers and should be short and lowercase.
Generate Application Code For Selected Harmony Components

Step 5: Building on the Application Template

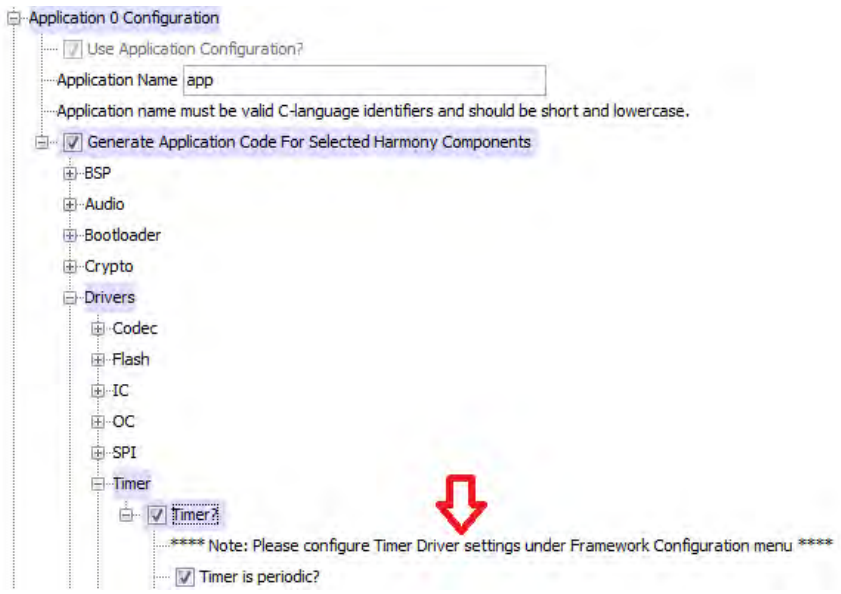
Describes how to build on the existing application template by modifying the configuration and template parameters.

Description

Modifying Configurations

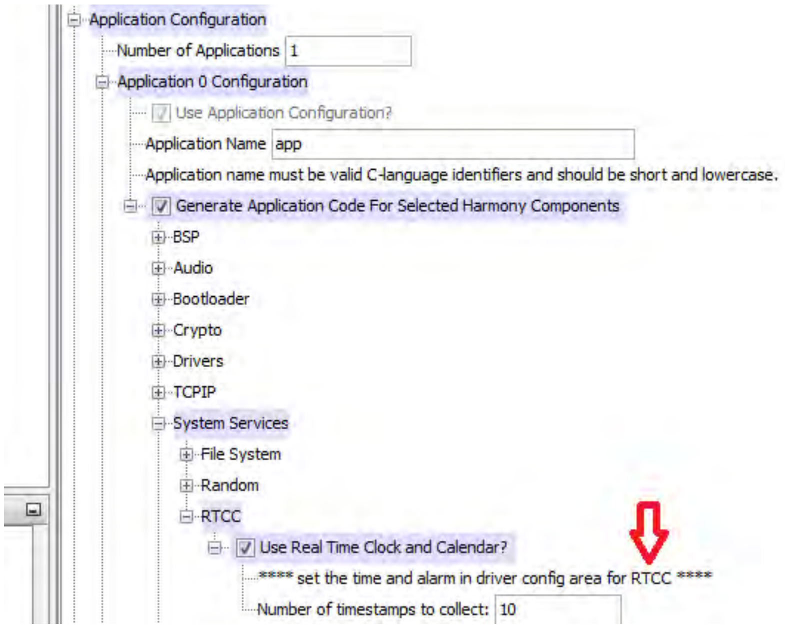
The configuration for supporting peripherals can be changed to new values, and then new code can be generated. For example, change the baud rate for the USART or change the alarm time mask for the RTCC.

Some application templates require that supporting modules be modified to be used. These templates will have a comment in the configuration menu that will state where the configuration change should be made, as shown in the following figure.


Application 0 Configuration
[✓] Use Application Configuration?
Application Name: app
Application name must be valid C-language identifiers and should be short and lowercase.
[✓] Generate Application Code For Selected Harmony Components
BSP
Audio
Bootloader
Crypto
Drivers
Codec
Flash
IC
OC
SPI
Timer
[✓] Timer?
**** Note: Please configure Timer Driver settings under Framework Configuration menu ****
[✓] Timer is periodic?

Changing Template Parameters

Some templates have parameters that can be changed, as shown in the following figure.


Application Configuration
Number of Applications: 1
Application 0 Configuration
[✓] Use Application Configuration?
Application Name: app
Application name must be valid C-language identifiers and should be short and lowercase.
[✓] Generate Application Code For Selected Harmony Components
BSP
Audio
Bootloader
Crypto
Drivers
TCPIP
System Services
File System
Random
RTCC
[✓] Use Real Time Clock and Calendar?
**** set the time and alarm in driver config area for RTCC ****
Number of timestamps to collect: 10

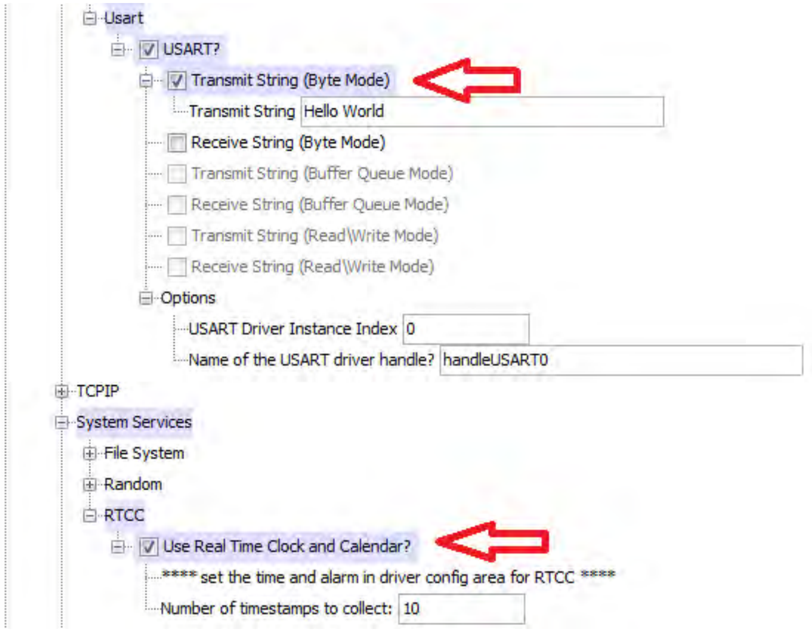
Step 6: Integrating Mutiple Modules

Describes how to integrate multiple modules into your application.

Description

When building an application, it is possible to select more than one module. For example, you can choose the SPI and USART modules in one application and modify the code with multiple modules to communicate with each other as producers and consumers.

As an example, the RTCC System Service can be added to the project as a producer, as shown in the following figure.



```
Usart
  [✓] USART?
    [✓] Transmit String (Byte Mode)
        Transmit String  Hello World
    [ ] Receive String (Byte Mode)
    [ ] Transmit String (Buffer Queue Mode)
    [ ] Receive String (Buffer Queue Mode)
    [ ] Transmit String (Read\Write Mode)
    [ ] Receive String (Read\Write Mode)
    Options
        USART Driver Instance Index  0
        Name of the USART driver handle?  handleUSART0
TCPIP
System Services
  File System
  Random
  RTCC
    [✓] Use Real Time Clock and Calendar?
        **** set the time and alarm in driver config area for RTCC ****
        Number of timestamps to collect:  10
```

After regenerating the code, modify the APP_RTCC_TIMESTAMP_Callback function to make it a ‘producer’. This function will put data into the array that the USART task checks. Then, modify the USART task to be the ‘consumer’. Note that the RTCC state machine will never quit because the code that increments the index for the timestamp array has been abandoned.

```
void APP_RTCC_TIMESTAMP_Callback(SYS_RTCC_ALARM_HANDLE handle, uintptr_t context)
{
    /* save typing and help readability */
    timestampsT *times = (timestampsT *)context;

    /* the handle will tell us it is our handle – check data sent */
    if ((handle == appHandle) && times && !appData.tx_count)
    {
        SYS_RTCC_BCD_TIME timestamp;
        SYS_RTCC_STATUS status = SYS_RTCC_TimeGet(&timestamp);
        if (SYS_RTCC_STATUS_OK == status)
        {
            /* get the seconds out that is in BDC and set the flag */
            app_tx_buf[0]= ((timestamp >> 8) & 0xF) + ‘0’;
            appData.tx_count = 1;
        }
    }
}
```

Modify the USART_Task so that it watches the tx_count, which is now just a flag for data present. If there is data, it will be sent out the USART and tx_count will be marked as clear. By default, the RTCC System Service is only running once per second.

```
static void USART_Task (void)
{
   if (appData.tx_count)
       {
           if(!DRV_USART_TransmitBufferIsFull(appData.handleUSART0))
           {
            DRV_USART_WriteByte(appData.handleUSART0, app_tx_buf[0]);
            appData.tx_count = 0;
        }
       }
}
```

Now the USART will output 0-9 at each second.

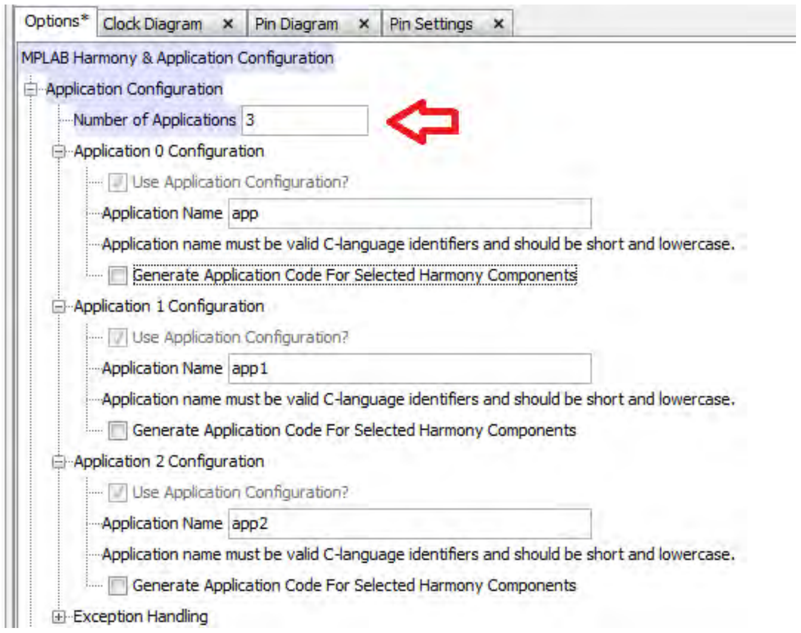
Step 7: Creating Multiple Applications

Provides information on creating multiple applications.

Description

With MHC, it is possible to create up to 10 applications. Using multiple applications allows you to perform multiple operations at the same time. For example, you can have one application read from the file system and another application can write to the file system.

Each application has its own state machine and variables with independent data scope of each other. The state machines of the individual applications run 'concurrently' with one state machine being called, and then the next instead of having one state machine to do all the work. This allows separating the work into more logical and understandable 'chunks'.


Options*
Clock Diagram
Pin Diagram
Pin Settings
MPLAB Harmony & Application Configuration
Application Configuration
Exception Handling

Number of Applications 3

Application 0 Configuration
- Use Application Configuration? ✓
- Application Name app
- Application name must be valid C-language identifiers and should be short and lowercase.
- Generate Application Code For Selected Harmony Components ☐

Application 1 Configuration
- Use Application Configuration? ✓
- Application Name app1
- Application name must be valid C-language identifiers and should be short and lowercase.
- Generate Application Code For Selected Harmony Components ☐

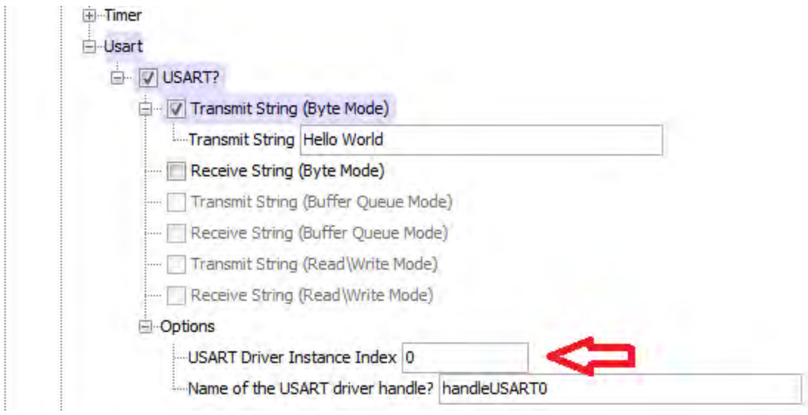
Application 2 Configuration
- Use Application Configuration? ✓
- Application Name app2
- Application name must be valid C-language identifiers and should be short and lowercase.
- Generate Application Code For Selected Harmony Components ☐

Step 8: Application Peripheral Integrity

Describes application templates that have peripherals associated by 'instances'.

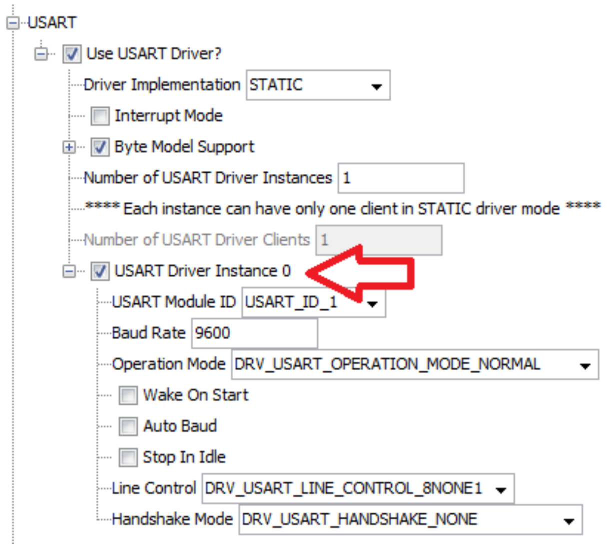
Description

Some of the application templates have peripherals associated by ‘instances’. These peripheral instances need to be configured in their respective section of the configuration tool. Make sure the peripheral instance specified in the application template configuration matches the instance number in the peripheral configuration. For the USART Application Template section the following option should be set:



```
⊞ Timer
⊟ Usart
  ⊟ [✓] USART?
    ⊟ [✓] Transmit String (Byte Mode)
        Transmit String  Hello World
      [ ] Receive String (Byte Mode)
      [ ] Transmit String (Buffer Queue Mode)
      [ ] Receive String (Buffer Queue Mode)
      [ ] Transmit String (Read\Write Mode)
      [ ] Receive String (Read\Write Mode)
    ⊟ Options
        USART Driver Instance Index  0
        Name of the USART driver handle?  handleUSART0
```

Next, connect to USART driver configuration:



```
⊟ USART
  ⊟ [✓] Use USART Driver?
      Driver Implementation  STATIC
      [ ] Interrupt Mode
    ⊞ [✓] Byte Model Support
      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
        USART Module ID  USART_ID_1
        Baud Rate  9600
        Operation Mode  DRV_USART_OPERATION_MODE_NORMAL
        [ ] Wake On Start
        [ ] Auto Baud
        [ ] Stop In Idle
        Line Control  DRV_USART_LINE_CONTROL_8NONE1
        Handshake Mode  DRV_USART_HANDSHAKE_NONE
```

Summary

Summarizes the teachings of this tutorial and describes where to find additional help.

Description

The application templates described in this tutorial are no substitute for good engineering and design in a large project, but are a valuable start to a project by providing configuration, initialization, 'glue' and an execution loop.

Break up your application into logical parts and select which parts need to be together (consume and produce) and which may benefit from being separate. Use multiple applications to support separate, unrelated functionality and combine both for a complex solution.

Additional Information

Please refer to the following MPLAB Harmony Help volumes for additional information:

- Volume II: MPLAB Harmony Configurator (MHC) - For details on how to use and develop the MHC
- Volume III: MPLAB Harmony Development - For a better understanding of key concepts of MPLAB Harmony and additional information on how to develop your own libraries, drivers and other items for MPLAB Harmony
- Volume IV: MPLAB Harmony Framework Reference - For interface details and information on how to use the libraries provided in the MPLAB Harmony installation
- Volume V: Third-Party Products - For information on third party products provided with the MPLAB Harmony installation
- Volume VI: Utilities - For information on utilities (used during development and testing) that are provided with the MPLAB Harmony installation

Enjoy!

Index