

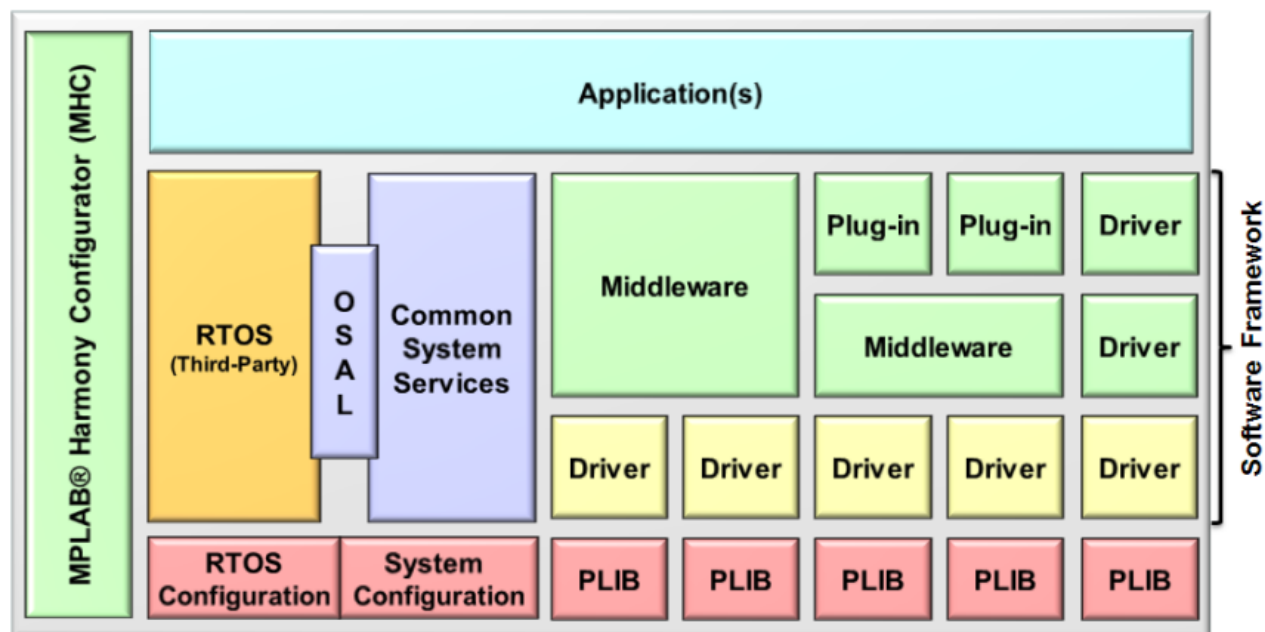
What is MPLAB Harmony?

This section provides an overview of MPLAB Harmony.

Description

Microchip MPLAB® Harmony is the result of a holistic, aggregate approach to creating firmware solutions for embedded systems using Microchip PIC32 microcontrollers. As shown in the following diagram, MPLAB Harmony consists of portable, modular and compatible libraries provided by Microchip and third-party ecosystem partners. MPLAB Harmony also includes easy-to-use development utilities like the MPLAB Harmony Configurator (MHC) plug-in for the MPLAB X IDE, which accelerate development of highly capable and reusable PIC32 embedded firmware applications.

MPLAB® Harmony Block Diagram



Designed almost completely in the C language (see **Note**), MPLAB Harmony takes key elements of modular and object-oriented design, adds in the flexibility to use a Real-Time Operating System (RTOS) or work without one if you prefer, and provides a framework of software modules that are easy to use, configurable for your specific needs, and that work together in complete harmony.



Note: MPLAB Harmony has not been tested with C++; therefore, support for this programming language is not provided.

Portability

Portability is a concern that is often overlooked when a silicon manufacturer provides software. However, breadth of solutions is a hallmark strength of Microchip, and MPLAB Harmony provides simple libraries to abstract away part-specific details and make a Microchip device easy to use, regardless of which device you choose. Any time you design a new product or update an existing one, cost must be balanced with capabilities; however, cost is more than just the bill of materials – it's also the Non-Refundable Engineering (NRE) cost to design and develop your solution. MPLAB Harmony provides peripheral libraries, device drivers, and other libraries that use clear and consistent interfaces, requiring little or no change in your application code and minimizing the engineering time and effort for each new design.

Device Drivers

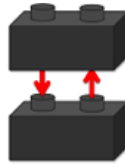
The primary purpose of a MPLAB Harmony device driver (or "driver") is to provide a simple and highly abstracted interface to a peripheral, allowing your application (or any other module in the system) to interact with a peripheral through a consistent set of functions. A driver is responsible for managing access to a peripheral, so that requests from different modules do not conflict with each other, and for managing the state of that peripheral so that it always operates correctly.

Peripheral Libraries

A Peripheral Library (PLIB) is a simple access library that provides a consistent (but very low level) interface to a peripheral that is "on board" the MCU. PLIBs hide register details, making it easier to write drivers that support multiple microcontroller families, but they are not normally used by applications directly to interact with peripherals, as they provide little abstraction, and because they require the caller to manage the detailed operation of a peripheral (including preventing conflicting requests from other modules). Because of the lack of conflict protection in a PLIB, only one module in a system should directly access the PLIB for a peripheral. Therefore, PLIBs are primarily used to implement device drivers (and some system services) to make them portable.

Modularity

MPLAB Harmony libraries are modular software "building blocks" that allow you to divide-and-conquer your firmware design. The interface to each library consists of a highly cohesive set of functions (not globally accessible variables or shared registers), so that each module can manage its own resources. If one module needs to use the resources of another module, it calls that module's interface functions to do so. Interfaces between modules are kept simple with minimal inter-dependencies so that modules are loosely coupled to each other. This approach helps to eliminate conflicts between modules and allows them to be more easily used together like building blocks to create the solutions you need.



Middleware Libraries

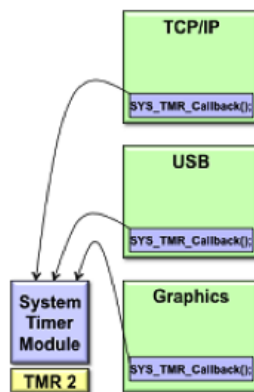
The normal usage models of some of the more complex peripherals, (i.e., USB or network interfaces) require interpreting complex protocols or may require substantial additional processing to produce useable results, such as drawing graphical images on an LCD screen with an LCD controller peripheral. Therefore, while a device driver may be completely sufficient for a simple peripheral like a UART, some peripherals require what is frequently called "middleware" (aptly named because it sits between your application and the hardware abstraction layer or "driver" layer). MPLAB Harmony provides several middleware library "stacks" to manage these more complex peripherals and provide the functionality you need and expect.

MPLAB Harmony middleware "stacks" are usually built upon device drivers and system services so that they can be supported on any Microchip microcontroller for which the required driver or service is supported. However, special purpose implementations may be available that integrate the driver, certain services, and various modules within the "stack" for efficiency.

System Services

MPLAB Harmony system services are responsible for managing shared resources so that other modules, such as drivers, middleware, and applications, do not conflict on shared resources. For example, if the TCP/IP, USB, and Graphics stacks attempted to concurrently use the Timer2 peripheral to perform some periodic task, they would very likely interfere with each other. However, if instead they used a timer system service (as the following image illustrates), it is the responsibility of the system service to keep the separate requests from interfering with each other. The timer service can be configured as desired for a specific system (for example, you may decide to use Timer3 instead of Timer2) isolating the necessary changes to the configuration of a single module and preventing potential conflicts.

The use of a system service is very similar the use of a device driver, except that a driver normally requires the caller to "open" it to create a unique client-to-driver association. A system service does not normally require the caller to open the service before using it because system services are frequently shared by many clients within the system.



Compatibility

MPLAB Harmony modules (drivers, system services, and middleware – excluding PLIBs) are "active". This means when an application calls a module's interface function, the call will usually return immediately (unless a RTOS is in use) and the module will continue working on its own to complete the operation. Most modules will then provide a notification mechanism so the caller (i.e., client) can determine when the operation has finished.

Most MPLAB Harmony modules are implemented as cooperative state machines. The following image shows the basic idea of how this works. Each module has an "Initialize" function and each module has one (or more) "Tasks" function(s) to maintain its state machine(s). The state machines of all modules are initialized, shortly after the system comes out of reset in "main". After that (in a polled configuration, with no OS), the system drops into a "super loop" where each module's state machine function is repeatedly called, one after the other, to allow it to do the next "task" necessary to keep its state machine running. This allows the system to keep all modules running using a cooperative or shared "multi-tasking" technique. Modules (under control of your application) interact with each other by calling the interface functions of other modules (as illustrated in the following figure) and the system-wide "super loop" keeps all modules in the system running so they stay "active" and do their jobs.

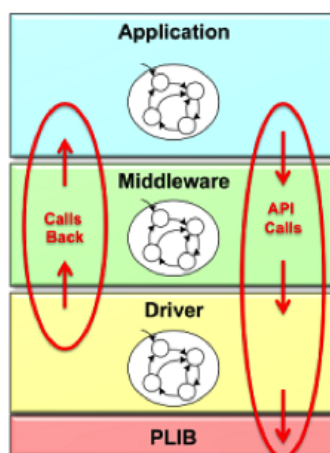
```

main()
{
    /* Call Function to Initialize State Machine 1 */
    /* Call Function to Initialize State Machine 2 */
    /* Call Function to Initialize State Machine 3 */

    while(true);
    {
        /* Call Function to Do Next
        State Machine 1 Task */
        /* Call Function to Do Next
        State Machine 2 Task */
        /* Call Function to Do Next
        State Machine 3 Task */
    }
}

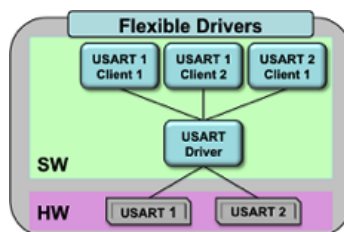
```

This method is not suitable for all needs; therefore, other configurations are possible. However, a polled configuration is the simplest to understand and it best illustrates the basic concept of how MPLAB Harmony allows independent modules to operate cooperatively within an embedded system. To interact with each other, otherwise independent library and application modules make calls to each other's Application Program Interface (API) functions, as shown in the following diagram. Calls *into* a library are made through well-defined API functions and calls back to the client may be made through *callback* functions, statically linked (at build time) or dynamically registered at run-time and called using a function pointer.



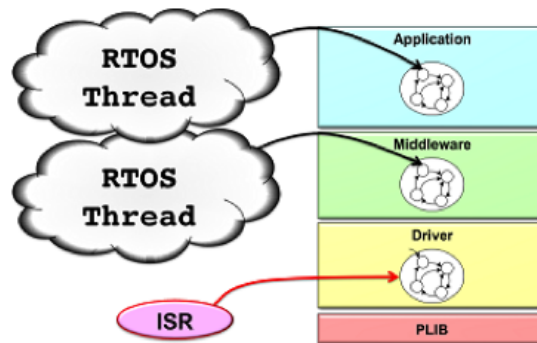
Flexibility

The basic MPLAB Harmony model of cooperating state machine driven modules, when combined with a little configurability, becomes flexible enough to meet the needs of almost any embedded system. For example, if you are using multiple identical peripherals, MPLAB Harmony has "dynamic" driver implementations that can manage all instances of a peripheral with a single instance of the driver code. You might also have a need for multiple "client" modules to use the same instance of a peripheral at the same time (such as the timer example, described previously). To manage this need, MPLAB Harmony has driver implementations that are intelligent enough to manage requests from multiple clients. On the other hand, your needs may be simpler than that. So, static and single client implementations are also available for key libraries to help reduce the amount of code and data storage needed for your system.



Or, your system may need to combine several middleware stacks and multiple, potentially independent, applications. If that is the case, the simple polling operation, using the "super loop" method frequently seen in simple embedded systems may not be sufficient. When you start adding more modules, it becomes more and more difficult to meet the timing requirements of all peripherals using a simple polled super loop.

Fortunately, MPLAB Harmony modules are written so that (where appropriate) their state machines can be run directly from an Interrupt Service Routine (ISR) or a RTOS thread. Using an ISR allows you to eliminate the latency of waiting for the execution of other modules in the loop to finish before a time-critical event is serviced, and it allows you to use the interrupt prioritization capabilities available on Microchip devices to ensure that your system responds to events in the real world in real-time.



Additionally, the ability to schedule and prioritize different tasks for different modules can be obtained for modules that are not associated with a specific processor interrupt (such as many middleware modules and your application) using a RTOS. In fact, that is one of the main reasons to use a RTOS. When your system becomes complex enough that you start struggling to meet your timing requirements using the super loop method, it's time to use a RTOS.

Fortunately, MPLAB Harmony module state machine functions can be called from a loop in a RTOS thread just as easily as they can be called from a polled "super loop" in a system without a RTOS. To allow this, modules are designed to be "thread safe" by calling semaphore, mutex, and critical section operations through an Operating System Abstraction Layer (OSAL). The OSAL provides a consistent set of functions to call, regardless of which RTOS is being used (or even if no RTOS is used). This method makes the choice of RTOS to use, if any, into a configuration option. MPLAB Harmony supports several OS and non-OS configurations and support for more operating systems is possible. All that is required is to implement the OSAL functions appropriately for the desired OS.

Configurability

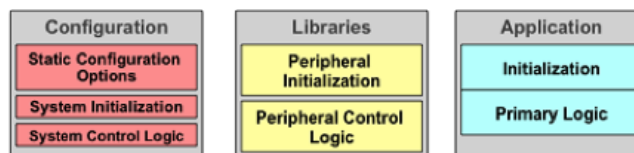
Most MPLAB Harmony libraries support a variety of build-time configuration options:

- Selection of the supported Microchip microcontroller
- Interrupt-driven or polled execution
- Static or Dynamic peripheral instance selection
- Single-client or Multi-client support
- Other library-specific options

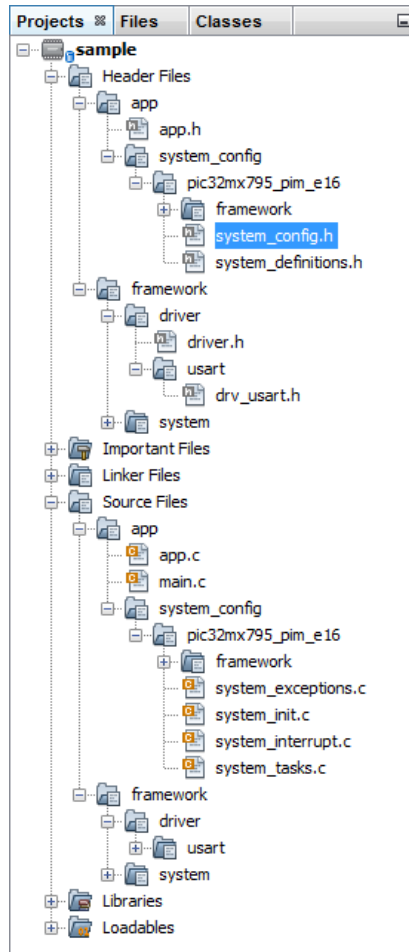
MPLAB Harmony libraries are designed to allow you to select a variety of configuration options to tailor them to your specific usage. For example, you may be able to select buffer sizes for data transfer modules or clock sources for timer modules. The set of configuration options for each library is identified and explained in the Help documentation (along with the interface and usage information) and the MPLAB Harmony Configurator (MHC) utility is provided to help simplify the process of configuring your system exactly the way you want and to get you started with a set of initial source files for your project.

Project Structure

To facilitate configurability, MPLAB Harmony projects are normally structured in a way that isolates the code necessary to configure a "system" from the library code and from your application code, as shown in the following figure.



The next figure shows how application, library, and configuration files are organized within the MPLAB X IDE project.



In a MPLAB Harmony project, the `main.c` file is kept very simple and consistent (containing primarily, just the super loop previously discussed). The application files (`app.c` and `app.h` in the previous figure) are separate from configuration files in the `system_config` sub-folders, so it is possible for a single application to have more than one configuration. (Usage of this capability can be seen in example and demonstration projects included with the installation of MPLAB Harmony.) The library modules that make up the MPLAB Harmony framework (in the `framework` folder) use the definitions provided in the selected configuration header (`system_config.h`, highlighted with a gray background in the previous figure) to specify the configuration options you selected when you configured the project. Finally the processor-specific peripheral libraries are provided as both a prebuilt binary (.a linker file) and as in-line source code to allow for maximum build efficiency for your firmware projects.

Summary

MPLAB Harmony provides a complete framework for developing your firmware solutions using Microchip microcontrollers and development tools. The firmware libraries and tools that make up the MPLAB Harmony framework are modular and compatible, making them simple to use. They're flexible and configurable, making them easy to tailor to your specific needs. And, they're portable across the full range of Microchip PIC32 microcontrollers, so you are sure to find a supported device that meets your needs.

Project Layout

This topic explains how a MPLAB Harmony project is organized.

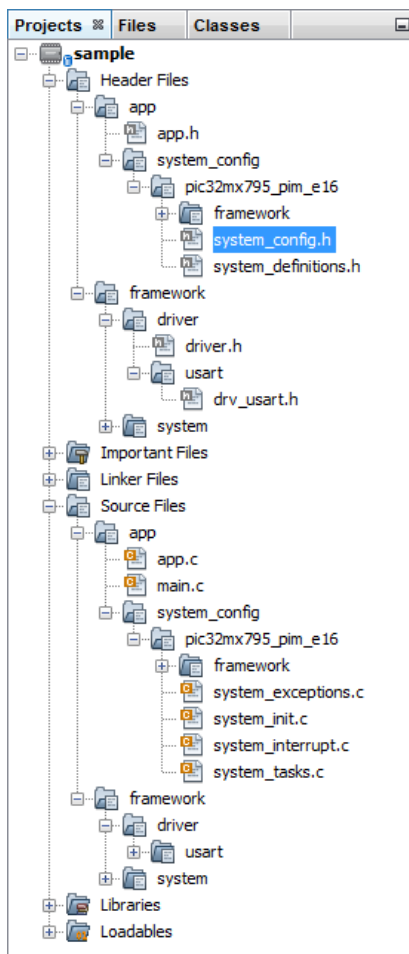
Description

A sample project has been created to show you the structure of a MPLAB Harmony project. The "sample" project is available in the following folder, within your MPLAB Harmony installation root folder:

```
<install-dir>/apps/examples/sample/firmware/sample.X
```

You should open this project in MPLAB X IDE and follow along with this guide.

A MPLAB Harmony project is organized within MPLAB X IDE, as shown in the following figure.



This organization consists of a few key "logical" folders and C language files, as follows.

The Header Files and Source Files Folders

The MPLAB X IDE separates C-language files into header (.h) files and source (.c) files by placing the header files in a top-level `Header Files` logical folder and the source files in a top-level `Source Files` logical folder. This distinction is for display only within the MPLAB X IDE and these folders do not appear on disk. Also, in most cases, logical folders that appear as sub-folders of these top-level folders are duplicated in both the `Header Files` and `Source Files` top-level logical folders because header and source files are kept together on disk.

The app Folder

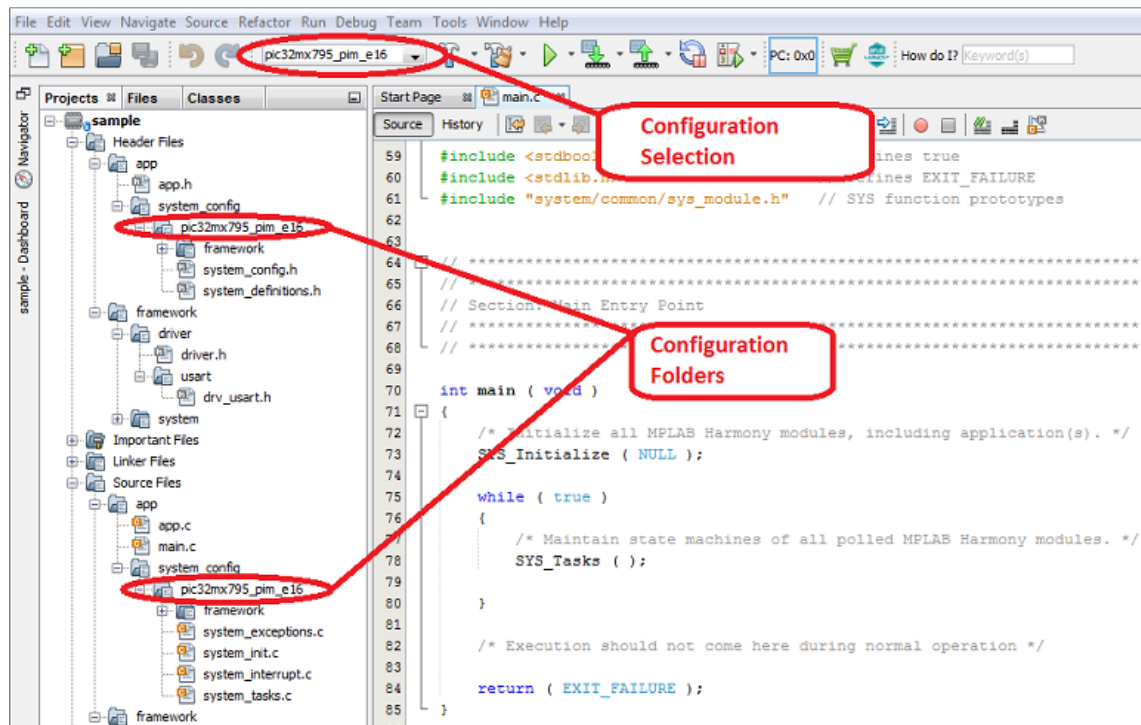
This folder contains the `main.c` and `app.c` source files and the `app.h` header file.

The `app` (i.e., application) folder and its sub-folders contain all of the project-specific source and header files (but not the shared files stored in the `framework` folder, which are discussed later). In a simple project, the `app` folder contains the `main.c`, `app.c`, and `app.h` files and a `system_config` sub-folder. More complex projects will very likely contain additional files, as needed. In a MPLAB Harmony project, the `main.c` file normally contains the C language main function and little or nothing else. The logic of the main function is consistent across all MPLAB Harmony projects and should not need to be changed. The `app.c` file normally contains the logic of the application itself. This is where the desired overall behavior of your system is usually implemented (although complex systems may have multiple applications).

The `app.h` file defines data types and other definitions required by the application or interface prototypes for functions the application wants to share with other applications or the system.

The system_config Folder

The `system_config` folder contains one or more subdirectories, each of which corresponds to an individual configuration of your project. MPLAB Harmony projects may have multiple configurations. Each project configuration creates a different variation of your embedded system with potentially different hardware or features. In each configuration, you can select a different set of libraries or modules, select different build parameters for each module and even select different source files for your application(s). A configuration consists of a specific set of properties (tools settings) in MPLAB X IDE, a set of source files that define the build parameters, and a set of source and header files that control which modules are initialized and maintained in your system.



In MPLAB X IDE, the project configuration can be selected by using a pull-down menu in the tool bar at the top of the window or by right-clicking the project name and selecting **Properties**. In most example and demonstration projects distributed with MPLAB Harmony, the name of each MPLAB X IDE configuration will match the name of the associated folder within the `system_config` folder in the project (the `795_pim_e16` folder in the sample project). When a specific MPLAB X IDE configuration is selected, the configuration files for that configuration are included in the build and the configuration files in other configuration folders are excluded from the build.

Note: This is the project convention used by the example and demonstration projects provided with MPLAB Harmony. You, of course, may organize your own projects any way you desire. However, it is recommended to follow this convention if you use multiple configurations in your projects. We think you'll appreciate the power and flexibility it provides.

Configuration Files:

- `system_config.h`
- `system_init.c`
- `system_tasks.c`
- `system_interrupt.c`
- `system_exceptions.c`
- `system_definitions.h`

This set of files define a configuration of the system. The purpose of each of these files is described in more detail in the following sections. But, the basic idea is that you may want different configurations of your application for different hardware boards, different Microchip microcontrollers, or different feature sets, depending on your specific needs.

Allowing different configurations of the same application logic makes your application more flexible and provides a well-organized way to deal with the sort of variation that usually occurs in any project of sufficient size and complexity. This technique helps to eliminate the duplication of code (and labor) that would otherwise be necessary to manage multiple related projects. Of course, if you don't need or want that flexibility, all of these files are specifically created for your project and you can make any modifications to them that you like. The choice is always yours.

Note: The relative path, from the MPLAB X IDE project folder to the configuration folder (containing the `system_config.h` file) for each project configuration is automatically placed in the "Includes directories" list in the compiler properties for each configuration of the MPLAB X IDE project by the MHC.

The framework Folder

The `framework` logical folder contains the source files for the MPLAB Harmony framework and libraries. Depending on your project configuration, there can be many, many files and sub-folders within this folder. These files are for MPLAB Harmony libraries that you should not need to edit. In fact, the framework source files are not normally located in your project. Instead, these files are included in your project directly from the MPLAB Harmony installation (using relative directory paths). All of the actual files stay in the MPLAB Harmony installation folder, out of the way.

**Notes:**

1. You always have the option of copying the framework files directly into your project's source folder, if desired. In fact, doing so is a good idea if you plan to move or distribute your project separately from the MPLAB Harmony installation.
2. In most cases, the "logical folder" organization within the MPLAB X IDE project matches exactly with the physical directory organization within the MPLAB Harmony installation (and within your project directory) on your disk drive. This is done to keep things simple and consistent so you only need to learn a single layout. But, there are a couple of notable exceptions.
 - MPLAB X IDE has a convention of splitting out "Header Files" (.h) and "Source Files" (.c), so that the virtual folder organization in project separates the files in to these two groups and the physical directories on disk do not
 - In a MPLAB Harmony example or demonstration project, the `app` folder will correspond to the `src` directory on disk within the `firmware` folder of the project

The Main File

This topic describes the logic of the `main.c` file and the C language `main` function in a MPLAB Harmony project.

Description

The C language entry point for a MPLAB Harmony embedded application is the `main` function. This function is defined in the `main.c` file, generated in the project's `app` folder (or `src` directory on disk) by the MHC. The `main` function (see the following example) implements a simple "super loop", commonly used in embedded applications that do not make use of an operating system.

Example main Function Logic

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

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

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

The SYS_Initialize Function

The first thing the `main` function does is to call a function named `SYS_Initialize`. The purpose of the `SYS_Initialize` function is to initialize every software module in the system. MPLAB Harmony is based upon a model of cooperating state machines. Therefore, this function must ensure that every module's state machine is placed in a valid initial state. The implementation of this function is one of the things generated by the MHC to configure a MPLAB Harmony system. This function's definition is generated in the `system_init.c` file, described in the [system_init.c](#) section.

**Note:**

The `SYS_Initialize` function signature has a "void *" input parameter. This is so that it may later be implemented in a library and an arbitrary initialization data structure may be passed into it. However, for a statically implemented `SYS_Initialize` function (which will normally be the case if you implement it yourself), this parameter is unnecessary and can be passed as "NULL".

The "Super Loop"

After all of the modules in the system have been initialized, the `main` function executes an infinite loop to keep the system running. This is commonly called a "super loop" as it is the outer-most loop, within which the entire system operates. This loop never exits. So, the code that exists after the end of that loop should never be executed and is only included there for safety, clarity, and syntactical completeness.

The SYS_Tasks File and the SYS_Tasks Function

Inside of the "super loop", the `main` function calls the `SYS_Tasks` function. The purpose of the `SYS_Tasks` function is to poll every module in the system to ensure that it continues to operate. This is how the system maintains the state machines of all polled modules. (Note that some modules may be interrupt driven and thus, not called from the `SYS_Tasks` function.) The implementation of the `SYS_Tasks` function is generated by the MHC in the `system_tasks.c` file, which is described in the [system_tasks.c](#) section.

The Application File(s)

This topic describes the normal structure of MPLAB Harmony application files.

Description

From the point of view of a MPLAB Harmony system, an application consists of two basic functions:

- `APP_Initialize`
- `APP_Tasks`

The application's initialization function (APP_Initialize) is normally called from the SYS_Initialize function, which is called from `main` before entering the top-level loop. The application's "tasks" function (APP_Tasks) is normally called from the SYS_Tasks function, which is called from `main` from inside the top-level loop. This is how the application's state machine is initialized and "polled" so that it can do its job. The SYS_Initialize function is normally implemented in the `system_init.c` file and the SYS_Tasks function is normally implemented in the `system_tasks.c` file. That is the convention for example and demonstration projects distributed with MPLAB Harmony and that is the case for projects generated by the MHC. You may do as you choose in your own projects, but it is recommended to follow this convention as it will make it easier to manage multiple configurations if you need them and it will be consistent with the MHC and other tools.

Application Initialization

An application's initialization function places the application's state machine in its initial state and may perform additional initialization if necessary. This function must not block and it should not call the routines of any other modules that may block. If something needs to be initialized that may take time to complete, that initialization should be done in the application's state machine (i.e., in its "Tasks" function).

Sample Application Initialization Function:

```
void APP_Initialize ( void )
{
    /* Place the App state machine in its initial state. */
    appData.state      = APP_STATE_INIT;
    appData.usartHandle = DRV_HANDLE_INVALID;
}
```

The sample project's initialization function initializes an internal variable and places the application's state machine in its initial state by assigning the APP_STATE_INIT enumeration value into the "state" member of the data structure that contains all of the data required by the application (appData). This structure is defined globally, but is only ever accessed by the application itself. The application's initialization function is called from the SYS_Initialize function (defined in `system_init.c`), which is called from `main` after a system Reset. Using this technique, the application is initialized (along with the rest of the system) whenever the system comes out of Reset.

Application Tasks

The application's state machine breaks up the job that the application must do into several short "tasks" that it can complete quickly, but between which it must wait for some other module to complete some tasks of its own. (In this case the other module is the USART driver.) Once each short task has completed successfully, the application transitions to another state to perform the next short task.

Example Application Tasks Function:

```
void APP_Tasks ( void )
{
    /* Handle returned by USART for buffer submitted */
    DRV_HANDLE usartBufferHandle;

    /* Check the application's current state. */
    switch ( appData.state )
    {
        /* Keep trying to open the driver until we succeed. */
        case APP_STATE_INIT:
        {
            /* open an instance of USART driver */
            appData.usartHandle = DRV_USART_Open(APP_UART_DRIVER_INDEX,
                                                  DRV_IO_INTENT_WRITE);

            if (appData.usartHandle != DRV_HANDLE_INVALID )
            {
                /* Update the state */
                appData.state = APP_STATE_SEND_MESSAGE;
            }
            break;
        }

        /* Send the message when the driver is ready. */
        case APP_STATE_SEND_MESSAGE:
        {
            /* Submit message to USART */
            DRV_USART_BufferAddWrite(appData.usartHandle, &usartBufferHandle,
                                     APP_HELLO_STRING, strlen(APP_HELLO_STRING));

            if ( usartBufferHandle != DRV_HANDLE_INVALID )
            {
                /* Message is accepted. Driver will transmit. */
                appData.state = APP_STATE_IDLE;
            }
            break;
        }
    }
}
```

```

    }

    /* Idle state */
    case APP_STATE_IDLE:
    default:
    {
        /* Do nothing. */
        break;
    }
}
}

```

Sample Application States

The sample application's "tasks" function breaks the operation of the application down in to the following states using a "switch" statement with the following "cases".

- APP_STATE_INIT
- APP_STATE_SEND_MESSAGE
- APP_STATE_IDLE

The sample application is placed into the APP_STATE_INIT state by the application's initialization function before the "tasks" function is ever called. So, the first time the APP_Tasks function is called, the switch statement executes the code under this case and the first short "task" the sample application attempts to do is open the USART driver to obtain a handle so that it can transfer data over the USART. Notice that the application checks the value of the handle returned from the DRV_USART_Open function to ensure that it is valid before it transitions to the APP_STATE_SEND_MESSAGE state. If the value of the handle returned by the driver's "open" function is invalid (equal to DRV_HANDLE_INVALID), the application stays in the APP_STATE_INIT state and continues trying to open the USART driver every time its "tasks" function is called. This technique allows a polled state machine to wait for something that it requires before continuing to avoid making inappropriate transitions to new states.

Once the application has a valid handle to the USART driver, it executes the code under the APP_STATE_SEND_MESSAGE case the next time its APP_Tasks function is called. In this state, the application calls a USART driver data transfer routine (DRV_USART_BufferAdd) to send the data data buffer string defined by the `system_config.h` header. Then, it checks the handle returned by the DRV_USART_BufferAddWrite function to see if it is valid. If the buffer handle is valid, it indicates that the USART driver has accepted the buffer and will take responsibility for the data transfer from that point forward. The application does not have to do anything else to cause the data transfer to occur. However, if the buffer is not accepted by the driver (in which case the handle returned by the DRV_USART_BufferAddWrite function would be invalid), the application stays in the APP_STATE_SEND_MESSAGE and tries again the next time the APP_Tasks function is called.

Once the application has successfully passed the buffer to the USART driver, it transitions to the APP_STATE_IDLE state where it stays and does nothing any time its "tasks" function is called. Its job is done! A more complex application would go on to some other task or potentially begin the process again. But, this is a simple "Hello World" sample application.



Note: The application is normally initialized last, after all other modules in the system have been initialized. But, it should never assume that any other module has completed its initialization when the application is initialized or when its "tasks" function is first called. Instead, it should always check the return value or status from any other module it calls to ensure that the call succeeded before moving on to the next state. Following this rule makes applications more robust and allows them to handle errors more effectively.

System Configurations

This section describes the files that make up a system configuration.

Description

In MPLAB Harmony, a system configuration consists of a set of files that define the build options, how the system is initialized, and how it runs after it has been initialized. The purpose of each of these files is described in the topics in this section.

system_config.h

This topic describes the purpose of system configuration header file.

Description

System Configuration

In MPLAB Harmony, most library modules require a set of build time configuration options that define a variety of parameters (such as buffer sizes, maximum or minimum limits, and default behavior). To configure a library for your specific needs, its configuration options can be defined using C language preprocessor `#define` statements. The set of configuration options supported is described for each library in the "Configuring the Library" section of its help document and most libraries provide a template and example configuration header files in a `config` sub-folder within their `src` folder.

To obtain its build configuration options, every library includes the same common top-level configuration file that is named `system_config.h`, and it is generated by the MHC as part of your system configuration. The relative directory path to configuration directory that contains this file is

defined in the build properties of your project configuration by the MHC so that the compiler can find it in its include file search path.

Example Configuration system_config.h Header

```
// *****
// *****
// Section: System Service Configuration
// *****
// *****

// *****
/* Common System Service Configuration Options
*/
#define SYS_VERSION_STR          "1.07"
#define SYS_VERSION              10700

// *****
/* Clock System Service Configuration Options
*/
#define SYS_CLK_FREQ              80000000ul
#define SYS_CLK_BUS_PERIPHERAL_1 80000000ul
#define SYS_CLK_UPLL_BEFORE_DIV2_FREQ 7999992ul
#define SYS_CLK_CONFIG_PRIMARY_XTAL 8000000ul
#define SYS_CLK_CONFIG_SECONDARY_XTAL 32768ul

/** Ports System Service Configuration */
#define SYS_PORT_AD1PCFG          ~0xffff
#define SYS_PORT_CNPUE            0x0
#define SYS_PORT_CNEN             0x0

// *****
// *****
// Section: Driver Configuration
// *****
// *****

// *****
/* USART Driver Configuration Options
*/

#define DRV_USART_INTERRUPT_MODE      false
#define DRV_USART_BYTE_MODEL_SUPPORT false
#define DRV_USART_READ_WRITE_MODEL_SUPPORT true
#define DRV_USART_BUFFER_QUEUE_SUPPORT true
#define DRV_USART_QUEUE_DEPTH_COMBINED 16
#define DRV_USART_CLIENTS_NUMBER      1
#define DRV_USART_SUPPORT_TRANSMIT_DMA false
#define DRV_USART_SUPPORT_RECEIVE_DMA false
#define DRV_USART_INSTANCES_NUMBER    1

#define DRV_USART_PERIPHERAL_ID_IDX0   USART_ID_2
#define DRV_USART_OPER_MODE_IDX0       DRV_USART_OPERATION_MODE_NORMAL
#define DRV_USART_OPER_MODE_DATA_IDX0  0x00
#define DRV_USART_INIT_FLAG_WAKE_ON_START_IDX0 false
#define DRV_USART_INIT_FLAG_AUTO_BAUD_IDX0 false
#define DRV_USART_INIT_FLAG_STOP_IN_IDLE_IDX0 false
#define DRV_USART_INIT_FLAGS_IDX0      0
#define DRV_USART_BRG_CLOCK_IDX0        80000000
#define DRV_USART_BAUD_RATE_IDX0        9600
#define DRV_USART_LINE_CNTRL_IDX0       DRV_USART_LINE_CONTROL_8NONE1
#define DRV_USART_HANDSHAKE_MODE_IDX0   DRV_USART_HANDSHAKE_NONE
#define DRV_USART_XMIT_INT_SRC_IDX0     INT_SOURCE_USART_2_TRANSMIT
#define DRV_USART_RCV_INT_SRC_IDX0      INT_SOURCE_USART_2_RECEIVE
#define DRV_USART_ERR_INT_SRC_IDX0      INT_SOURCE_USART_2_ERROR

#define DRV_USART_XMIT_QUEUE_SIZE_IDX0  10
#define DRV_USART_RCV_QUEUE_SIZE_IDX0   10

#define DRV_USART_POWER_STATE_IDX0      SYS_MODULE_POWER_RUN_FULL
```

```
// *****
// *****
// Section: Application Configuration
// *****
// *****
```

```
#define APP_UART_DRIVER_INDEX    DRV_USART_INDEX_0
#define APP_HELLO_STRING        "Hello World\r\n"
```

The previous example defines configuration options for the application, system services, and USART driver used in the "pic32mx795_pim_e16" configuration of the sample project.

system_init.c

This topic describes the purpose of the system initialization file.

Description

In a MPLAB Harmony project, the SYS_Initialization function is called from the main function in order to initialize all modules in the system. This function is implemented as part of a system configuration by the MHC in a file named system_init.c. This file may also include other necessary global system items that must be implemented in order to initialize a system such as processor configuration bits and module initialization global data structures.

Example system_init.c File

```
// *****
// *****
// Section: Configuration Bits
// *****
// *****

/** DEVCFG0 */

#pragma config DEBUG = OFF
#pragma config ICESEL = ICS_PGx2
#pragma config PWP = OFF
#pragma config BWP = OFF
#pragma config CP = OFF

/** DEVCFG1 */

#pragma config FNOSC = PRIPLL
#pragma config FSOSCEN = OFF
#pragma config IESO = ON
#pragma config POSCMOD = XT
#pragma config OSCIOFNC = OFF
#pragma config FPBDIV = DIV_1
#pragma config FCKSM = CSDCMD
#pragma config WDTPS = PS1048576
#pragma config FWDTEN = OFF

/** DEVCFG2 */

#pragma config FPLLIDIV = DIV_2
#pragma config FPLLMUL = MUL_20
#pragma config FPLLODIV = DIV_1
#pragma config UPLLIDIV = DIV_12
#pragma config UPLLEN = OFF

/** DEVCFG3 */

#pragma config USERID = 0xffff
#pragma config FSRSEL = PRIORITY_7
#pragma config FMIIEN = OFF
#pragma config FETHIO = OFF
#pragma config FCANIO = OFF
#pragma config FUSBIDIO = OFF
#pragma config FVBUSONIO = OFF
```

```

// *****
// *****
// Section: Driver Initialization Data
// *****
// *****

const DRV_USART_INIT drvUsart0InitData =
{
    .moduleInit.value = DRV_USART_POWER_STATE_IDX0,
    .usartID = DRV_USART_PERIPHERAL_ID_IDX0,
    .mode = DRV_USART_OPER_MODE_IDX0,
    .modeData.AddressedModeInit.address = DRV_USART_OPER_MODE_DATA_IDX0,
    .flags = DRV_USART_INIT_FLAGS_IDX0,
    .brgClock = DRV_USART_BRG_CLOCK_IDX0,
    .lineControl = DRV_USART_LINE_CNTRL_IDX0,
    .baud = DRV_USART_BAUD_RATE_IDX0,
    .handshake = DRV_USART_HANDSHAKE_MODE_IDX0,
    .interruptTransmit = DRV_USART_XMIT_INT_SRC_IDX0,
    .interruptReceive = DRV_USART_RCV_INT_SRC_IDX0,
    .queueSizeTransmit = DRV_USART_XMIT_QUEUE_SIZE_IDX0,
    .queueSizeReceive = DRV_USART_RCV_QUEUE_SIZE_IDX0,
};

// *****
// *****
// Section: Module Initialization Data
// *****
// *****

const SYS_DEVCON_INIT sysDevconInit =
{
    .moduleInit = {0},
};

// *****
// *****
// Section: System Data
// *****
// *****

/* Structure to hold the object handles for the modules in the system. */
SYSTEM_OBJECTS sysObj;

// *****
// *****
// Section: System Initialization
// *****
// *****

void SYS_Initialize ( void* data )
{
    /* Core Processor Initialization */
    SYS_CLK_Initialize( NULL );
    sysObj.sysDevcon = SYS_DEVCON_Initialize(SYS_DEVCON_INDEX_0, (SYS_MODULE_INIT*)&sysDevconInit);
    SYS_DEVCON_PerformanceConfig(SYS_CLK_SystemFrequencyGet());
    SYS_PORTS_Initialize();

    /* Initialize Drivers */
    sysObj.drvUsart0 = DRV_USART_Initialize(DRV_USART_INDEX_0, (SYS_MODULE_INIT *)&drvUsart0InitData);

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

```

In addition to the SYS_Initialize function implementation, the previous example, the system_init.c file from the "pic32mx_795_pim_e16" configuration of the "sample" project, defines the processor configuration bits, data structures used to initialize the USART driver and device control service, and a global sysObj data structure used for the USART driver Device Control System Service returned by their initialization functions.

Note the `SYSTEM_OBJECTS` data type for the `sysObj` data structure is defined in the [system_definitions.h](#) section.

system_tasks.c

This topic describes the purpose of the system tasks file.

Description

Since MPLAB Harmony modules are state machine driven, they each have a "Tasks" function that must be called repeatedly (from the system-wide "super loop" in `main` or from an ISR or OS thread). The "Tasks" functions are all called from the top-level `SYS_Initialize` function that is normally implemented in a file called `system_tasks.c` that is generated by the MHC as part of a system configuration.

Example `system_tasks.c` File

```
void SYS_Tasks ( void )
{
    /* Maintain system services */
    SYS_DEVCON_Tasks( sysObj.sysDevcon );

    /* Maintain Device Drivers */
    DRV_USART_TasksTransmit( sysObj.drivUsart0 );
    DRV_USART_TasksReceive( sysObj.drivUsart0 );
    DRV_USART_TasksError ( sysObj.drivUsart0 );

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

The `system_tasks.c` file for the "pic32mx_795_pim_e16" configuration of the "sample" project, contains only the implementation of the `SYS_Tasks` function for that configuration. This function calls the tasks function of the Device Control System Service, the USART driver's tasks functions (it has three, one each for transmitter, receiver, and error-handling tasks), passing in the object handle returned from the driver's initialization routine, and it calls the application's tasks function `APP_Tasks` to keep the state machines of all three modules running.

system_interrupt.c

This topic describes the purpose of the system interrupts file.

Description

In an interrupt-driven configuration, any modules (such as drivers or system services) that can be driven from an interrupt must have their interrupt-capable tasks function(s) called from an Interrupt Service Routine (ISR) "vector" function instead of from the `SYS_Tasks` function. The form of the definition of the ISR vector function is dependent on what type of PIC32 microcontroller on which the system is running. So, any vector functions required are normally implemented as part of the a specific system configuration in a file normally named `system_interrupt.c`.

Since the sample application is entirely polled, its `system_interrupt.c` file does not contain any ISR vector functions. Refer to any interrupt-driven demonstration or example application to see how vector functions are implemented.

system_definitions.h

This topic describes the purpose of the system definitions header file.

Description

The system configuration source files (`system_init.c`, `system_tasks.c`, and `system_interrupt.c`) all require a definition of the system objects data structure and an `extern` declaration of it. The MHC generates these items in the `system_definitions.h` header file and the system source files all include that header file.

For example, the sample application defines the following structure definition and `extern` declaration.

```
typedef struct
{
    SYS_MODULE_OBJ sysDevcon;
    SYS_MODULE_OBJ drivUsart0;

} SYSTEM_OBJECTS;

extern SYSTEM_OBJECTS sysObj;
```

This structure holds the object handles returned by the Initialize functions for the device control and USART modules (in `system_init.c`) because they must be passed into the associated Tasks functions (called in `system_tasks.c`), which is why the system global `sysObj` structure requires an `extern` declaration. The MHC generates object handle variables in this structure for every instance of an active module in the system. Additionally, the system configuration source files require the interface headers for all libraries and applications included in the system so that they have prototypes for their Initialize and Tasks functions. In the sample application, the `system_definitions.h` file includes the following interface headers (and standard C headers).

```
#include <stdint.h>
#include <stddef.h>
#include <stdbool.h>
#include "system/common/sys_common.h"
#include "system/common/sys_module.h"
#include "system/clk/sys_clk.h"
#include "system/clk/sys_clk_static.h"
#include "system/devcon/sys_devcon.h"
#include "driver/usart/drv_usart.h"
#include "system/ports/sys_ports.h"
```



Note: The `system_configuration.h` header file should not be included by the application (`app.c`, `app.h`, or other) source files because it provides direct extern access to system objects and these objects should not be utilized by the application directly. The application (or other modules) should only interact with a module through its defined Application Program Interface (API), or client interface, not through the system objects or system functions that require that object.

system_exceptions.c

This topic describes the purpose of the system exceptions source file.

Description

The `system_exceptions.c` source file provides a skeletal implementation of the general exception handler function (shown below), overriding the weak function implementation provide by the MPLAB XC32 C/C++ Compiler that simply hangs in an endless loop.

```
void _general_exception_handler ( void )
{
    /* Mask off the ExcCode Field from the Cause Register.
       Refer to the MIPS Software User's manual. */
    _excep_code = (_CP0_GET_CAUSE() & 0x0000007C) >> 2;
    _excep_addr = _CP0_GET_EPC();
    _cause_str = cause[_excep_code];

    SYS_DEBUG_PRINT(SYS_ERROR_ERROR,
        "\nGeneral Exception %s (cause=%d, addr=%x).\n",
        _cause_str, _excep_code, _excep_addr);

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

If a general exception occurs, this implementation will capture the address of the instruction that caused the exception in the `_excep_addr` variable, the cause code in the `_excep_code` variable and use a look-up table indexed by the cause to provide a debug message describing the exception if debug message support is enabled. Then, the function will hit a hard-coded debug breakpoint (in Debug mode) and hang in a loop to prevent runaway execution.

This implementation is provided to assist with development and debugging. The user is encouraged to modify this implementation to suit the needs of their system.

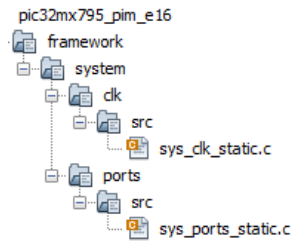
The Configuration-specific "framework" Folder

This topic describes the configuration-specific `framework` folder.

Description

The interface (i.e., API) headers and the source files for the dynamic implementations of all MPLAB Harmony libraries are contained in the main framework folder (`<install-dir>\framework`). However, the MHC generates any static implementations of the MPLAB Harmony libraries. These generated source files are all configuration specific because they are generated with knowledge of the configuration selections. Thus, they are placed in a configuration-specific framework folder. For consistency, the organization of the sub-folder tree of the configuration-specific framework folder matches the organization of the main framework folder.

For example, the configuration-specific `framework` folder for the `pic32mx795_pim_e16` configuration of the sample project contains the source files for the static, MHC-generated implementations of the Clock System Service and the Ports System Service, as shown in the following figure.



Other Configuration-specific Files

This topic describes two other (non C-language) configuration-specific files.

Description

There are two additional (non-C language) files generated by the MHC and placed into the configuration-specific folder. The first file, `<config-name>.mhc`, captures the configuration selections made by the user. The second file, which is always named `configuration.xml`, captures the various checksums and additional information required by the MHC to identify which generated files have been edited externally and to store miscellaneous other information it requires (such as the path to the MPLAB Harmony installation).