Creating Your First Project

MPLAB Harmony Integrated Software Framework

Volume I: Getting Started With MPLAB Harmony Libraries and Applications

This volume introduces the MPLAB® Harmony Integrated Software Framework.

Description

MPLAB Harmony is a layered framework of modular libraries that provide flexible and interoperable software "building blocks" for developing embedded PIC32 applications. MPLAB Harmony is also part of a broad and expandable ecosystem, providing demonstration applications, third-party offerings, and convenient development tools, such as the MPLAB Harmony Configurator (MHC), which integrate with the MPLAB X IDE and MPLAB XC32 language tools.

Legal Notices

Please review the *Software License Agreement* prior to using MPLAB Harmony. It is the responsibility of the end-user to know and understand the software license agreement terms regarding the Microchip and third-party software that is provided in this installation. A copy of the agreement is available in the `<install-dir>/doc` folder of your MPLAB Harmony installation.

The OPENRTOS® demonstrations provided in MPLAB Harmony use the OPENRTOS evaluation license, which is meant for demonstration purposes only. Customers desiring development and production on OPENRTOS must procure a suitable license. Please refer to one of the following documents, which are located in the `<install-dir>/third_party/rtos/OPENRTOS/Documents` folder of your MPLAB Harmony installation, for information on obtaining an evaluation license for your device:

- `OpenRTOS Click Thru Eval License PIC32MXxx.pdf`
- `OpenRTOS Click Thru Eval License PIC32MZxx.pdf`

TIP! Throughout this documentation, occurrences of `<install-dir>` refer to the default MPLAB Harmony installation path:
- Windows: `C:/microchip/harmony/<version>`
- Mac OS/Linux: `~/microchip/harmony/<version>`

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.

Part I: Creating Your First MPLAB Harmony Application in MHC

This section provides information on creating your first project in MPLAB Harmony.

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 project. It also shows how to create a simple "heartbeat" LED application that flashes an LED. The project created can then serve as a test bed for understanding additional features of MPLAB Harmony, including error handling, system console, and debugging services, and using MPLAB Harmony Middleware and Drivers. You can also reuse the heartbeat LED application in future projects as a simple indicator of system health.

Getting Started

Provides information on getting started with creating your first project.

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, ensure that you have installed MPLAB Harmony on your hard drive and that you have the correct MHC plug-in installed in the MPLAB X IDE.

You may want to check out your development board by first loading and running a MPLAB Harmony example that uses your board. Follow the instructions in *Volume I: Getting Started With MPLAB Harmony > Applications Help > Examples* for the demonstration you chose. Set up the board as detailed in the related User's Guide.

The example project in this tutorial can be used with any of the following boards:

- PIC32MZ Embedded Connectivity with Floating Point Unit (EF) Starter Kit (DM320007).
- PIC32 USB Starter Kit III (DM320003-3)
- Explorer 16 Development Board (DM240001-2) with PIC32MX795F512L PIM (MA320003)

The tutorial steps are equally valid on any other development board, but may be slightly different. In the event you do not have any of these boards, refer to *Volume II: Supported Hardware > Supported Development Boards* for a list of available development boards that you could use to complete this tutorial. If you are using some other development board, you will need to know what processor is on the board to select the correct Board Support Package.

Finally, this tutorial assumes that you have some familiarity with the following:

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

What You Will Learn

- How to set up your hardware
- How to create a new MPLAB Harmony project from within MPLAB X IDE
- How to use System Services, in this case Timer System Services
- How to use the Board Support Package (BSP) to toggle an LED
- How to add new application states to the application task loop
- How to run and build your project

Tutorial Steps

Describes the necessary steps to create your project.

Step 1: Setting Up Your Hardware

Provides information for setting up your hardware.

Description

PIC32MZ Embedded Connectivity (EF) Starter Kit

Connect the "USB Debug" port on the starter kit board to a USB port on your PC using a Mini-B to Type-A USB cable. See PIC32MZ Embedded Connectivity with Floating Point Unit (EF) Starter Kit for additional information on this hardware.

PIC32 USB Starter Kit III

Connect the debug port on the upper left side of the board to your PIC using a Mini-B to Type-A USB cable. Refer to the PIC32 USB Starter Kit III for additional information on this hardware.

Explorer 16 Development Board with the PIC32MX795F512L Plug-in Module (PIM)

Mount the PIC32MX795F512L PIM to PIM socket. Set switch S2 to PIM. Power the board with 9V to 15V DC using the J12 connector. Attach a REAL ICE In-circuit Emulator to the RJ12 jack on the board.

Other Boards

Consult the Information Sheet or User's Guide for your hardware. Refer to *Volume II: Supported Hardware > Supported Development Boards* for the list of hardware supported by MPLAB Harmony.

Step 2: Create a New MPLAB X IDE Project

Provides the required steps to create a new MPLAB X IDE project.

Description

Note: Prior to starting this tutorial, please ensure that the software requirements are met, as described in *Volume I: Getting Started With MPLAB Harmony >Prerequisites.*

1. Start MPLAB X IDE and select *File > New Project*. The New Project dialog appears.
2. In the New Project dialog, ensure that Microchip Embedded is selected, and that the project type is 32-bit MPLAB Harmony Project, and then click **Next**.

Note: If the option "32-Bit MPLAB Harmony Project" is not visible, you need to stop and download/install MPLAB Harmony before continuing with this tutorial.
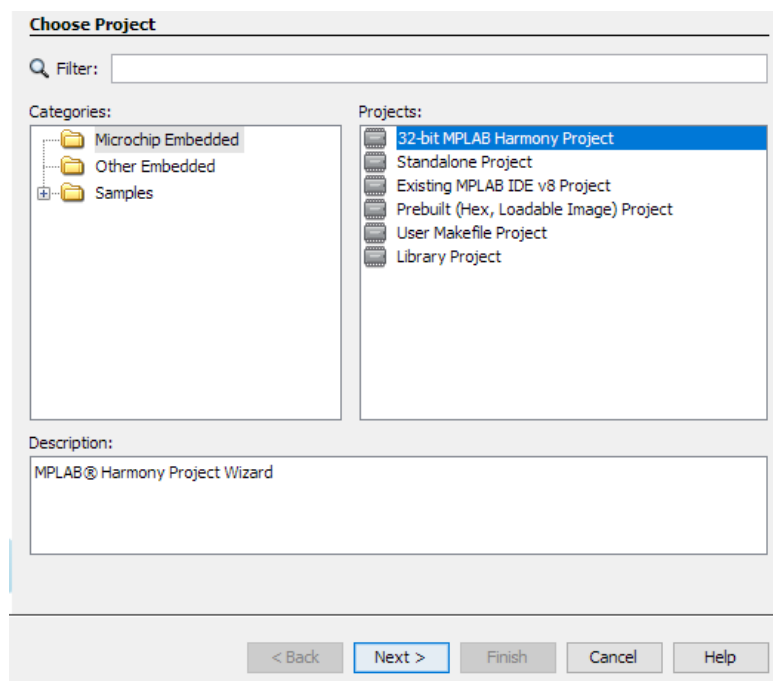
4. In the updated New Project dialog, make the following changes:
   - Harmony Path: Ensure that the path you enter is the path to your installation of MPLAB Harmony
   - Project Name: Enter heartbeat (all lowercase)
   - Device Family: Select the device family that includes your board's processor.
     - For the PIC32MZ EF Starter Kit board, select PIC32MZ

- For the PIC32 USB Starter Kit III and the Explorer 16 Development Board with PIC32MX795F512L PIM, select PIC32MX
  - Target Board: Select the board you are using (alternately, you can choose the Target Device first, and then look through a smaller list of Target Boards):
    - For the PIC32MZ EF Starter Kit board, select PIC32MZ (EF) Starter Kit. You will have to scroll down the list to find this board.
    - For the PIC32 USB Starter Kit III, select PIC32MX USB Starter Kit III.
    - For the Explorer 16 Development Board with PIC32MX795F512L Plug In Module, select PIC32MX795F512L PIM w/Explorer 16 Development Board.

5. The New Project dialog should appear, as follows. Descriptions of each field follow the image.

1: The path to your installation of MPLAB Harmony.

2: The location of your MPLAB project.

3: The name of the MPLAB project (the name must be all lowercase characters).

4: The path to the MPLAB project file.

5: The MPLAB project configuration name.

6: The selected device family (PIC32MZ or PIC32MX).

7: The selected target device.

8: The selected target board (PIC32MZ EF Starter Kit, PIC32MZ USB Starter Kit III, or PIC32MX795F5123L with the Explorer 16 Development Board).

6. Click **Finish** when done. A new empty project named heartbeat will be created in MPLAB X IDE, which opens the MPLAB Harmony Configurator (MHC) plug-in.

Note: The selected Board Support Package (BSP) assigns device pins to various board functions and sets up the device's clock tree based on the board's clock source.

You can review the pin assignments using the *Pin Diagram* or *Pin Settings* tabs in MHC. The *Clock Diagram* tab shows the board and application clock setup.
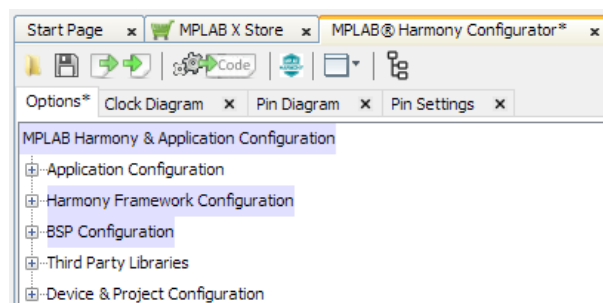

Step 3: Configure MPLAB Harmony and the Application

Describes how to configure MPLAB Harmony and the application.

Description

Within the MPLAB Harmony Configurator window, change the Application Name from "app" to "heartbeat". Be sure to make the new application name all lowercase. The name is reused in source code for function and data type definitions and using lowercase will stay consistent with the naming conventions used in MPLAB Harmony code.
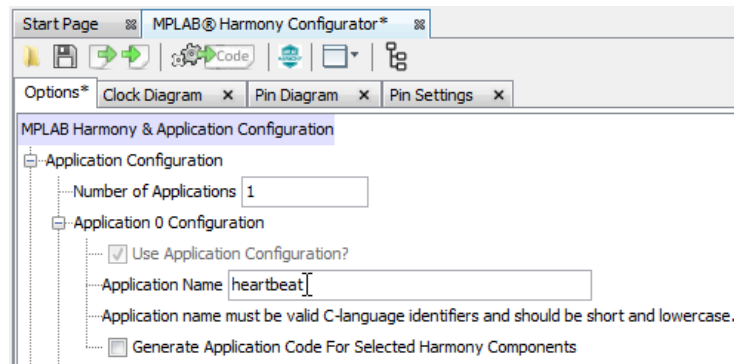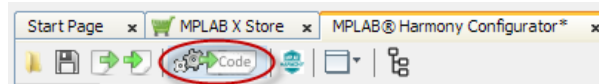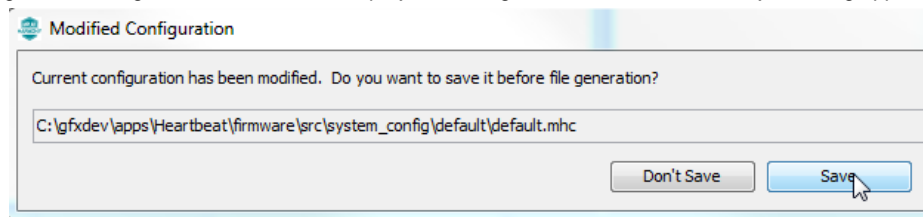

Step 4: Generate the Configured Source Code

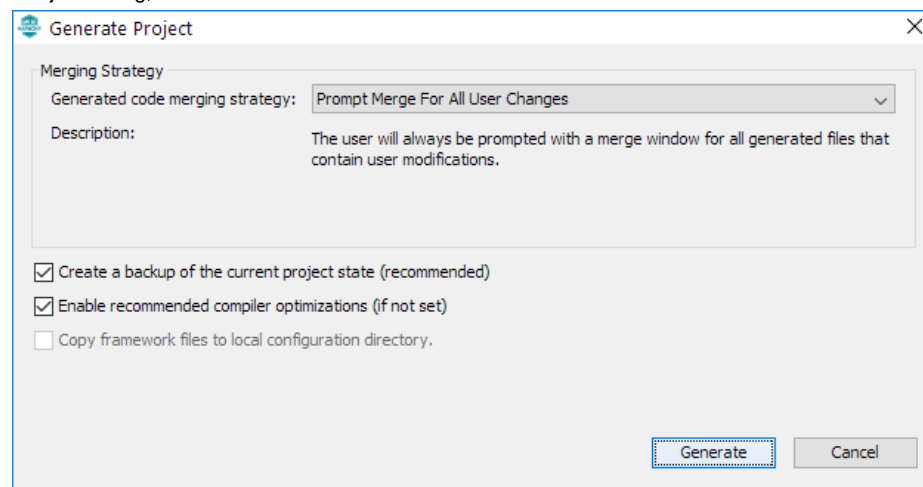Describes how to generate the configured source code.

Description

1. In MHC, click **Generate Code** to generate the application's code for the first time.


2. In the Modified Configuration dialog, click **Save** to save the project's configuration. The Generate Project dialog appears.


3. Next, in the Generate Project dialog, click **Generate.**

At this point, the project's initial software has been configured. Let's examine the software just created in the **Projects** panel of MPLAB X IDE, by expanding the Header Files and Source Files folders. Note the icons used in this image of the project's organization make it seem like the files of the project are organized this way. Actually, this is a virtual organization of these files, not an actual one.
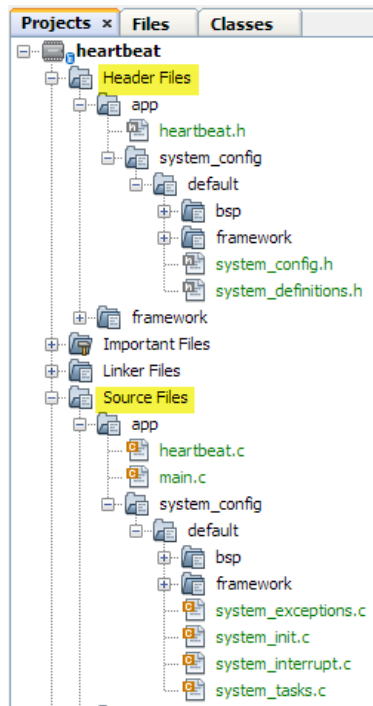
If you click the Files tab you will see the actual organization of these files on your drive.
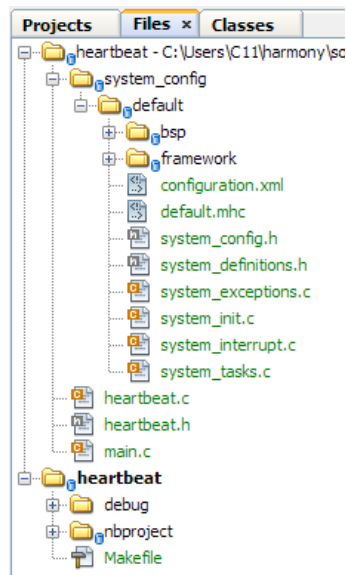

Step 5: Use a Delay Timer to Toggle an LED on the Target Board

Describes how to use a delay timer to toggle an LED.
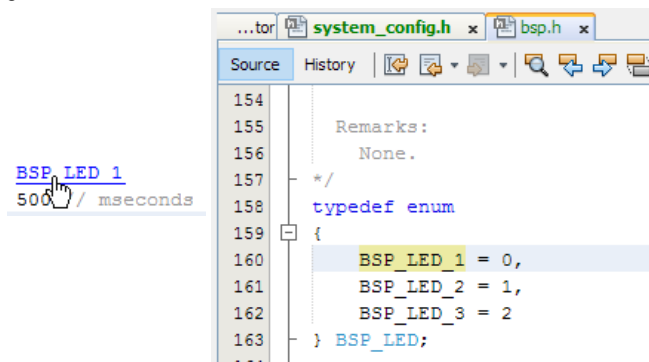
Description

In this project we will use a delay timer to toggle an LED on the board using a delay of 500 milliseconds and LED1 on the PIC32MZ EF Starter Kit.

1. Double click `system_config.h` to open the file in an editor.
2. Add the following code to the end of the file, immediately after the line `/*** Application Instance 0 Configuration ***/`.
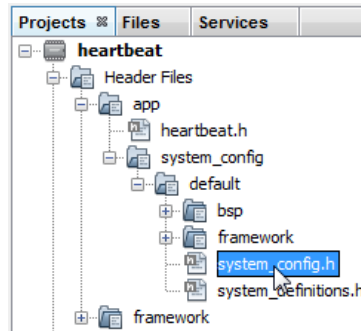
```
// CUSTOM CODE - DO NOT DELETE
#define HEARTBEAT_LED   BSP_LED_1
#define HEARTBEAT_DELAY 500 // milliseconds
// END OF CUSTOM CODE
```

2. In the `system_config.h` file within the editor, hold down the CTRL key and click **BSP_LED_1**. The editor will locate where this token is defined in the Board Support Package `bsp.h` file for the PIC32MZ EF Starter Kit.
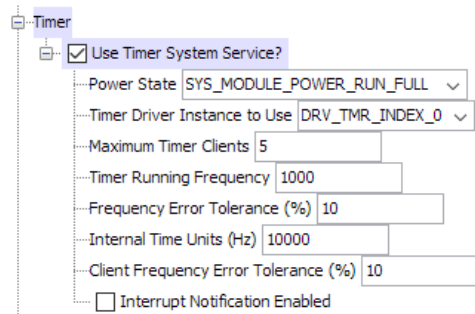

Step 6: Add the Timer System Service to Your Project

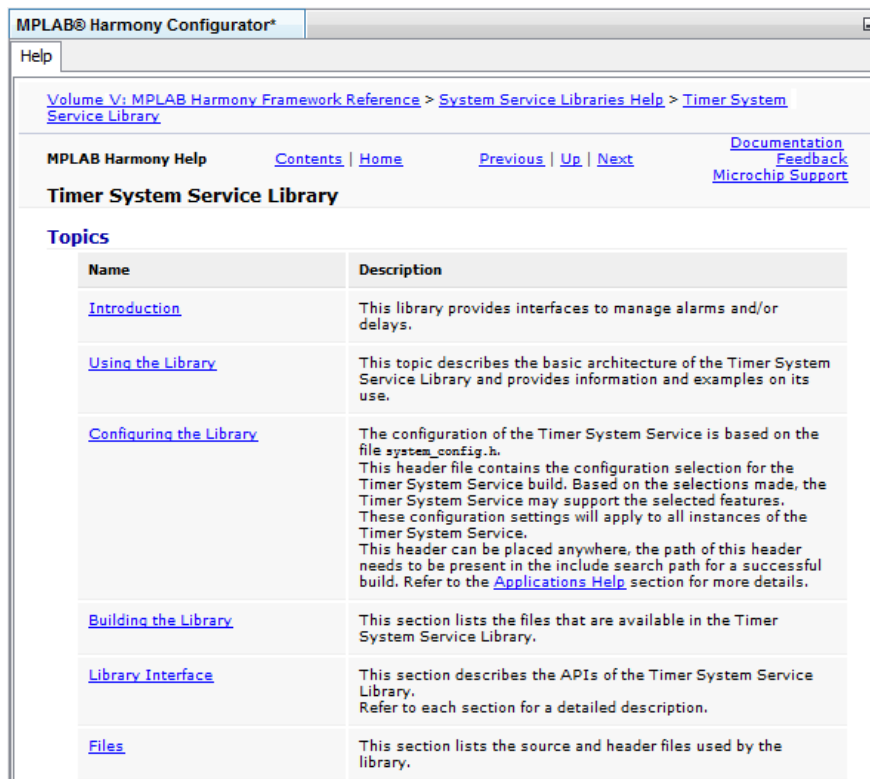Describes how to add the Timer System Service to your project.

Description

Next, the Timer System Service needs to be selected in MHC.

1. Select the MHC tab, expand *Harmony Framework Configuration* > *System Services* > *Timer*, and then select **Use Timer System Service?**

2. As observed in the Help window, the documentation for the Timer System Service is displayed. Using the Help, we can explore what this library provides and choose how to implement the timer delay we need to blink LED1. Click **Library Interface** and scroll down to Timed Delay Functions.

The SYS_TMR_DelayMS function can be used to create a one-shot delay timer, and then poll that timer status using SYS_TMR_DElayStatusGet. When the timer times out, we can then toggle the LED.

3. Click **SYS_TMR_DelayMS** to open the related Help for this function.

The code example in the documentation provides all that is needed to create the delay. First, the SYS_TMR_HANDLE variable is needed, which is assigned when the timer is created. Then, use SYS_TMR_DelayStatusGet to poll whether the timer has timed out using this handle. So now, we know what to do.

Step 7: Add the Timer System Service Source Code to Your Project

Describes how to add the source code for the Timer System Service to your project.

Description

1. Before adding the Timer to the application, we need to regenerate the application to add the Timer System Service library to our code, using the same process as described in Generate the Configured Source Code. The merge will open a difference window for `system_config.h` that was modified, as described in Add the Timer System Service to Your Project. Accept all the changes using the icon shown in the following
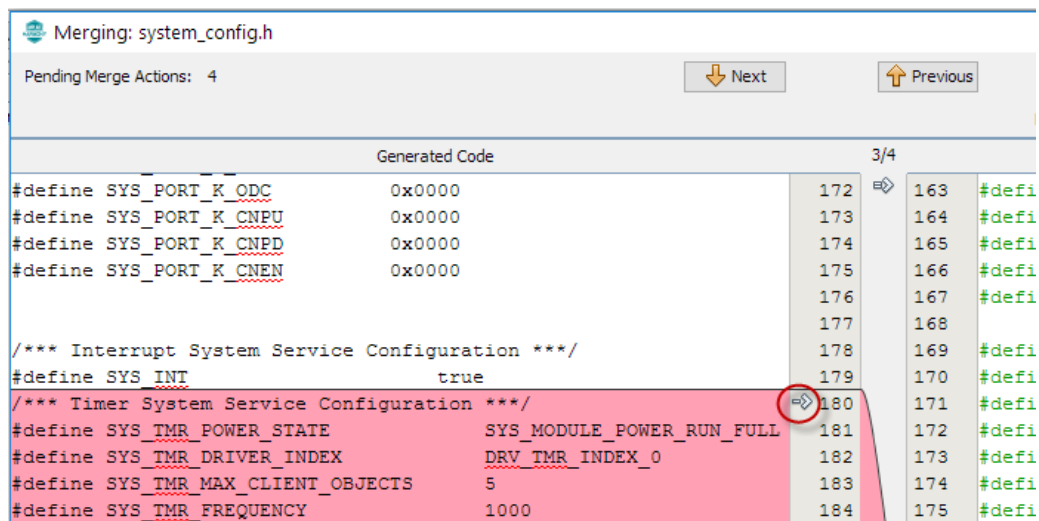
figure.

- The next figure shows the customer code that was added previously, which we want to retain. Therefore, *do not* click the icon for this merge.
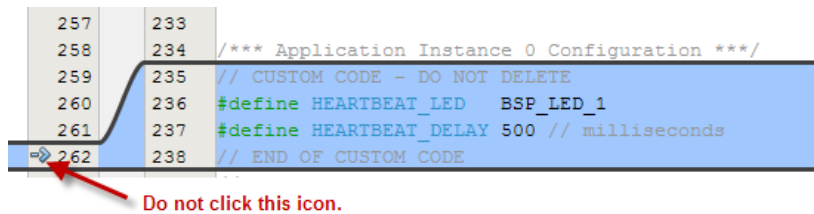

Do not click this icon.

Step 8: Use the Timer System Service in Your Application

Describes how to use the Timer System Service in your application.

Description

1. Next, from the Project tab in MPLAB X IDE, double-click the heartbeat.h file to open it in the editor.
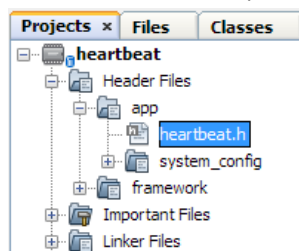
2. Then, add the new state, HEARTBEAT_RESTART_TIMER, to the application's state enumeration, as shown in the following figure. We will show how that state is used later in the tutorial.

```
87   typedef enum
88   {
89       /* Application's state machine's initial state. */
90       HEARTBEAT_STATE_INIT=0,
91       HEARTBEAT_STATE_SERVICE_TASKS,
92
93       /* TODO: Define states used by the application state machine. */
94       HEARTBEAT_RESTART_TIMER
95
96   } HEARTBEAT_STATES;
```

```
// HEARTBEAT_STATES:
    HEARTBEAT_RESTART_TIMER
```

3. Now, add the delay timer handle, `SYS_TMR_HANDLE hDelayTimer; // Handle for delay timer`, to the application's data structure, as shown in the following figure.

```
112     typedef struct
113   {
114         /* The application's current state */
115         HEARTBEAT_STATES state;
116
117         /* TODO: Define any additional data used by the application. */
118         SYS_TMR_HANDLE hDelayTimer;   // Handle for delay timer
119
120   } HEARTBEAT_DATA;
```
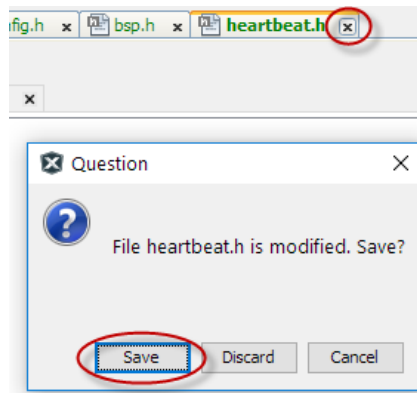
```
// HEARTBEAT_DATA
    SYS_TMR_HANDLE hDelayTimer;   // Handle for delay timer
```

4. Close and save `heartbeat.h` by clicking the 'x', and then clicking **Save**.

5. Next, from the Projects tab in MPLAB X IDE, double-click `heartbeat.c` from the `Source Files > app` folder to open it in the editor. We need to update `heartbeat.c` to add the first timer delay, execute the time-out wait, and restart timer code. Refer to the following figure for the locations to insert the different code blocks.

- Insert the following code to start the first delay timer

```
heartbeatData.hDelayTimer = SYS_TMR_DelayMS(HEARTBEAT_DELAY);
if (heartbeatData.hDelayTimer != SYS_TMR_HANDLE_INVALID)
{ // Valid handle returned
    BSP_LEDOn(HEARTBEAT_LED);
    heartbeatData.state = HEARTBEAT_STATE_SERVICE_TASKS;
}
```

- Insert the following code to wait for a time-out

```
if (SYS_TMR_DelayStatusGet(heartbeatData.hDelayTimer))
{ // Single shot timer has now timed out.
    BSP_LEDToggle(HEARTBEAT_LED);
    heartbeatData.state = HEARTBEAT_RESTART_TIMER;
}
```

- Finally, insert the following code to add the state to the restart timer.

```
case HEARTBEAT_RESTART_TIMER:
{ // Create a new timer
    heartbeatData.hDelayTimer = SYS_TMR_DelayMS(HEARTBEAT_DELAY);
    if (heartbeatData.hDelayTimer != SYS_TMR_HANDLE_INVALID)
    { // Valid handle returned
        heartbeatData.state = HEARTBEAT_STATE_SERVICE_TASKS;
    }
    break;
}
```

```
135     void HEARTBEAT_Tasks ( void )
136  ┌  {
137
138          /* Check the application's current state. */
139          switch ( heartbeatData.state )
140          {
141              /* Application's initial state. */
142              case HEARTBEAT_STATE_INIT:
143              {
144                  bool appInitialized = true;
145
146
147                  if (appInitialized)
148                  {                          <-----  Insert the start of the first timer delay code here
149
150                      heartbeatData.state = HEARTBEAT_STATE_SERVICE_TASKS;
151                  }
152                  break;
153              }
154
155              case HEARTBEAT_STATE_SERVICE_TASKS:
156              {                          <-----  Insert wait for time-out code here
157
158                  break;
159              }
160
161              /* TODO: implement your application state machine.*/
162                                         <-----  Insert add state to restart timer code here
163
164              /* The default state should never be executed. */
165              default:
166              {
167                  /* TODO: Handle error in application's state machine. */
168                  break;
169              }
170          }
171  └  }
```

6. Once you have finished inserting the code blocks, the `heartbeat.c` file should appear like the following figure.

```
1   /*******************************************************************
2     Function:
3       void HEARTBEAT_Tasks ( void )
4
5     Remarks:
6       See prototype in heartbeat.h.
7   */
8
9   void HEARTBEAT_Tasks ( void )
10  {
11
12      /* Check the application's current state. */
13      switch ( heartbeatData.state )
14      {
15          /* Application's initial state. */
16          case HEARTBEAT_STATE_INIT:
17          {
18              bool appInitialized = true;
19
20
21              if (appInitialized)
22              {
23                  heartbeatData.hDelayTimer = SYS_TMR_DelayMS(HEARTBEAT_DELAY);
24                  if (heartbeatData.hDelayTimer != SYS_TMR_HANDLE_INVALID)
25                  { // Valid handle returned
26                      BSP_LEDOn(HEARTBEAT_LED);
27                      heartbeatData.state = HEARTBEAT_STATE_SERVICE_TASKS;
28                  }
29                  heartbeatData.state = HEARTBEAT_STATE_SERVICE_TASKS;
30              }
31              break;
32          }
33
34          case HEARTBEAT_STATE_SERVICE_TASKS:
35          {
36              if (SYS_TMR_DelayStatusGet(heartbeatData.hDelayTimer))
37              { // Single shot timer has now timed out.
38                  BSP_LEDToggle(HEARTBEAT_LED);
39                  heartbeatData.state = HEARTBEAT_RESTART_TIMER;
40              }
41              break;
42          }
43
44          /* TODO: implement your application state machine.*/
45          case HEARTBEAT_RESTART_TIMER:
46          { // Create a new timer
47              heartbeatData.hDelayTimer = SYS_TMR_DelayMS(HEARTBEAT_DELAY);
48              if (heartbeatData.hDelayTimer != SYS_TMR_HANDLE_INVALID)
49              { // Valid handle returned
50                  heartbeatData.state = HEARTBEAT_STATE_SERVICE_TASKS;
51              }
52          }
53
54          /* The default state should never be executed. */
55          default:
56          {
57              /* TODO: Handle error in application's state machine. */
58              break;
59          }
60      }
61  }
62
```

7. After updating the code, close and save `heartbeat.c`.

Step 9: Build and Run Your Project

Describes how to build and run your project.

Description

1. For the PIC32MZ EF Starter Kit, click the Run Main Project icon to build and run your project in MPLAB X IDE. If prompted, select the on-board debugger to load the project.
   - For the PIC32 USB Starter Kit III, select the PICkit On Board (PKOB) debugger
   - For the Explorer 16 Development Board, select REAL ICE
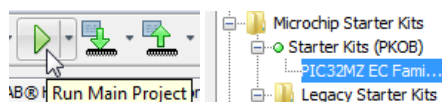
3. After making these selections click **OK** and close the Properties window.
4. Run the project using the Run Main Project button.

For both boards, the LED should flash with a one second period.

Part II: Debugging With Your Project

This section discusses how to debug problems in your project from within MPLAB's debugger

Overview

This section discusses how to debug problems in your project from within the MPLAB X IDE debugger. Part III: Debugging While Running Stand-alone discusses how to debug problems when running without the debugger, including using diagnostic messages to a HyperTerminal equivalent application on your computer.

Description

Two important tools in debugging any embedded software application are asserts and exception handling. By default, asserts in the PIC32 code write out an error message to USART2, and then jump into a `while(1){}` loop. However, if you have not set up USART2 you have no information. Even with USART2 set up, you can miss the message if your HyperTerminal isn't set up correctly. By default, exceptions (e.g., divide by zero) cause the application to jump into a `while(1){}` loop, preventing the application from continuing, but providing no additional information. Therefore, in both cases your application stops working and you have no idea why.

As a first step in developing any new application, you will be writing code and debugging it using the MPLAB debugger. This lesson shows you how to enable asserts and exception handling while running the debugger, so that you don't have to setup USART2. The next tutorial will show how to support asserts and exception handling outside of the debugger. It will also show how to add other diagnostic messages to aid in debugging.

Getting Started

Provides information on getting started with project debugging.

Description

The following steps can be applied to any MPLAB Harmony-based project, but for the sake of clarity it is assumed that you have completed Part I: Creating Your First MPLAB Harmony Application in MHC. The project created in Part I will be used as the basis for this lesson, s it will be necessary to set up your board using the instructions from that tutorial.

What You Will Learn

- How to enable asserts from the debugger
- How to enable Harmony's built-in exception handler
- How to decode the information reported by the exception handler to find where exception occurred in your code and what type of exception it was
- How to test asserts and the exception handler
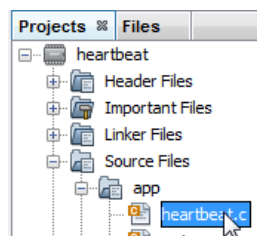
Tutorial Steps

This part of the tutorial explores how to use the debugger with your project.

Asserts Under the Debugger

This section explores how to use the debugger with asserts.

Description

1. Launch MPLAB X IDE and load the project you created in Part I: Creating Your First MPLAB Harmony Application in MHC.
2. Open `heartbeat.c`.

3. Add `assert(0);` to the start of the HEARTBEAT_Initialize function, as shown in red in the following code example.

```
void HEARTBEAT_Initialize ( void )
{
 assert(0);


/* Place the App state machine in its initial state. */
heartbeatData.state = HEARTBEAT_STATE_INIT;
```

4. Build and run the application. You will see that the LED no longer flashes. This is because the assert(0) fired, and the application is now in an infinite loop within the compiler's built-in assert function. However, if we hadn't installed an assert(0) in the code in the first place, how would we know what happened? This is where the debugger can help.

5. As shown in the following figure, if you press and hold the Ctrl key and hover your cursor over the assert call, the Macro assert appears. However, where is this `#define` located?
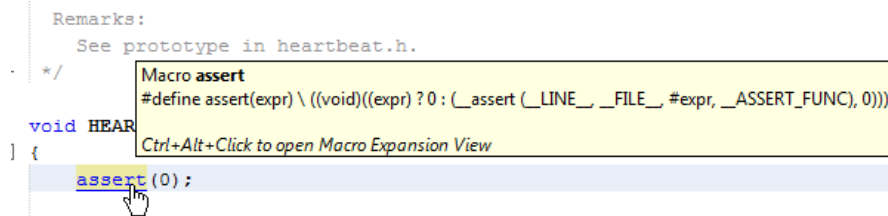

6. Press and hold the Ctrl key and click the assert call. This will open the `assert.h` file that provides the definition of the assert. As shown in the next example, in the header file you can see how the assert is defined.

```
extern void __attribute__((noreturn)) _fassert(int, const char *, const char *, const char*);
#define __assert(line,file,expression,func)  \
  _fassert(line,file,expression,func)
#define assert(expr)     \
  ((void)((expr) ? 0 : (__assert (__LINE__, __FILE__, #expr, __ASSERT_FUNC), 0)))
```

7. The function _fassert is the built-in assert handler provided by the compiler. Modify the file to allow the debugger to fire a breakpoint when running the debugger (when defined(__DEBUG) is true) by adding the code shown in red in the following example.

```
#if defined(NDEBUG) || !defined(__DEBUG)
# define __conditional_software_breakpoint(X) ((void)0)
#else
# define __conditional_software_breakpoint(X) \
((X) ? \
(void) 0 : \
__builtin_software_breakpoint())
// Added to support debugger:
# undef assert
# define assert(expr) __conditional_software_breakpoint(expr)
// End of addition
#endif
```

8. Save your edits to `assert.h` by pressing Ctrl+S and closing the window.

9. Build and run the project under the debugger by clicking **Debug Project**.
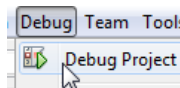
10. The debugger should now stop at the assert(0) call. So by a slight modification to the compiler's `assert.h` file, the debugger now stops with a breakpoint at the location of the failing assert.

```
115     void HEARTBEAT_Initialize ( void )
116   {
        assert(0);
118
119         /* Place the App state machine in its initial state. */
120         heartbeatData.state = HEARTBEAT_STATE_INIT;
121
122
123         /* TODO: Initialize your application's state machine and other
124          * parameters.
125          */
126   }
```

For more information on how to use the built-in debugger in the MPLAB X IDE, refer to the IDE's built-in help (*Help Menu > Tool Help Contents > MPLAB X IDE >*)for these topics:

- Tutorial > Running and Debugging Code
- Basic Tasks > Debug Run code
- Basic Tasks > Control Program Execution with Breakpoints
- Basic Tasks > Step Through Code
- Basic Tasks > Watch Symbol Value Change
- Basic Tasks > Watch Local Variable Values Change

SYS_ASSERT Macro

MPLAB Harmony uses a SYS_ASSERT macro in many places. Other libraries may have a localized assert. For example, the Graphics Library has its own macro GFX_ASSERT, which can help with debugging graphics development. By default, these macros are not defined. You can turn SYS_ASSERT on by the simply including the following code in your `system_config.h` file, which is located in `Header Files > app > system_config > default`).

```
/*** Application Instance 0 Configuration ***/
// CUSTOM CODE - DO NOT DELETE
#define HEARTBEAT_LED BSP_LED_1
#define HEARTBEAT_DELAY 500 // milliseconds
#if defined( SYS_ASSERT )
#undef SYS_ASSERT
#endif
#define SYS_ASSERT(test,message) assert(test)

// END OF CUSTOM CODE
```

This code converts all of the MPLAB Harmony SYS_ASSERTs found into simple assert calls. However, this can greatly affect how the code works, depending on where the SYS_ASSERTs are located. Therefore, this method is best used sparingly.

For more information on the SYS_ASSERT macro, refer to *Volume V: MPLAB Harmony Framework > System Services Library Help > System Service Overview > Using the SYS_ASSERT Macro*. By default, SYS_ASSERT is not defined. There are two alternatives provided in the MPLAB Harmony documentation, one for the debugger and a second for running outside of the debugger (stand-alone). Combining these two yields:

```
#include "system/debug/sys_debug.h"

#if !defined(NDEBUG)
  /*** SYS_DEBUG_Breakpoint Definition ***/
  #if defined(__DEBUG)
    #define SYS_DEBUG_Breakpoint()   __asm__ volatile (" sdbbp 0")
  #else
    #define SYS_DEBUG_BreakPoint()
  #endif//defined(__DEBUG)

  /*** SYS_ASSERT Definition ***/
  #if defined( SYS_ASSERT )
  //Remove prior definition - necessary to prevent ugly builds
    #undef SYS_ASSERT
  #endif
  #if defined(__DEBUG)
  //SYS_ASSERT for the debugger
    #define SYS_ASSERT(test, message) \
            do{ if(!(test)) SYS_DEBUG_Breakpoint(); }while(false)
  #else
  //SYS_ASSERT for Standalone:
    #define SYS_ASSERT(test, message) \
            do{ if(!(test)){ \
                    SYS_MESSAGE((message)); \
                    SYS_MESSAGE("\r\n"); \
                while(1);} \
```

```
                }while(false)
    #endif//defined(__DEBUG)


#endif//!defined(NDEBUG)
```

The details of how to enable SYS_MESSAGE to allow output to a HyperTerminal are discussed in *Part II: Debugging With Your First Project >
Part III: Debugging While Running Stand-alone*.

Exception Handling in the Debugger

This section explores how the debugger can be used in exception handling.

Description

1. The first step in exploring how MPLAB Harmony handles exceptions is to verify that the exception handler is enabled. Before launching MHC, the project must be the Main Project within the MPLAB X IDE. To set the project as the Main Project, right click the project name, and select **Set as Main Project**.

2. Start MHC by clicking the MPLAB Harmony icon ( ). If this icon is not visible, this indicates the MHC plug-in is not installed (refer to *Volume III: MPLAB harmony Configurator (MHC) > MPLAB Harmony Configurator User's Guide > Installing MHC* to install MHC).

3. Verify that the MPLAB Harmony Exception Handler will be used. If correctly configured, the project should have the file `system_exception.c` within *Source Files > app > system_config > default* in the MPLAB X IDE Project tab. If this file is missing, go to MHC and select **Use MPLAB Harmony Exception Handler Template?** to enable MPLAB Harmony's exception handler. Then, regenerate the application's code to add Harmony's exception handler.
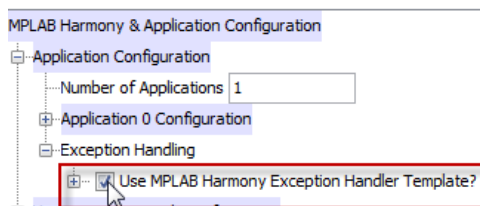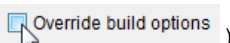

4. Next, we need to create an exception in the code to observe how exceptions are handled. Replace the `assert(0);` in HEARTBEAT_Initialize with the following code. However, if we jump to trying out this code in the debugger, nothing will happen. Under the default optimization level (-01) the compiler will recognize that this code does not do anything useful, and will not include it in the build. Therefore, we will have to change the optimization level for the file to zero before proceeding.
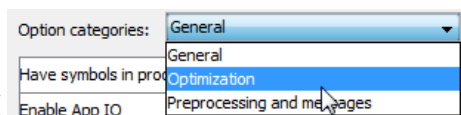
```
// Test out error handling under Optimization Level Zero for system_init.c
{
    uint8_t x, y, z;
    x = 1;
    y = 0;
    z = x/y;
}
```
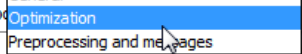
5. Right click `heartbeat.c` and choose Properties from the resulting menu and make the following selections:

   • In File Properties, select Override build options ( )

   • Next, select the compiler ( )

   • Within the Optimization category ( ), set the optimization level to zero

   After making the selections, click **OK** to close the window

6. Next, build and run the application under the debugger. The application should stop at the debugger breakpoint in `system_exceptions.c`.

```
124     void _general_exception_handler ( void )
125     {
126         /* Mask off Mask of the ExcCode Field from the Cause Register
127         Refer to the MIPs Software User's manual */
128         _excep_code = (_CP0_GET_CAUSE() & 0x0000007C) >> 2;
129         _excep_addr = _CP0_GET_EPC();
130         _cause_str  = cause[_excep_code];
131
132         SYS_DEBUG_PRINT(SYS_ERROR_ERROR, "\n\rGeneral Exception %s (cause=%d, addr=%x).\n\r",
133                         _cause_str, _excep_code, _excep_addr);
134
135         while (1)
136         {
⇨               SYS_DEBUG_BreakPoint();
138         }
139     }
```

Hovering your cursor above the variable, _excep_code, will reveal the exception code.

```
Address = 0x800003AC,   _excep_code = 0x0000000D
_excep_code = (_CP0_GET_CAUSE() & 0x0000007C) >> 2;
```

In this case, 0xD or 13, corresponds to an arithmetic trap (see the following table for a list of PIC32 Exception Codes. Hovering your cursor over the variable, _excep_addr, will reveal where in the code the exception occurred.

```
Address = 0x800003A8,   _excep_addr = 0x9D0025D8
_excep_addr = _CP0_GET_EPC();
```

Now we need to find out where 0x9D00_25D8 is located in the code (this address may be different for your appliation). Before going to the next

step, stop the debugger session by pressing Shift+F5 or by clicking the Finish Debugger Session icon ( ).

Exception Codes for PIC32

```
typedef enum {
EXCEP_IRQ = 1, // interrupt (coded as zero)
EXCEP_AdEL = 4, // address error exception (load or ifetch)
EXCEP_AdES = 5, // address error exception (store)
EXCEP_IBE = 6, // bus error (ifetch)
EXCEP_DBE = 7, // bus error (load/store)
EXCEP_Sys = 8, // syscall
EXCEP_Bp = 9, // breakpoint
EXCEP_RI = 10, // reserved instruction
EXCEP_CpU = 11, // coprocessor unusable
EXCEP_Overflow = 12, // arithmetic overflow
EXCEP_Trap = 13, // trap (possible divide by zero)
EXCEP_IS1 = 16, // implementation specfic 1
EXCEP_CEU = 17, // CorExtend Unuseable
EXCEP_C2E = 18, // coprocessor 2
} EXCEPTION_CODES;
```

7. If the debugger is inside of a function, you can look at a disassembly of the code, but this is impractical when you don't know where to look for the cause of the exception. Instead, you can build a list of all the application's assembly code at build time. (Of course, this step can cause the build to take longer, so only use it when trying to debug an exception.)
   - Right click the project name and select **Properties**
   - Within the Building properties, enable **Excecute this line after build**, and enter the following text (Windows):
     - `${MP_CC_DIR}\xc32-objdump -S ${ImageDir}\${PROJECTNAME}.${IMAGE_TYPE}.elf > disassembly.lst`
     - For Linux: `${MP_CC_DIR}/xc32-objdump -S ${ImageDir}/${PROJECTNAME}.${IMAGE_TYPE}.elf > disassembly.lst`
   - At the end, the window should show:

   ☑ Execute this line after build
   `${MP_CC_DIR}\xc32-objdump -S ${ImageDir}\${PROJECTNAME}.${IMAGE_TYPE}.elf > disassembly.lst`

- Click **OK** to finish.
8. Run the project under the debugger again. When the breakpoint fires verify that the address is the same as before.

9. Now we need to examine the `disassembly.lst` file that was just generated. This file is located in the `./heartbeat/firmware/heartbeat.X` folder. Load the file with your favorite text editor and search for 9D0025D8 (or the address you found) in the listing. The following example illustrates what you should see:

```
void HEARTBEAT_Initialize ( void )
{
9d0025b4:    27bdfff0    addiu    sp,sp,-16

9d0025b8 <.LCFI0>:
9d0025b8:    afbe000c    sw    s8,12(sp)
9d0025bc:    03a0f021    move    s8,sp

9d0025c0 <.LBB2>:
    // Test out error handling under Optimization Level Zero for system_init.c
    {
        uint8_t x, y, z;
        x = 1;
9d0025c0:    24020001    li    v0,1
9d0025c4:    a3c20000    sb    v0,0(s8)

9d0025c8 <.LVL0>:
        y = 0;
9d0025c8:    a3c00001    sb    zero,1(s8)

9d0025cc <.LVL1>:
        z = x/y;
9d0025cc:    93c30000    lbu    v1,0(s8)
9d0025d0:    93c20001    lbu    v0,1(s8)

9d0025d4 <.LVL2>:
9d0025d4:    0062001b    divu    zero,v1,v0
9d0025d8:    004001f4    teq    v0,zero,0x7  <----- Exception Address
9d0025dc:    00001010    mfhi    v0
9d0025e0:    00001012    mflo    v0
9d0025e4:    a3c20002    sb    v0,2(s8)

9d0025e8 <.LBE2>:
    }
```

So the exception occurred during the assembly execution of the C instruction z = x/y, as expected.

10. Before proceeding, comment out the exception code in the `heartbeat.c` file. You could delete it from the file, but leaving it in as a comment provides a convenient way to validate that the exception handler is working; just uncomment and run to verify it still works as expected.

11. You should also remove the Override build options from `heartbeat.c`, returning it back to the projects default of Optimization Level One (-O1).

> **Note:** You might expect from the code in `system_exceptions.c` that it would also print out a message reporting the exception, but pressing and holding the Ctrl key while hovering your cursor reveals that SYS_DEBUG_PRINT is not defined, so nothing really happens in the code.
>
> Enabling this feature is discussed in the next tutorial.

```
void _general_exception_handler ( void )
{
    /* Mask off Mask of the ExcCode Field from the Cause Register
    Refer to the MIPs Software User's manual */
    _excep                                     000007C) >> 2;
    _excep    Macro SYS_DEBUG_PRINT
    _cause    #define SYS_DEBUG_PRINT

             Ctrl+Alt+Click to open Macro Expansion View
    SYS_DEBUG_PRINT(SYS_ERROR_ERROR, "\n\rGeneral Exception %s (cause=%d, addr=%x).\n\r",
                _cause_str, _excep_code, _excep_addr);

    while (1)
    {
        SYS_DEBUG_BreakPoint();
    }
}
```

Part III: Debugging While Running Stand-alone

This section discusses how to debug problems when running without the debugger, including using diagnostic messages to a HyperTerminal or equivalent application on your computer.

Description

While the debugger in the MPLAB X IDE can help identify many bugs, there are cases where running the application outside of the debugger (i.e., Stand-alone mode), is necessary. The ability to dump error messages from failing asserts or from the exception handler is key to debugging an application outside of the debugger. Transmitting diagnostic or debug messages can also be key, both with and without the debugger.

Getting Started

Provides information on getting started with project debugging while running stand-alone.

Description

The steps outlined in this section can be applied to any MPLAB Harmony-based project, but for the sake of clarity we assume you have completed Part I: Creating Your First MPLAB Harmony Application in MHC. We will use the project from the this tutorial as the basis for this lesson. Board setup will be different than in the prior tutorials, primarily because of the need to support a USART connection to a COM port on your PC.

Setting Up the Hardware

PIC32MZ Embedded Connectivity (EF) Starter Kit

Setting up this board is easy, since it has an on-board MCP2221A USART-to-USB Bridge. Therefore, all that is required to connect the device to a COM port is to plug in a mini-B to Type-A cable from the mini-B port beneath the Ethernet PHY to a USB port on your PC.
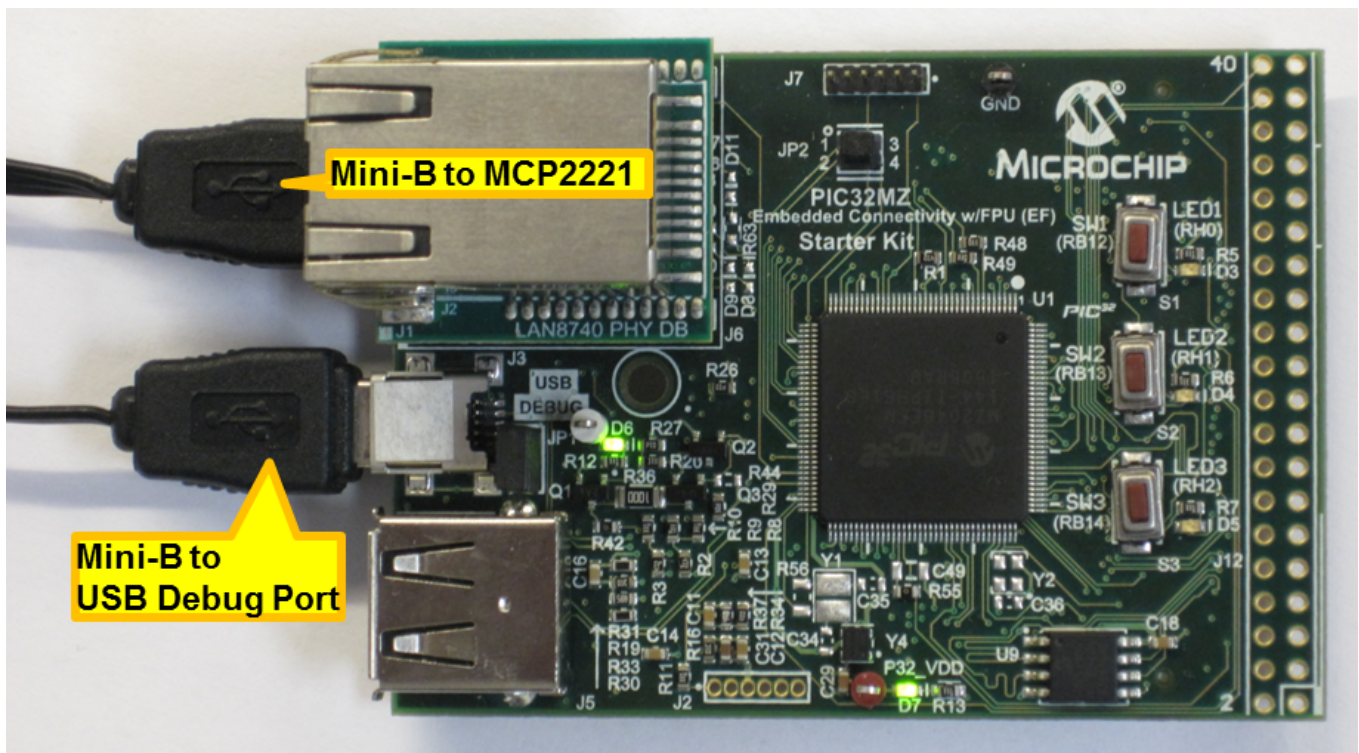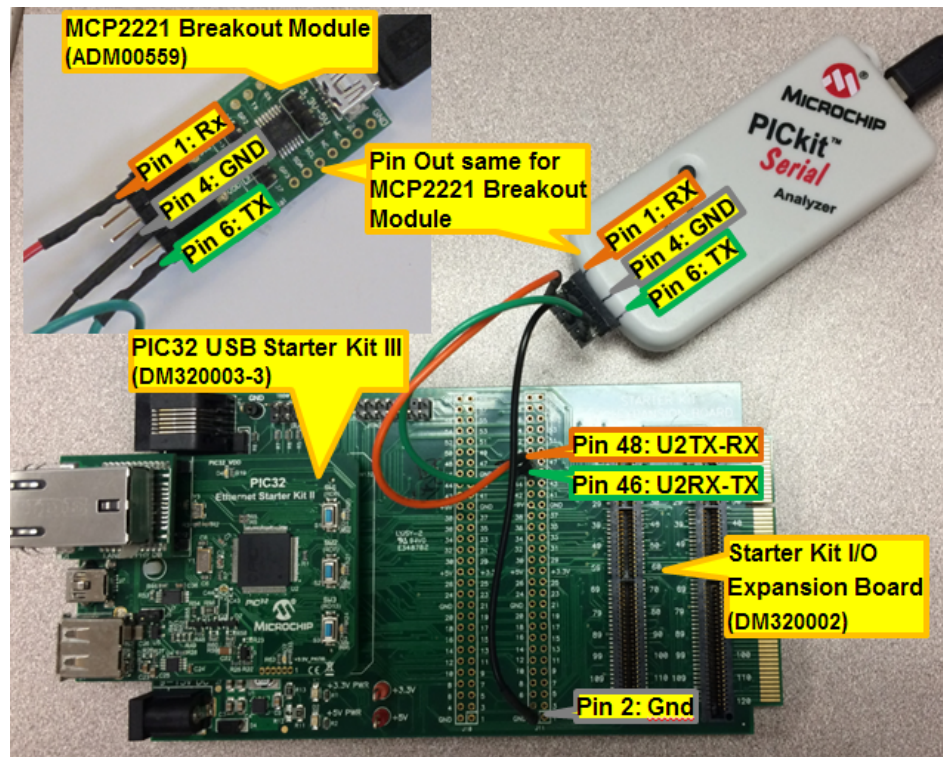

PIC32 USB Starter Kit III

As an older board design, there is no MCP2221A on this board, so you will need the following additional hardware:

• MCP2221 Breakout Module (ADM00559)

• Mini-B to Type-A USB cable for the Breakout Module

• Starter Kit I/O Expansion Board (DM320002)

• Jumper Wires

• 0.1" Pitch Header Pins to connect jumpers between the breakout module and I/O expansion board.

The hardware is set up as follows:

Explorer 16 Development Board with the PIC32MX795F512L Plug-in Module (PIM)

As an older board design, there is no MCP2221A on this board, so you will need the following additional hardware:

• MCP2221 Breakout Module (ADM00559)

• Mini-B to Type-A USB cable for the Breakout Module

• Prototype PICTail Plus Daughter Board (AC164126 for a three pack)

• Jumper Wires

• 0.1" Pitch Header Pins to connect jumpers between the breakout module and I/O expansion board.

The hardware is set up as follows:
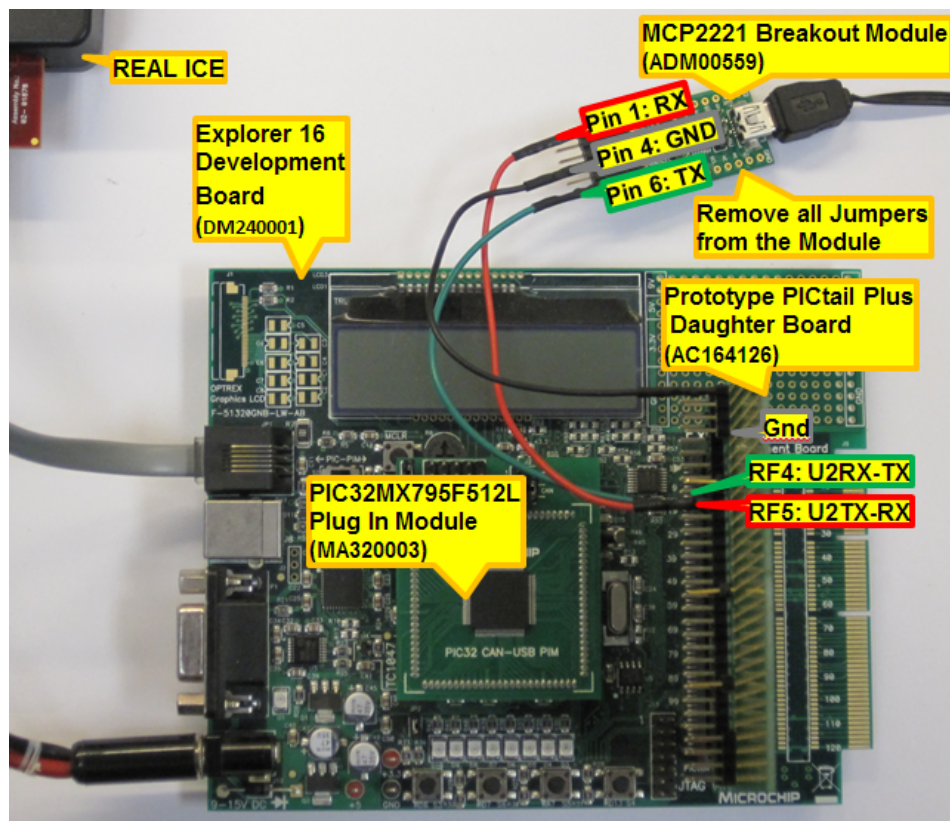
What You Will Learn

- How the MPLAB Harmony Configurator (MHC) configures USARTs and device pins for USARTS
- How MHC configures the Console System Service and the Debug System Service to support output via a USART
- How to output diagnostic and debug messages via these system services
- How to customize the assert handler and exception handler used in the application

Tutorial Steps

This part of the tutorial explores how to debug when running without the debugger (i.e., stand-alone).

### Enabling USART Output Using System Console and Debug System Services

This section describes how to set up the System Console and Debug System Services using a USART port.

Description

1. Launch the MPLAB X IDE and load the project you created in the Part I: Creating Your First MPLAB Harmony Application in MHC.
2. Set the project as the IDE's Main Project and launch the MPLAB Harmony Configurator (MHC).
3. If you are using a board with a built-in MCP2221A USART-to-USB Bridge, in the BSP Configuration enable the USART-to-USB Bridge. This will assign device pins for use by the USART connected to the bridge.
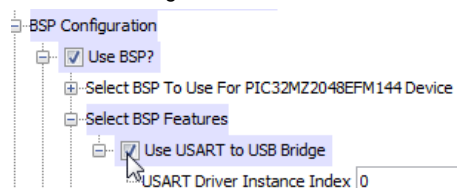

4. Within *Harmony Framework Configuration > Drivers > USART,* configure USART 2.
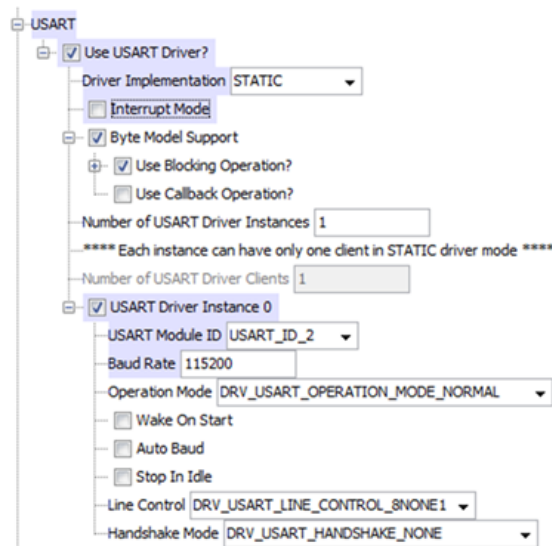
> **Note:** There are many options on how the USART driver is configured, but the simplest is always the best in this situation, since the USART must work after an assert has failed or an exception has fired. Therefore, the simplest set up is best.

5. Within *Harmony Framework Configuration > System Services*, configure the application to use the Console System Service. (The STATIC configuration is hard-wired to use USART Driver Instance 0 (the first one defined), which we set up in the previous step. To use another USART Driver Instance you must use the DYNAMIC service mode.) Again, the simplest setup is the best approach to handle asserts and exceptions.
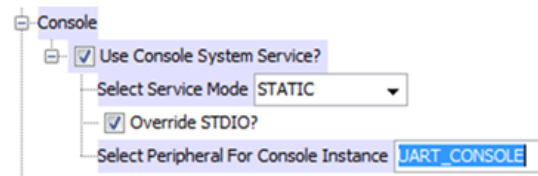
6. Also under System Services, configure the application to use the Debug System Service. Set the System Error Level to SYS_ERROR_DEBUG to support SYS_DEBUG_PRINT. The pull-down menu for System Error Level has the options shown in the second figure.
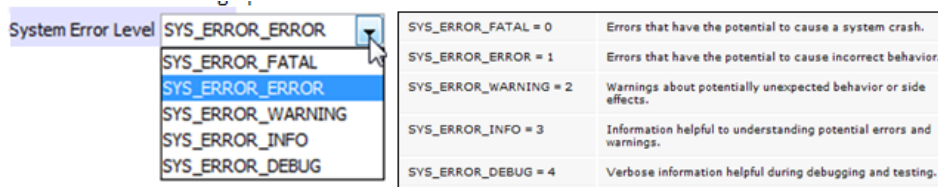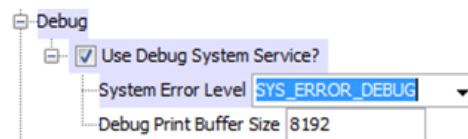
> **Note:** The System Error Level determines which SYS_DEBUG_PRINT messages are actually printed on the USART port. Set to SYS_ERROR_DEBUG, all levels are printed.

7. Open the Pin Settings Tab in MHC. For boards with a MCP2221A, verify that the BSP has correctly set the USART pins. For boards without a MCP2221A, set the USART pins as shown. (Click the Function column to select the correct pin function.)
   - PIC32 USB Starter Kit III:

Pin Number	Pin ID	Voltagea Tolerance	Name	Function
49	RF4	5V	U2RX	U2RX
50	RF5	5V	U2TX	U2TX

   - PIC32MZ EF Starter Kit:

Pin Number	Pin ID	Voltagea Tolerance	Name	Function
14	RG6	N/A	USART-to-USB Bridge (USB)	U2RX
61	RB14	N/A	USART-to-USB Bridge (BSP)	U2TX

- Explorer 16 Development Board with PIC32MX795F512L Plug In Module:

Pin Number	Pin ID	Voltagea Tolerance	Name	Function
49	RF4	5V	U2RX	U2RX
50	RF5	5V	U2TX	U2TX

8. Generate this new code configuration. For `system_config.h`, accept the changes, but do not discard the `// CUSTOM CODE` segment that you added in Part I: Creating Your First MPLAB Harmony Application in MHC.

9. Open `heartbeat.c` and add the following code shown in red to HEARTBEAT_Initialize.

```
void HEARTBEAT_Initialize ( void )
{
SYS_MESSAGE(
"\r\nApplication created " __DATE__ " " __TIME__ " initialized!\r\n");
//Test out error handling
// assert(0);
// {
// uint8_t x, y, z;
// x = 1;
// y = 0;
// z = x/y;
// SYS_DEBUG_PRINT(SYS_ERROR_DEBUG,"x: %d, y: %d, z: %d\r\n",x,y,z);
// }


/* Place the App state machine in its initial state. */
heartbeatData.state = HEARTBEAT_STATE_INIT;



/* TODO: Initialize your application's state machine and other
* parameters.
*/
}
```

10. Save the file by pressing Ctrl+S, and then close the window.

Note: The portion of this addition that is commented can be uncommented to support testing that asserts and exceptions are correctly reported. Since we have enabled the Debug System Service, SYS_DEBUG_PRINT here actually works. Therefore, the compiler will not drop this this code when it is enabled, thereby eliminating the need to modify the code's optimization level as before.

11. Set up your PC's HyperTerminal to 115200 baud, 8 bits, 1 Stop Bit, No Parity.

12. Run the project.

If you have correctly setup you HyperTerminal application you should see something similar to the following on its display:

```
Application created Aug 8 2017 12:14:04 initialized!
```

Getting the HyperTerminal to correctly identify the COM port belonging to the MCP2221A (either on-board or in a Breakout Module) can be a fussy and frustrating process. There will be times when you can't find the COM port. In those cases, at least on a Windows PC, try the following:

- In the Control Panel, select *System > Device Manager*
- Within Ports (COM & LPT), identify the COM port belonging to the MCP2221A. Double click on this port to open its Properties window.
- Select the Driver tab and disable, and then enable the driver
- Close the window.

This should allow your HyperTerminal to see the port.

Note: If all else fails, you may need to put the SYS_DEBUG_PRINT statement in a `while(1)` loop and, worst case, use an oscilloscope to make sure the USART TX signal is getting to the MCP2221A and that the USB data lines are working as well.

If your board has a built-in MCP2221A but does not have an independent power supply, as with the PIC32MZ EF Starter Kit, where power is supplied by the debug port, you will not see the initial startup message when power cycling the application by unplugging and plugging in the debug port. When power is supplied to the board, the application starts, but the MCP2221A has to enumerate as a USB device with your PC before COM port output is accepted. So, the initial message has long since passed on the port before the COM port is working.

If your board has a Master Clear (MCLR) button, you can simply press the button to reset the application after the MCP2221A enumerates. Then, the initial message will be seen on your HyperTerminal application. If your board does not have a MCLR button, you can still assert a Master Clear by grounding pin 1 of the ICSP header. The following figure show how this is done on the PIC32MZ EF Starter Kit.
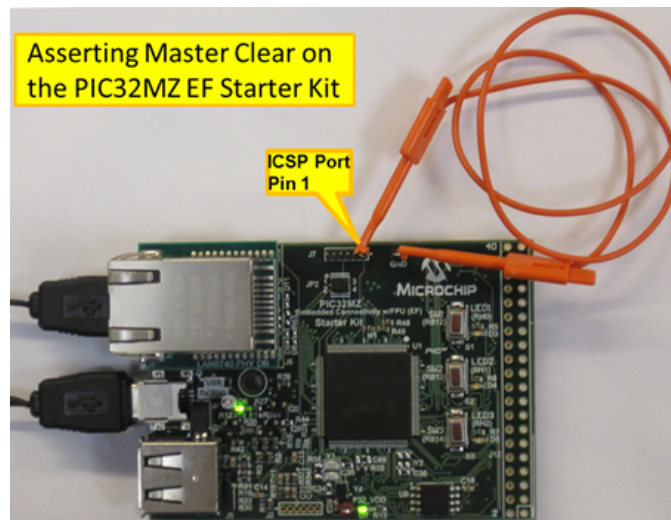
Adding Customized Assert and Exception Handling

This section describes how to add customized assert and exception handling without a debugger.

Description

The default assert function provided by the PIC32 compiler is called _fassert. It is "weakly" defined, meaning that you can provide a customized replacement for it in your project. You may want to replace the default (compiler) assert for two reasons:

- The default function is hardwired to use USART2 and you want to use another USART
- You have a lot of asserts in your code, but do not need them to report out the line number, file name, failed expression, and function name in the message, because storing all of these string constants for every assert uses too much memory. Instead, you can invent a customized _fassert that only reports out, for example, only the line number and function name to save memory.

The default exception handler is hardwired to USART2, so using MPLAB Harmony's replacement handler, as we did in Part I: Creating Your First MPLAB Harmony Application in MHC, will at least give you the flexibility to control which USART is used. However, both handlers only report out the cause and program address of the exception, nothing more. Please note that there will be cases where this information is not sufficient to locate what went wrong. MHC provides two additional, advanced exception handlers that can be used instead.

1. The MHC menu Advanced Exception and Error Handling shows the options available.
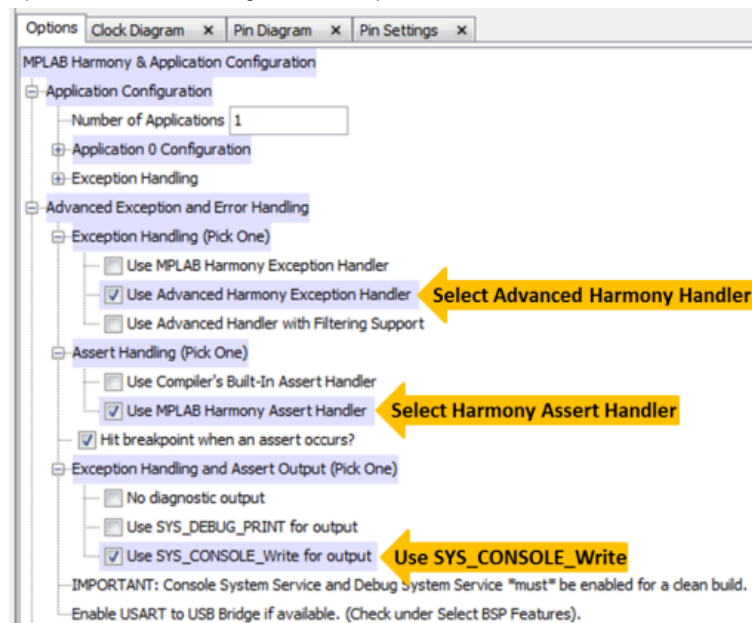
2. Select the Advanced Harmony Exception Handler. (The other advanced handler shown supports filtering by saturating rather than overflowing integer arithmetic.) Select the MPLAB Harmony Assert Handler to replace the compiler's built-in assert handler. By default, output uses SYS_DEBUG_PRINT, but using SYS_CONSOLE_Write can be chosen instead since it is a more robust way of reporting exceptions and errors.

3. Generate this configuration by clicking Generate Code. After completing this, your default project folder should contain the files shown in the following figure. Note the assembly (.S) file is required to report out extra information via the new exception handler.
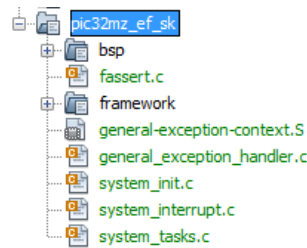
4. Double click the new exception handler, general_exception_handler.c, to see what it reports.

```c
void __attribute__((nomips16)) _general_exception_handler (XCPT_FRAME* const pXFrame)
{
    register uint32_t _localStackPointerValue asm("sp");

    _excep_addr = pXFrame->epc;
    _excep_code = pXFrame->cause;    // capture exception type
    _excep_code = (_excep_code & 0x0000007C) >> 2;

    _CP0_StatusValue   = _CP0_GET_STATUS();
    _StackPointerValue = _localStackPointerValue;
    _BadVirtualAddress = _CP0_GET_BADVADDR();
    _ReturnAddress     = pXFrame->ra;

    sprintf(msgBuffer,"**EXCEPTION:*\r\n"
                    " ECode: %d, EAddr: 0x%08X, CPO Status: 0x%08X\r\n"
                    " Stack Ptr: 0x%08X, Bad Addr: 0x%08X, Return Addr: 0x%08X\r\n"
                    "**EXCEPTION:*\r\n",
                    _excep_code,_excep_addr,_CP0_StatusValue,
                    _StackPointerValue,_BadVirtualAddress,_ReturnAddress);
    SYS_CONSOLE_Write(SYS_CONSOLE_INDEX_0,STDOUT_FILENO,msgBuffer,strlen(msgBuffer));

    SYS_DEBUG_BreakPoint();  // Stop here if in debugger.

    while(1) {
        //Do Nothing
    }
}
```

So, in addition to the cause (ECode) and address (EAddr) of the exception, this exception handler also reports the Core Register CP0 value (CPO Status), the Stack Pointer value (Stack Ptr), Bad Address value (Bad Addr), and the Return Address value (Return Addr).

For more information on the bit fields in the CP0 register refer to one of these PIC32 Family Reference Manual sections:

- PIC32MX: Section 02. "CPU for Devices with M4K Core" (DS6000113)
- PIC32MK and PIC32MZ: Section 50. "CPU for Devices with microAptiv Core" (DS60001192)

Both of these documents are available for download from the Microchip website at: www.microchip.com.

5. Add post processing to the project's configuration to produce a disassembly listing.
    - Right click the project name and selecting **Properties**
    - Within the Building ( ) properties, enable **Execute this line after build** and enter the following text:
        - ${MP_CC_DIR}\xc32-objdump -S ${ImageDir}\${PROJECTNAME}.${IMAGE_TYPE}.elf > disassembly.lst
    - At the end the window should show:
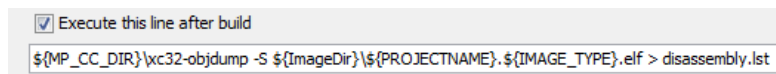
- Click **OK** to close the window

6. To explore how we can use the extra information reported by the new exception handler, make the following modifications shown in red text to heartbeat.c:

```c
void DivideByZero(void)
{
uint8_t x, y, z;
x = 1;
y = 0;
```

```
z = x/y;
SYS_DEBUG_PRINT(SYS_ERROR_DEBUG,"x: %d, y: %d, z: %d\r\n",x,y,z);
}


void Dereference_Bad_Address(void)
{
uint32_t * pointer;
uint32_t value;


pointer = (uint32_t *)0xDEADBEEF;
value = *pointer;
SYS_DEBUG_PRINT(SYS_ERROR_DEBUG,"Value: %d\r\n",value);
}


void HEARTBEAT_Initialize ( void )
{
SYS_MESSAGE(
"\r\nApplication created " __DATE__ " " __TIME__ " initialized!\r\n");
// Test out error handling
// assert(0);
// {
// uint8_t x, y, z;
// x = 1;
// y = 0;
// z = x/y;
// SYS_DEBUG_PRINT(SYS_ERROR_DEBUG,"x: %d, y: %d, z: %d\r\n",x,y,z);
// }


//assert(0);
//DivideByZero();
//Dereference_Bad_Address();


/* Place the App state machine in its initial state. */
heartbeatData.state = HEARTBEAT_STATE_INIT;



/* TODO: Initialize your application's state machine and other
* parameters.
*/
}
```

7. Uncomment the `//assert(0);`, abd then build and run the application. In your HyperTerminal application, you should see something similar to the following:

```
Application created Aug 8 2017 16:51:05 initialized!
ASSERTION '0' FAILED! File: ../src/heartbeat.c, Line: 148, Function: HEARTBEAT_Initialize
```

8. Now comment out the assert and uncomment the call to DivideByZero, and then build and run. In your HyperTerminal application, you should see something similar to the following:

```
Application created Aug 8 2017 16:37:51 initialized!
**EXCEPTION:*
ECode: 13, EAddr: 0x9D006CAC, CPO Status: 0x25000003
Stack Ptr: 0x8007FED8, Bad Addr: 0x25651D53, Return Addr: 0x9D007020
**EXCEPTION:*
```

In the disassembly listing you will see:

```
  void DivideByZero(void)
  {
      uint8_t x, y, z;
      x = 1;
      y = 0;
      z = x/y;
  9d006ca0:   24070001    li  a3,1
```

```
9d006ca4:   00001021    move    v0,zero
9d006ca8:   00e2001b    divu    zero,a3,v0
9d006cac:   004001f4    teq v0,zero,0x7 <<------------------ EAddr
9d006cb0:   00003812    mflo    a3
        SYS_DEBUG_PRINT(SYS_ERROR_DEBUG,"x: %d, y: %d, z: %d\r\n",x,y,z);
9d006cb4:   3c049d00    lui a0,0x9d00
9d006cb8:   24846968    addiu   a0,a0,26984
9d006cbc:   24050001    li  a1,1
9d006cc0:   00003021    move    a2,zero
9d006cc4:   0f4015fd    jal 9d0057f4 <SYS_DEBUG_Print>
9d006cc8:   30e700ff    andi    a3,a3,0xff


9d006ccc <.LVL2>:
}


    //assert(0);
    DivideByZero();
9d007018:   0f401b22    jal 9d006c88 <.LFE1152>
9d00701c:   00000000    nop


9d007020 <.LVL9>:                       <<------------ Return Address
    //Dereference_Bad_Address();

    /* Place the App state machine in its initial state. */
        heartbeatData.state = HEARTBEAT_STATE_INIT;
9d007020:   af808054    sw  zero,-32684(gp)  <<---- Return Address
```

In this example we see, as before, that the exception address correctly identifies the instruction that caused the exception. Note also that the return address points to the instruction after the call to the DivideByZero function.

Note: The actual addresses may vary depending on the target device.

9. Now comment out the DivideByZero and uncomment the call to Dereference_Bad_Address, and then build and run. In your HyperTerminal application you should see something similar to the following:

```
Application created Aug 8 2017 16:43:01 initialized!
**EXCEPTION:*
ECode: 4, EAddr: 0x9D006F38, CP0 Status: 0x25000003
Stack Ptr: 0x8007FED8, Bad Addr: 0xDEADBEEF, Return Addr: 0x9D006F40
**EXCEPTION:*
```

In the disassembly listing you will see:

```
  void Dereference_Bad_Address(void)
  {
  9d006f24: 27bdffe8 addiu sp,sp,-24
  9d006f28: afbf0014 sw ra,20(sp)
  9d006f2c: afb00010 sw s0,16(sp)
  uint32_t * pointer;
  uint32_t value;

  pointer = (uint32_t *)0xDEADBEEF;
  value = *pointer;
  9d006f30: 3c02dead lui v0,0xdead
  9d006f34: 3442beef ori v0,v0,0xbeef

  9d006f38 <.LVL4>: <<------------ EAddr
  SYS_DEBUG_PRINT(SYS_ERROR_DEBUG,"Value: %d\r\n",value);
  9d006f38: 0f40003e jal 9d0000f8 <.LFE1164>
  9d006f3c: 8c500000 lw s0,0(v0)

  9d006f40 <.LVL5>: <<----- Return Address
  9d006f40: 10400006 beqz v0,9d006f5c <.LVL6+0x4> <<--
  9d006f44: 8fbf0014 lw ra,20(sp)
  9d006f48: 3c049d00 lui a0,0x9d00
  9d006f4c: 24846980 addiu a0,a0,27008
  9d006f50: 0f4015fd jal 9d0057f4 <SYS_DEBUG_Print>
  9d006f54: 02002821 move a1,s0

  9d006f58 <.LVL6>:
  }
```

The exception code reported, 4, corresponds to an "address error exception (load or ifetch)".

In this example, the return address didn't provide much information but we see that the bad address used in the pointer dereference was correctly reported.

As an additional step, replace `value = *pointer` with `*pointer = value` in the code and verify that an exception code of 5, "address error exception (store)", is reported instead.
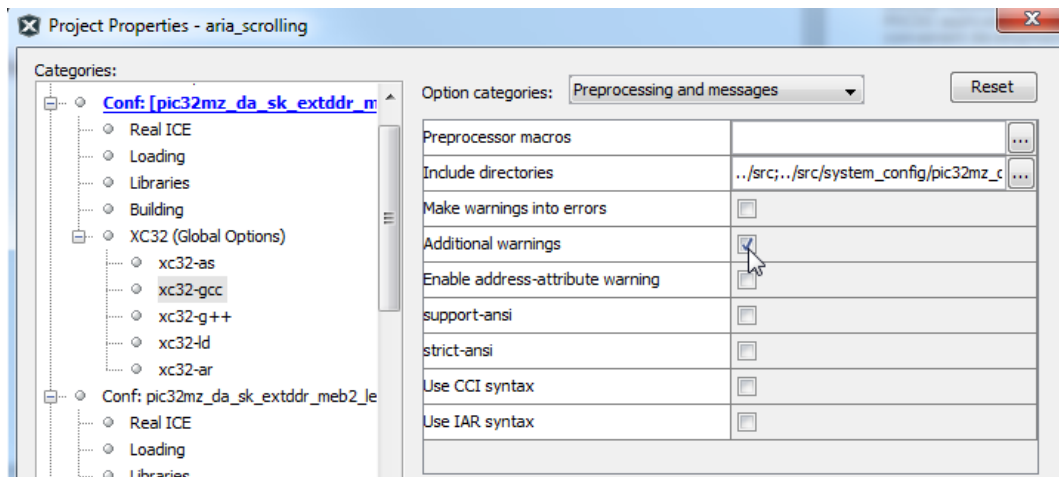
Tips

Provides tips for effective use.

Description

Warnings All Compiler Switch

It is recommended that you enable the compiler switch, –Wall (Warnings-All), for your project. This will warn you of potential problems that may turn into bugs. In the configuration's properties, select the compiler compiler and select **Additional Warnings**.

Project Locations on Your Hard Drive

In the first tutorial of this series, the project was created in the following folder (for Windows): `C:\microchip\harmony\<Version>\apps`.

In reality, you can create a MPLAB Harmony project anywhere on the same hard drive that contains the MPLAB Harmony installation you are using. For MAC OS and Linux, that is the only limitation. For Windows, there is an operating system limitation that all paths in the project must be less than 256 characters in length. Therefore, you may run into trouble on Windows if the project is created too deep into the drive's directory tree.

Moving and Copying Projects

All of files in your project are referenced by their relative path from the ".X" directory (`heartbeat\firmware\heartbeat.X`), which contains the `Makefile` make file and `nbproject` sub-directory. This provides flexibility in relocating and copying projects, since as long as the relative paths to files in the MPLAB Harmony installation (typically `C:\microchip\harmony\<version>`) still work the project can be anywhere.

Example

You can move/copy a project:

- Old Location: `C:\MyWork\MyProject\heartbeat`
- New Location: `C:\MyWork\MyNewProject\heartbeat` (Good)

However, neither of these new locations work, since it breaks the project's relative paths:

- Old Location: `C:\MyWork\MyProject\heartbeat`
- New Location: C:\MyProject\heartbeat (Not Good), or
- New Location: `C:\MyWork \MyNewProject\Rev2\heartbeat` (Not Good)

Next Steps

Provides information on where to find additional resources.

Description

To learn more about MPLAB Harmony, refer to *Volume I: Getting Started With MPLAB Harmony > What is MPLAB Harmony?*

Revisit *Volume I: Getting Started With MPLAB Harmony > Guided Tour* for suggestions on where to begin learning more.

Try an existing MPLAB Harmony demonstration that runs on a PIC32MZ EF Starter Kit:

Peripheral Examples (<install-dir>/apps/examples/)**:**

peripheral

Basic Bootloader (<install-dir>/apps/bootloader/)**:**

basic

Graphics with MEB II Display (<install-dir>/apps/gfx/)**:**

- aria_quickstart
- aria_showcase
- aria_weather_forecast

TCP/IP Stack (<install-dir>/apps/tcpip/)**:**

- tcpip_client_server
- tcpip_tcp_server
- tcpip_udp_client
- tcpip_udp_client_server
- tcpip_udp_server

USB Device (<install-dir>/apps/usb/device/)**:**

- cdc_msd_basic
- cdc_serial_emulator
- hid_basic
- hid_joystick
- hid_msd_basic
- msd_basic
- hid_mouse

USB Host (<install-dir>/apps/usb/host)**:**

- audio_speaker
- cdc_basic
- cdc_msd
- hid_basic_keyboard
- hid_basic_mouse_usart
- hub_msd
- msd_basic

Index