MPLAB Harmony Help

MPLAB Harmony Integrated Software Framework

Volume IV: MPLAB Harmony Development

This volume provides information about how to develop MPLAB Harmony-compatible libraries and applications and how to best distribute and integrate them into an existing installation. Information on porting and updating is also included.

Description

Information on the tools, techniques, and knowledge required to develop, distribute, and integrate MPLAB Harmony-compatible libraries and applications into existing installations, as well as updating existing MPLAB Harmony projects to a newer version of MPLAB Harmony is provided throughout the help documentation. This section also introduces key topics and provides a starting point for new developers.

Key Concepts

Explains several key MPLAB Harmony concepts.

Description

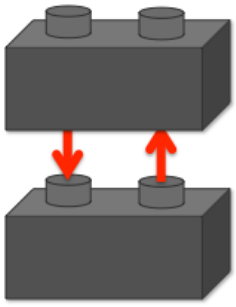
MPLAB Harmony utilizes several key concepts from operating systems and modular and object-oriented programming methodologies. This section introduces these concepts and explains how they affect the design and implementation of MPLAB Harmony libraries.

Modularity and Encapsulation

MPLAB Harmony libraries are modular, which encapsulates resources and allows them to be used as building blocks.

Description

MPLAB Harmony libraries are modular and modular libraries have interfaces that consist of functions that control and provide access to internal variables, registers, or other resources. Every function has input and output and may have some side-effects. In most cases, the side-effect is the real reason to call the function, especially for modules that manage hardware such as device drivers.



By convention, the functions in a library's interface must be treated like black boxes, completely encapsulating internal resources. No other code in the system should access or make any assumptions about what is inside the box. Clients (callers) of the library's interface should only use documented features of the library. Directly accessing internal resources or using undocumented features is dangerous, because there may be a number of different implementations of the library, optimized for different purposes or hardware that may not have those features or resources or that may implement them differently.

Also, using functions to access a library's resources provides a well-defined way for the client and the library to interact. A function can protect shared resources and prevent conflicts that might occur if those resources were accessed directly. A good, modular library will have a well-abstracted and clearly documented interface made up of as small a set of functions as possible that will still do the job the library is intended to do. Well-designed, modular libraries allow users to more easily divide-and-conquer the problems they need to solve instead of managing the details that each library should manage for them.

The set of functions that make up a library's interface must be highly cohesive, relating only to the specific feature provided by the library. They must not relate to control of any common resources or other modules in the system. This reduces coupling between libraries, making them largely independent of each other, improving modularity.

Important!

Since MPLAB Harmony is primarily a source code solution, modularity must be maintained at a source code level. *It is not necessarily at an object code level*. This means that changes in configuration or implementation may require binary object (`.o` or `.a`) files and applications or other clients that call the library to be rebuilt. Therefore, it is permissible to use macros or inline functions that provide a consistent source code interface; however, the library and application may both require recompilation to work properly when the library configuration or implementation changes. This allows MPLAB Harmony systems to more directly benefit from optimized static implementations of libraries. See Static vs. Dynamic for details.

Key One-to-Many Relationships

Describes key one-to-many relationships and how they affect MPLAB Harmony library design and implementation.

Description

There are a number of one-to-many relationships that are key determining factors in the design of a MPLAB Harmony library. How these relationships are managed shapes the way a library is designed and implemented. This section summarizes what these relationships are and describes how they affect MPLAB Harmony libraries. The following table lists and describes the relationships and the related section that describes each one in detail.

One-	-To-Many	Described in Section
Interface	Implementation	Interface vs. Implementation
Implementation	Instances	Implementation vs. Instances and Static vs. Dynamic

Instance	Clients	Single Client vs. Multiple Client

A MPLAB Harmony library has a single interface, but can have many different implementations that are optimized or configured for specific situations. Because of this, there can be multiple mutually exclusive sets of source code for a single library. However, there can be only one interface definition for each type of library. See Interface vs. Implementation for additional information.

Each implementation of a library may manage one or more instances of a peripheral or other resource, depending upon how that implementation was designed and configured. A static implementation can manage only one instance of a resource and a dynamic implementation can mange multiple instances of a (given type of) resource. This relationship informs and shapes the design of both the interface and the implementation of a library as described in Implementation vs. Instances. Also, the assumptions and simplifications that can be made when an implementation of a library supports only a single instance of the resources it manages, which is described in Static vs. Dynamic.

An instance of a library, particularly drivers or interface busses, may need to support multiple independent clients, accessing the same instance of the resource concurrently. In this context, a client is any application or library module in a system that calls the client interface functions of the library in question. How this relationship affects the design of the interface and implementation of a library is described in Single Client vs. Multiple Client.

Interface vs. Implementation

Describes the difference between a library's interface and its implementation.

Description

A library's interface is a contract between an application (or any other client in the system that uses the that interface) and the library. It consists of more than just the data definitions and function prototypes used to call the library. It also comprises knowledge of the intended functionality and usage model of the library.

That knowledge is not fully defined by the library's interface header. The interface header defines the syntax of the library's interface, but it does not usually fully define the meaning of the interface (its *semantics*). And, the semantics may also not fully be defined by the library's source code. Any specific implementation of a library may not actually provide all of the functionality that the library interface defines. Implementations may be partial implementations, optimized or configured for a specific usage. Ultimately, it is the library's documentation (its *Help* section) that defines the library's interface. The Help documentation defines the intended usage models of the driver, the supported configuration options, and the syntax of the interface. However, the exact sub-set of features implemented is selected during configuration. This is a key concept of MPLAB Harmony because of the degree of configurability that can be provided by MPLAB Harmony libraries.

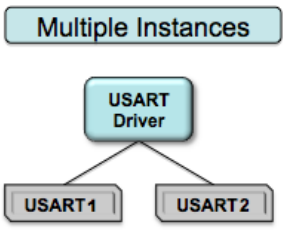
Also, while a MPLAB Harmony library interface may define more options and features than are provided by an implementation of that library, if an implementation does provide a documented feature, it must provide the feature exactly as documented. If it does not, it is not an implementation of the library in question. It is a completely different library. *This is a key requirement.* Maintaining a consistent interface from one implementation to another is necessary to allow different implementations of libraries to be used to satisfy the needs of different configurations without requiring changes to the calling application or to any other client of the library.

Implementation vs. Instances

Describes how a dynamic driver library implementation manages multiple peripheral instances.

Description

For some types of peripherals, it is common to have multiple identical (or nearly identical) instances of that peripheral on a single microcontroller. When that occurs, it is more efficient and effective for a single driver library to manage all instances of the peripheral that are used than it is to write multiple different drivers, with each managing a single instance of the peripheral, as shown in the following diagram.



MPLAB Harmony multi-client drivers are designed to handle multiple clients for the same peripheral instance, which affects both the library's interface and its implementation.

For a single set of functions to manage multiple instances of a peripheral, the functions must accept a parameter that identifies which instance of the peripheral the caller wants to use. This parameter, and exactly how it's used, may be different for different types of interface functions and different implementations. However, the concept can be best illustrated by a driver's system interface functions, as shown by the following (somewhat simplified) example.

Example: Running a Multiple Instance Driver

```
DRV_USART_INIT initData0 = { .usartID = USART_ID_2 };
DRV_USART_INIT initData1 = { .usartID = USART_ID_3 };
SYS_MODULE_OBJ usart0;
SYS_MODULE_OBJ usart1;
```

```
int main ( void )
{
    SYS_MODULE_OBJ usart0;
    SYS_MODULE_OBJ usart1;
    /* Initialize two USART driver instances. */
    usart0 = DRV_USART_Initialize(0, (SYS_MODULE_INIT *)&initData0);
    usart1 = DRV_USART_Initialize(1, (SYS_MODULE_INIT *)&initData1);

    while ( true )
    {
        /* Run USART Driver instance 0 (physical USART 2) State Machines. */
        DRV_USART_TasksTransmit(usart0);
        DRV_USART_TasksReceive(usart0);
        DRV_USART_TasksError(usart0);

        /* Run USART Driver instance 1 (physical USART 3) State Machines. */
        DRV_USART_TasksTransmit(usart1);
        DRV_USART_TasksReceive(usart1);
        DRV_USART_TasksError(usart1);
    }

    return ( EXIT_FAILURE );
}
```

As shown in the previous example, a single set of functions can initialize and run multiple instances of a driver, allowing the driver to initialize and run multiple instances of a peripheral. Notice that the first parameter of the DRV_USART_Initialize function is an index number. That index identifies the instance of the driver being initialized. Think of it as a typical C-language zero-based array index. (Which is what it may actually be in some implementations.) Therefore, the previous example initializes two USART driver instances – USART driver instance 0 and USART driver instance 1. But, that may not be the same as initializing physical USART0 and USART1, as follows.

Note: Normally, you would want to give a driver index number a meaningful label that identifies its purpose, such as `#define MY_DEBUG_USART 0` or `#define MY_WIFI_USART 1`. However, for this example, it is only relevant that they are zero-based indexes.

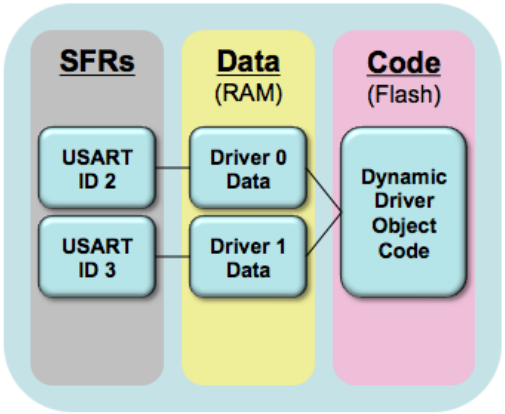
The second parameter is a pointer to a driver-specific data structure that contains any data necessary to initialize the specific instance of the driver and peripheral. This data can include items such as initial baud rates, handshaking settings, or any other data required to initialize the driver and peripheral. That data can be different for different instances or implementations of the driver. For example, different USARTs could be initialized at different baud rates or with different handshaking settings.

After having initialized the driver instances, the previous code example executes an infinite “while” loop, calling the driver’s state machine *Tasks* functions (DRV_USART_TasksTransmit, DRV_USART_TasksReceive, and DRV_USART_TasksError) over and over again to “run” the driver instances.

Note: In this example, the USART driver has three state machines: one to manage data transmission, one to manage data reception, and one handle error events. Some drivers will have only a single state machine, while others may have several depending on the peripheral and the driver implementation details.

Notice that the tasks functions do not use the driver index number to identify the driver instance. Instead, the tasks functions (and all other system interface functions) rely on an object handle (the SYS_MODULE_OBJ data type) to identify the driver instance. You can think of the object handle as a pointer to the driver’s internal data structure, which is the same as indexing into the array of structures using the driver’s instance index number that was passed into the driver’s initialize function. This object handle is used by all other system interface functions to identify the driver instance and the peripheral instance. This method is generally more efficient than using an index variable and it forces the driver’s initialize function to be called before any other system interface function can be called.

One particularly important item in the initialization data structure is the identifier that specifies which physical peripheral instance the driver should use. In this example, the name of that item is `usartID`. Having a physical peripheral ID in the initialization data structure allows any driver instance to be mapped to (or associated with) any desired instance of the physical peripheral. This is a system configuration choice, illustrated by the following diagram.



In this example, a single *dynamic* driver manages two USART instances. To do this, it uses two sets of data variables, identified by the driver instance index. Driver data zero (0) holds the data to manage physical USART peripheral instance 2 (`USART_ID_2`) and Driver Data 1 holds the data to manage physical USART peripheral instance 3 (`USART_ID_3`). The peripheral ID labels (and their values) are defined by the peripheral library (PLIB) that is associated with the type of peripheral managed by this driver (the USART PLIB in this example). But, even though there are multiple sets of SFRs and variables, there is only one instance of the driver object code generated by the compiler. The following code example shows how this might be done.

Example: Dynamic Driver Data Structure

```
typedef struct
{
    USART_MODULE_ID  moduleId;
    uint8_t         *buffer;
    size_t           bufferSize;

    /* Other driver instance data variables. */

} DRV_USART_OBJ;

DRV_USART_OBJ drvUsart[DRV_INSTANCES_NUMBER];
```

The previous code example shows how a dynamic driver implementation could define a structure to contain all of the variables necessary to manage a single instance of a USART peripheral. It also shows how it could allocate an array of those structures using a configuration option (DRV_INSTANCES_NUMBER) to define the number of instances. Assuming that each structure was initialized appropriately and that some caller had passed a buffer of a known size to the driver, the following example shows how this might be used to transmit data from that buffer over the desired USART using the USART Peripheral Library.

Example: Dynamic Driver Indexed Implementation

```
for (i=0; i < drvUsart[index].bufferSize; i++)
{
    if (PLIB_USART_TransmitterIsFull(drvUsart[index].moduleID) )
    {
        break;
    }
    else
    {
        PLIB_USART_ByteSend(drvUsart[index].moduleID, drvUsart[index].buffer[i]);
    }
}
```

Of course, this is just a partial example, but it demonstrates how the driver index number (in an aptly-named variable, `index`) can be used to choose the desired structure from the global array of driver objects (`drvUsart`). In this example, `drvUsart[index]` holds the driver data variables associated with the desired peripheral instance. The variable holding the ID of the physical USART peripheral module is `drvUsart[index].moduleID`, the variable holding the caller's buffer pointer is `drvUsart[index].buffer`, and the size of the caller's buffer is held in the `drvUsart[index].bufferSize` variable. The driver then loops through the buffer and uses USART PLIB functions to send the data to the desired USART.

However, as shown in the first example, the tasks function (and other driver functions) do not normally use the driver instance `index`. Instead, they normally only have access to a driver object handle. That handle is usually just a pointer to the data structure in the driver object array. Therefore, assuming that `pObj = &drvUsart [index]` (from the previous indexed example), the following pointer example shows a more realistic example of how this code might look.

Example: Dynamic Driver Implementation Pointer

```
for (i=0; i < pObj->bufferSize; i++)
{
    if (PLIB_USART_TransmitterBufferIsFull(pObj->plibID) )
    {
        break;
```

```
    }
    else
    {
        PLIB_USART_TransmitterByteSend(pObj->plibID, pObj->buffer[i]);
    }
}
```

In this example, using a pointer is a little more efficient and it does not force a driver's instance data object to be statically allocated. If necessary, it could potentially be dynamically allocated (using `malloc`). However, the specific technique used may be different for different implementations of the library. What is important is that the driver is capable of managing multiple instances of the desired peripheral.

This dynamic, multiple instance driver design mechanism is the technique by which MPLAB Harmony manages the fact that a single implementation of a driver can potentially manage multiple instances of a peripheral. This mechanism allows a single set of driver (or other resource) interface functions to control multiple instances of a peripheral (or other resource), but it affects the definition of both the interface and the implementation, and it is accounted for in the initial design of the driver or library.

Static vs. Dynamic

Describes how a static library or driver implementation can save code space.

Description

The techniques used to implement a dynamic driver (described previously in Implementation vs. Instances) allow one instance of a driver's object code to manage multiple instances of a physical peripheral. However, in systems where only a single instance of a peripheral (or other resource) is used, there may be some savings in code size by utilizing a static implementation.

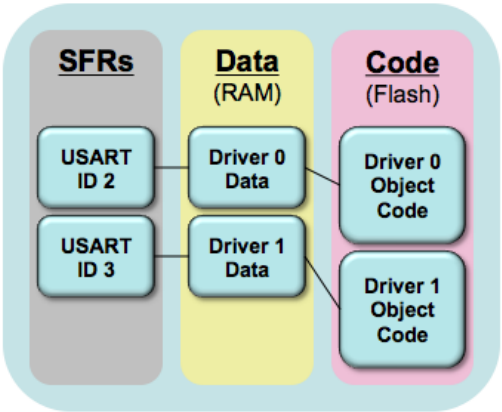
The primary difference between a static driver implementation and a dynamic driver implementation is that the dynamic driver uses an index or pointer variable to access its internal data and the instance of the peripheral it will use. Conversely, a static driver does this by hard-coding the peripheral instance ID directly into the driver's source code (using a constant), as shown in the following example.

Example: Static Driver Implementation

```
for (i=0; i < bufferSize; i++)
{
    if ( PLIB_USART_TransmitterBufferIsFull(USART_ID_2) )
    {
        break;
    }
    else
    {
        PLIB_USART_TransmitterByteSend(USART_ID_2, buffer[i]);
    }
}
```

This technique saves some code size by eliminating the additional code generated to remove the reference to the pointer or index variable. Also, since the peripheral library functions are all defined as inline functions, it allows the compiler to identify the peripheral instance at compile time, instead of generating code to do so at run-time. This results in a significant reduction in generated object code size.

However, there is a drawback to this method. If a system did require multiple instances of a peripheral, this method would result in multiple copies of the driver's object code, each directly accessing different SFRs and different variables, as shown in the following diagram.



Therefore, unlike a dynamic driver where a single instance of the driver's object code manages multiple instances of the data and SFRs, each instance of a static driver requires its own SFRs, data, and object code. This drawback is not a problem if only a single instance of the peripheral is used in the system. Also, overall code size is reduced because a single instance of the object code for a static driver is smaller than a single instance of the object code for a dynamic driver (assuming both implement the exact same functionality). However, two or more instances of a static driver implementation are normally larger than a single instance of a dynamic driver implementation (again, assuming both implement the exact same functionality). This means that while it is possible to use multiple instances of a static driver, there is usually no reason to do so.

In addition, using multiple implementations of a static driver adds some complexity to the process of calling a driver, because the interface (the

function names, parameters, return values, data types, etc.) must remain exactly the same when using either a static and dynamic implementation. If it does not, the source code of any application (or other client module) that uses the library would have to be changed to use the new interface. However, the functions for a static driver do not necessarily need all of the parameters and cannot have the same names as the dynamic driver, as shown by the following examples.

Example: Dynamic Driver Functions

```
SYS_MODULE_OBJ DRV_USART_Initialize ( const SYS_MODULE_INDEX drvIndex,
                                      const SYS_MODULE_INIT * const init );
void DRV_USART_TasksTransmit ( SYS_MODULE_OBJ object );
void DRV_USART_TasksReceive (  SYS_MODULE_OBJ object );
void DRV_USART_TasksError (    SYS_MODULE_OBJ object );
```

The previous function prototypes are part of the USART driver's system interface. These functions are called, as shown in the Implementation Vs. Instances section. They can be called multiple times using different driver index values to initialize and run multiple instances of the driver. However, the equivalent static versions for the first two instances would be as shown in the following example.

Example: Static Driver Functions

```
/* Static USART Driver Instance 0 functions */
void DRV_USART0_Initialize ( void );
void DRV_USART0_TasksTransmit ( void );
void DRV_USART0_TasksReceive ( void);
void DRV_USART0_TasksError ( void );

/* Static USART Driver Instance 1 functions */
void DRV_USART1_Initialize ( void );
void DRV_USART1_TasksTransmit ( void );
void DRV_USART1_TasksReceive ( void);
void DRV_USART1_TasksError ( void );
```

The names must be different because the driver instance number must be encoded into the name to differentiate the functions of one driver instance from the functions of another. Also, the function parameters are different because they can be hard-coded directly into the functions at build time. And, in some cases, even the return values are different because they are no longer needed.

However, client applications and modules must call the interface defined for the dynamic diver. The interface definition must match the interface of a dynamic implementation or no driver would ever be able to provide a dynamic implementation and every client would be hard-coded to a specific instance of the driver. To resolve this difference, mapping functions can be implemented, as shown in the following example.

Example: Mapping Functions

```
#define DRV_USART_INDEX_0  0
#define DRV_USART_INDEX_1  1

extern DRV_USART_OBJ drvUsart0;
extern DRV_USART_OBJ drvUsart1;

SYS_MODULE_OBJ DRV_USART_Initialize ( const SYS_MODULE_INDEX drvIndex,
                                      const SYS_MODULE_INIT * const init )
{
    SYS_MODULE_OBJ obj = SYS_MODULE_OBJ_INVALID;

    switch( drvIndex )
    {
        case DRV_USART_INDEX_0:
        {
            DRV_USART0_Initialize();
            obj = (SYS_MODULE_OBJ)drvUsart0;
            break;
        }

        case DRV_USART_INDEX_1:
        {
            DRV_USART1_Initialize();
            obj = (SYS_MODULE_OBJ)drvUsart1;
            break;
        }

        default:
        /* invalid instance. */
    }

    return obj;
}

void DRV_USART_TasksTransmit ( SYS_MODULE_OBJ object )
```

```
{
    switch( (DRV_USART_OBJ *)object )
    {
        case &drvUsart0:
        {
            DRV_USART0_TasksTransmit();
            break;
        }

        case &drvUsart1:
        {
            DRV_USART1_TasksTransmit();
            break;
        }

        default:
        /* invalid instance. */
    }
}
```

Mapping functions, such as those shown previously, can translate between the dynamic interface called by clients and the static implementation. These functions discard unneeded parameters whose values are hard-coded into the static implementation. They provide constant return values, and they translate dynamic non-instance-specific interface function names to instance-specific static function names. However, this method does add a small amount of overhead.

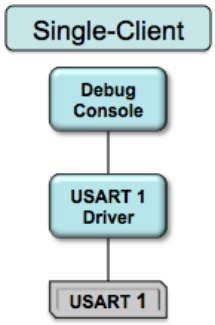
The mapping functions shown previously are for the system interface functions. The client interface mapping functions are a little more complex, particularly if the driver is multiple client, as explained in Single Client vs. Multiple Client.

Single Client vs. Multiple Client

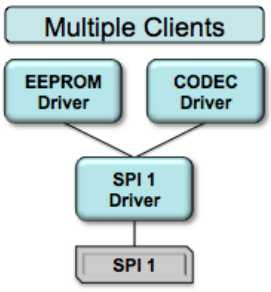
Describes how drivers manage concurrent use by multiple independent clients.

Description

It is common to think of a driver or other library as only being used by a single client, as shown by the following diagram.



However, in a modular environment it is possible that a single driver may need to support multiple, completely independent client modules operating concurrently, as shown by the following diagram.



To do this, the driver module needs to prevent requests from one client from conflicting with requests from another. If a driver does not manage such conflict prevention itself, some method of handing off the driver, such as relay racers passing a baton, would have to be used, and every module that needed to use the driver would have to be aware of all other modules in the system that also needed to use it. This situation would create complex interrelationships between otherwise unrelated modules that would require modification of all such modules every time a new one was added to the system.

To avoid this situation, it is necessary to encapsulate all logic related to both controlling and sharing a peripheral within a single driver library. Doing so allows clients of the shared driver to be completely independent and unaware of other each other. MPLAB Harmony drivers support this by requiring clients to call an open function and obtain a handle before using an instance of the driver, as shown by the following example.

Example: Opening a Driver Handle

```
#define MY_USART_INDEX DRV_USART_INDEX_0
DRV_HANDLE myUsart;

myUsart = DRV_USART_Open(MY_USART_INDEX, DRV_IO_INTENT_READWRITE);
if (myUsart == DRV_HANDLE_INVALID)
{
    /* Retry later or handle as an error. */
}
```

A handle (`myUsart`, in the previous example) is a value that identifies both the client that is calling as well as the instance of the driver that the client intends to use. To identify the instance of the driver (and thus the physical peripheral), the open function that accepts the same instance index value used by the driver's initialize function as a parameter. (It is the only other interface function to do so.) This allows the client to specify the instance of the driver with which it wishes to interact. The open function then returns a unique opened instance handle value that is different from the value it provides to any other active client.

The opened handle is the link between client and driver. The client must pass this handle into all other client interface functions provided by the driver (as shown in the following example). Doing so, allows the driver to identify which client is calling and which instance of the driver it wants to use.

Example: Using a Driver Handle

```
char   buffer[MY_BUFFER_SIZE];
size_t count;

count  = DRV_USART_Read(myUsart, buffer, MY_BUFFER_SIZE);
if (count < MY_BUFFER_SIZE)
{
    /* Did not transfer all data. */
}
```

This technique is very similar to the mechanism used by most file systems where a caller must first open a file and obtain a file handle or descriptor before it can read or write data to or from the file. This mechanism is in fact the distinguishing feature of a MPLAB Harmony driver, as apposed to system services or middleware libraries that do not commonly have an open function.

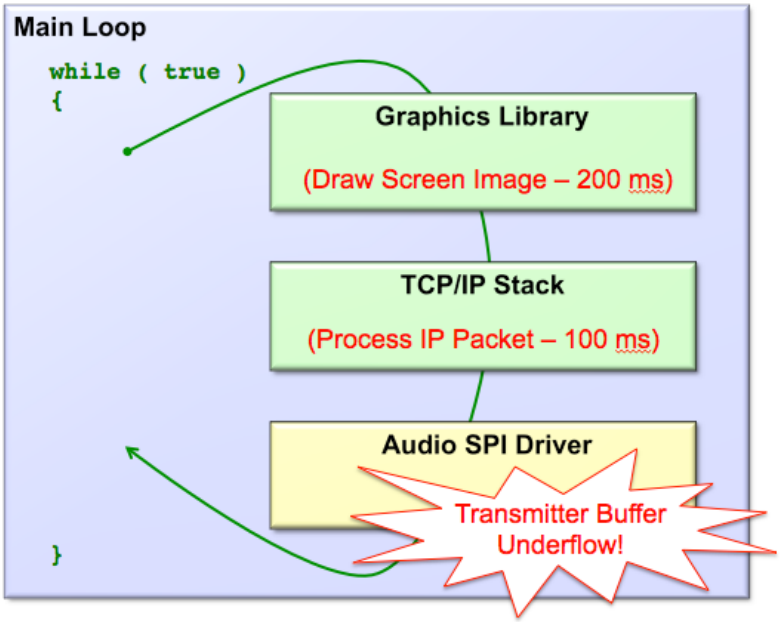
State Machine Programming Model

Explains why MPLAB Harmony uses a state machine-based programming model.

Description

Why does MPLAB Harmony utilize a state machine-based programming model?

The state machine programming model may seem a bit unfamiliar if you are not used to dividing up the job that your software performs into expeditious tasks. However, it soon becomes necessary in the polled *super loop* environment with which most embedded developers are very familiar. Once an application acquires a significant set of capabilities, the processing required by all of the necessary modules becomes too much to simply do one thing at a time, as shown in the following diagram.



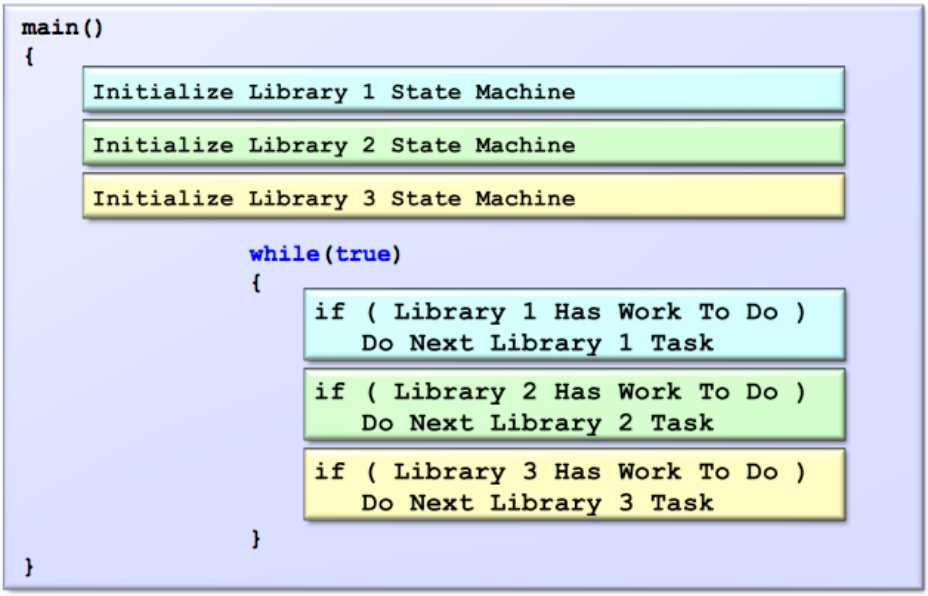
In this example, an application utilizes a graphics library, a TCP/IP networking stack and a UART driver working within the same main loop. In this system it is very likely that by the time the graphics library finishes drawing a complete screen image, followed by the TCP/IP stack processing a large network packet, the SPI driver sending audio data to a Codec is going to underflow. Even if the SPI driver is interrupt-driven, it must still work

out of a buffer that is very likely filled with data and managed by an application or other module that executes in the polled loop. Unfortunately, just using a bigger buffer is ultimately a losing proposition.

In this situation it becomes very difficult, if not impossible, to effectively share the processor execution bandwidth between these different modules. Also, simply using a faster processor will not always solve this issue. The root cause of the problem is not the processor speed or memory size. It is the one-thing-at-a-time blocking approach being used. Such libraries frequently spend a large amount of time polling in a loop, doing nothing but waiting for something to occur in the real world. The problem is often not the processor workload; it is the time spent *busy waiting*.

To solve this problem, a library's work must be broken down into smaller tasks that can be performed from start-to-finish in an acceptable amount of time. Any time a library must wait for some significant period of time for something to occur, it should not monopolize the CPU and block the rest of the system. Instead, it should release the processor to do something useful and let the rest of the system execute. To do this requires either the capabilities of an operating system (discussed in Interrupt and Thread Safety), or it requires breaking up the application's job into smaller tasks and using a state machine methodology to step through these short tasks as it transitions from one state to another. This is the reason why MPLAB Harmony libraries are designed using a state machine-based model, which facilitates the creation of incredibly capable systems, either with or without an OS.

The best way to understand the overall picture of how a MPLAB Harmony system executes is to examine a strictly polled configuration, as shown in the following diagram.



This diagram shows a simplified view of a polled super loop system with three library modules. Each library has its own state machine. Before entering the main loop, the system calls the initialize function for each library module. This puts each library's state machine in its initial state. Then, the system enters the main loop where it repeatedly and endlessly calls each library's tasks function. Each library's tasks function checks the current state of the driver and, if it has some work to do, performs the next task necessary to keep that library running. If a library's current state indicates that it has no work to do, its tasks function does nothing and returns immediately. This method keeps all of the libraries in the system running by allowing them to share the processor and only use it when they have work to do.

While explained in more detail in Interrupt and Thread Safety, it is also worth noting here that libraries use this same mechanism when they are interrupt-driven or in a RTOS environment.

When interrupt-driven, a library's tasks function is no longer called from the main loop. Instead, it is called from the appropriate Interrupt Service Routine (ISR). It behaves in exactly the same way (checking its state, which includes its interrupt flag if it has an associated interrupt) and only taking action when appropriate. Checking the flag may seem redundant since the ISR is only called when it is set, but it is also more robust as it avoids taking inappropriate action when a spurious interrupt occurs or if ISR vectors are shared between multiple interrupt flags as they are in some processors.

When operating in a RTOS-based environment, a library's tasks function can either be interrupt-driven as described previously or polled. If it is polled, it is polled from loop within a RTOS thread and it works exactly the same as if it were polled from a *bare metal* (i.e., no OS) super loop.

The only differences between polled super loop-driven, interrupt-driven, and RTOS-driven operation of a library is that it has to be configured correctly at build time to use the OSAL and/or enable and disable its interrupt(s) when appropriate. These configuration options affect how certain internal functions are implemented, but they do not change the logic or fundamental operation of the library itself.

Blocking Guidelines

Describes the guidelines and limitations on driver function blocking behavior.

Description

In general, no MPLAB Harmony library or application should waste CPU processing power by performing any sort of *busy waiting* loop. Specifically, under a bare metal (i.e., no OS) configuration, a module's functions must not block waiting for external operations to complete (especially on anything that has any possibility of never completing) or it may block the entire system. If the function must wait for external operations the module must break up the operation into smaller tasks to be performed later. This is how to develop a module's state machine; by

identifying when and where the module must wait for the module to complete some task.

For example, the job of a driver is to manage the sequence of interrupts that a peripheral supplies. When a client calls an API function, the driver must do whatever is necessary to ensure that the first interrupt will occur. Then, when the interrupt occurs, the driver's state machine tasks function(s) must do everything necessary to ensure that the next interrupt occurs. This sequence must continue until the entire job, as requested by the client, is completed.

If the client needs to know when the operation completes, the driver can either provide a status function which the client can poll (from its own state machine, not in a blocking loop) until the operation completes or it can provide a way for the client to register a function pointer that will be called later by the driver. The latter mechanism is commonly called a *callback* function.

If a library's functions require an excessive amount of time to complete, even if it is doing useful processing the entire time, it becomes very difficult to create a working system in a bare metal environment. Unfortunately, determining how much time is or is not excessive cannot be normally done when a library is developed because it is entirely dependent upon the requirements of a given application.

The fundamental idea is that the processor must be shared. It should always be doing some useful work (filling a FIFO, processing some data, making useful decisions, etc.). Any time the processor is busy waiting on some event, it is wasting time and energy. Therefore, the following blocking guidelines provide a good basis for developing interoperable MPLAB Harmony libraries.

- Never block on any condition whose timing is under control of an external signal or that has a possibility of never happening at all
- Do not block using a "busy waiting" loop that does nothing, but repeatedly check for some condition to change, even if the condition is based on something internal to the processor that will always happen in a timely manner (with a potential exception, described in the following paragraphs)
- For best interoperability with other modules in the system, limit the amount to time spent processing in a function (even doing useful work) to a reasonable period of time, as follows

Ultimately, it is all a matter of how long a function takes to execute. Any sort of busy waiting loop is undesirable and should be avoided if it is possible. But, sometimes it is more effective to waste a (very) few microseconds than it is to entirely switch context and spend several milliseconds. Following these general timing guidelines are intended to help ensure that your drivers and libraries are flexible and interoperable.

- Functions taking less than 10 µs are ideal
- No function should perform a busy waiting loop for more than 50 µs
- The outside limit for time spent in a function (busy working, not waiting) is around 200 µs. Functions taking longer than that are problematic at best.
- Functions taking any number of milliseconds are unacceptable unless using a RTOS mechanism capable of switching tasks

Blocking on anything outside of the processor's control is not a good idea. For example, doing nothing and looping until the UART receives data is usually not a good idea because it will block the rest of the system until it does. It's also not a good idea to block in a loop until UART data is transmitted, because how long it takes to transmit the data depends on the BAUD rate and slow BAUD rates will cause excessive delays (greater than 200 µs).

However, this should not be interpreted as blocking is always bad. For example, it is a very good idea to loop writing bytes to the UART transmitter FIFO until it is full, because any FIFO can only hold a fixed number of bytes and once it is full, the hardware may operate independently of the software until it runs low on data again.

Also, blocking in a RTOS environment is not only acceptable it is encouraged. It is one of the advantages of using a RTOS and it effectively becomes an optimization. However, blocking in a RTOS environment (for a MPLAB Harmony library) always occurs inside of OSAL functions and within an appropriate thread. See the section on Interrupt and Thread Safety and the topic on Synchronization for additional information on how to utilize the OSAL to support blocking and to synchronize between tasks and interface functions.

Porting from MLA to MPLAB Harmony

Information on updating or porting to this release of MPLAB Harmony, integrating third-party libraries, or packaging and distributing MPLAB Harmony projects.

Description

Please read this section if you are attempting one of the following tasks:

- Updating an existing MPLAB Harmony project from a previous release of MPLAB Harmony
- Porting to MPLAB Harmony from the Microchip Libraries for Applications (MLA)
- Integrating your own or third-party libraries into this MPLAB Harmony installation
- Packaging and distributing your own MPLAB Harmony projects

Updating From a Previous Release

Describes how to update an existing MPLAB Harmony project to a newer release of MPLAB Harmony.

Description

The client interfaces (function signatures and data type definitions) of new releases of production-ready MPLAB Harmony libraries maintain backward-compatibility with previous versions. Therefore, no changes to existing applications are required when updating an existing MPLAB Harmony project from a previous release of MPLAB Harmony to a newer release (see **Note 1**). If a library used by an existing MPLAB Harmony project was updated in a new release, the MPLAB Harmony Configurator (MHC) should be used to update the project's configuration-specific files to avoid potential build errors. The update process, which is quite simple, is as follows:

1. Copy the existing project folder to the same relative location in the `<install-dir>/apps` folder of the new MPLAB Harmony installation (see **Note 2**).
2. Using MPLAB X IDE, open the existing project.
3. Start MHC and open the project's first (or only) configuration. Typically, it is not necessary to make any changes to the configuration selections; however, you may want to review them to determine whether you want to use any newly available options.
4. In MHC, click **Generate** to regenerate and save the project's configuration using the options available in the new installation of MPLAB Harmony.
5. Build and test the project's current configuration. The project will build and behave functionally as it did before while taking advantage of any improvements in the implementations of any libraries used by it.
6. Repeat the process for any additional configurations and projects that were also updated.

This process may also update the format of the files used by MHC to store the selections for the current configuration of the project. Therefore, it is recommended to always perform this process when updating an existing MPLAB Harmony project to a new installation of MPLAB Harmony. All configurations of all demonstration and example projects provided with an installation will always be updated to use the version of MHC and the libraries in the installation with which they are provided; therefore, it is not necessary to update them and regenerate their configurations *unless* you modify them.

Notes:

1. Interface changes to alpha and beta releases of MPLAB Harmony libraries are possible and production releases of a library may add new interface functions; however, existing interface function signatures will always stay the same on production MPLAB Harmony libraries to maintain backward-compatibility in new releases.
2. Currently, MPLAB Harmony projects only support relative paths to source files. To avoid complications when updating projects to a new release, it is necessary to maintain the same relative path from the project folder to the source files in the installation's `framework` folder. The easiest way to ensure this is to create MPLAB Harmony projects within the `apps` folder of the MPLAB Harmony installation, and maintaining the same relative location in the `<install-dir>/apps` folder hierarchy when updating the project for a new installation.

Porting to MPLAB Harmony

This section provides information on porting an existing Microchip Library of Applications (MLA) library to a MPLAB Harmony library.

Description

If you are a Microchip Libraries for Applications (MLA) user, porting your application from the MLA TCP/IP, File System, USB Device, Graphics, or peripheral libraries to the MPLAB Harmony equivalents, be sure to first consult the information in Creating Your First Project, and then refer to the following porting help for additional guidance:

- Peripheral Library Porting Example
- File System Service Porting Guide
- TCP/IP Stack Library Porting Guide
- USB Device Stack Library Porting Guide

Integrating Third-Party Products

Describes how to integrate third party libraries and applications into an existing MPLAB Harmony installation.

Description

This topic describes how and where to best integrate the third-party products into MPLAB Harmony. To utilize third-party libraries and applications, it is best to insert them into an existing MPLAB Harmony installation. To do this, the library or application files need to be copied into the appropriate directory within the installation and changes may need to be made to one or more top-level Hconfig (`*.hconfig`) files to enable a third-party library to appear within the MHC.

Application Installation

Third-party (and other) applications should be installed into a sub-folder of the `<install-dir>/apps` folder within an existing MPLAB Harmony installation. Refer to Packaging and Distributing Applications for additional details.

Library or Other Third-Party Installations

Libraries, tools, and other third-party collateral to be integrated into a MPLAB Harmony installation should be installed into a sub-folder of the `<install-dir>/third_party` folder within an existing MPLAB Harmony installation. Third-party RTOS offerings should be installed into a sub-folder of the `<install-dir>/third_party/rtos` folder.

MHC Integration

The MHC plug-in provides an open and expandable environment into which third-party (and other) libraries may be integrated without any need to edit the source code of the MHC plug-in. To generate the configuration options tree, the MHC plug-in reads a hierarchy of Hconfig (`*.hconfig`) files that are simple text files that follow a specified format and syntax.

To support inclusion in and configuration by the MHC, third-party products must provide their own Hconfig text file(s) that follow the Hconfig grammar. This grammar allows definition of menus and configuration options that, once integrated into the hierarchy of Hconfig files for a specific MPLAB Harmony installation, will be presented to the user for selection. To perform this integration properly requires modification of the appropriate top-level Hconfig file to insert a "source" command at the appropriate location to make the MHC aware of the third-party product. The following identifies the appropriate files.

Application-specific Hconfig File:

`<install-dir>/apps/<project-dir>/firmware/<project-name>.hconfig`

Insert application-specific modifications in this file under the `#insert application-specific configuration here` comment and before the `endmenu` statement for the application configuration menu.

Third-Party Hconfig File:

`<install-dir>/third_party/config/third_party.hconfig`

Insert this file as a source statement for any provided third-party library's Hconfig file after the menu `Third Party Libraries` statement and before the associated `endmenu` statement.

Once the third-party library's Hconfig file has been integrated into the MHC, the configuration options provided by it will be presented to the user and the library can be properly inserted into the projects. The MHC will then create definitions for any selections made and will capture them in the `.mhc` file of any project that uses the library. These definitions are then available for use by FreeMarker template (`*.ftl`) files (which must also be provided by the third-party library) to generate any necessary configuration code required by the library.

Refer to the *Utilities Help* > *MPLAB Harmony Configurator Help* > MPLAB Harmony Configurator Developer's Guide section for additional details.

Packaging and Distributing Applications

This topic describes recommended practices for packaging and distributing MPLAB Harmony applications.

Description

Use the practices recommended in this section to easily utilize libraries and tools provided by MPLAB Harmony when packaging and distributing applications outside of the original development workstation.

Licensing

Libraries distributed in the MPLAB Harmony installation are governed by Microchip and/or third-party Software License Agreements (SLAs). Refer to Software License Agreement for complete licensing information. A copy of this agreement is available in the `<install-dir>/doc` folder of your MPLAB Harmony installation.

Packaging

Use the installation or compression tool of your choice. Capture or compress the project folder in place, from within the `apps` folder hierarchy of the MPLAB Harmony installation in which it was developed such that, upon extraction, the project will install into a sub-folder of the `apps` folder that is at the same level in the folder hierarchy, relative to the installation root folder in the installation into which it is extracted. For example, if the original project was developed using MPLAB Harmony v1.00 under the `C:\Microchip\harmony\v1_00\apps\my_app` folder, it should extract

into the `C:\Microchip\harmony\v1)03\apps\my_app` folder when extracted into the MPLAB Harmony v1.03 installation (assuming default installation locations on Windows development workstations.

This is necessary because all MPLAB Harmony projects refer to common library files using paths relative to the installation root folder (ex. `C:\Microchip\harmony\v1_00` for MPLAB Harmony v1.00 or `C:\Microchip\harmony\v1_03` for MPLAB Harmony v1.03). Following this method will avoid build failures caused by moving the project from one installation to another. Build failures may still occur if library interface (API) functions are used by the project not supported in the MPLAB Harmony installation into which the project is extracted. New releases of production-level MPLAB Harmony libraries provide backward API compatibility. However, API changes are likely in new releases alpha-level libraries and possible in new releases of beta-level libraries.

When distributing projects for use outside of an existing MPLAB Harmony installation, the easiest method is to use the MPLAB X IDE "package" facility (see "package" within the right-click menu for the project from within the Project window in MPLAB X IDE).

When packaging a MPLAB Harmony project, it is not necessary to include generated such as the `.o`, `.d`, and `.mk` files are automatically generated by MPLAB X IDE. These files are, in fact, not source files and they will be regenerated by MPLAB X IDE when needed.

This includes the following:

- Object (`.o`, `.d`, `.elf`) and Hex files
- MPLAB X IDE project files:
  - Shell script (`.bash`) files
  - `nbproject/Makefile-*`
  - `nbproject/Package-*`
  - `nbproject/private`
  - `nbproject/private/*.*`

(where `nbproject/private` is a directory.)

Prebuilt Libraries

This section provides information on the MPLAB X IDE projects that are provided to build binary (i.e., .a file) versions of key MPLAB Harmony libraries.

Introduction

This section describes the MPLAB X IDE projects that are provided to build binary (i.e., `.a` file) versions of key MPLAB Harmony libraries.

Description

Source code is provided for most of the libraries included in the MPLAB Harmony installation. However, some MPLAB Harmony libraries are provided in prebuilt binary format (`.a` file) because source code is not released or to provide optimal performance for users of the free version of the compiler. The MPLAB X IDE projects used to build these libraries are provided so that you can rebuild these libraries with different build parameters, optimization settings, and/or debug symbols if desired.

The Build Projects that are provided in MPLAB Harmony are located in the following folder: `<install-dir>/build/framework`.

Note: Building these libraries with high optimization settings requires a fully licensed version of the necessary MPLAB XC compiler.

Supported Libraries

This topic provides information on the supported libraries that are provided in MPLAB Harmony.

DSP Fixed-Point Math Library

This section describes MPLAB X IDE projects used to build the MPLAB Harmony DSP math libraries.

Description

The DSP Fixed-Point Math Library consists of routines that are optimized in assembly to take advantage of the microAptiv™ core of PIC32MZ devices. The library operates on fixed point integers, which are scaled to represent floating point numbers. Many functions are available in both 16-bit and 32-bit numerical formats.

The library contains functions for mathematical operations on a vector (or array) of values, complex scalar values and matrixes. Operations include add, subtract, multiply, power, data generation, and in the cases of vectors more complex functions including statistics.

The DSP Math Library also contains a full set of digital filtering functions which include FIR and IIR primitives, as well as more complex architectures (e.g., cascade and parallel bi-quad structures).

Finally the library contains a number of transform functions include FFT and inverse FFT, as well as a number of windowing functions.

Some functions in the DSP Math Library require the use of the LibQ Fixed Point Math library as well. In those cases the LibQ library must also be installed with your project. Details on dependent functions are found in the DSP Fixed-Point Library section within the remarks section for that specific function.

LibQ Fixed-Point Math Library

This section describes MPLAB X IDE projects used to build the MPLAB Harmony fixed-point math LibQ libraries.

Description

The LibQ Fixed-Point Math Library consists of routines in optimized assembly for performing advanced mathematical operations on a scalar. This is similar in construction to the floating point equivalent `math.h` file that is included with the compiler. All functions are performed using fixed-point integer operations and generally available in both high precision and low precision variants. Lower precision functions generally perform faster and may be suited for time critical operations.

PIC32 Bluetooth Stack Library

This section describes MPLAB X IDE projects used to build the MPLAB Harmony PIC32 Bluetooth Stack libraries.

Description

The PIC32 Bluetooth Stack Library is provided only in binary form. It consists of a large number of routines that enable the interface of a PIC32 system to a Bluetooth radio via a Hardware Communication Interface (HCI) controller and a UART port. The communication is enabled by a Simple Secured Pairing (SSP) and data is transmitted through the Bluetooth Serial Port Profile (SPP).

The SPP-only library, `lib_bluetooth_spp_only_<version>.a`, is located in the directory `<install-dir>/bin/framework/bluetooth`.

Support for audio is provided as well, as part of a premium package, in the library, `lib_bluetooth_<version>.a`, which is located in `<install-dir>/bin/framework/bluetooth/premium/audio`.

Premium Bluetooth Audio applications also need at least one audio decoder. The Bluetooth specification requires that every Bluetooth audio application support a Sub-Band Coding (SBC) decoder. Support for a SBC decoder is provided in the library, `lib_sbc_decoder_<version>.a`, which is located in the same directory as the Premium Bluetooth Stack Library: `<install-dir>/bin/framework/bluetooth/premium/audio`.

The library interface allows for an external header file to control the port via Service Discovery Protocol (SDP), so customized interfaces are possible through the API. All API functions are exposed in the header file group, which is located within the `<install-dir>/framework/bluetooth/cdbt` folder of your MPLAB Harmony installation.

Assembly Language Issue

There is an assembly language issue in the SBC Decoder Library that misaligns the stack pointer. Contrary to the MIPS programming specification, this error is benign except when the application is built for a device with a hardware Floating Point Unit (FPU), such as the PIC32MZ EF family of devices. In this case, the sub-routine context-saving operations will produce an exception and stop the application. A work around exists for this issue, which involves disabling FPU context-saving for Interrupt Service Routines (ISRs).

The work around is to convert all ISRs from:

```
void __ISR(vector,ipl) MyInterruptHandler(void)
```

and change them to:

```
void __attribute__((vector),interrupt(ipl),nomips16,no_fpu)) MyInterruptHandler(void)
```

where, `vector` and `ipl` are replaced with the actual interrupt vector and interrupt priority level.

So as long as no FPU operations occur in ISRs and the routines are called by ISRs, this work around will prevent the misaligned stack pointer from causing an exception. If FPU operations result from an ISR with no_fpu enabled, a general exception will result.

Peripheral Libraries

This section describes MPLAB X IDE projects used to build the MPLAB Harmony peripheral libraries.

Description

The MPLAB Harmony peripheral libraries are implemented almost entirely using C language inline functions. This is done for efficiency so that the compiler can generate a few simple instructions in place of multiple layers of function calls when function parameters are passed as constants. However, if the project that uses the peripheral library is built with low optimization settings, the compiler may generate a function call instead of "inlining" the function implementation. Unfortunately, this will cause an undefined symbol error at link time if the linker cannot find an actual implementation of the function to which it can link the call.

To satisfy the linker (and to provide optimized PLIB operation, even when low compiler optimization settings are used in your project), the MPLAB Harmony peripheral libraries must be prebuilt as `.a` file, linkable libraries. The appropriate MPLAB X IDE peripheral library `.a` file for the processor in use must be added to the MPLAB X IDE project so that the linker can use them.

Note: Prebuilt binary `.a` files are provided for all supported processors in the MPLAB Harmony installation in the following folder: `<install-dir>/bin/framework/peripheral`.

The MPLAB Harmony peripheral library `.a` file build projects are provided so that you may rebuild the binary peripheral library `.a` files using any desired optimization settings. However, if you do this, you must have the appropriate XC compiler and you must copy the resultant `.a` file to your project folder and add it to your MPLAB X IDE project.

Build Configurations

There are two MPLAB X IDE configurations from which to choose when building peripheral libraries for MPLAB Harmony:

- device_with_ethernet - Select this configuration if the device for which the PLIB `.a` file is to be created *has* an Ethernet module. The default device for this configuration is PIC32MX795F512L.
- device_without_ethernet - Select this configuration if the device for which the PLIB `.a` file is to be created *does not* have an Ethernet module. The default device for this configuration is PIC32MX450F256L.

The default Optimization level for both configurations is O3.

Key Configuration Options

To build the peripheral library binary `.a` files, there are two key configuration options that must be defined, as follows:

```
#define PLIB_INLINE_API extern
#define PLIB_INLINE static inline
```

These two configuration items are defined in the source file (`peripheral.c`), which then directly includes the peripheral library implementation headers.

Note: This has already been done for you. This information is provided strictly for reference.

During normal peripheral library use, both of these options are defined as "extern inline" so that the compiler can choose between generating a

function call or generating inline code, directly in the calling function (as described previously). However, to build the binary library `.a` files the peripheral library Application Program Interface (API) functions must be exposed as global "external" symbols. The "PLIB_INLINE_API" attribute is placed as an attribute on all PLIB API function implementations. So, defining it as "extern" allows the compiler to expose these functions within the library `.a` file so that the linker can find them. The VREG and other "internal" functions prefixed by the "PLIB_INLINE" attribute are then defined as "static inline" functions for efficiency.

Rebuilding the Prebuilt Libraries

This topic provides information on how to use the MPLAB X IDE projects provided in MPLAB Harmony to rebuild the prebuilt libraries.

Description

Prerequisites

To build MPLAB Harmony libraries, you must have the following installed on your development workstation:

- MPLAB X IDE
- MPLAB XC32 C/C++ Compiler
- Appropriate compiler license for your desired optimization settings

Note: Refer to the Release Notes for the specific version numbers. A PDF copy of the release notes is provided in the `<install-dir>/doc` folder of your installation.

Using the MPLAB X IDE Projects

Do the following in MPLAB X IDE:

1. Select the appropriate configuration that is applicable for your device.
2. Select the desired processor in the project properties in MPLAB X IDE.
3. Select the desired optimization levels, debug, capabilities, etc.
4. Build the project to generate the `.a` file.

A post-build step in the project will copy the output (`.a`) of the build process to the `<install_dir>/bin` folder.

Note: Refer to the MPLAB X IDE Help for detailed information on using the Integrated Development Environment.

MPLAB Harmony Compatibility Guide

This section provides information for making software libraries compatible with MPLAB Harmony.

1: Objective

The objective of this document is to provide a set of design, implementation, and porting guidelines for making software libraries compatible with MPLAB Harmony.

Description

Following 2: Overview, MPLAB Harmony compatibility guidelines are broken down into the following primary subsets.

- 3: Modularity Guidelines
- 4: Flexibility Guidelines
- 5: Testing Guidelines
- 6: Documentation Guidelines

Modularity guidelines ensure that libraries do not interfere with each other. Adherence to modularity guidelines is essential to making libraries compatible and interoperable with other MPLAB Harmony libraries. Flexibility guidelines affect the number of different environments in which a library can be used as well as the ability of a library to be customized for any particular solution. Testing guidelines make recommendations on how to verify correct and robust behavior of libraries. And, documentation guidelines describe the recommended documentation support for MPLAB Harmony compatible libraries.

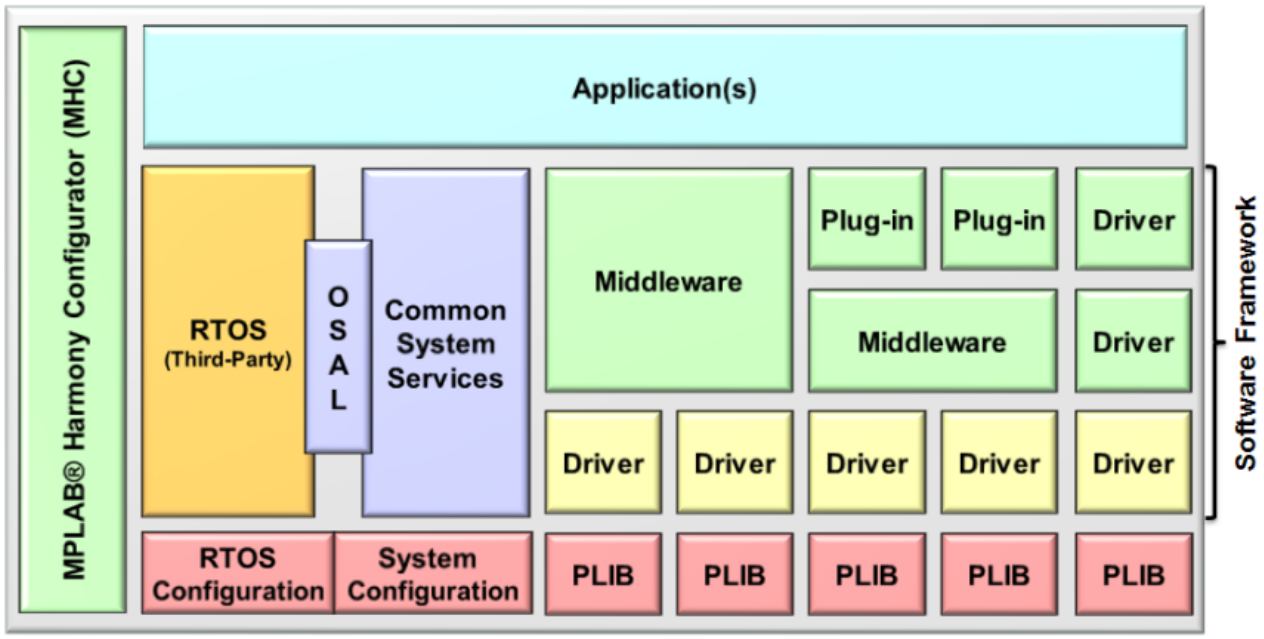
A range of compatibility and compliance to these guidelines is possible. One key example is that it is not necessary that a library support all possible execution models to be considered MPLAB Harmony compatible. However, it does affect the level of flexibility supported and restrictions must be clearly documented. Another key example is that libraries that are released as part of the MPLAB Harmony framework follow more stringent coding style and naming convention guidelines than can be required of pre-existing libraries. This document describes the acceptable levels of compatibility with each rule.

2: Overview

This section provides an overview of MPLAB Harmony and describes libraries and the execution model.

Description

MPLAB Harmony is a modular, layered, cross-microcontroller, RTOS-friendly, integrated software framework, illustrated by the following block diagram.



2.1: Libraries

MPLAB Harmony libraries provide simple C-language application software interfaces to peripherals and middleware supported by Microchip microcontrollers

Description

A MPLAB Harmony library can be an active state machine driven module or a fully re-entrant library with no internal state of its own. Libraries are highly adaptable to support a wide variety of hardware, RTOS, and functional configurations. All modules are designed to be fully interoperable, respecting the abstractions of other modules in the system. Modules only access the resources owned by other modules through the interfaces provided by them, never directly. Libraries may be dynamic, supporting multiple client modules and/or multiple identical instances of themselves, or they may be static, supporting a single client and/or single instance operation, saving code size when dynamic behavior is not required.

Device drivers are library modules that support internal (on-chip) or external (off-chip) peripherals and provide abstracted file-system like interfaces that stay consistent across different implementations to simplify porting from one Microchip microcontroller or system configuration to another. System services help prevent conflicts between modules by providing access to common resources such as interrupts, clocks, and I/O ports. Drivers and services may be built upon part-specific Peripheral Libraries (PLIBs) or they may use other driver and/or service modules if they need the resources owned by them. PLIBs provide a functional breakdown of internal (on-chip) peripherals and hide register details while providing direct (unprotected) low-level access to microcontroller hardware. PLIBs do not maintain any internal state data, except that which is maintained by the hardware itself, and are primarily implemented as inline functions. Middleware libraries support complex protocols such as USB and TCP/IP, or advanced functionality, such as generation of display graphics and cryptographic capabilities. Finally, the Operating System Abstraction Layer (OSAL) enables execution of MPLAB Harmony libraries in supported third-party RTOS environments and it can be effectively removed to support non-RTOS environments.

2.2: Execution Model

Describes the Polled Execution Model.

Description

In its most basic configuration, the MPLAB Harmony framework follows a polled non-blocking (cooperative multi-tasking) execution model within a system-wide "super loop" as shown in the following pseudo-code example.

MPLAB Harmony Polled Execution Model

```
main()
{
    Initialize State Machine 1
    Initialize State Machine 2
    Initialize State Machine 3

        while(true)
        {
            if ( Task 1 Condition )
                Do State Machine 1 Task
            if ( Task 2 Condition )
                Do State Machine 2 Task
            if ( Task 3 Condition )
                Do State Machine 3 Task
        }
}
```

Each module (library or application) can implement its own state machine functions, which can be called from the main “super loop" (or from an appropriate interrupt service routine, for libraries that support an interrupt-driven execution model). Libraries can support task/thread safety and blocking behavior by calling OSAL functions, allowing them to be used with any OS for which the OSAL has been implemented. However, MPLAB Harmony does not get between the OS and the application. The application is free to utilize any functionality provided by the selected RTOS and the system can be configured to use almost any desired thread or task model simply by breaking up the super loop into separate loops, each in its own thread with its own priorities. Projects can support one or more configurations, allowing a single application project to be easily supported on different hardware platforms or to support different related feature sets.

Note: MPLAB Harmony has no process-space awareness. It assumes kernel-level access to all resources (although it does perform virtual-to-physical and physical-to-virtual address translation when necessary). Thus, libraries may require proxy support if used in operating environments that provide task-or-process address space protection.

This framework architecture reduces the amount of microcontroller support code that must be must be implemented for each new embedded design, allowing MPLAB Harmony developers to focus on application development instead of infrastructure work.

3: Modularity Guidelines

This section provides a modularity overview and related topics.

Description

The fundamental principal of MPLAB Harmony is one of modularity, at a source code level. This means that changes to the implementation of a library must not cause subsequent changes in the source code of an application or client module that uses the interface of that library. However, it is acceptable for changes in the implementation of a library to require recompilation of clients that use it. Also if necessary, modules may be written that integrate the functionality of other modules as long as it is clearly documented so that the user will not accidentally use conflicting modules in the same system at the same time.

3.1: Isolate Interface From Implementation

This topic introduces the functional interface, which is required for MPLAB Harmony modules.

Description

All MPLAB Harmony modules must have a clearly defined and well-documented "functional" interface. The documented interface should define function prototypes and any necessary data types and macros, but it must not utilize direct access to global data, internal resources or implementation details. Use of C-language functions is preferred (to support binary compatibility), but inline functions and macros are acceptable because they provide source-code compatibility. For example, if a module has internal global variables (as shown in the following example), it is legal to expose the values of these variables through a function, inline function, or macro. However, it is not legal to use direct access to the variables as part of the interface.

Example (mylib.h): Library Interface Header

```
/************************** Legal Interfaces ********************/

/* Function Access */
int MYLIB_Counter1Get ( void );

/* Warning!  Do not use counter2 or counter3 variables directly. */
extern int counter2;
extern int counter3;

/* Inline Function Access */
inline int MYLIB_Counter2Get ( void )
{
     return counter2;
}

/* Macro Access */
#define MYLIB_Counter3Get()  counter3;

/************************** Illegal Interface ********************/

/* Direct Global Variable Access */
extern int counter4;
```

Example (mylib.c): Library Implementation

```
int counter1;
int counter2;
int counter3;
int counter4;


int MYLIB_Counter1Get ( void )
{
    return counter1;
}
```

Example (myapp.c): Client Implementation

```
#include "mylib.h"

int c1, c2, c3, c4;
```

```
/***** Legal Interface Use *****/

c1 = MYLIB_Counter1Get();
c2 = MYLIB_Counter2Get();
c3 = MYLIB_Counter3Get();

/***** Illegal Interface Use ****/

c4 = counter4;
```

Any access that requires the client source code to use function call syntax allows for an implementation that can protect owned resources. Direct global access to owned resources does not allow such protection.

Note: The previous examples do not show any sort of protection mechanism. They only show the form of the interface.

3.2: Respect Abstractions

This topic provides information on preventing abstraction violations.

Description

Any code that is expected to be MPLAB Harmony compatible must not violate the abstraction provided by another MPLAB Harmony module in use in the system.

No module may access the internal modifiable resources (data or hardware) of another module except through a legal supported interface that allows the module to control all access to that resource in a safe and correct manner within the supported usage configuration and environment.

For example, the direct client access to "counter4" in the `myapp.c` example (see 3.1: Isolate Interface From Implementation) is a violation of the "mylib" interface abstraction because it directly accesses an internal resource owned by "mylib" without going through a legal interface. Calling a PLIB function for a different module's peripheral is another example of violating an abstraction because it constitutes "direct" access to owned resources. PLIBs cannot utilize any sort of protection mechanism (which may require OS support). Therefore, if a timer module owns a specific timer instance, it can call that timer's PLIB directly (see the following example). But, if it also calls the interrupt PLIB, then it is violating the Interrupt System Service's abstraction.

Example: PLIB Application

```
/******* Legal.  Module "owns" Timer2 *******/

/* Set up Timer2 - PBCLK as the source, prescaler is 256 (PBCLK / 256),
   enable 32-bit counter mode, clear the counter, set the period to 312,500
   - 80 MHz PBCLK / 256 = 312,500 Hz Timer2 clock, so period is set to
   - 312,500 to trigger an interrupt every 1 second */

PLIB_TMR_ClockSourceSelect(TMR_ID_2, TMR_CLOCK_SOURCE_PERIPHERAL_CLOCK);
PLIB_TMR_PrescaleSelect(TMR_ID_2, TMR_PRESCALE_VALUE_256);
PLIB_TMR_Mode32BitEnable(TMR_ID_2);
PLIB_TMR_Counter32BitClear(TMR_ID_2);
PLIB_TMR_Period32BitSet(TMR_ID_2, 312500);

/***** Illegal! Module does not "own" the Interrupt Controller  *****/

/* Enable the Timer 2 interrupt source (Timer2 is used for interrupts in
   32-bit mode), set its priority level to 2, set its sub-priority level to 0 */

PLIB_INT_SourceEnable(INT_ID_0, INT_SOURCE_TIMER_2);
PLIB_INT_VectorPrioritySet(INT_ID_0, INT_VECTOR_T2, INT_PRIORITY_LEVEL2);
PLIB_INT_VectorSubPrioritySet(INT_ID_0, INT_VECTOR_T2, INT_SUBPRIORITY_LEVEL0);
```

In the previous example, this application must use the Interrupt System Service Library (which can control access to the Interrupt Controller), instead of direct access via the PLIB (unless it is the only module in the system that uses the Interrupt Controller).

3.3: Protect Owned Resources

This provides details on protecting owned resources.

Description

With the singular exception of PLIBs (which implement a hardware access layer and are not technically MPLAB Harmony "modules"), every module owns all internal resources that it modifies directly or by using a PLIB.

Owned resources include global data structures, static or dynamically allocated RAM (including client buffers whose ownership is temporarily passed to the module) and special function registers (accessed via PLIBs). This covers any addressable memory space not allocated on the

program stack. The intended usage model of all MPLAB Harmony compatible libraries must provide for the protection of all modifiable, non-stack resources owned by library from potential corruption.

Owned resources are protected from direct intentional access by the rules governing the respect of each module's abstraction. However, there are protected a few other specific possibilities addressed in the following sections.

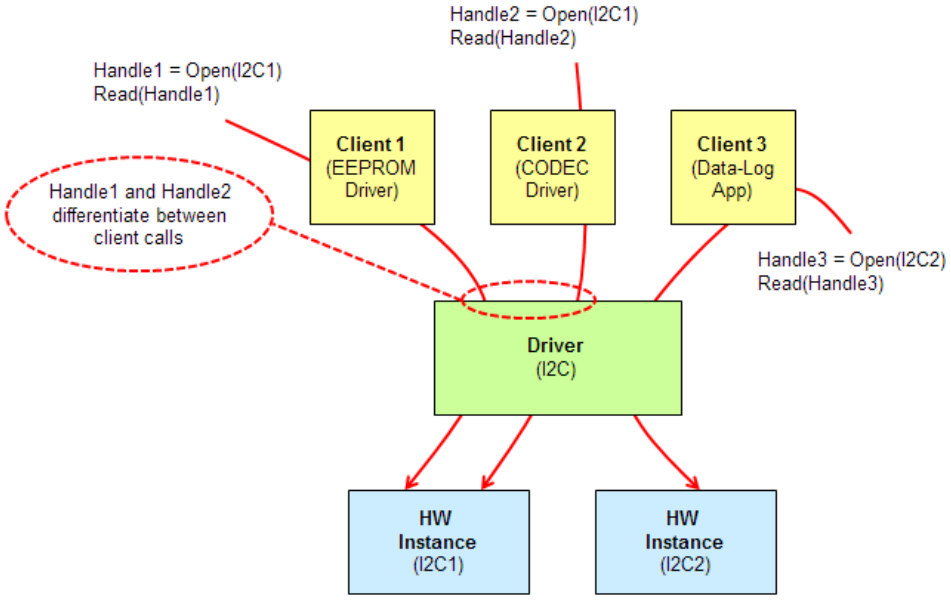
3.3.1: Protect Owned Resources From Accesses by Multiple Clients

This topic provides details on protecting owned resources from accesses by multiple clients.

Description

If more than one module is allowed to interact with a library's interface at a time, that library must implement a mechanism to identify the calling client and prevent requests from one client from interfering with requests from another client. For example, a dynamic multi-client MPLAB Harmony driver requires the client to call an "Open" function and obtain a "handle" value that identifies the client (and the hardware instance if the driver dynamically manages multiple instances of the hardware, as shown in the following example).

Example: Driver Handle Mechanism



The handle obtained from the "Open" function must be passed into all other functions provided by the driver's client interface. If a client interface function, such as the "Read" example shown above, is called with an invalid handle the driver must fail the request. Using this mechanism, a driver can prevent requests from one client from interfering with requests from another client by failing the conflicting request (which must then be retried by the client), by temporarily suspending/blocking the request using operating system capabilities, or by queuing the request and servicing it when the hardware resource becomes available.

3.3.2: Protect Owned Resources From Accesses by Multiple Threads

This topic provides details on protecting owned resources from accesses by multiple threads.

Description

Any library that supports operation in one or more multi-threading RTOS environments must use operating system methods to protect non-atomic accesses to modifiable resources that it owns.

The preferred method for doing this is to use the appropriate MPLAB Harmony OSAL functions, unless the library is restricted to use within a specific operating system (in which case OS-specific methods may be used). For example, if multiple clients can use a module (as shown in the diagram in 3.3.1: Protect Owned Resources From Accesses by Multiple Clients), it may be called from multiple operating system threads. If that is the case, the module must protect its shared resources using operating system methods. For example, the MPLAB Harmony OSAL provides a mutex mechanism that can be used to protect access to a shared data structure, as shown in the following example.

Example: OSAL Mutex Protecting Shared Data Structure

```
/* Perform operations on a shared data structure */
struct myDataStructure
{
    uint16_t  x;
    uint8_t   y;

} myDataStructure;
```

```
...
OSAL_MUTEX_DECLARE(mutexDS);
OSAL_MUTEX_Create(&mutexDS);

...
/* Wait 2 seconds to obtain the mutex */
if (OSAL_MUTEX_Lock(mutexDS, 2000) == OSAL_RESULT_TRUE)
{
    /* Operate on the data structure */
    myDataStructure.x = 32;
    myDataStructure.y = 42;
    OSAL_MUTEX_Unlock(mutexDS);
}
```

3.3.3: Protect Owned Resources From Accesses by ISR

This topic provides details on protecting own resources from accesses by an Interrupt Service Routine.

Description

Any library that supports calling of one or more of its functions from within an Interrupt Service Routine (ISR) context must implement a mechanism by which it prevents that interrupt from corrupting modifiable resources owned by that library that may be accessed non-atomically by non-ISR routines. For example, the following code sequence example shows how this can be done using the MPLAB Harmony Interrupt System Service.

Example: Protecting Critical of Code from an Interrupt

```
bool interruptWasEnabled;

/* Saves interrupt source enable status before disabling it. */
interruptWasEnabled = SYS_INT_SourceDisable(MY_DRIVER_INTERRUPT_SOURCE);

/* Do something critical. */

/* Re-enable interrupt source if it was enabled before. */
if (interruptWasEnabled)
{
    SYS_INT_SourceEnable(MY_DRIVER_INTERRUPT_SOURCE);
}
```

Note: Normally, only a device driver or system service library's state machine "Tasks" function (as described in Support One or More "Tasks" Functions if Required) can be called from an ISR. In these cases, the library's other routines may use the MPLAB Harmony Interrupt System Service to temporarily disable the interrupt in question (and only that interrupt). However, care must be taken if other interface or callback routines can be called from within another module's interrupt context to not make non-atomic accesses to resources as one module may not disable another module's interrupt.

3.4: Access Shared System Resources Only Through MPLAB Harmony Drivers and System Services

This provides information on shared system resources.

Description

Shared system resources such as the interrupt controller, I/O ports, timers, and other peripherals are common resources that must be owned by a MPLAB Harmony system service or a device driver module to be shared by other modules (including middleware libraries and applications).

Access to these shared system resources must only occur via the controlling module's interface, never by direct access (which is a violation of the abstraction, as described in 3.2: Respect Abstractions) unless that resource is dedicated to the module accessing it.

System services provide interfaces to allocate, access, and control resources for which an active link need not normally be opened between the library and the client. This is because once a resource is allocated to a specific client; no other client may attempt to use it.

Device drivers provide a mechanism for clients to maintain open link to a shared resource and share its ongoing use with other clients or use it until done and close the link to allow the resource to be used by other clients.

3.5: Support MPLAB Harmony System Module Model When Applicable

This describes the MPLAB Harmony system module model.

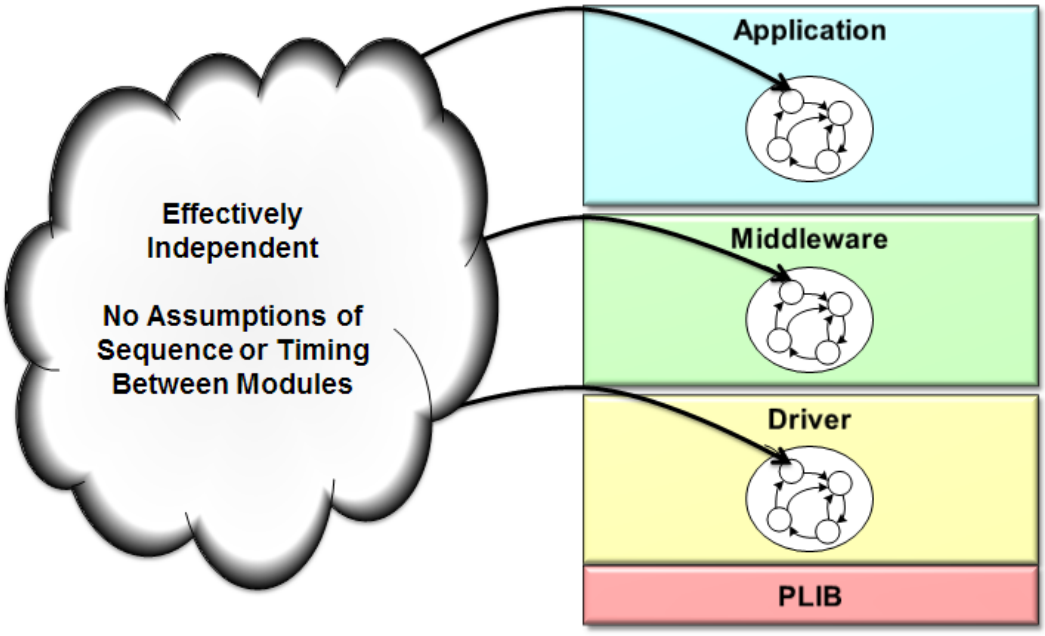
Description

As mentioned in the introduction, a MPLAB Harmony library can be an active state machine driven module or a fully re-entrant library with no internal state of its own. If a library does not need to be initialized (i.e., it is fully re-entrant and owns no internal modifiable resources), it does not

need to implement an initialization routine and it is not considered a MPLAB Harmony "module".

Any library or application that requires initialization, that owns any internal resources (that are not automatically allocated on the stack), or that maintains any sort of state machine is considered a "module" and must implement some portion of the MPLAB Harmony "system module" interface. This interface defines the function signatures of specific routines used by the system configuration code (and/or system libraries) to initialize and maintain correct operation of the module within the overall system.

Horizontal "System" Interface



Clients of the module do not use this interface; only the system uses it to keep the module "running".

3.5.1: Support an "Initialize" Function if Required

This topic describes the Initialize function.

Description

Support of an "Initialize" function is the minimal criterion for a MPLAB Harmony library to be considered a module.

The primary purpose of the "Initialize" function is to place an instance of a module into its initial state and return an object handle used by the other system interface functions to access the instance of the module that was initialized. The "Initialize" routine must store any required data passed to it, place the module's state machine into its initial state and return immediately (without blocking). Any initialization that may take significant time to complete must be carried out in the module's state machine.

Function

```
SYS_MODULE_OBJ <module>_Initialize( const SYS_MODULE_INDEX index,
                                    const SYS_MODULE_INIT * const data )
```

Summary

Initializes the module with data for the index instance of the `<module>` module.

Description

This routine initializes the module for the index instance of the <module> module, using the initialization given data, placing the module in its initial state. It will be called only once, at system initialization and will not be called again unless the system first calls the module's "Deinitialize" routine, if supported.

Required or Optional?

Optional – if the library does not require any initialization.

Preconditions

The low-level processor and board initialization must be completed before the system will call the initialization routine for any modules.

Parameters

index	Zero-based index, identifying the instance of the module to be initialized.
data	Pointer to a module implementation specific data structure. Although the data type of this pointer is to a standard structure, structure actually used can (and usually will) be extended to include additional implementation-specific data as long as the first member of the structure is a SYS_MODULE_INIT structure. If this pointer is NULL, then build-time configuration options must be given to override the items contained in the structure.

Returns

A handle to the instance of the system module that was initialized or SYS_MODULE_OBJ_INVALID if an error occurred. This handle is a necessary parameter to all of the other system-module interface routines for that module.

Code Example

```
// Initialize the I2C driver
sysObjects.i2cDriver = DRV_I2C_Initialize(I2C_ID_1, &i2c_init_data);
```

Blocking Behavior

This function must not block waiting for external I/O, other modules, or anything that may take significant time.

Remarks

The "Status" operation will return SYS_STATUS_READY when the module is fully initialized and ready for client access.

Every module should define its own initialization data structure type named <*module*>_INIT, where <*module*> is the abbreviation identifying the module to be initialized. This structure must be an extension of the SYS_MODULE_INIT structure (i.e., its first member must be the SYS_MODULE_INIT structure or equivalent). Any parameter that can change the power state of the module must be included in the data structure.

Once the Initialize operation has been called, the Deinitialize operation must be called before the Initialize operation can be called again.

Refer to the `<install-dir>/framework/system/common/sys_module.h` header file or System Service Library Introduction for additional information and definitions of the additional data types used.

3.5.2: Support One or More "Tasks" Functions if Required

This topic describes the Tasks function(s).

Description

If a module is an active component of the system, it will implement one or more "Tasks" functions.

Task functions manage the state of a module and perform the "background" work of a module (any work that does not occur in a client interface routine). Task routines can be called from a system-wide super loop, from OS threads, or possibly from ISRs, depending on the specific requirements and configuration of the module. Tasks routines must not make any implicit assumptions about state. They must always check the module's current state data (be that state stored in variables or in registers) and take any appropriate actions necessary to move to the next appropriate state. If no action is necessary, the tasks routine may stay in the same state and should return as quickly as possible.

Most device drivers (and some system services) use ISR safe tasks routines to manage their state. If a tasks routine is designed to be called from an ISR, it must not call any potentially blocking routines. A module may implement both ISR-safe tasks functions and non-ISR-safe tasks functions if necessary.

Function

```
void <module>_Tasks ( SYS_MODULE_OBJ object )
```

Summary

Used to maintain the module's state machine and/or implement its ISR logic.

Description

This routine is used to maintain the module's internal state machine and/or implement its ISR logic for interrupt-driven tasks.

Required or Optional?

Optional – But required for any module that requires a state machine.

Preconditions

The driver module's "Initialize" operation must have completed.

Parameters

object	Module's instance data handle.

Returns

None.

Code Example

```
DRV_I2C_Tasks(sysObjects.i2cDriver);
```

Blocking Behavior

If intended to be called from an ISR this routine must NEVER block. If polled this routine may block on OS objects, but not on external I/O.

Remarks

This routine is not called by a client application. It is only called by the super loop, from an RTOS thread, or from an ISR by the system's configuration code.

Refer to the `<install-dir>/framework/system/common/sys_module.h` header file or the System Service Library Introduction for additional information and definitions of the additional data types used.

3.5.3: Support a "Deinitialize" Function if Required

This topic describes the Deinitialize function.

Description

Some modules are always required for a working system and do not support a "Deinitialize" function. However, if a module is intended to support calling of its "Initialize" function a second or subsequent time (without first going through an entire system reset), it must implement a "Deinitialize" function that stops all current activity, frees any allocated resources, and disables all operations.

Task functions manage the state of a module and perform the "background" work of a module (any work that does not occur in a client interface routine). Task routines can be called from a system-wide super loop, from OS threads, or possibly from ISRs, depending on the specific requirements and configuration of the module. Tasks routines must not make any implicit assumptions about state. They must always check the module's current state data (be that state stored in variables or in registers) and take any appropriate actions necessary to move to the next appropriate state. If no action is necessary, the tasks routine may stay in the same state and should return as quickly as possible.

Most device drivers (and some system services) use ISR safe tasks routines to manage their state. If a tasks routine is designed to be called from an ISR, it must not call any potentially blocking routines. A module may implement both ISR-safe tasks functions and non-ISR-safe tasks functions if necessary.

Function

```
void <module>_Deinitialize( SYS_MODULE_OBJ object )
```

Summary

Deinitializes the specified instance of the module

Description

This routine deinitializes the specified instance of the module, stopping all current activity, freeing any allocated resources, and disabling all operations.

Required or Optional?

Optional – unless the module can be disabled and initialized a second or subsequent time (without performing an entire system reset), which means it is required.

Preconditions

The module's "Initialize" function must have been called once and returned a valid object handle, before this routine may be called.

Parameters

object	Module's instance data handle.

Returns

None.

Code Example

```
DRV_I2C_Deinitialize(sysObjects.i2cDriver);
```

Blocking Behavior

This function must not block or wait for ongoing operations to complete. If time is required to complete the deinitialization operation, then the process must be completed by the module's state machine and the module's "Status" must indicate a status greater than SYS_STATUS_UNINITIALIZED until the operation is complete.

Remarks

The Status operation will return SYS_STATUS_UNINITIALIZED when this operation has completed.

This routine may abort any queued operations, and possibly even operations in progress (if it they can be aborted without adverse effect on the interface) or it may allow them to complete, but it must prevent any new operations from being initiated by any clients.

Once the Initialize operation has been called, the Deinitialize operation must be called before the Initialize operation can be called again and vice versa.

Refer to the `<install-dir>/framework/system/common/sys_module.h` header file or the System Service Library Introduction for additional information and definitions of the additional data types used.

3.5.4: Support a "Status" Function if Required

This topic describes the Status function.

Description

If a module implements the "Deinitialize" or "Reinitialize" system interface functions, it must also implement the "Status" function. A module may also implement the "Status" function if it provides error status reporting to the system.

Function

```
SYS_STATUS <module>_Status( SYS_MODULE_OBJ object )
```

Summary

Provides the current status of the identified instance of the module.

Description

This routine provides the current status of the instance of the module identified by the object handle.

Required or Optional?

Optional – But, required if the module implements either of the "Deinitialize" or "Reinitialize" routines.

Preconditions

None. (The "Status" routine must return SYS_STATUS_UNINITIALIZED if called before the module has been initialized.)

Parameters

object	Module's instance data handle.

Returns

- SYS_STATUS_ERROR_EXTENDED = -10 - Any value less or equal to this indicates that a non-system defined (module specific) error has occurred.
- SYS_STATUS_ERROR = -1 - Indicates that the specified module is in an error state. Any value less or equal to this value indicates an error has occurred.
- SYS_STATUS_UNINITIALIZED = 0 - Indicates that the module has not yet been initialized (its "Initialize" function has not been called).
- SYS_STATUS_BUSY = 1 - Indicates that a previous system operation for the specified module instance has not yet completed and the module is not yet ready to receive calls to other system interface operations.
- SYS_STATUS_READY = 2 - Indicates that any previous module operation for the specified module has completed. Any value greater than or equal to value this indicates that the module is ready to receive calls to other operations.
- SYS_STATUS_READY_EXTENDED = 10 - Any value greater or equal to this indicates that the specified module is in a non-system defined (module specific) ready or running state.

Code Example

```
// Initialize the I2C driver module
sysObjects.i2cDriver = DRV_I2C_Initialize(I2C_ID_1, &i2c_init_data);
```

```
// Check driver's status
status = DRV_I2C_Status(sysObjects.i2cDriver);
if (status <= SYS_STATUS_ERROR)
{
    // Handle error
}
else if (status < SYS_STATUS_READY)
{
    // Module not ready yet, check again later
}
```

Blocking Behavior

This function must not block waiting for external I/O, other modules, or anything that may take significant time.

Remarks

This operation can be used to determine when other system operations have completed, if a module has been initialized, or if an error has occurred. If the Status operation returns SYS_STATUS_BUSY, the previous operation has not yet completed. Once the Status operation returns SYS_STATUS_READY (or greater), any previous operations have completed.

The value of SYS_STATUS_READY is 2. Values between 2 and 10 are reserved for system defined "ready" states. A module may define module-specific "ready" or "run" states greater than or equal to 10 (SYS_STATUS_READY_EXTENDED).

The value of SYS_STATUS_ERROR is -1. Values between -1 and -10 are reserved for system-defined errors. A module may define module-specific error values of less than or equal to -10 (SYS_STATUS_ERROR_EXTENDED).

If the Status operation returns an error value, the error may be cleared by calling the reinitialize operation. If that fails, the Deinitialize operation will need to be called, followed by the initialize operation to return to normal operations. (The system must check the value returned by the Status routine after calling any of the module operations to find out when they have completed.)

Refer to the `<install-dir>/framework/system/common/sys_module.h` header file or the System Service Library Introduction for additional information and definitions of the additional data types used.

3.5.5: Support a "Reinitialize" Function if Required

This topic describes the Reinitialize function.

Description

A module may support a "Reinitialize" function if it provides the ability to change initialization parameters dynamically, while the system is running.

The primary purpose of this operation is to support dynamic power management by allowing initial power states and parameters to be changed without damaging ongoing client activity. If supported, this operation must either temporarily suspend ongoing client activity (but not invalidate opened client handles or ongoing requests) before making the requested change in initial parameters or it must make the change in a safe (non-destructive) way while client operations are ongoing (if that is possible).

This operation will primarily be used by a system power management module (if utilized) or by safety monitoring system module to guarantee active modules are in the correct power state.

Function

```
void <module>_Reinitialize (SYS_MODULE_OBJ object,
const SYS_MODULE_INIT * const data )
```

Summary

Reinitializes and refreshes the state data for the identified instance of the module.

Description

This routine reinitializes and refreshes the state data (particularly the hardware settings) for the identified instance of the module using the initialization given data. It does this in a client safe manner, not by arbitrarily clearing or reinitializing internal data structures and it does not disconnect or interrupt any ongoing client operations (although, it may temporarily suspend them).

Required or Optional?

Optional – not required if the module is not power-state aware or safety critical.

Preconditions

The associated initialization routine must have been called and the module's state machine(s) must be running.

Parameters

object	Module's instance data handle.
data	Pointer to a module implementation specific data structure. This is identical to the structure passed to the module's "Initialize" function.

Returns

None.

Code Example

```
DRV_I2C_Reinitialize(sysObjects.i2cDriver, &i2c_init_low_power_state);
```

Blocking Behavior

This function must not block or wait for ongoing operations to complete. If time is required to complete the re-initialization operation, then the process must be completed by the module's state machine and the module's "Status" must indicate a status less than SYS_STATUS_READY until the operation is complete.

Remarks

The Status operation will return SYS_STATUS_READY (or greater) when this operation has completed.

This operation uses the same initialization data structure as the Initialize operation.

This operation can be used to change the power state of the peripheral the module manages.

This operation will not interrupt any ongoing client operations. (Effects on queued operations are module dependent, but the module must provide a way to identify and handle any errors in queued operations that it allows to occur.)

This operation must refresh the hardware and state data (as affected by the initialization data) to ensure that the module operates using the new settings.

Refer to the `<install-dir>/framework/system/common/sys_module.h` header file or the System Service Library Introduction for additional information and definitions of the additional data types used.

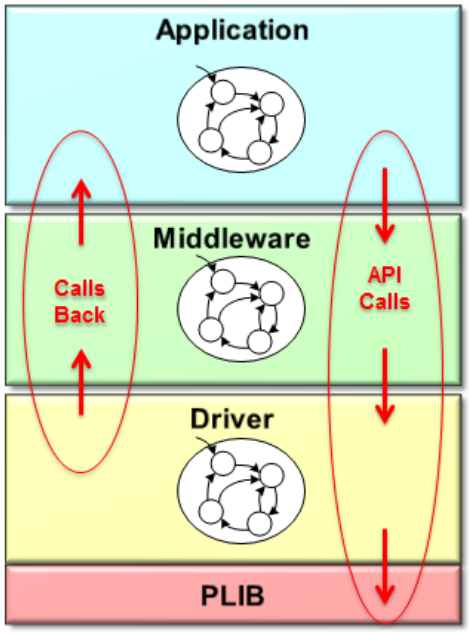
3.6: Drivers Use Driver-Client Model

This describes the Driver-Client model used by drivers.

Description

All libraries implement a vertical "client" interface through which clients of the library interact with it. This allows libraries, especially active modules in the system, to be "stacked" so that their state machines work together.

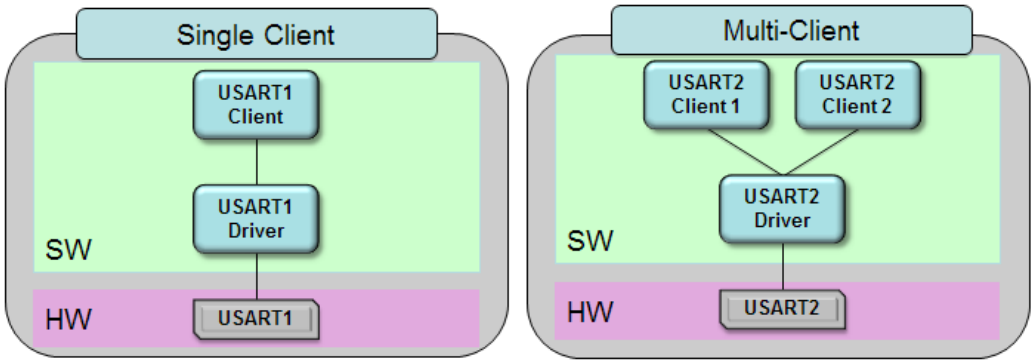
Vertical "Client" Interface



Any MPLAB Harmony module that can be considered a device "driver" must implement the MPLAB Harmony driver interface. The driver interface defines function signatures of open and close routines to allow a client module to open an individual link to a driver before using it. It must hold onto the link as long as required and then close the link after it has finished using it. This allows drivers to identify requests from different clients so

that it may be implemented in a way that allows clients to safely share the resources managed by the driver without interfering with each other.

Single Client Versus Multi-client Drivers



Drivers can also be designed to only support a single client at a time, but the driver must still emulate the multi-client interface even if it does so by providing macros or inline functions that effectively "throw away" the call to the "Open" and "Close" functions to provide source-code level compatibility with the multi-client dynamic interface.

3.6.1: Support Driver "Open" Function

This topic describes the Open driver function.

Description

The purpose of a driver's "Open" function is to initialize a link between the driver and the client and provide the client a "handle" that identifies that link.

The "Open" function is modeled after a POSIX file system "open" and "fopen" functions and it allows the client to indicate the intended usage of the driver for input and/or output, as well as the desired level of sharing with other modules.

Function

```
DRV_HANDLE DRV_<module>_Open (const SYS_MODULE_INDEX index,
                              const DRV_IO_INTENT intent )
```

Summary

Opens a driver for client use and provides an "open-instance" handle.

Description

This routine opens a driver for use by a client module and provides an "open-instance" handle that must be provided to any of the driver's other client operations to identify the caller and the instance of the driver module.

Required or Optional?

Required by any driver module.

Preconditions

The driver module's "Initialize" operation must have completed (meaning that the DRV_<module>_Status routine should have returned SYS_STATUS_READY (or greater)).

Parameters

index Index, identifying the instance of the driver module to be opened (example DRV_I2C_1, or DRV_I2C_2).

intent Flags parameter identifying the intended use of the driver:

One of:

- DRV_IO_INTENT_READ – Driver opened in read-only mode
- DRV_IO_INTENT_WRITE – Driver opened in write-only mode
- DRV_IO_INTENT_READWRITE – Driver opened in read-write mode, equivalent to DRV_IO_INTENT_READ|DRV_IO_INTENT_WRITE

One of:

- DRV_IO_INTENT_NONBLOCKING – Routines return immediately
- DRV_IO_INTENT_BLOCKING – Routines return after operation is complete

One of:

- DRV_IO_INTENT_EXCLUSIVE – Driver does no SW buffering
- DRV_IO_INTENT_SHARED – Driver buffers reads internally

One flag from each group may be ORed together to fully define the intended use.

Returns

If successful, the routine returns a valid open-instance handle (a number identifying both the caller and the module instance). If an error occurs, the routine value is DRV_HANDLE_INVALID.

Code Example

```
handle = DRV_I2C_Open(DRV_I2C_1, DRV_IO_INTENT_READWRITE|
                                 DRV_IO_INTENT_NONBLOCKING|
                                 DRV_IO_INTENT_SHARED);
if (DRV_HANDLE_INVALID == handle)
{
    // Handle error
}
```

Blocking Behavior

This function may block when operating in an OS environment. It must never block waiting on I/O in a non-OS environment (i.e., blocking is only supported in an OS environment).

Remarks

To support blocking behavior, the driver must be appropriately configured and built.

Drivers that are opened with a mode that is not supported must fail the open call by returning DRV_HANDLE_INVALID.

The default mode (if no flags are set (i.e., zero (0) is passed in the intent parameter), is non-blocking, no read access, no write access, and shared access so the mode parameter is usually required to do anything useful with a driver.

Refer to the `<install-dir>/framework/driver/driver_common.h` header file or the Driver Library Introduction for additional information and definitions of the additional data types used.

3.6.2: Support Driver Handle

This topic provides information on driver handle usage.

Description

To use a driver, an application (or other client) must first call the driver's open routine to get an "open-instance" handle to the driver for the desired peripheral instance. This handle must be passed into all of the driver's other client interface routines to identify both the caller and the specific instance of the driver (or peripheral hardware or other resource owned by it).

Once the driver has been initialized and opened, the application (or other client layer) can use the other operations provided by the driver to read or write data and generally control the device in question. Normally, an application will open a driver, keep it open while it needs the device, and then close the device only when it's done with the device. In simple systems, a client might never close the driver.

3.6.3: Support Driver "Close" Function

This topic describes the Close driver function.

Description

The purpose of the driver "Close" function is to allow a client module to release an opened link to a driver module, freeing any allocated resources and making the link available to other modules.

Simple drivers might not support this operation if they are normally never closed without a complete system reset.

Function

```
void DRV_<module>_Close ( const DRV_HANDLE handle )
```

Summary

Closes an opened-instance of a driver.

Description

This routine closes an opened-instance of a driver, invalidating the given handle and freeing any allocated resources.

Required or Optional?

Optional – Not required if the driver is designed to never be closed.

Preconditions

The driver module's "Initialize" operation must have completed (meaning that the DRV_<module>_Status routine should have returned SYS_STATUS_READY (or greater)) and the driver's "Open" routine must have returned a valid device handle.

Parameters

handle	A valid open-instance handle, returned from the driver's open routine

Returns

None.

Code Example

```
// Close the driver
DRV_I2C_Close(handle);
```

Blocking Behavior

This routine can block until all pending driver operations have completed if the driver was appropriately configured and built and if the system is OS-based. Otherwise, it must not block.

Remarks

Once this routine has been called, the given handle will become invalid.

In non-blocking configurations, a closed driver may go into a "zombie" mode until the driver's state machine has finished closing out any ongoing client operations. In "zombie" mode, a client may continue to use the handle only to call a driver's ClientStatus routine (if supported). When the client link has been successfully closed, the ClientStatus routine will return the DRV_CLIENT_STATUS_CLOSED value from the DRV_CLIENT_STATUS enumeration.

Refer to the `<install-dir>/framework/driver/driver_common.h` header file or the Driver Library Introduction for additional information and definitions of the additional data types used.

3.7: Use Common Data Transfer Models When Applicable

This describes common data transfer models.

Description

The primary purpose of many different types of modules is to transfer data into or out of client accessible buffers. MPLAB Harmony drivers (and other modules) provide a consistent set of programming models (described in the following section) for accomplishing this transfer of data. If possible (and appropriate), it is best to use a data transfer model that is similar to one used by other modules. This consistency makes MPLAB Harmony libraries easier to use and improves overall quality by using tried and true methods.

When these methods are used, they should be identified as such in the documentation.

3.7.1: Support Byte-by-Byte for FIFO-based Data Transfer Model

This topic describes byte-by-byte data transfers.

Description

A byte-by-byte data transfers data one byte or word at a time in small batches.

In pseudo-code, this model appears as follows (from the client point of view).

Reading Data:

```
while ( DataIsAvailable() )
{
    DataWord[i++] = DataWordGet();
}
```

Writing Data:

```
while ( !TransmitterIsFull() )
{
    DataWordSend(DataWord[i++]);
}
```

The exact function names, parameters, and data types for the operations shown, which are shown in bold type in the previous examples, may vary, but the basic usage is consistent. Modules that use this transfer model may also provide operations to tell when all data transmission is complete (effectively a transmitter under run) and when the receive FIFO buffer is full and new data is received (effectively a receiver overrun).

This data transfer has the advantage of being useful for lightweight systems or whenever data is transferred a single data byte (or data word) at a time through a relatively small FIFO buffer. However, it has the disadvantage of encouraging use of potentially blocking loops, forcing the client to provide the loop to transfer multiple data words, and only working when a single client has exclusive access to a module. It is best used only when the data words are internally buffered (in hardware or software) in a small FIFO or queue. The operations should not block waiting on the actual transfer of data, instead they should only access the FIFO buffer and return FIFO full (or empty) so that the client stops looping when there is no more space (or data) in the FIFO. However, depending upon the exact parameters and return values used, the data transfer operations may still result in incorrect data being sent or received if an error occurs because there is no way to detect from within the loop it unless the client checks the status between every byte or word transferred.

3.7.2: Support File System Style Read-Write Data Transfer Model

This topic describes the File System style read/write data transfer model.

Description

The File System style read/write data transfer model is intended to be similar to the POSIX read/write and fread/fwrite operations. This is the most basic data transfer model that any data source or sink driver should support, if feasible. In pseudo-code, this model appears as follows (from the client point of view).

Reading Data:

```
Count = Read(&DataBuffer[SIZE], SIZE);
```

Writing Data:

```
Count = Write(&DataBuffer[SIZE], SIZE) )
```

The exact function names, parameters, and data types for the operations shown (in bold in the previous examples) may vary, but the basic usage is consistent. This model has the advantage that it may allow clients to use the file system abstraction layer to read and/or write data to a peripheral without the need to know the actual function names of the module (as long as that module is mounted to the file system). However, it has the disadvantage that, in non-blocking configurations, it may only transfer the amount of data that can be buffered by the driver or hardware, thus the client may need to check the return count and call the function again (potentially several times) to complete the transfer. Also, like the byte-by-byte model, this method requires the client to respond within the time it takes to fill or empty the FIFO to keep a continuous stream of data transfers. This model is most useful in a blocking environment where it can block and only return when the entire transfer is complete.

3.7.3: Support Buffer Queuing Data Transfer Model

This topic describes the buffer queuing data transfer model.

Description

The buffer queuing data transfer model is similar to a network packet method.

In pseudo-code, this model appears as follows (from the client point of view).

Reading Data:

```
TransferHandle = BufferAddRead(&DataBuffer[SIZE], SIZE);
```

Writing Data:

```
TransferHandle = BufferAddWrite(&DataBuffer[SIZE], SIZE) )
```

The exact function names, parameters, and data types for the operations shown (in bold in the previous examples, above) may vary, but the basic usage is consistent. Some modules may combine the operations into one and identify the direction of the transfer in a parameter or packet header. Most modules will provide a callback mechanism and possibly a status function to allow the client to identify when the buffer has been transferred. The operations can be called multiple times to queue up multiple buffers. Each call will return a handle identifying the transfer buffer (or an invalid handle indicating the queue was full).

This model has the advantage that it is easier to use and it allows the client the ability to easily maintain a continuous stream of data, if desired.

(The client only needs to respond before all queued buffers have been transferred to avoid over run or under run situations.) However, it has the disadvantage that it usually requires a larger more complex implementation.

3.8: Use Existing Abstraction Models When Applicable

This describes abstraction model usage.

Description

Key benefits of a layered, modular software framework are the ability to easily interchange one implementation of a module for another and the ability to treat similar modules in a common way. This allows use of implementations that are optimized for specific purposes (or that have added features) without the need to change existing client code. In addition, it allows higher abstraction layers to use different modules without requiring modifications. Whenever a new module is intended to support an existing library stack or application, it must support the existing abstraction model.

3.8.1: File System Modules Use the SYS FS Plug-in Interface

This topic describes the SYS FS Plug-in interface.

Description

Modules that implement a file system used to organize data on a storage medium must provide the interface required by the MPLAB Harmony virtual file system layer (SYS FS) and they must use the MPLAB Harmony File System Media Manager interface to access supported storage media. Refer to the `<install-dir>/framework/system/fs/sys_fs.h` and `<install-dir>/framework/system/fs/sys_fs_media_manager.h` header files and the File System Service Library for a complete description of this interface.

3.8.2: Media Drivers Use the File System Media Manager Driver Model

This topic describes the File System Media Manager Driver model.

Description

Modules that implement a storage media driver to be used with the MPLAB Harmony virtual file system layer (SYS FS) must provide a media manager driver interface. Refer to the `<install-dir>/framework/system/fs/sys_fs_media_manager.h` header file and the File System Service Library for a complete description of this interface.

3.8.3: TCPIP MAC Drivers Use the TCPIP Virtual MAC Driver Model

This topic describes the TCP/IP virtual MAC Driver model.

Description

Modules that implement a network Media Access Controller (MAC) driver to be used with the MPLAB Harmony TCP/IP library stack must provide a MAC driver interface as required by the virtual MAC layer. Refer to the `<install-dir>/framework/tcpip/tcpip_mac*.h` header files and the MAC Driver Module for a complete description of this interface.

3.9: Emulate and Extend Existing Interface Models When Applicable

This provides information on emulating and extending existing interface models.

Description

Even if a new module is not intended to replace an existing module or to "plug in" to an existing library stack, it should attempt to emulate an existing interface and extend it if required. To emulate an existing interface the module should provide functions, whose names are different, but whose parameters, return values, and functionality are identical (if possible) or very similar to those of an existing module or set of modules. If additional functionality is required that existing modules do not similarly support, then it is acceptable to extend the existing interface model with additional features. It is unacceptable for modules with similar functionality to use completely dissimilar interface models.

4: Flexibility Guidelines

This section provides guidelines for flexibility when creating a MPLAB Harmony library.

Description

To address a broad range of individual customer's needs, MPLAB Harmony libraries provide a high degree of flexibility and configurability, supporting a wide range of user-selectable build options and execution environments. Libraries can be compatible with MPLAB Harmony without necessarily supporting all of the options and execution environments for which fully flexible libraries can be configured as long as any restrictions are fully understood and clearly documented.

4.1: Support One or More Execution Environments

This topic provides information on support for one or more execution environments.

Description

There are three fundamental execution models that may be supported by MPLAB Harmony libraries:

- Polled
- Interrupt-driven
- RTOS multi-threaded

It is generally preferable for libraries to be fully flexible, use the abstracted state machine model and OSAL provided by MPLAB Harmony to allow them to be appropriately configured for any of these three of execution models in any supported execution environment. However, it is acceptable for a library to be optimized and implemented specifically for one such execution environment and not support one or more of the others as long as the restrictions are clearly documented.

4.1.1: Execution in One or More RTOS Environment

This topic describes execution in an RTOS environment.

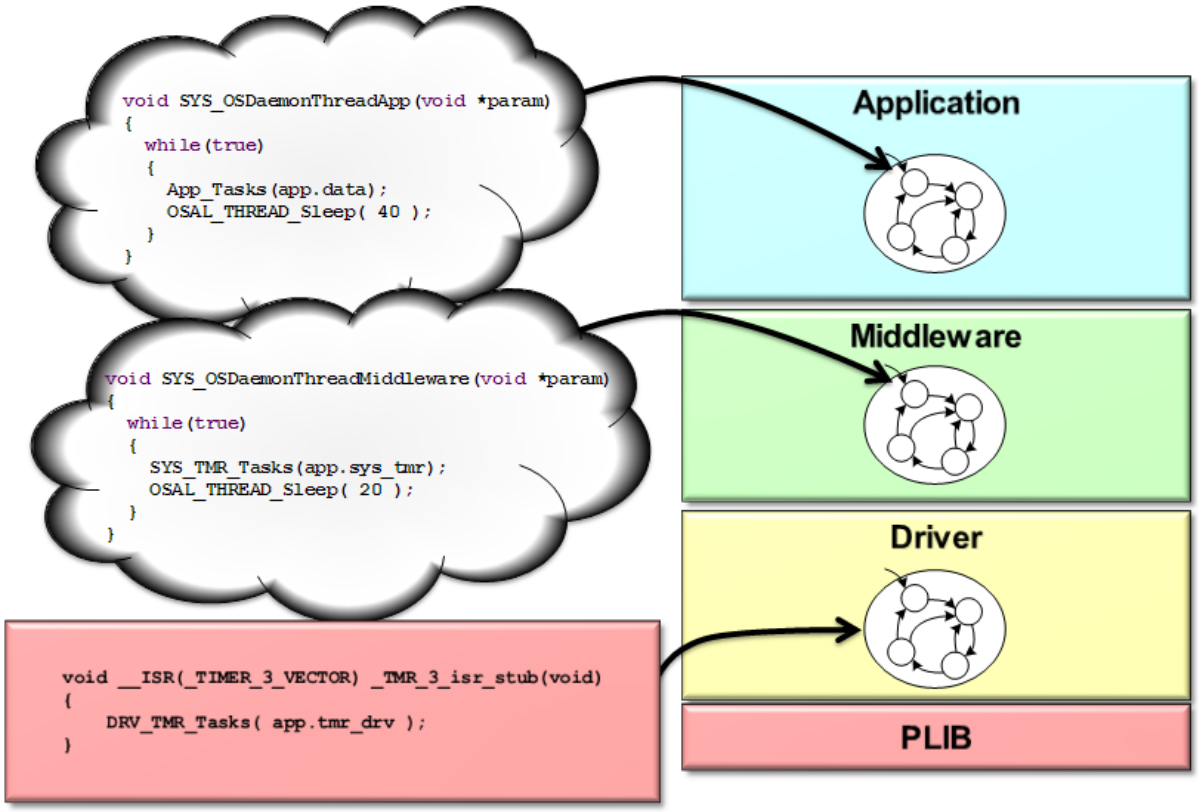
Description

The MPLAB Harmony OSAL defines a set of functions that libraries can call to provide thread safety and blocking behavior when executing in a supported OS environment. MPAB Harmony libraries call the appropriate OSAL functions to provide thread safety when accessing shared resources or providing blocking functionality. For non-OS environments, MPLAB Harmony provides "basic" and "none" implementations of the OSAL, allowing the OSAL calls (or "hooks") to remain in the library code.

The OSAL can be implemented for most OS or non-OS environments. So, using the OSAL provides maximum flexibility for MPLAB Harmony compatible libraries and is highly recommended. However, a library can be implemented for a specific OS (or for no OS) and still be MPLAB Harmony compatible in the environment for which it was designed as long as the library's documentation clearly and conspicuously identifies the OS (or non-OS) environment(s) it supports, if it does not make use of the MPLAB Harmony OSAL.

It is preferable that a module that supports execution in one or more RTOS environments should utilize the MPLAB Harmony abstracted state machine architecture and that the module's tasks functions be called from a loop in an OS thread (if polled) or from an ISR (if interrupt driven), as illustrated in the following diagram.

OS-Based Execution



But, it is acceptable to use any desired method of implementing the module if it is restricted to an RTOS environment and all restrictions are clearly and conspicuously documented.

4.1.2: Interrupt Driven Execution

This topic describes interrupt driven execution.

Description

Different microcontrollers handle interrupts in different ways. Some support multi-vectored interrupts so that each instance of each device can have its own interrupt vector function. Some microcontrollers only have a single interrupt vector function and the ISR that implements the vector function must interrogate the interrupt controller and/or peripherals to identify which peripheral caused the interrupt. Other microcontrollers do something in between; grouping some interrupts so that the ISR may need to identify which specific flag from a group caused the interrupt.

In all cases, the ISR itself is only responsible for identifying the specific source of the interrupt and for calling the appropriate driver tasks function. But, since the exact combination of interrupt-source flags and vector functions is always processor-specific, the actual interrupt vector functions must be implemented as part of the system configuration code. However, the driver's tasks function is the routine that actually services the peripheral as required and clears the interrupt flag. So, the ISR must call the appropriate tasks function to service the interrupt, as shown in the following example.

Interrupt-Driven Execution

```
void __ISR ( _TIMER_3_VECTOR ) _InterruptHandler_TMR_3_stub( void )
{
    /* Call the timer driver's "Tasks" routine */
    DRV_TMR_Tasks( appDrvObject.tmrDrvObject );
}
```

```
void DRV_TMR_Tasks( SYS_MODULE_OBJ object )
{
    DRV_TMR_OBJ *dObj = (DRV_TMR_OBJ *) object;

    if ( true == SYS_INT_SourceStatusGet(dObj->interruptSource) )
    {
        _DRV_TMR_PeriodSet(dObj->tmrId, dObj->timerPeriod);
        dObj->elapseStatus = true;

        /* Call the client back if the alarm is active */
        if ( dObj->alarmInUse == true )
        {
            dObj->alarmCount = dObj->alarmCount + 1;
            if( dObj->alarmCallback != NULL )
            {
                dObj->alarmCallback();
            }
        }

        /* Clear Timer Interrupt/Status Flag */
        SYS_INT_SourceClear(dObj->interruptSource);
```

It is preferable that any driver or module that supports interrupt-driven operation supports this mechanism, making it flexible enough to be easily configured for any microcontroller that supports the peripheral in question. However, it is acceptable to embed the ISR vector function into the driver or module's implementation. But, doing so will likely restrict the driver to only supporting a single instance of a single module on a small set of microcontrollers. So, all resulting restrictions must be clearly and conspicuously documented.

Additionally, it is preferable any tasks routine that is designed so that it can be called from an ISR, should also be configurable so that it can be called from a polled loop. At a minimum, testing ISR tasks functions in a polled environment will improve the robustness of the function by ensuring that it correctly handles spurious interrupts (since a polled tasks routine will be called many times when it has no actual task to perform and it must be able to distinguish when it does and does not have anything to do).

4.1.3: Polled Execution in a Super Loop With No RTOS

This topic describes Polled execution.

Description

The simplest MPLAB Harmony execution environment (for systems with relaxed timing requirements) is a polled "super loop" environment, as illustrated in the following example.

Polled Execution

```
int main(void)
{
    SYS_Initialize(NULL);

    while(true)
    {
        SYS_Tasks();
    }

    retur
}
```

```
void SYS_Initialize( void* data )
{
    SYS_CLK_Initialize( &clkInit );
    BSP_Initialize();
    modules.usart  = DRV_USART_Initialize( APP_DRV_USART_INDEX, &usartInit );
    modules.tmr    = DRV_TMR_Initialize(   APP_DRV_TMR_INDEX,   &tmrInit );
    modules.sysTmr = SYS_TMR_Initialize(   APP_SYS_TMR_INDEX,   &sysTmrInit );

    APP_Initialize();
}
```

```
void SYS_Tasks ( void )
{
    DRV_USART_TasksTX( modules.usart );
    DRV_TMR_Tasks(     modules.tmr );
    SYS_TMR_Tasks(     modules.sysTmr );

    APP_Tasks();
}
```

It is preferable that all modules support execution in this environment and that any tasks functions provided by the module be callable using a polled method. At a minimum, testing tasks functions in a polled environment will improve the robustness of the function, ensuring that it correctly handles spurious calls. However, it is acceptable for a module to be limited to an interrupt-driven or RTOS-based configuration as long as all restrictions are clearly and conspicuously documented.

4.2: Support a Broad Set of PIC32 Microcontrollers

This topic discusses PIC32 MCU support.

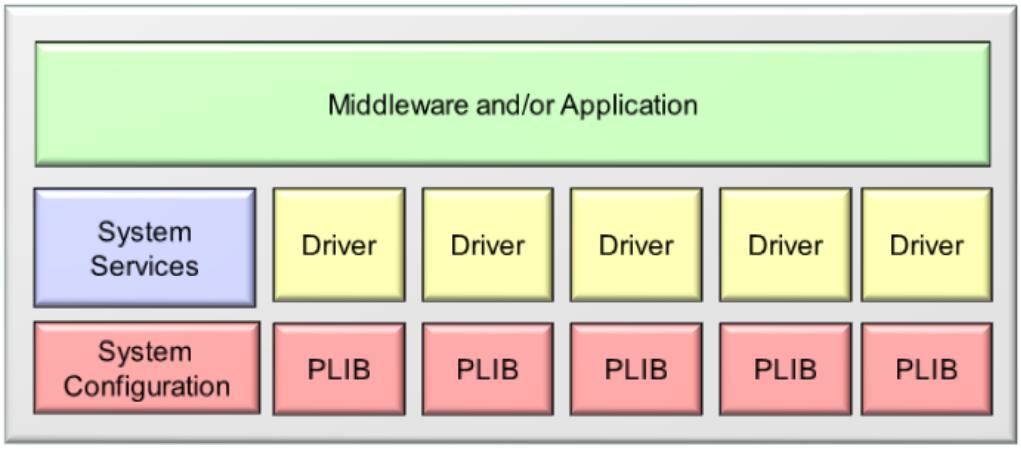
Description

Different PIC32 microcontrollers support different features and different sets of peripherals. These differences are abstracted away from middleware and application modules by three different layers in the MPLAB Harmony framework:

- Device Drivers
- System Services
- Peripheral Libraries

Device drivers provide highly abstracted file-system like interfaces to most types of peripherals. System services provide highly abstracted interfaces to common system-resource types of peripherals, like interrupt controllers, I/O ports, and DMA. Above these interfaces, any part-specific knowledge that is required is contained in parameters that can be defined at build time by the system configuration header (`system_config.h`) or passed into an initialization function at system boot/reset time. So, MPLAB Harmony drivers and system services together can be thought of as a Hardware Abstraction Layer (HAL) as shown by the following block diagram.

Drivers and Services Abstract Peripherals

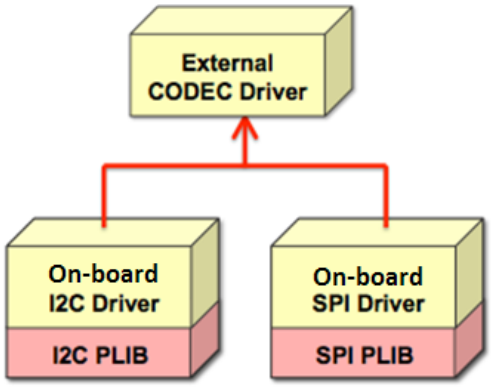


Note: Some driver or system services features may not be supported on specific microcontrollers as MPLAB Harmony libraries do not usually emulate hardware features in software.

Beneath the module's interface, device drivers and system services may have different implementations for different microcontroller families. However, to dramatically reduce the number of implementations required, drivers and system services are normally implemented using peripheral libraries (or by using other drivers or system services in a "stack"). Peripheral libraries provide consistent interfaces for each type of peripheral supported on Microchip microcontrollers, but they may have different implementations for the different variants of that peripheral that are available on different part families.

Note: Peripheral libraries are only supported for internal (on board) microcontroller peripherals. External peripherals are always connected to the microcontroller through one or more internal peripheral interfaces and therefore do not require peripheral library support. They usually use a driver for an on-board peripheral (that does use a PLIB) to access the external device, as shown in the following diagram.

Example External Driver "Stack"



It is preferable that other library or application modules interact with microcontroller peripherals only through abstracted device driver and system service interfaces for portability across the widest range of PIC32 microcontrollers and usages. However, it is possible to obtain portability across one or more PIC32 families (assuming they support the same type of peripheral) by interacting directly with the peripheral library. However, if library modules access peripheral registers directly, they will be restricted to usage only on the set of microcontrollers that support the specific registers used. While direct register access is not advised, it is still possible to us that method and be MPLAB Harmony-compatible as long as all restrictions are clearly and conspicuously documented and the following two key rules are followed.

Key Rules:

1. Any library that directly accesses a peripheral (either through a PLIB or by its registers) "owns" that peripheral and no other code may access the peripheral except through the interface to the library that owns it.
2. The library that "owns" a peripheral (usually a driver for that peripheral) is responsible for ensuring that clients only access that peripheral in safe ways that prevent them from interfering with each other. (Explicit tests and protection can be implemented in the driver in ways that are available for testing and debugging, but can be removed in production builds to reduce run-time overhead.)

Ultimately, PLIB usage is not a MPLAB Harmony requirement. PLIBs are used to support implementations of drivers and services that are flexible enough to be easily configured for different part families. However, ownership and encapsulation of peripherals by a driver or service that allows it to be shared (either one client at a time or multiple clients at once) is a requirement.

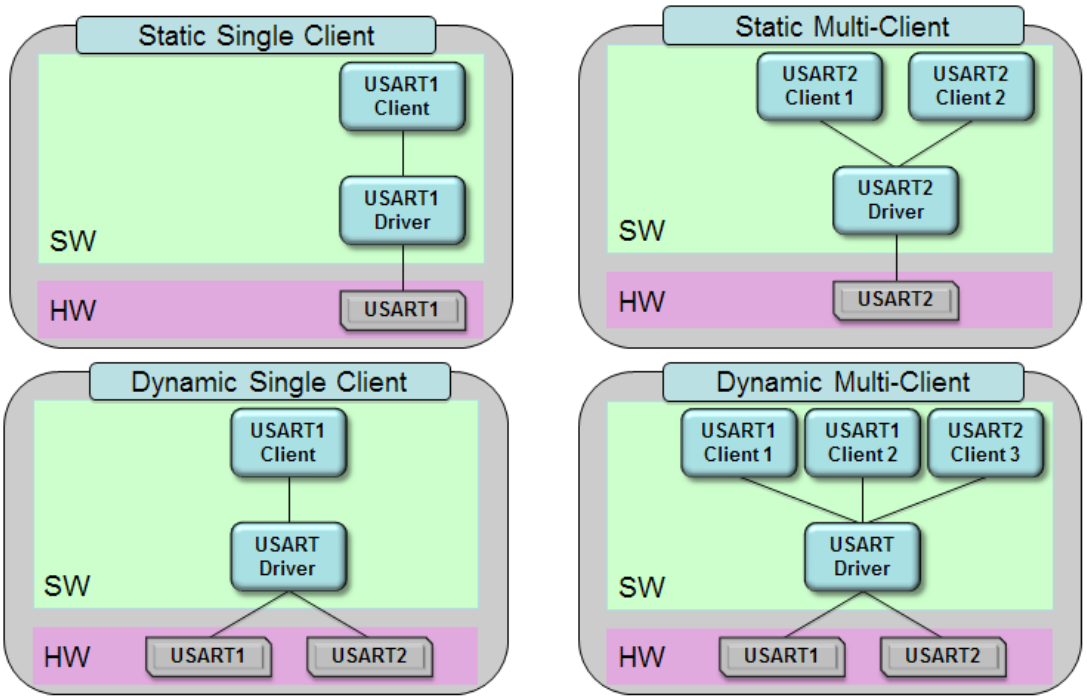
4.3: Support a Dynamic Interface

This topic discusses support for a dynamic interface.

Description

Most MPLAB Harmony library modules are intended to support multiple instances of themselves, each of which may support multiple clients.

Dynamic Versus Static and Single-Client Versus Multi-Client Modules



The previous diagram represents examples all four possible implementations for a USART driver. The top-Left example is a single driver that supports a single client (or user) of a single USART peripheral. The one to the Right of it shows a driver that would allow two clients (individual modules, completely unaware of each other) to share the same USART peripheral. The bottom-Right diagram shows a single driver that allows multiple clients to share multiple USART peripherals (notice that there are two clients to the USART1 instance and a third client for the USART2 instance). The last combination (to the bottom-Left), where a driver can support multiple hardware instances, but only one client at a time, is a special case of a fully dynamic, multi-client driver and not usually a separate implementation.

To support the ability to dynamically manage multiple instances, each potentially having multiple independent clients, the set of interface (or API) routines must support the ability to initialize each instance of the module and the ability to create a logical link between different clients for each instance of the module. MPLAB Harmony modules all require an "Initialize" function (which use standard return values and parameters) that must be called once during system initialization for each instance of the module to be used in the system. MPLAB Harmony device drivers all require the use of an "Open" function (which also uses standard return values and parameters) that provides a "handle" which links each client to a specific instance of the module. Other libraries (such as system services and middleware) may use different methods, but the interface must allow a fully dynamic implementation of the module to manage both multiple instances and multiple clients effectively.

However, it is acceptable for a specific implementation of a module may support only a single static instance of itself and/or support only a single client as long as the interface to the module allows for the ability to provide a fully dynamic (multi-instance, multi-client) implementation. The interface must not need to be changed if such an implementation is required in the future.

Note: It is acceptable to expose a static (single-instance or single-client) interface as long as macros or inline functions are provided that translate calls to the dynamic interface into calls to the static interface. This method provides for source level compatibility, but may require a client to be rebuilt when switching from a dynamic to a static implementation (or vice versa).

4.4: Support Dynamic, Static, and Feature Implementation Variants

This topic discusses support for dynamic, static, and feature implementations.

Description

The basic implementation of a MPLAB Harmony library module is assumed to be fully dynamic (multi-instance and multi-client) and fully featured module, unless otherwise documented. However, it is possible (and usually desirable) to provide different implementation variants so that a user can select the specific implementation or set of features that suits their needs.

Different variants of the same feature(s) can be implemented in different source (`.c`) files (and potentially, in associated header (`.h`) files). The user must select the correct set of source files to build the version of the library he or she desires. Thus, the purpose or use of each source file must be clearly documented. If an implementation is provided that is static or single client, it must be clearly documented as such. If optional features are implemented in separate source files (a practice that is recommended over using `#if` statements to add or remove code from source files that are always included), then the purpose of each file must be clearly documented so that the user knows which files to include in their project. Also, if project configuration tools are available, it is recommended that they be supported.

4.5: Support Build Time Configuration Options

This topic describes build time configuration options.

Description

MPLAB Harmony libraries usually support several different types of build-time configuration options.

Types of Configuration Options

- Selection of supported Microchip microcontroller
- Interrupt-driven or polled execution
- Static or dynamic peripheral instance selection (for drivers)
- Single- or multi-client support (for drivers)
- Library-specific options

MPLAB Harmony drivers and some system services are built upon MPLAB Harmony PLIBs. The appropriate PLIB implementation is selected when the specific microcontroller is selected in the build environment. Selection of the supported microcontroller is done from within the MPLAB X IDE (and, ultimately, by passing the correct build switch on the compiler command line). This mechanism supports easy porting from one Microchip microcontroller to another when using PLIBs, provided the required features are supported. Selection of all other options is done using static configuration options that need to be defined as C-language macros (`#define` statements) by the user in the `system_config.h` header (a file that is included by all libraries) or by selection of appropriate implementation (`.c`) files.

Drivers should be easily configured to operate in an interrupt-driven mode or in a polled mode. If the driver (or other library) is configured to be interrupt driven, then the driver's "Tasks" routine will be called from the appropriate ISR. If the driver is configured for polled operation, then the driver's "Tasks" routine will be called from the main super loop (or an RTOS thread). The selection of which method is used is usually a required configuration option.

The interfaces to MPLAB Harmony drivers are designed to support dynamic (run-time) selection of the specific instance of a peripheral with which the driver interacts. However, since drivers can be implemented so that they are statically associated with a specific instance of a peripheral, the mechanism used to make this choice must be clearly documented.

Similarly, since a driver can allow only a single client to open and use it at one time or it can allow multiple clients to open and use it at the same time, the mechanism used to make this choice must be clearly documented.

Finally, there may be any number of driver or library-specific (build time) configuration options that can be supported. The meaning and usage of each option must be clearly documented.

4.5.1: Support Required Configuration Options

This topic discusses required configuration options.

Description

Required configuration options do not have default definitions. The user must define them to build the library or application. Use of such options must be minimized and clearly documented. It is preferable to utilize the compiler's preprocessor capabilities to test for appropriate definition of required configuration options before they are used and provide a meaningful explanation to the user (via `#error` or `#warning` directives) if they are not correctly defined (see the following example).

Example: Providing an Error for Required Options

```
/* The DRV_USART_INTERRUPT_DRIVEN option must be defined as "true" if this driver is
   used in a interrupt-driven mode or as "false" if used in a polled mode. */
#ifndef DRV_USART_INTERRUPT_DRIVEN
    #error "DRV_USART_INTERRUPT_DRIVEN must be defined as either true or false"
#endif
```

4.5.2: Support Optional Configuration Options

This topic discusses optional configuration options.

Description

Optional configuration options must have default definitions that provide the most commonly desired or safe definitions so that the user need not explicitly define them. These options are normally used to "fine tune" a library to optimize resource usage or performance. The default definitions should allow the expected functionality of a library to be available. Optional configuration options must still be clearly documented.

Example: Supporting an Optional Configuration Option

```
/* This default value of the DRV_USART_READ_BUFFER_SIZE option can be overridden
   to select a different buffer size (in bytes). */
#ifndef DRV_USART_READ_BUFFER_SIZE
    #define DRV_USART_READ_BUFFER_SIZE  10
#endif
```

5: Testing Guidelines

This describes testing guidelines.

Description

Testing is necessary to verify that a library functions as designed and that it follows the compatibility guidelines in this document. While no amount of testing can guarantee the absence of all possible issues, this makes recommendations on key types of testing that will help identify common issues.

Errors in behavior must be reported in release documentation if they cannot be fixed in time for a specific release. Reasonable efforts must be made to correct all reproducible errors in behavior. Reporting the complete results of all tests in published documentation is recommend so that customers can make informed decisions on the suitability of a specific solution. If feasible it is also recommended to publish the tests themselves (and descriptions of the required test platforms) so that customers can reproduce the results themselves if they so desire.

Generation of a suite of tests is also recommended. Each test should be designed as a MPLAB Harmony state-driven module that can be easily integrated with other tests or applications and that can be easily configured to operate in multiple different execution environments so that a multitude of possible uses can be tested. Such a suite of tests will help simplify the task of implementing and executing the tests recommended in this section.

5.1: Test All Possible Build Configurations

This topic discusses testing all possible build configurations.

Description

It is not acceptable to publish source code that does not build correctly in all supported configurations. So, it is recommended that build tests be performed that verify all superset combinations of supported configuration settings. It is not usually practical to build test all possible combinations. However, it is much more feasible to build configurations that define the largest sets of all compatible options. Some configurations may be incompatible with each other (for example, you cannot simultaneously configure a library to run both interrupt driven and polled). Therefore, it may take more than one build test to verify all supersets of configuration options.

MPLAB Harmony compatible libraries (distributed in source code) are expected to build with zero errors and warnings on the MPLAB XC32 C/C++ Compiler with all warnings enabled (i.e., with the `-Wall` switch defined on the command line).

5.2: Test for Correct Functionality

This topic discusses testing for correct functionality.

Description

To validate that a library provides the intended functionality, it is necessary to create applications that exercise that functionality. With the potential flexibility of MPLAB Harmony compatible libraries, there may be many possible environments and platforms on which the tests may be executed. It is best to develop test applications that follow the guidelines for Harmony modules so that these applications can be easily integrated into different test harnesses in different environments. In all cases, the tests should verify that the library meets the desired requirements.

5.3: Stress Test and Measure Performance

This topic discusses stress testing and measuring for performance.

Description

So that customers can select the appropriate hardware platform and verify that a library meets their performance needs, tests must be developed that measure the performance of a library (using appropriate metrics) and that execute the tests in both lightly loaded and heavily loaded environments. To do this, it may be necessary to measure any number of metrics (especially time) and to artificially generate processor or bus-loading conditions, spurious interrupts, and other resource restrictions or optimization levels.

5.4: Test Error Handling

This topic discusses error handling testing.

Description

Testing error handling will help ensure that libraries fail "gracefully" when passed bad parameters or used incorrectly. To do this, call library functions with incorrect parameter values or in incorrect sequences. Verify that the library provides an acceptable mechanism for detecting and recovering from the error (if possible) and that it does not respond in an unacceptable way. Key things to look for are potential causes of exceptions, lockups and memory violations. Sometimes it is best to perform error tests and stress tests in combination to help identify any possible transitory error conditions.

5.5: Test All Supported Execution Environments

This topic discusses testing libraries for all environments.

Description

Fully compliant MPLAB Harmony Libraries can be configured to execute in three fundamentally different ways:

- Under a supported RTOS
- Interrupt-driven (when an appropriate interrupt is supported)
- Polled in a non-RTOS "bare metal" super loop environment

Libraries may or may not support all three environments, but it is important to test libraries in all of the environments that they do support to validate correct behavior. If feasible, test for correct functionality, acceptable performance, and robust error handling in all supported environments.

5.5.1: Test for Thread Safe Execution in All Supported RTOS Configurations

This topic discusses testing for thread safe execution in RTOS configurations.

Description

There are several different possible situations that may occur that are unique to execution in an RTOS environment.

- Some library "Tasks" functions may be executed interrupt-driven or polled from an RTOS thread
- For libraries that support multiple "Tasks" functions:
  - Each tasks function may be called from a different thread,
  - Multiple tasks functions may be called from the same thread, or
  - Some tasks functions may be called from an ISR and some called from threads
- Interface functions may be called from the same thread as tasks functions
- Interface functions may be called from different threads than tasks functions
- Interface functions may be called from multiple different threads that are different from tasks functions
- Interface functions may be called from threads while some tasks functions are called from an ISR

Libraries must be tested for correct operation within each of these situations under each supported RTOS. Also, it is necessary to test with different combinations of priorities for each of the previously mentioned threading models and document any restrictions on relative thread priorities.

5.5.2: Test for Correct Interrupt-driven Execution (if Supported)

This topic discusses testing interrupt driven execution.

Description

Some libraries (especially device drivers) are designed such that one or more of their "Tasks" functions can be called from an ISR. For libraries that support such execution, configurations must be tested where these tasks functions are called from the appropriate ISR. This must be tested in all supported RTOS and bare-metal (non-OS) environments. These ISR-supporting tasks functions must also be tested in a polled execution model, if that model is supported.

5.5.3: Test for Correct Polled Execution

This topic discusses testing polled execution.

Description

Libraries that support execution in a bare-metal (non-RTOS) "super loop" environment must be tested in this configuration. This is the most basic execution model for MPLAB Harmony compliant execution, but it is also the most restrictive due to potential timing requirements. Thus, testing in this environment should measure and report execution time of module "Tasks" functions and interface functions.

5.6: Test Multi-instance Support (if Supported)

This topic discusses multi-instance support testing.

Description

Libraries that support dynamically managing multiple instances of themselves (and potentially multiple instances of peripheral hardware) must be tested to verify that they behave correctly when instantiated multiple times. Multiple instance configurations should be tested in all supported

execution environments.

5.7: Test Multi-client Support (if Supported)

This topic discusses multi-client support testing.

Description

Libraries that are intended to support multiple clients must be tested for correct behavior when used by multiple clients at the same time in all supported execution environments.

5.8: Test for Correct Interoperability With Other MPLAB Harmony Modules

This topic discusses interoperability testing with other MPLAB Harmony modules.

Description

While it may not practical to test a specific library with every other MPLAB Harmony library available, it is recommended to test libraries with each of the major middleware stacks and commonly used drivers and system services. If tests for middleware, drivers, and services are not available, interoperability testing may be performed by integrating the specific library's tests into demonstration programs provided in the MPLAB Harmony release.

5.9: Test On All Major PIC32 Device Families

This topic discusses testing libraries on the major PIC32 device families.

Description

While it may not be practical to test a library on every supported PIC32 processor, it is recommended to test libraries on the largest members of each major PIC32 family.

6: Documentation Guidelines

This provide guidelines for creating documentation.

Description

Libraries released as part of the MPLAB Harmony installation follow specific documentation guidelines. At a minimum, any third-party MPLAB Harmony compatible libraries or applications that do not support the recommended MPLAB Harmony capabilities or requirements must clearly and conspicuously document the restrictions. However, it is recommended that third-party documentation for MPLAB Harmony compatible libraries provide similar information to that provided by MPLAB Harmony's own libraries.

In addition to documenting the functions, data types, and other C-language elements, the interface documentation must explain (or provide a reference to) any intrinsic knowledge necessary to use the library interface.

Recommended Help Document Hierarchy

Introduction

Give a brief introduction of the module. This should include a brief description basic concepts of the module, but do not attempt to explain detailed standards or protocols. Instead provide links to specifications or detailed explanations. Use simple diagrams (if applicable).

See the Timer Driver Library Introduction section for an example.

Using the Library

This topic describes the basic architecture of the library and provides information and examples on how to use the functions provided by the library to satisfy common usage scenarios.

See the Timer Driver Library Using the Library section for an example.

Abstraction Model (sub-topic)

Briefly describe the abstraction model or usage model of the library. For example, a device driver library abstracts a peripheral device and presents a file-system like interface with one or more data transfer and control models. Briefly explain each model or refer to more detailed usage topics.

See the Timer Driver Library Abstraction Model section for an example.

How the Library Works (sub-topic)

Explain the common usages. Describe how to use the library's interface functions to perform common tasks. Provide a subtopic for each common usage scenario (especially initializing the library and setting it up for common usage models).

See the Timer Driver Library How the Library Works section for an example.

Configuring the Library

List and explain the build-time configuration requirements and options of the library.

See the Timer Driver Library Configuring the Library section for an example.

Building the Library

List and explain the source files necessary to build this library. If some files are optional (i.e., implement optional features) explain why the user would want to include then in the build. Also list all modules on which the library depends (but let the documentation for those modules explain how to build them).

See the Timer Driver Library Building the Library section for an example.

Library Interface

Logically partition the interface routines in the sets and provide a "programmer's reference" for each that describes its purpose, behavior, usage, parameters, return values, and side effects. A good guideline for partitioning of the library interface is that, if the library abstraction model can be drawn using a block diagram, then partition the library interface routines in similar sets.

See the Timer Driver Library Library Interface section for an example.

7: MPLAB Harmony Version Compatibility

This discusses version compatibility.

Description

It is common for one library to use (or depend upon) another. Any library distributed as part of a MPLAB Harmony installation it tested with the libraries (on which it depends) that are also part of the same installation. However, any library distributed separately (from an installation of MPLAB Harmony) must identify the version of MPLAB Harmony with which it was tested and is known to be compatible. Future versions of such libraries should maintain equivalent "backward" compatibility (i.e., the newly released library should still be compatible with the version of MPLAB Harmony with which it was originally tested.)

However, if the library is later updated to use new MPLAB Harmony functionality (functionality that is available in a newer release and was not available in the MPLAB Harmony release with which it was originally tested) then the updated library has the following options, in order of preference.

- Maintain backward compatibility to the older MPLAB Harmony version if the new functionality is not used
- Maintain backward compatibility to the older MPLAB Harmony version as a build or installation option
- Update the minimum MPLAB Harmony version number with which the library is compatible, but provide a reasonable transition period where the older version of the library is still made available and supported

It is not acceptable for an updated version of the library to simply stop working with the older version of MPLAB Harmony without a warning or explanation to the user.

8: Compatibility Worksheet

This provides information on the compatibility worksheet.

Description

Use this fillable and printable PDF form to determine the level of MPLAB Harmony compatibility and to capture any exceptions or restrictions to the compatibility guidelines. The form allows you to enter and save information as you progress with your library.

A copy of the worksheet is available in the `<install-dir>/doc` folder. It is suggested to make a personal copy of the original form for your own use. The following image shows a portion of the first page of the worksheet form. The blue shaded areas indicate the portions of the worksheet that can be filled in.

Portion of First Page of the Worksheet

MPLAB Harmony Compatibility Worksheet

The first column in this worksheet references by section number, the related information in the *MPLAB Harmony Compatibility Guide*. This guide is located within the *MPLAB Harmony Help* in the *Understanding MPLAB Harmony* section.

Use this compatibility worksheet to determine the level of MPLAB Harmony compatibility and to capture any exceptions or restrictions to the compatibility guidelines. In the Compliant column, enter one of the following values:

- Yes – If supported and fully compliant
- No – If not compliant (list exceptions or provide an explanation)
- Not Applicable – If not applicable (list exceptions or provide an explanation)

Section Number	Description	Compliant
3	List module name, describe what it abstracts, and identify if it integrates the functionality of any other known modules.	
Module Name & Abstraction:		

Section Number	Description	Compliant
3.1	Interface completely documented and isolated from implementation	
Exceptions/Restrictions:		

Section Number	Description	Compliant
3.2	Respects all other abstractions (or list any globally accessed resources).	
Exceptions/Restrictions:		

MPLAB Harmony Driver Development Guide

This guide provides information on developing MPLAB Harmony device drivers.

Introduction

Describes how to develop MPLAB Harmony device drivers.

Description

This development guide describes how to develop device drivers for MPLAB Harmony. MPLAB Harmony device drivers, or simply "drivers", typically utilize MPLAB Harmony Peripheral Libraries (PLIBS) to access and control built-in peripheral hardware. A driver is the *glue* logic between an application and the peripheral library. The peripheral library provides a low-level interface to a specific peripheral, but use of that interface would place a lot of responsibility on the application for maintaining the state of the device and ensuring that other (usually unrelated) modules do not interfere with its operation. Instead, these functions become the responsibility of the driver, freeing the application from managing devices and considerably simplifying the interface to the peripheral.

Drivers provide simple C-language interfaces (see **Note**). A driver's interface should be highly abstracted and provide file system style "*open*" and "*close*", and "*read*" and "*write*" functions (for data transfer peripherals) that allow applications (or other client modules) to easily interact with them in a consistent manner. Most drivers also provide additional functions that are unique to a particular type of driver or peripheral, but are independent of the details of how that peripheral is implemented on any specific hardware or how many instances of that driver or peripheral exist in a given system.

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

Drivers can also indirectly support external peripheral hardware, which has no peripheral library, by accessing another driver. For example, a SD Card driver may use a SPI driver to access an external Flash device. Or, a driver may be completely abstracted, utilizing no peripheral hardware at all and simply providing some device-like service to higher layers of software.

Regardless of the type or lack of hardware a MPLAB Harmony driver manages, it has the following fundamental responsibilities:

- Providing a common system-level interface to a peripheral
- Providing a highly-abstracted file system style client interface to a peripheral
- Controlling access to a peripheral
- Managing the state of a peripheral

Information on how a driver can fulfill these responsibilities and other key concepts and requirements for design and development of MPLAB Harmony device drivers is provided in the following sections.

Using This Document

Describes how to best use this document, depending upon your experience level and familiarity with MPLAB Harmony.

Description

Experienced MPLAB Harmony Driver Developer

If you are an experienced MPLAB Harmony driver developer, you can skip to the Checklist and Information section at the end of this document for a list of the tasks necessary to develop a MPLAB Harmony driver and a handy reminder of the file and folder naming and organization conventions.

Experienced Embedded Software Developer

If you are an experienced embedded software developer who is familiar with modular design and state-machine based development, but are not familiar with MPLAB Harmony driver development, you can briefly scan the Key Design Concepts section and jump to the System Interface and Client Interface sections. Also, be sure to review the Interrupt and Thread Safety section for examples of recommended methods to use in MPLAB Harmony drivers and read the remaining sections in detail.

MPLAB Harmony User Who Has Yet to Develop a Driver

If you are a MPLAB Harmony user who would like to start developing MPLAB Harmony drivers, please read this entire document.

New to MPLAB Harmony

If you are new to MPLAB Harmony, please read the What is MPLAB Harmony? section first.

System Interface

Describes the system interface requirements for driver design.

Description

In MPLAB Harmony, almost everything running in the system is considered a module. A module usually has an internal state machine that runs when the system runs. Exactly what it means to *run* in the system depends on the configuration that is selected. In the simplest polled bare metal configuration a MPLAB Harmony system runs modules in a single polled super loop, implemented in the main function, as shown in the following example.

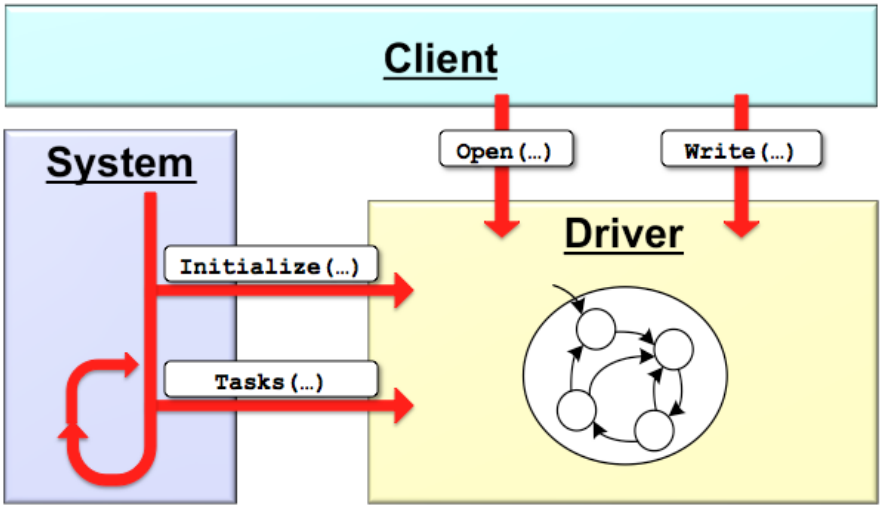
The Main Function

```
int main ( void )
{
    SYS_Initialize(NULL);

    while( true )
    {
        SYS_Tasks();

    }

    return(EXIT_FAILURE);
}
```

The SYS_Initialize and SYS_Tasks functions are implemented in system configuration-specific files, normally generated by the MPLAB Harmony Configurator (MHC). The SYS_Initialize function calls the initialization functions of all modules in the system. The initialization function of a module must (at a minimum) initialize the state machine of that module so that its tasks function can be safely called. Then, the SYS_Tasks function calls the tasks functions of all modules in the system. This continues indefinitely, keeping all of the modules in the system running. Effectively, the main loop and the SYS_Initialize and SYS_Tasks functions implement a simple system kernel scheduler as a round-robin polled super loop.

The signatures of these functions are always consistent, using the same parameters and return values. Only the names change from module to module. This provides a consistent execution model for polled, interrupt-driven, and RTOS-based environments and supports the ability to implement system executives, power mangers, and test harnesses that initialize, deinitialize and maintain multiple modules.

In any module, the system interface should be thought of as conceptually separate from the application or client Interface (what is commonly referred to as the API), as shown in the following diagram.



These two interfaces serve completely separate purposes. The system interface allows system kernel or scheduler to initialize and run the module (as described previously) and the client interface allows the application or other client module to interact with the driver or module. The system code does not normally interact with the client-level API of the module and the application or any other client should not call the system interface functions or have access to the object handle (explained in the following sections). Once the system has been initialized, client or application code can call the driver's client interface functions (such as open, read, and write functions). The client interface is described in detail in the Client Interface section.

Once a driver module has been initialized, its state machine must be placed into its initial state and it can be considered ready to use. Although its internal state machine may have still several initial transitions to complete, its other system and client interface routines may be called.

The initial power state of the module can be defined by build-time configuration parameters or be dependent upon the hardware initialization data passed into the initialize operation. Driver modules can be initialized in a power-on, full running state or a power-off or low-power state and reinitialized later under system control to a power-on state when they are needed. Drivers may also be reinitialized to refresh the hardware state.

A driver can be deinitialized if it does not need to be used any longer, after which none of its other interface functions may be called without first calling the initialize function again. The initialize function must not be called more than once without first calling the deinitialize function. The reinitialize operation can be called any time the module is in a ready state.

A driver's system interface consists of the following functions, where `<module>` matches the module abbreviation for the driver or peripheral.

Driver's System Interface Functions

Function	Description
DRV_<module>_Initialize	Place the driver in its initial state.
DRV_<module>_Tasks	Manage the running state of the driver.
DRV_<module>_Reinitialize	Change a running driver's initial parameters.
DRV_<module>_Deinitialize	Disable the driver and stop it from running.
DRV_<module>_Status	Provide the current status of the driver

A driver can have only one of each of these functions, except for the DRV_<module>_Tasks function. It is possible for a driver to have multiple different tasks functions, each maintaining a different state machine within the driver. The naming format for device-driver system module routines adds the DRV_ prefix to identify that the function belongs to a device driver module and to be consistent with the driver's other interface routines.

Full descriptions of the driver-module interface routines that implement these operations are provided in the following sections.

Module Initialization

Describes the details of the module initialization function.

Description

The function signature of a module's initialize function is defined by a pointer data type that is defined by the system module interface header, in the <install-dir>/framework/system/common/sys_module.h file, as shown in the following example.

Example: Module Initialization Function Signature

```
/*********************************************************************
  System Module Initialization Function Pointer

  Function Pointer:
    SYS_MODULE_OBJ (* SYS_MODULE_INITIALIZE_ROUTINE) (
                        const SYS_MODULE_INDEX index,
                        const SYS_MODULE_INIT * const init )

  Description:
    This data type is a pointer to a function that initializes a system module
    (driver, library, or system-maintained application).

  Preconditions:
    The low-level processor and board initialization must be completed before the
    system can call the initialization functions for any modules.

  Parameters:
    index           - Zero-based index of the module instance to be initialized.

    init            - Pointer to the data structure containing any data
                      necessary to initialize the module. This pointer may
                      be null if no data is required.

  Returns:
    A handle to the instance of the module that was initialized.  This handle is
    a necessary parameter to all of the other system level routines for that
    module.

  Remarks:
    This function will normally only be called once during system initialization.
*/

typedef SYS_MODULE_OBJ (* SYS_MODULE_INITIALIZE_ROUTINE) (
                            const SYS_MODULE_INDEX index,
                            const SYS_MODULE_INIT * const init );
```

An implementation of a module's initialization function might look like the following example.

Example: Sample Module Initialization Function

```
SYS_MODULE_OBJ SAMPLE_Initialize ( const SYS_MODULE_INDEX index,
                                   const SYS_MODULE_INIT * const init )
{
    SAMPLE_MODULE_DATA      *pObj  = (SAMPLE_MODULE_DATA *)&gObj[index];
    SAMPLE_MODULE_INIT_DATA *pInit = (SAMPLE_MODULE_INIT_DATA *)init;
```

```
    /* Initialize module object. */

    pObj->state                = SAMPLE_STATE_INITALIZE;
    pObj->status               = SYS_STATUS_BUSY;
    pObj->dataNewIsValid       = false;
    pObj->dataProcessedIsValid = false;

    if ( null != init )
    {
        pObj->dataNew          = pInit->dataSome;
        pObj->dataNewIsValid   = true;
    }

    return (SYS_MODULE_OBJ)pObj;
}
```

Ignoring the index parameter and the return value for now, you can see from the sample code above that a module's initialization function accepts a pointer to an initial data (`init`) structure, casts the pointer to a new type, and then stores data from the `init` structure into an internal module object structure. Because of the requirement for consistency in function signature, the initialize function for any module must use a common data type for the `init` data pointer parameter. This data type, shown in the following example, is also defined in the `sys_module.h` header file.

SYS_MODULE_INIT Structure

```
typedef union
{
    uint8_t         value;

    struct
    {
        uint8_t     powerState  : 4;

        uint8_t     reserved    : 4;
    }sys;

} SYS_MODULE_INIT;
```

The SYS_MODULE_INIT structure allows the system to pass in a pass in a parameter (the `powerState` member) to identify the requested initial power state of the module. The following values and labels are predefined (in `system_module.h`) to provide a common set of power states for all modules.

Predefined Power States

Volume	Label	Description
0	SYS_MODULE_POWER_OFF	Module Power-off state code.
1	SYS_MODULE_POWER_SLEEP	Module Sleep state code.
2	SYS_MODULE_POWER_IDLE_STOP	Module Idle-Stop state code.
3	SYS_MODULE_POWER_IDLE_RUN	Module Idle-Run state code.
4 through 14	<Module-specific Definition>	Module-specific meaning.
15	SYS_MODULE_POWER_RUN_FULL	Module Run-Full state code.

Note:

Note: Refer to the Help documentation for each individual label for a more detailed description of what each value indicates. See Framework Help > System Service Libraries Help > System Service Overview.

Using a pointer to this structure as the `init` parameter allows system code to treat all modules in a consistent way. However, any specific module or implementation of a module may have its own unique `init` data requirements and may define its own unique structure type. Unfortunately, the C language does not provide a syntactical mechanism for managing this sort of polymorphism. Polymorphism is an object oriented programming (OOP) concept that allows different types (or classes) of data (or other objects) to support multiple forms. To achieve this flexibility in the C language, the module must cast the pointer to an internally defined data type. But, it is reasonable to think of the SYS_MODULE_INIT structure as a base class, extended as necessary by any individual module class or implementation to contain the specific additional initialization data it requires. While this is a slight abuse of the C language, it works as required as long as the first member of any module's extended `init` structure is a SYS_MODULE_INIT structure, which is (of course) a requirement of any MPLAB Harmony module.

Whether or not a module requires any initialization data, the primary purpose of its initialization function is to place the module's state machine into its initial state. In the sample module initialization function above, the following line of code does this.

```
pObj->state = SAMPLE_STATE_INITALIZE;
```

In this line, state is simply a structure member variable used to keep track of the current state of the module's state machine and the SAMPLE_STATE_INITIALIZE value is its initial state. This variable is contained within in a structure and accessed using the `pObj` pointer along with all other variables that are specific to a single instance of the driver module. This allows an instance of a driver to be referenced by a single

driver *object* pointer, which is cast to the SYS_MODULE_OBJ data type and returned from the driver's initialize function to be used by the system to identify an instance of the driver and passed into the driver's other system interface functions to access its instance-specific data.

Note: Any module that has a state machine must implement an initialize function.

Module Tasks

Describes the details of the module "Tasks" function.

Description

The function signature of a module's tasks function is defined by a pointer data type that is defined by the system module interface header, in the `<install-dir>/framework/system/common/sys_module.h` file, as shown in the following example.

Example: Module Tasks Function Signature

```
// *****************************************************************************
/* System Module Tasks Routine Pointer

  Function:
    void (* SYS_MODULE_TASKS_ROUTINE) ( SYS_MODULE_OBJ object )

  Summary:
    Pointer to a routine that performs the tasks necessary to maintain a state
    machine in a module.

  Description:
    This data type is a pointer to a routine that performs the tasks necessary
    to maintain a state machine in a module (driver, library, or application).

  Preconditions:
    The low-level board initialization must have been completed and the module's
    initialization function must have been called before the system can call the
    tasks routine for any module.

  Parameters:
    object          - Handle to the module instance

  Returns:
    None.

  Remarks:
    If the module is interrupt driven, the system will call this routine from
    an interrupt context.
*/

typedef void (* SYS_MODULE_TASKS_ROUTINE) ( SYS_MODULE_OBJ object );
```

An implementation of a module's tasks function might look like the following example.

Example: Sample Module Tasks Function

```
void SAMPLE_Tasks( SYS_MODULE_OBJ object )
{
    SAMPLE_MODULE_DATA     *pObj  = (SAMPLE_MODULE_DATA *)object;

    // Sample Module State Machine
    switch ( pObj->state )
    {
        case SAMPLE_STATE_INITIALIZE:
        {
            pObj->status = SYS_STATUS_READY;
            pObj->state  = SAMPLE_STATE_PROCESS;
            break;
        }

        case SAMPLE_STATE_PROCESS:
        {
            if (pObj->dataNewIsValid && !pObj->dataProcessedIsValid)
            {
                pObj->dataProcessed         = pObj->dataNew;
```

```
                pObj->dataNewIsValid        = false;
                pObj->dataProcessedIsValid  = true;
            }
            break;
        }

        default:
        {
            pObj->status = SYS_STATUS_ERROR;
            break;
        }
    }

    return;
}
```

As shown by the previous sample code, the object handle returned from the initialize function is passed into the tasks function and cast back into a pointer to the module's internal data structure type. This allows the function to access the instance-specific data it contains. The module may then utilize the data to determine what its next appropriate action may be. This is often implemented using a state variable (`pObj->state`) and a switch statement with different states defined by an enumeration.

Each case in the switch statement corresponds to a different state transition that the module's state machine can make. The module's initialize function placed the state machine in its initial state (SAMPLE_STATE_INITIALIZE). The initialize state transitions to the next state (SAMPLE_STATE_PROCESS) automatically and, as a side effect, changes the module's status to ready (SYS_STATUS_READY). Therefore, the next time the tasks function is called, it is in the process state and is ready to process data.

In the process state, the sample module checks to see if it has any valid new data (using the Boolean flag `pObj->dataNewIsValid`) and if it is able to process that new data. It can only process one data item at a time. So, if it does not currently have any processed data (as indicated by the `pObj->dataProcessedIsValid` Boolean flag being false), it can then it processes the new data item and update the Boolean flags.

In this sample module, the initialize state transition is not really necessary and could have been eliminated by appropriately setting the module's status variable in the initialize function. Also, the default case is unnecessary and should never occur, although it can be used for error handling. The only active state transition occurs in the process state and it never transitions out of that state to a new one.

It is important to notice that the state machine checks status flags before taking any action. This prevents it from taking action until some change has occurred, which allows it to be polled from the main super loop. The flags in the sample module are modified by the module's API functions, when a client calls them. However, these flags could just as easily have been hardware interrupt flags. If that were the case, this state machine could just as easily be called from an ISR, which is exactly how it would be called in an interrupt-driven configuration. In that configuration, testing the interrupt flag could be redundant. But, doing so is safer as it avoids taking inappropriate actions if spurious interrupts occur or if the interrupt vector is shared between multiple interrupt sources.

Note: Use of the switch-case technique is not a requirement, but it is a convenient way to explain the purpose of the tasks function. Simpler and more sophisticated techniques are possible. Any method that correctly implements the necessary state machine logic is acceptable.

For a complete working example, refer to the sample module library, located in the `<install-dir>/framework/sample` folder.

A module's tasks function may assume that the initialize function has been called once (and only once) before the tasks function is called. But, it must not associate any meaning to when or how often the tasks function is called. It cannot assume that it is called before or after any other tasks has been called and it cannot assume it has been called once for every time any other tasks function has been called. It is entirely possible (particularly in a RTOS-based system) that the module's tasks function is running at a different priority level and is called more frequently or less frequently than other tasks functions in the system. It is also possible that it may be called from within an ISR, but only if it was designed to support an ISR. It is possible to design a tasks function that has no associated interrupt. Every time it is called, a module's tasks function must use current state or status to determine the appropriate state transition to make or if it needs to make any transition at all.

Note: Any module that has a state machine must implement a tasks function.

Module Status

Describes the details of the module "Status" function.

Description

The function signature of a module's status function is defined by a pointer data type that is defined by the system module interface header, in the `<install-dir>/framework/system/common/sys_module.h` file, as shown in the following example.

Example: Module Status Function Signature

```
// *****************************************************************************
/* System Module Status Routine Pointer

  Function:
    SYS_STATUS (* SYS_MODULE_STATUS_ROUTINE) ( SYS_MODULE_OBJ object )
```

```
  Summary:
    Pointer to a function that gets the current status of a module.

  Description:
    This data type is a pointer to a function that gets the current status of a
    system module (driver, library, or application).

  Preconditions:
    The low-level board initialization must have been completed and the module's
    initialization function must have been called before the system can call the
    status function for any module.

  Parameters:
    object          - Handle to the module instance

  Returns:
    One of the possible status codes from SYS_STATUS

  Remarks:
    A module's status function can be used to determine when any of the
    other system level operations has completed as well as to obtain general
    status of the module.
    If the status function returns SYS_STATUS_BUSY, a previous operation
    has not yet completed. Once the status function returns SYS_STATUS_READY,
    any previous operations have completed.
    The value of SYS_STATUS_ERROR is negative (-1). A module may define
    module-specific error values of less or equal SYS_STATUS_ERROR_EXTENDED
    (-10).
    The status function must NEVER block.
    If the status function returns an error value, the error may be cleared by
    calling the reinitialize function. If that fails, the deinitialize
    function will need to be called, followed by the initialize function to
    return to normal operations.
*/

typedef SYS_STATUS (* SYS_MODULE_STATUS_ROUTINE) (  SYS_MODULE_OBJ object );
```

An implementation of a module's status function might look like the following example.

Example: Sample Module Status Function

```
SYS_STATUS SAMPLE_Status ( SYS_MODULE_OBJ object )
{
    SAMPLE_MODULE_DATA     *pObj  = (SAMPLE_MODULE_DATA *)object;

    return pObj->status;
}
```

In most cases, a module's status function will just return the current value of the status variable in the module-instance data structure referred to by the object handle. However, it is possible to deduce the module instance's current status using more complex logic. But, be aware of potential race conditions that could be caused by checking multiple individual variables and/or hardware status flags.

A module's status function must return a value from the SYS_STATUS enumeration (or an extension of it), as shown in the following example.

SYS_STATUS Enumeration

```
// *****************************************************************************
/* System Module Status

  Summary:
    Identifies the current status/state of a system module

  Description:
    This enumeration identifies the current status/state of a system module
    (driver, library, or application).

  Remarks:
    This enumeration is the return type for the system-level status routine
    defined by each driver, library, or application (for example, DRV_I2C_Status).
*/

typedef enum
{
```

```
    /* Indicates that a non-system defined error has occurred.  The caller
       must call an extended status routine for the module in question to
       identify the error. */
    SYS_STATUS_ERROR_EXTENDED   = -10,

    /* An unspecified error has occurred. */
    SYS_STATUS_ERROR            = -1,

    /* The module has not yet been initialized. */
    SYS_STATUS_UNINITIALIZED    = 0,

    /* An operation is currently in progress. */
    SYS_STATUS_BUSY             = 1,

    /* Any previous operations have completed and the module is ready for
       additional operations. */
    SYS_STATUS_READY            = 2,

    /* Indicates that the module is in a non-system defined ready/run state.
       The caller must call an extended status routine for the module in
       question to identify the state. */
    SYS_STATUS_READY_EXTENDED   = 10

} SYS_STATUS;
```

However, a module may define its own module-specific status enumerations that extend the system status enumeration by equating their values and defining new values in the extended error and ready ranges, as shown in the following example.

Example: Sample Module Specific Status Enumeration

```
typedef enum
{
    SAMPLE_STATUS_ERROR_PARITY     = SYS_STATUS_ERROR_EXTENDED - 2,
    SAMPLE_STATUS_ERROR_UNDERRUN   = SYS_STATUS_ERROR_EXTENDED - 1,
    SAMPLE_STATUS_ERROR_OVERFLOW   = SYS_STATUS_ERROR_EXTENDED,
    SAMPLE_STATUS_ERROR            = SYS_STATUS_ERROR,
    SAMPLE_STATUS_UNINITIALIZED    = SYS_STATUS_UNINITIALIZED,
    SAMPLE_STATUS_BUSY             = SYS_STATUS_BUSY,
    SAMPLE_STATUS_READY            = SYS_STATUS_READY,
    SAMPLE_STATUS_READY_BUS_IDLE   = SYS_STATUS_READY_EXTENDED,
    SAMPLE_STATUS_READY_RECEIVING  = SYS_STATUS_READY_EXTENDED + 1,
    SAMPLE_STATUS_READY_SENDING    = SYS_STATUS_READY_EXTENDED + 2

} SAMPLE_STATUS;
```

This allows the module to utilize its own status labels internally in its implementation code and allows its clients to do the same when using the module's API while still allowing the system to utilize the standard values or to check for error ranges below SYS_STATUS_ERROR or ready ranges above SYS_STATUS_READY.

Note: Any module that has a state machine must implement a status function.

Module Deinitialize

Describes the details of the module "Deinitialize" function.

Description

The function signature of a module's deinitialize function is defined by a pointer data type that is defined by the system module interface header, in the `<install-dir>/framework/system/common/sys_module.h` file, as shown in the following example.

Example: Module Deinitialize Function Signature

```
/*********************************************************************
  Function:
    void (* SYS_MODULE_DEINITIALIZE_ROUTINE) (  SYS_MODULE_OBJ object )

  Summary:
    Pointer to a routine that deinitializes a system module.

  Description:
    This data type is a pointer to a routine that deinitializes a system
    module (driver, library, or application).
```

```
  Preconditions:
    The low-level board initialization must have (and will be) completed
    and the module's initialization function will have been called before
    the system will call the deinitialization function for any module.

  Parameters:
    object -  Handle to the module instance

  Returns:
    None.

  Remarks:
    If the module instance has to be used again, the module's "initialize"
    function must first be called.
*/

typedef void (* SYS_MODULE_DEINITIALIZE_ROUTINE) (  SYS_MODULE_OBJ object );
```

An implementation of a module's deinitialize function might look like the following example.

Example: Sample Module Deinitialize Function

```
void SAMPLE_Deinitialize ( SYS_MODULE_OBJ object )
{
    SAMPLE_MODULE_DATA *pObj  = (SAMPLE_MODULE_DATA *)object;

    pObj->dataNewIsValid        = false;
    pObj->dataProcessedIsValid  = false;
    pObj->status                = SYS_STATUS_UNINITIALIZED;

    return;
}
```

In the previous example, the deinitialize function for the sample module simply clears a few key flags and returns. For a simple module, where its other system and client interface routines take no action and return appropriate values when uninitialized, this may be all that is necessary. Other, more complex modules may need their state machines to go through a deinitialize sequence before the modules can be safely considered deinitialized. Such modules must return SYS_STATUS_BUSY from their status functions until the sequence has completed to signal to the system that it must continue calling the module's tasks function(s). Once the system has called a module's deinitialize function and received a SYS_STATUS_UNINITIALIZED status from its status function, it may stop calling the module's state machine. However, it is safer to not make that assumption and place the module in an uninitialized state in which its state machine does nothing. See the Module Tasks section for details.

Note: This function is optional. If a module is not intended to be deinitialized in normal operation (i.e., only initialized after a reset and always running thereafter) it does not need to implement the deinitialize function.

Module Reinitialize

Describes the details of the module "Reinitialize" function.

Description

The function signature of a module's reinitialize function is defined by a pointer data type that is defined by the system module interface header, in the `<install-dir>/framework/system/common/sys_module.h` file, as shown in the following example.

Example: Module Reinitialize Function Signature

```
// *****************************************************************************
/* System Module Reinitialization Function Pointer

  Function:
    void (* SYS_MODULE_REINITIALIZE_ROUTINE) ( SYS_MODULE_OBJ object,
                                               const SYS_MODULE_INIT * const init)

  Summary:
    Pointer to a routine that reinitializes a module.

  Description:
    This data type is a pointer to a routine that reinitializes a system
    module (driver, library, or application).

  Preconditions:
    The low-level board initialization must have been completed and the
    module's initialization function must have been called before the system
```

```
    will call the reinitialization function for any module.

  Parameters:
    object          - Handle to the module instance

    init            - Pointer to the data structure containing any data
                      necessary to initialize the module. This pointer may
                      be null if no data is required and default initialization
                      is to be used.

  Returns:
    None.

  Remarks:
    This function uses the same initialization data structure as the Initialize
    function.
    This function can be used to change the power state of a module by passing
    in a different set of initial data values to reconfigure the module to a
    different power level.
    This function can also be used to refresh the hardware state as defined
    by the initialization data by passing in initial data values that match the
    previously given initial data values.  Thus, this function should guarantee
    that all hardware state is refreshed.
    This function can be called multiple times to reinitialize the module.
*/

typedef void (* SYS_MODULE_REINITIALIZE_ROUTINE) ( SYS_MODULE_OBJ object,
                                                   const SYS_MODULE_INIT * const init );
```

An implementation of a module's reinitialize function might look like the following example.

Example: Sample Module Reinitialize Function

```
void SAMPLE_Reinitialize ( SYS_MODULE_OBJ                object,
                           const SYS_MODULE_INIT * const init )
{
    SAMPLE_MODULE_DATA         *pObj  = (SAMPLE_MODULE_DATA *)object;
    SAMPLE_MODULE_INIT_DATA    *pInit = (SAMPLE_MODULE_INIT_DATA *)init;

    if (NULL == pInit)
    {
        pObj->status            = SYS_STATUS_READY;
        pObj->dataNewIsValid    = false;
    }
    else
    {
        pObj->status            = SYS_STATUS_BUSY;
        pObj->dataNew           = pInit->dataSome;
        pObj->dataNewIsValid    = true;
    }

    return;
}
```

The previous example initialize function for the sample module simulates resetting the hardware by resetting the `dataNew` value. If no `init` data is provided, it simply invalidates the current value by clearing the `dataNewIsValid` flag to false. However, if `init` data is provided, it stores the value and sets the library into a busy status that will be cleared once the library's tasks function has processed the data.

This may not be a particularly useful example, because it is possible to lose data given by the client if the `dataNew` value was currently valid. However, it does illustrate that the decision of what is preserved and what is not preserved when a module is reinitialized is a module-specific design decision that will depend on the type of module and how it operates safely.

Note: This function is optional. Any module that does not support power management or the ability to dynamically change initial settings while it is running does not need to implement this function.

Client Interface

Describes the client interface requirements for driver design.

Description

A driver's client interface is what is commonly thought of as its Application Program Interface (API). It is the interface through which any application

or other client interacts with the driver. It should be considered conceptually separate from the system interface (see the System Interface section). It represents the highest-level of abstraction at which an application or other module directly interacts with a specific peripheral device.

Middleware layers may implement protocol modules that simplify usage of specific types of peripherals (such as network interfaces or storage devices), but the driver interface is the highest level at which a client will interact directly with a peripheral. As such, a device driver should have a very simple usage model, especially for the most common uses of the device. The basic driver operations should allow an application to access the device, read and write data as if it were a simple file. In some cases, this will be all that is required. However, in most cases, device-type specific operations will also be required.

Driver-Client Usage Model

MPLAB Harmony drivers are required to provide a consistent "open/close" driver-client usage model.

Description

MPLAB Harmony device drivers follow a file system style device driver model, similar to that of POSIX-based operating systems, with a few primary differences. First, clients can directly access the drivers, through their own driver-specific functions instead of (or in addition to) indirectly through file system functions. Second, instead of having a single read/write data transfer model, there are multiple common data transfer models and some drivers may provide their own unique data transfer models. Third, instead of grouping all other input/output control operations into a single I/O control or IOCTL function, MPLAB Harmony drivers each provide a set of functions used to control the device.

In fact, the primary distinguishing feature of a MPLAB Harmony driver is that it uses a consistent driver-client usage model. This means that it has an open function and (optionally) a close function, and that before a client may use a device driver, it must call the driver's open function to obtain a driver handle. The handle is then passed into all other client-level interface functions as a parameter to identify the instance of the client that is calling and the instance of the driver (and, by implication the instance of the peripheral device) it is using, as shown in the following example.

Example: Driver-Client Usage Model

```
DRV_HANDLE  myUsart;
char       *message = “Hello World\n”;

/* Obtain an open handle to the USART driver. */
myUsart = DRV_USART_Open(MY_UART_INDEX, DRV_IO_INTENT_READWRITE);

/* Interact with the USART driver. */
DRV_USART_Write(myUsart, message, sizeof(message));

/* Continue using other USART driver client-interface functions as needed. */

/* Close USART driver if no longer needed. */
DRV_USART_Close(myUsart);
```

This example is somewhat simplified for the purpose of explanation. In normal usage, a client should test the value of the handle returned from the open function to ensure that it is not invalid (equal to DRV_HANDLE_INVALID). If the value of the handle returned is invalid, the caller should retry the open function later as it is possible that the driver is not yet ready for client usage. A client that requires the usage of a driver will normally prevent its state machine from advancing until a valid open handle has been obtained or it may eventually time-out and go into an error handling state.

The requirements of the open and close functions are described in the following section. The need for this model and the driver's internal usage of the handle are described in the Single Client vs. Multiple Client section.

Driver Client Interface Functions

Describes the format and naming convention for the interface functions of a driver client.

Description

The interface functions for the client of a driver follow a consistent format and naming convention, where `<module>` matches the module abbreviation for the driver or peripheral and `<operation>` is the name of the driver-specific operation to be performed.

Function	Description
`DRV_<module>_Open`	Open a link to the driver and start using it.
`DRV_<module>_<operation>`	Perform some driver-specific operation.
`DRV_<module>_Close`	Close link to the driver and stop using it.

The usage of the open and close functions is described previously in the Driver-Client Usage Model section and example implementations and key concepts are described in the Single Client vs. Multiple Client section. The details of the interface requirements (parameter data types, return values, etc.) are described in the following section. Requirements of data transfer operations are described in the Common Data Transfer Models section and general client interface requirements are described in the General Guidelines section.

Open

Describes the interface requirements for driver *open* functions.

Description

The purpose of a driver's *open* function is described in the Single Client vs. Multiple Client section. Driver *open* functions must meet the following interface requirements.

Function:

```
DRV_HANDLE DRV_<module>_Open ( const SYS_MODULE_INDEX index, const DRV_IO_INTENT intent )
```

Summary:

Opens a driver for use and provides an open-instance handle.

Descriptions:

This function opens a driver for use by a client module and provides an open-instance handle that must be provided to all of the other client-interface operations to identify the caller and the instance of the driver module.

Required or Optional:

Required.

Preconditions:

The driver module's initialize operation must have been called.

Parameters:

index	A zero-based index, identifying the instance of the driver to be opened. This value matches the index passed to the driver's initialize function.
intent	Flags parameter identifying the intended use of the driver: One of: • DRV_IO_INTENT_READ – Driver opened in read-only mode • DRV_IO_INTENT_WRITE – Driver opened in write-only mode • DRV_IO_INTENT_READWRITE – Driver opened in read-write mode One of: • DRV_IO_INTENT_NON_BLOCKING – Routines return immediately • DRV_IO_INTENT_BLOCKING – Routines return after operation is complete One of: • DRV_IO_INTENT_EXCLUSIVE – Will support only a single client. • DRV_IO_INTENT_SHARED – Can be used by multiple clients concurrently One flag from each group may be ORed together to fully define the intended use. However, the zero values and thus the default for each group is: DRV_IO_INTENT_READ, DRV_IO_INTENT_BLOCKING, and DRV_IO_INTENT_SHARED.

Returns:

If successful, the function returns a valid open-instance handle (an opaque value identifying both the caller and the driver instance). If an error occurs, the value returned is DRV_HANDLE_INVALID.

Example:

```
#define MY_I2C  0

handle = DRV_I2C_Open(MY_I2C, DRV_IO_BLOCKING|DRV_IO_RW|DRV_IO_NON_BUFFERED);
if (DRV_HANDLE_INVALID == handle)
{
    // Handle error
}
```

Blocking Behavior:

This function (and other client interface functions) may block on IO operations in an OS environment if DRV_IO_INTENT_BLOCKING (the default) is passed in the `intent` parameter. However, it (and other client interface functions) should never block waiting on I/O in a non-OS environment or it will block the entire system.

Remarks:

To support blocking behavior, the driver must be appropriately configured and built.

Drivers that are opened with a mode that is not supported must fail the open call by returning DRV_HANDLE_INVALID.

The default mode (if no flags are set, i.e., zero (0) is passed in the intent parameter), is blocking, read access only, and shared access.

Close

Describes the interface requirements for driver *close* functions.

Description

The purpose of a driver's *close* function is described in the Single Client vs. Multiple Client section. Driver *close* functions must meet the following interface requirements.

Function:

```
void DRV_<module>_Close ( DRV_HANDLE handle )
```

Summary:

Closes an opened-instance of a driver.

Descriptions:

This routine closes an opened-instance of a driver, invalidating the handle provided.

Required or Optional:

Optional – Not required if the driver is designed to never be closed.

Preconditions:

The driver's initialize function must have been called and the driver's open function must have returned a valid open-instance handle.

Parameters:

handle	A valid open-instance handle, returned from the driver's open function.

Returns:

None.

However, the driver's system-level status function will return SYS_STATUS_BUSY until the close operation has completed.

Example:

```
// Close the driver
DRV_I2C_Close(handle);
```

Blocking Behavior:

This function (and other client interface functions) may block on I/O operations in an OS environment if DRV_IO_INTENT_BLOCKING (the default) is passed in the `intent` parameter of the open function. However, it (and other client interface functions) should never block waiting on I/O in a non-OS environment or it will block the entire system.

Remarks:

Once this routine has been called, the handle provided will become invalid.

Common Data Transfer Models

Describes common data transfer usage models used by MPLAB Harmony drivers.

Description

MPLAB Harmony drivers for data transfer (or data source or data sink) peripherals normally follow one or more consistent data transfer usage models, described in this section. Each data transfer model has its own advantages and disadvantages, depending on the type of usage required and the execution environment. A single MPLAB Harmony driver may provide multiple data transfer models, depending upon the needs of the peripheral device. Usually, one particular model is the base or normal model, and others can be selected as optional features for broader compatibility.

Byte-by-Byte (Single Client)

Describes the byte-by-byte (single client) data transfer model.

Description

The byte-by-byte data transfer model provides a very simple mechanism for transferring data to and receiving data from a driver. It is very similar to the method commonly used by simple serial devices such as a USART that have transmitter and receiver FIFO buffers. This method transfers data one byte (or word) at a time until the driver's FIFO is either full (if transmitting data) or empty (if receiving data), as shown in the following examples.

Example: Reading Data Using the Byte-by-Byte Model

```
char buffer[MY_BUFFER_SIZE];
int  count;

for (count=0; count < MY_BUFFER_SIZE; count++)
```

```
{
    if (DRV_USART_ReceiverBufferIsEmpty(myUsart))
    {
        break;
    }
    else
    {
        buffer[count] = DRV_USART_ByteRead(myUsart);
    }
}
```

The previous example code assumes the USART driver has been successfully opened and the handle was stored in the `myUsart` variable. The `for` loop counts from 0 through MY_BUFFER_SIZE, unless the driver runs out of data. Each time through the loop, the code calls the DRV_USART_ReceiverBufferIsEmpty function to see if there is any data available. If there is no data available in the driver's receiver buffer FIFO (indicated by DRV_USART_ReceiverBufferIsEmpty returning true), the loop is aborted and the count is not incremented. If data is available in the driver's FIFO (indicated by DRV_USART_ReceiverBufferIsEmpty returning false), the example calls the driver's DRV_USART_ByteRead function and stores the data returned into the current position in the buffer, indexed by the count variable. Then, the `for` loop increments `count` before checking the loop exit condition and potentially starting over. When this loop exits, either because `count` reached MY_BUFFER_SIZE or because the driver had no more data available, the buffer contains count bytes of data received from the driver.

A very similar method is used to transmit data to the driver.

Example: Writing Data Using the Byte-by-Byte Model

```
char *buffer = "Hello World\n";
int   count;

for (count=0; count < strlen(buffer); count++)
{
    if (DRV_USART_TransmitBufferIsFull(myUsart))
    {
        break;
    }
    else
    {
        DRV_USART_ByteWrite(myUsart, buffer[count]);
    }
}
```

Again, it is assumed that the USART driver was previously opened and a valid `myUsart` handle obtained. The `for` loop counts from 0 through `strlen(buffer)`, unless it fills the driver's transmitter buffer FIFO first. Each time through the loop, it checks to see if the driver's transmitter FIFO is full. If it is (as indicated by DRV_USART_TransmitBufferIsFull returning true), it aborts the loop and does not increment the `count` variable. If the driver's transmitter FIFO buffer is not full, it will then call DRV_USART_ByteWrite to send the byte of data in `buffer` currently indexed by the `count` variable. It will then increment the `count` variable, check the loop exit condition and potentially start over. When the loop exits, `count` bytes of data from `buffer` have been sent from `buffer` to the driver's transmitter FIFO.

This data transfer model has the advantage that it is usually very lightweight and simple to implement, resulting in very little RAM and Flash required by the driver. However, this data transfer model is only safe for usage with single client drivers or with multiple client drivers that have been successfully opened in DRV_IO_INTENT_EXCLUSIVE mode. It is not safe for use with multiple clients or in a preemptive multi-tasking environment because it requires calling multiple functions to completely read or write a buffer of data. Refer to the Interrupt and Thread Safety section for an explanation on the types of issues that could occur in that environment.

Also, this data transfer model is not particularly easy to use, as it requires the caller to manage and adjust its current buffer pointer each time the loop exits. A synchronous file system style read/write data transfer model, described in the next section, may be simpler and safer from a caller's point of view, especially when used in a RTOS environment.

File System Read/Write

Describes the "file system style" read/write data transfer model.

Description

The synchronous, file system style, read/write data transfer model is intended to be similar to the POSIX read and write (and fread and fwrite) operations, as shown in the following example.

Example: Reading Data Using the File System Style Model

```
char    buffer[MY_BUFFER_SIZE];
size_t  count;

count = DRV_USART_Read(myUsart, buffer, MY_BUFFER_SIZE);
```

In the previous data read example, the single DRV_USART_Read function is called to transfer the entire contents of the `buffer` array. In a RTOS or file system model environment (described as follows), this function should not return until all data has been transferred or some form of error or time-out occurs. The success of this operation will be indicated by the return value stored in the `count` variable. If it is equal to the value of MY_BUFFER_SIZE, the operation completed successfully.

Example: Writing Data Using the File System Style Model

```
char    buffer = “Hello World\n”;
size_t  count;

count = DRV_USART_Write(myUsart, buffer, strlen(buffer));
```

In the previous data write example, the single DRV_USART_Write function is called to transfer the entire contents of the buffer string. Like the data reading example, in a RTOS or file system model environment, this function should not return until all data has been transferred or some form of error or time-out occurs. The success of this operation is indicated by the return value stored in the `count` variable. If its value is equal to `strlen(buffer)`, the operation completed successfully.

For simple data stream peripherals, such as UART, SPI, I2S, etc., this is normally the most basic data transfer model that a driver should support. It has the advantage that it provides a simple single-function interface and, depending on the configuration and execution environment, it manages advancing through the caller's buffer automatically and does not return until all data has been transferred.

If an error occurs, most drivers will return (`(size_t)-1`) (equivalent to a maximum unsigned integer value) from these functions. Or, in a non-blocking environment, they will only transfer the amount of data that can be buffered by the driver or hardware. So, this data transfer model has the disadvantage that the caller should always check the return count and may need to call a driver-specific status function to identify if an error has occurred. If no error has occurred, the caller may need to call the function again (potentially several times) to complete the desired transfer. So, this model is most suitable for configurations that provide sufficient buffering or that support a RTOS or where blocking behavior is acceptable, such as a file system model environment where operation of other modules is not required.

Buffer Queuing

Describes the buffer queuing data transfer model.

Description

The buffer queuing data transfer model is an asynchronous transfer mode. It is always non-blocking, so it allows the client to call the buffer add function multiple times, without waiting for each transfer to complete. This allows the caller to queue up more than one buffer at a time, potentially before the first buffer has finished (depending on buffer size and data transfer speed). The following examples show how this is done.

Example: Reading Data Using the Buffer Queuing Model

```
DRV_USART_BUFFER_HANDLE handle1;
DRV_USART_BUFFER_HANDLE handle2;
char buffer1[BUFFER_1_SIZE];
char buffer2[BUFFER_2_SIZE];

DRV_USART_BufferAddRead(myUsart, &handle1, buffer1, BUFFER_1_SIZE);
DRV_USART_BufferAddRead(myUsart, &handle2, buffer2, BUFFER_2_SIZE);
```

The previous example shows the caller queuing up two buffers to read data from the USART driver. When the first call to DRV_USART_BufferAddRead occurs, the driver will place the address and size of `buffer1` into its queue. Then, it will store a unique handle, identifying the data transfer request, into the `handle1` variable and begin copying data into the buffer (unless the driver was already busy placing data into a different buffer). Then the call will return. When the second call to DRV_USART_BufferAddRead occurs, the process repeats. If the driver has not yet finished filling `buffer1`, it will add the address and size of `buffer2` into its queue, provide a handle to it, and return.

Example: Writing Data Using the Buffer Queuing Model

```
DRV_USART_BUFFER_HANDLE handle1;
DRV_USART_BUFFER_HANDLE handle2;
char                    buffer1 = “Hello World\n”;
char                    buffer2 = “Hello Again\n”;

DRV_USART_BufferAddWrite(myUsart, &handle1, buffer1, strlen(buffer1));
DRV_USART_BufferAddWrite(myUsart, &handle2, buffer2, strlen(buffer2));
```

Similarly, the previous example shows the caller queuing up two buffers to write data to the USART driver. When the first call to DRV_USART_BufferAddWrite occurs, the driver will place the address and size of `buffer1` into its queue. Then, it will store a unique handle, identifying the data transfer request, into the `handle1` variable and begin copying data from the buffer (unless the driver was already busy copying data from a different buffer). Then, the call will return. When the second call to DRV_USART_BufferAddWrite occurs, the process repeats. If the driver has not yet finished copying all data from `buffer1`, it will add the address and size of `buffer2` into its queue, provide a handle to it, and return.

A driver that supports the buffer queuing model usually provides either a callback notification function or a status function (or both) so that a client can determine when the data transfer request has completed, as shown by the following examples.

Example: Using Callback Notification

```
DRV_USART_BUFFER_HANDLE handle1;
char                    buffer1 = “Hello World\n”;

DRV_USART_BufferEventHandlerSet(myUsart, MyUsartCallback, NULL);

DRV_USART_BufferAddWrite(myUsart, &handle1, buffer1, strlen(buffer1));
```

In the previous example, the client registers a callback function called MyUsartCallback with the USART driver before calling the DRV_USART_BufferAddWrite function to transmit of the contents of buffer1 on the USART. When the driver has completely transferred all data

from buffer1 by the USART, it will call the MyUsartCallback function, as shown in the following example.

Example: Callback Implementation

```
void MyUsartCallback ( DRV_USART_BUFFER_EVENT event,
                       DRV_USART_BUFFER_HANDLE bufferHandle,
                       uintptr_t context )
{
    switch ( event )
    {
        case DRV_USART_BUFFER_EVENT_COMPLETE:
        {
            if (bufferHandle == handle1)
            {
                 /* buffer1 data transfer complete */
            }
            break;
        }

        /* Handle other possible transfer events. */
    }
}
```

The MyUsartCallback function (described previously) is an example of how the client might handle the callback from the USART driver. In this example, the USART driver passes the DRV_USART_BUFFER_EVENT_COMPLETE ID in the `event` parameter to indicate that the data from `buffer1` has been completely transmitted by the USART. The value of the `bufferHandle` parameter will match the value assigned into the `handle1` parameter of the DRV_USART_BufferAddWrite function call so that the client can verify which buffer transfer has completed. (Recall that the driver can queue multiple transfers. It may, in fact, have multiple read and multiple write transfers queued at the same time.) For now, ignore the `context` parameter, which is explained in the Single Client vs. Multiple Client section.

The callback mechanism allows a client to synchronize to the timing of when the data transfers have completed. However, this adds some additional complexity to the interface and has a few potential subtle concerns. For one, a very short transfer may actually complete and the callback may occur before the DRV_USART_BufferAddWrite function actually returns. This is a potential *race* condition and it is why the transfer handle value is given as an output parameter and not as a return value. It allows the driver to ensure that the client has a valid handle before it starts the transfer so that the handle value is valid when the callback occurs. Also, in an interrupt-configuration, the callback may occur in an ISR context. Therefore, the client should be careful to not call anything that may block. In particular, the client should be careful to not call any of the driver's own API functions as they are not normally designed to be called from within the driver's own ISR.

Alternately, client may not require hard real time notification of the completion of the transfer of the buffer data. It may just need to know when it can safely reuse the buffer. If that is the case, the driver may also provide an interface function that will allow the client to poll the driver at its convenience to determine if the buffer has completed, as shown by the following example.

Example: Checking for Buffer Status

```
DRV_USART_TRANSFER_STATUS status;

status = DRV_USART_BufferStatusGet(myUsart, handle1);
if (status == DRV_USART_BUFFER_COMPLETE)
{
    /* Buffer Transfer Has Completed */
}
```

The previous example shows how the client can call the driver, passing in the `handle1` value given by the DRV_USART_BufferAddWrite function, to poll at its leisure to find out when the driver has completed using `buffer1` and has transferred all the data it contained. Most drivers will provide both the buffer status function and the callback mechanisms so that the client can choose the most appropriate one.

Using the buffer queuing model, a client can keep a driver at 100% throughput utilization by queuing up at least two buffers. When the first buffer completes, the client has until the second buffer completes to queue another buffer. Using either the byte-by-byte or file system style models requires the client to respond within the time it takes to fill or empty the driver's built-in FIFO buffer to keep a continuous stream of data transfers. This can be a very short period, potentially as short as the time it takes to transfer a single byte on the peripheral. However, the buffer queuing method is somewhat more complex to use and usually requires more RAM and Flash to implement. Therefore, a driver may not offer it or may only offer it as an optional feature, unless the normal operation of the peripheral requires continuous, uninterrupted data transfer.

General Guidelines

Provides general guidelines for device driver client interface design.

Description

In addition to the open, close, and data transfer functions, most device drivers will require a number of device type-specific interface functions. What these functions do will depend on exactly what type of device is being controlled. It is generally best to provide a solution for the most basic usage of a given type of peripheral first. It is easy to add unique optional capabilities later, but it is hard to fix a bad interface without breaking existing client code. Keep in mind that the driver should manage the state of the device so the client does not have to. Intermediate steps that are part of normal operation should be hidden by the interface as much as possible. The following list provides more general guidelines that may help when defining a driver's interface.

1. A driver that supports only a single type of peripheral is easier to maintain and more flexible to use. Only integrate drivers for different types of peripherals together when the merged driver serves another purpose and cannot expose any of the underlying functionality. For example, if the merged driver is a touch screen driver that fully utilizes the underlying Timer and ADC resources, leaving neither available for other uses. Otherwise, the merged driver will need to expose multiple driver interfaces so as to be transparent to clients.
2. Define the interface based on the client's point of view of an idealized and abstracted version of the peripheral, not on a detailed understanding of the device. Any details that may be different from one implementation of a device to another should be hidden from the client. Features that exist on one piece of hardware and not on another that need to be exposed to the interface can be added or removed as configuration selections. Operations related to features that do not exist on one part can be grouped together and removed from the interface when not supported.
3. Although it may be appropriate for a driver to maintain its own buffer(s) in which to collect data, it is generally preferable to use the caller's buffer. This places decisions about buffer size and allocation with the caller, who has better knowledge of exactly how the data is to be used. Ownership of the buffer is passed to the driver when a data transfer function is called and released from the driver either when the call returns or when the transfer completion status has been given to the client.
4. Many driver functions will perform operations that require some time to complete (usually waiting for some status to change). These functions should report status to indicate whether an operation is complete or not and provide a handle or identifier with which the client can identify the operation later when it completes.
5. An interface should be appropriately sized (i.e., it should not contain too many operations or too few operations). A smaller interface is generally better, if it can support all of the required features. However, performance, ease of use, and compatibility are more important. Do not sacrifice any of these considerations to eliminate interface functions.
6. Try to use data types that can be easily ported to an appropriate size (8-, 16-, 32-bit) if the data value range or processor changes for parameters or return types, unless the usage model of the driver requires a specific bit width. When a specific data size is required, use the C99 data types defined in `stdint.h`, `stdbool.h`, and `stddef.h`.
7. If DMA is supported (for peripherals that would benefit from it), it should be hidden behind the same data transfer operations used when it is not available and either enabled as a build-time configuration option or enabled and disabled by a setup function at run-time, as appropriate.
8. Drivers must use system services for memory allocation, interrupt control, system clocks and timers, power management, physical/virtual address conversion, etc. They must also use the OSAL for thread safety and synchronization. Generally, these facts should be hidden from the client interface. (Refer to the System Services and OSAL help for information on what services are available and how to use them.)
9. Interrupt specifics, such as the interrupt ID and vector numbers, should be abstracted away as build-time configuration options (for static implementations) or instance-specific initialization options (for dynamic implementations) unless there is a need to change them dynamically at run-time.

As always, it is best to follow generally accepted programming practices and a consistent programming style. Remember that source code frequently outlives its original purpose and well-designed and easily readable and maintainable code will be a joy to work with long after the original project is completed.

Interrupt and Thread Safety

Describes key concepts and concerns related to safe driver operation when using interrupts or RTOS threads.

Description

MPLAB Harmony allows libraries to be configured for any one of several different environments. This is accomplished by designing libraries that comprehend the restrictions of all supported environments and that are configurable for each.

The basic concept is to always consider the context in which a function can be called. Since a driver has both system and client interface functions, the developer must consider both of these interfaces. In particular a driver's tasks function(s) may be called from any one of the following three contexts (but only from one of them in any given configuration).

- The main polling loop
- An ISR, if an appropriate one is available
- A loop in a RTOS thread or task function

It is also important to keep in mind that a driver's client interface functions are normally called from an application's tasks function (or some other client module's tasks function), which is called from a potentially different one of the previous three contexts. And, if a driver supports callback functions, the developer must consider from what context it will call the client's function (again, from a potentially different one of the previous three contexts).

Supporting (and testing) the capability to configure a driver for any of these environments, while potentially challenging, helps to produce robust and flexible drivers that can be used and reused in the widest range applications and that are easily portable to future projects. It also helps to ensure that they are reliable in the project for which they were originally developed by helping to find corner cases and ensure reliable operation.

Of the three contexts listed previously, most developers consider the polled environment is first when designing and implementing libraries. This environment is the easiest to understand, but it is also the most restrictive and limiting in terms of system capabilities and real-time responsiveness. So, most systems will utilize at least some interrupt-driven code or use a RTOS, requiring the developer to consider the issues described in this section.

Atomicity

Describes atomic code sequences and data types at a fundamental level.

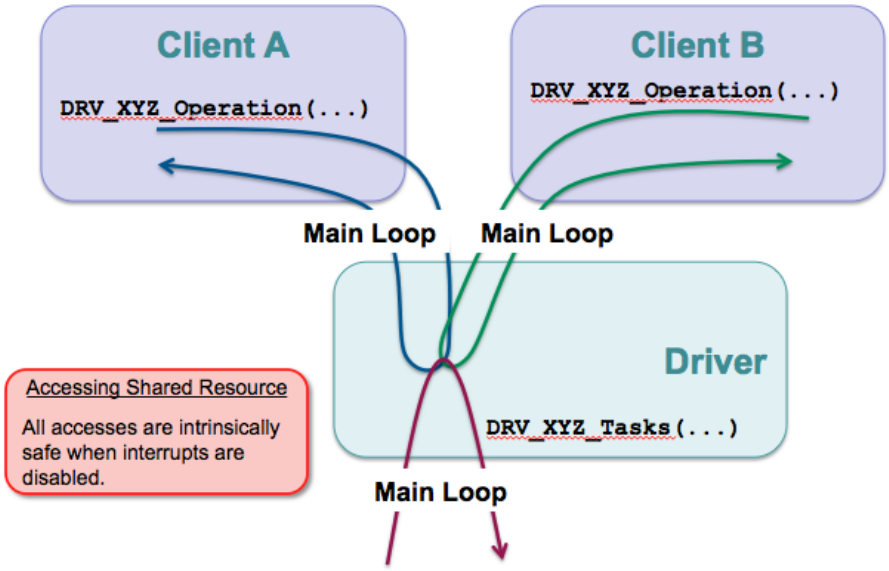
Description

The English word *atomic* derives from a Greek word meaning *indivisible*. A sequence of instructions that is indivisible once started and cannot be

interrupted until it has completed is called atomic. A data item is considered atomic if it cannot be subdivided as it is read or written. These are critical concepts for all software, including real-time embedded systems, because interrupts and context switches can occur at particularly inopportune times and potentially cause data corruption or incorrect behavior or even complete system failure. Such possibilities must be prevented to guarantee correct and robust functioning of the system.

When a processor is reset, interrupts are disabled and the processor executes instructions one after the next and will continue to do so until powered off or reset. Therefore, all code is effectively atomic in a polled environment where interrupts are globally disabled. In such a strictly polled configuration, conflicts due to non-atomic accesses to resources shared between clients and drivers do not occur.

Polled Configuration



Even though different clients may call the same driver function and access, the same resources that are also accessed by the driver's tasks function, all functions are called from the context of the main system loop and only one such access will occur at a time and it will complete without interruption.

However, once interrupts are enabled, there is a very limited set of situations where atomicity can be guaranteed. The execution of a single instruction is atomic. When an interrupt occurs, an instruction will either execute completely or it will not be started. Interrupts are synchronized to the instruction flow and they effectively occur between the instructions. Or, more accurately, the CPU can only respond to an interrupt after completing the instruction it is currently decoding and executing. That may seem to be obvious, but it is the basis for all atomicity of both data items and sequences of instructions in a computing system.

Atomicity is also guaranteed when reading or writing a single data word with a single instruction. A data word contains the same number of bits as the data bus width. In a correctly functioning system, a data value that is the width of the processor's data bus (or less if half word or smaller values are supported, as they are on PIC32 microcontrollers) will be read or written in its entirety. The data word will never be partially read or partially written.

However, if a variable, data structure, or array is larger than a single data word (for example, a 64-bit value on a 32-bit processor), it will take more than one instruction to read or write the entire value and interrupts could occur in between those instructions. If that happens, it is possible that part of the data value could be changed by the interrupt and may not be consistent with the part that was read before the interrupt (or written after the interrupt) when the sequence of instructions completes. Such data types should not be considered atomic. Instruction sequences that access them should be protected to guarantee that there is no possibility of corruption.

Even if a data type is atomic (only one word or less in size), it may still be necessary to protect a code sequence that performs a read-modify-write operation on it because almost all read-modify-write operations require execution of multiple instructions, which could be interrupted. This concern applies to both data stored in memory (in variables, structures, arrays, and buffers) as well as registers (or SFRs). If non-atomic accesses are made to these resources from both an ISR and the main loop or from more than a single thread context when using a RTOS, that resource is considered shared and access to it must be made atomic by guarding or protecting it as described in the following sections.

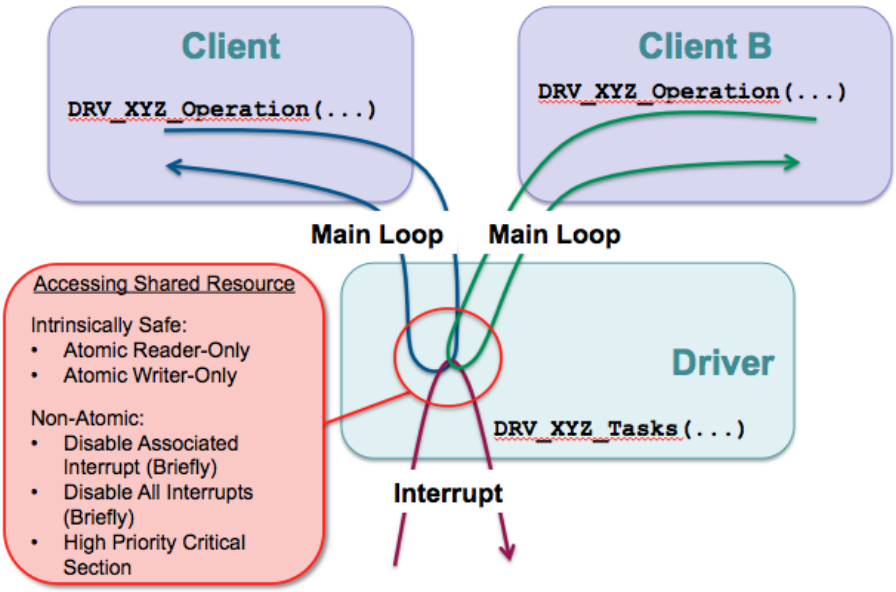
Interrupt Safety

Describes how to guard against the potential conflicts that interrupts can cause.

Description

If a sequence of code must be atomic (indivisible and uninterruptible), the relevant interrupts must be disabled before entering the sequence. In MPLAB Harmony, interrupts can be disabled globally by using the interrupt system service or by using a high-priority mutex. However, MPLAB Harmony libraries are modular and by convention they respect the abstractions of other libraries, never attempting to directly access their internal resources. Because of this convention, the only code in the system that should ever attempt to access the internal resources owned by a driver is the driver code itself, as shown in the following diagram.

Interrupt-Driven (No RTOS)



This means that it is not necessary to globally disable interrupts in most cases to guarantee correct and reliable operation of a MPLAB Harmony driver. It is usually sufficient for a driver to temporarily mask just the interrupt(s) of the peripheral it owns while performing non-atomic accesses to its own data structures or peripheral hardware. Doing so will prevent the driver's own interrupt-driven tasks function(s) from potentially corrupting data that is also accessed by the driver's interface function(s).

This can be done using the interrupt system service and it is more efficient than globally disabling all interrupts because it allows higher priority interrupts (which do not affect the driver in question) to occur, protecting their response time latency. This can be done using the Interrupt System Service, as shown in the following example.

Example: Temporarily Disabling an Interrupt Source

```
#define MY_INTERRUPT_SOURCE  INT_SOURCE_TIMER_2

bool enabled;

enabled = SYS_INT_SourceDisable(MY_INTERRUPT_SOURCE);

/* Access resource shared with interrupt-driven tasks function. */

if (enabled)
{
    SYS_INT_SourceEnable(MY_INTERRUPT_SOURCE);
}
```

This method is safe to use, even if the driver in question is not running in an interrupt-driven mode because it only re-enables the interrupt source if it was enabled before the sequence was entered. However, when a driver is not configured for interrupt-driven operation, it must not enable its own interrupt and the code that is necessary to disable the interrupt and restore its previous state is not necessary and could be removed to save code space. Fortunately, this can be accomplished fairly easily by abstracting the interrupt management code behind functions that switch implementations depending upon the configuration of the driver, as shown in the following example.

Example: Interrupt Management Functions

```
#if (SAMPLE_MODULE_INTERRUPT_MODE == true)

    #define _SAMPLE_InterruptDisable(s)    SYS_INT_SourceDisable(s)

#else

    #define _SAMPLE_InterruptDisable(s)    false

#endif


#if (SAMPLE_MODULE_INTERRUPT_MODE == true)

    static inline void _SAMPLE_InterruptRestore ( INT_SOURCE source, bool enabled )
    {
        if (enabled)
        {
            SYS_INT_SourceEnable(source);
        }
```

```
    }

#else

    #define _SAMPLE_InterruptRestore(s,e)

#endif
```

This method effectively compiles away the interrupt management code, adding little or no object code when used in a polled configuration (when SAMPLE_MODULE_INTERRUPT_MODE is false). But, when used in an interrupt-driven configuration (when SAMPLE_MODULE_INTERRUPT_MODE is true), these functions use the system interrupt service to manage the driver's interrupt source(s). These functions can then be used to guard non-atomic accesses by the driver's interface functions to resources that are shared with the driver's ISR, as shown in the following code example.

Example: Interrupt Management

```
bool SAMPLE_DataGet ( const SYS_MODULE_INDEX index, int *data )
{
    SAMPLE_MODULE_DATA *pObj;
    bool                intState;
    bool                result = false;

    pObj = (SAMPLE_MODULE_DATA *)&gObj[index];

    // Guard against interrupts
    intState = _SAMPLE_InterruptDisable(pObj->interrupt);

    if (pObj->dataProcessedIsValid)
    {
        // Provide data
        *data = pObj->dataProcessed;
        pObj->dataProcessedIsValid = false;
        result = true;
    }

    // Restore interrupt state.
    _SAMPLE_InterruptRestore(pObj->interrupt, intState);

    return result;
}
```

In this code example, the driver interface function SAMPLE_DataGet calls _SAMPLE_InterruptDisable before checking a flag and potentially updating an internal data structure, which is a non-atomic process that will take multiple instructions. When SAMPLE_MODULE_INTERRUPT_MODE is true, the _SAMPLE_InterruptDisable function will call the SYS_INT_SourceDisable system service function to atomically disable the interrupt source and capture its current state (whether it was enabled or disabled before being disabled). Once it is done accessing the data structure, it calls the _SAMPLE_InterruptRestore function to restore the interrupt to its previous state. This ensures that the ISR cannot fire and call the tasks function to modify this data until the interface function has finished with the data. However, if SAMPLE_MODULE_INTERRUPT_MODE is false, the _SAMPLE_InterruptDisable function will be replaced with a constant false value (to avoid a syntax error in the assignment of the function return value) and the _SAMPLE_InterruptRestore function will be completely removed.

This method works to ensure safe access to shared resources without disabling interrupts globally when using a bare-metal configuration. Of course, if a section of code must be truly atomic (uninterruptible), interrupts can be globally disabled for a short period of time using either the Interrupt System Service functions or a high-priority critical section.

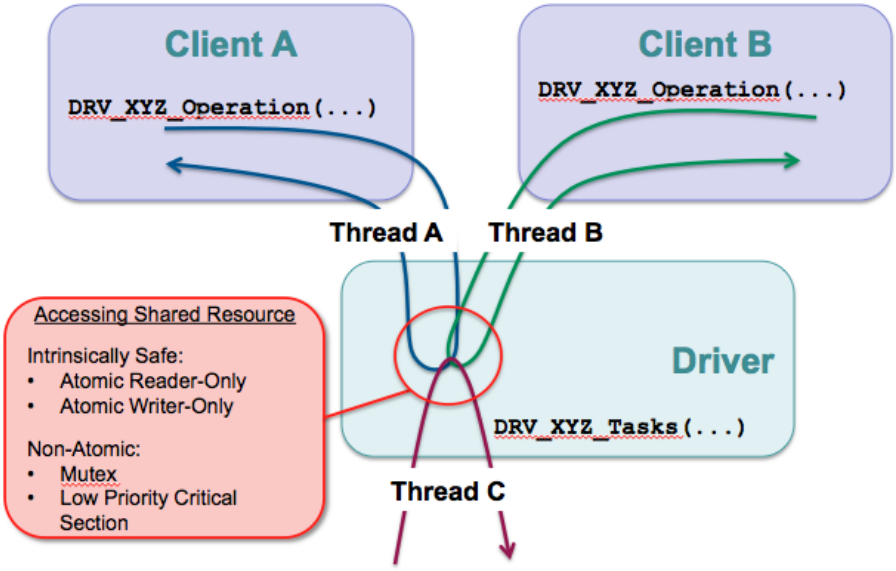
RTOS Thread Safety

Describes how to protect accesses to shared resources and critical sections of code from corruption by simultaneous access by multiple RTOS threads (tasks) and interrupts.

Description

When utilizing a RTOS, it is possible that a driver and its clients may each run in its own RTOS thread. If the RTOS is preemptive, it is possible that the scheduler may interrupt any of these threads at any time and switch to another. If the other thread happens to also non-atomically access the same shared resource or execute the same critical section of code (as shown by the following diagram), that section of code must be guarded and made atomic using the methods provided by the MPLAB Harmony Operating System Abstraction Layer (OSAL).

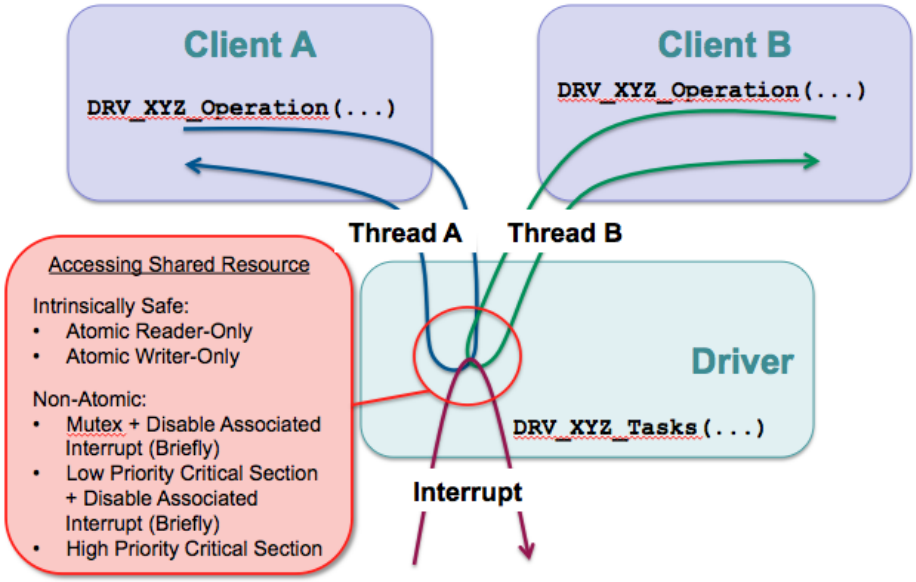
RTOS Polled



In a RTOS configuration, where the MPLAB Harmony driver and all of its clients are running strictly polled in their own threads, a mutex or low-priority critical section is sufficient to protect non-atomic accesses to shared resources or critical sections of code. Both of these mechanisms will instruct the RTOS scheduler to only allow a single thread to access to the resource or execute the critical code sequence at a time and will block all other threads, making them idle until the first thread has released the mutex or exited the critical section.

However, MPLAB Harmony drivers normally run most efficiently when interrupt driven. So, even when utilizing a RTOS, it is common for a driver's tasks function to be called from an interrupt context, as shown in the following diagram.

RTOS Interrupt-Driven



In this situation, a mutex or a low-priority critical section can still be used to guard against simultaneous access to shared resources by different threads (for example client A's thread and client B's thread, as shown previously). However, neither will prevent an ISR from accessing the shared resource or critical code section right in the middle of the thread's attempt to access it. So, if either of these methods is used, they must be augmented by also temporarily disabling (masking) the associated interrupt source, exactly the same way it is done in a bare metal environment as described in the Interrupt Safety topic. It is also possible to simply disable interrupts globally and prevent context switches by using a high-priority critical section. But, this is a brute force solution that is not recommended unless the timing of the code sequence is absolutely critical, and then only for very brief periods (see the Blocking Guidelines section for recommendations on what is an acceptably brief period).

The following example uses the _SAMPLE_InterruptDisable and _SAMPLE_InterruptRestore functions from the previous (Interrupt Safety) topic in conjunction with the OSAL mutex functions to effectively guard non-atomic accesses to shared resources from within client interface function code.

Example: Guarding Shared Resources

```
result = false;

if (OSAL_MUTEX_Lock(&pObj->mutex, SAMPLE_MODULE_TIMEOUT) == OSAL_RESULT_TRUE)
{
    /* Guard against interrupts */
    intState = _SAMPLE_InterruptDisable(pObj->interrupt);
```

```
    /* Check for storage space */
    if (!pObj->dataNewIsValid)
    {
        /* Store data */
        pObj->dataNew           = data;
        pObj->dataNewIsValid    = true;
        pObj->status            = SYS_STATUS_BUSY;
        result                  = true;
    }

    /* Restore interrupt state and unlock module object. */
    _SAMPLE_InterruptRestore(pObj->interrupt, intState);
    OSAL_MUTEX_Unlock(&pObj->mutex);
}
```

In the previous code example, the OSAL_MUTEX_lock function is called first to lock the mutex. When using a RTOS, this is the point at which it will block any subsequent thread entering the sequence before the first exits. So, the locked mutex protects the current thread against accesses by multiple clients. However, the non-atomic access sequence still needs to be guarded against ill-timed interrupts by calling the local _SAMPLE_InterruptDisable function and passing in the appropriate interrupt flag ID. When interrupt-driven, this will disable the interrupt and (atomically) capture the previous status of the interrupt sourced passed in. Then, once finished accessing the shared resource, this example calls the inverse functions (_SAMPLE_InterruptRestore and OSAL_MUTEX_Unlock) in reverse order to restore the previous state, unlock the mutex and continue safely.

In a bare metal configuration, if the mutex is already locked, the OSAL_MUTEX_lock function will return OSAL_RESULT_FALSE and the if will fail the result variable will stay false, allowing the interface function that contains this code to provide a negative result to the caller. If the mutex is not locked, it will become locked and the `if` clause will be taken, emulating the behavior of a RTOS. Of course, as described in the Interrupt Safety section, the interrupt enable and disable functions also change implementation depending on whether the driver is configured for interrupt-driven or polling operation. So, this method works for client interface functions in all possible execution environments (bare metal polled, bare metal interrupt-driven, RTOS polled and RTOS interrupt-driven). It has full flexibility and configurability when implementing client-interface functions. However, a driver's tasks function requires a slightly different set of behavior, as described in the following paragraph.

When interrupt-driven, the driver's tasks function is called from within the ISR. In an ISR, the associated interrupt source is already masked and it will be automatically unmasked when the ISR returns. This means that the driver's tasks function does not need to disable and restore its own interrupt source, only the driver's other functions need to do that. However, driver's tasks function can only call potentially blocking OSAL functions when in a polled configuration (RTOS-based or bare metal). When built in an interrupt-driven configuration, any interrupt-driven tasks functions must not call OSAL functions that might block. This means that any OSAL functions called from a tasks function must switch implementations and compile away to nothing when the driver is built in an interrupt-driven configuration. The following example shows a simple way to do this based on the interrupt configuration option.

Example: OSAL Use in Interrupt Tasks Functions

```
#if (SAMPLE_MODULE_INTERRUPT_MODE == false)

    static inline bool _SAMPLE_TasksMutexLock (SAMPLE_MODULE_DATA *pObj)
    {
        if (OSAL_MUTEX_Lock(&pObj->mutex, SAMPLE_MODULE_TIMEOUT) == OSAL_RESULT_TRUE)
        {
            return true;
        }

        return false;
    }

#else

    #define _SAMPLE_TasksMutexLock(p)      true

#endif


#if (SAMPLE_MODULE_INTERRUPT_MODE == false)

    static inline void _SAMPLE_TasksMutexUnlock ( SAMPLE_MODULE_DATA *pObj )
    {
        OSAL_MUTEX_Unlock(&pObj->mutex);
    }

#else

    #define _SAMPLE_TasksMutexUnlock(p)

#endif
```

These functions effectively compile away when used in an interrupt-driven configuration (when SAMPLE_MODULE_INTERRUPT_MODE is true), adding little or no object code. But, when used in an interrupt-driven configuration (when SAMPLE_MODULE_INTERRUPT_MODE is false), these functions translate to the desired OSAL mutex functions and can be used to guard non-atomic accesses in the tasks function, as shown in the following example.

Note: The OSAL functions themselves map to the appropriate RTOS-specific or bare metal implementation, so no additional steps need be taken to ensure flexibility to use or not use a RTOS. Only the interrupt and non-interrupt behavior needs to be mapped by functions defined in the driver's source code.

Example: Guarding Shared Resources in Interrupt-Driven Tasks

```
void SAMPLE_Tasks( SYS_MODULE_OBJ object )
{
    SAMPLE_MODULE_DATA     *pObj  = (SAMPLE_MODULE_DATA *)object;

    SYS_ASSERT(_ObjectIsValid(object), "Invalid object handle");

    if (!_SAMPLE_TasksMutexLock(pObj))
    {
        return;
    }

    // Process data when ready.
    if (pObj->dataNewIsValid && !pObj->dataProcessedIsValid)
    {
        pObj->dataProcessed         = pObj->dataNew;
        pObj->dataNewIsValid        = false;
        pObj->dataProcessedIsValid  = true;
        pObj->status                = SYS_STATUS_READY;
    }

    _SAMPLE_TasksMutexUnlock(pObj);

    return;
}
```

This previous example shows how to use the mapped mutex lock functions within an ISR-driven tasks function. Note that if the _SAMPLE_TasksMutexLock function returns false, it is potentially an error condition and the tasks function will be unable to perform its task because the resources are unavailable. So, be sure to perform appropriate error recovery or management if necessary.

A task function that manages an interrupt is, by nature, interrupt safe because it is either called from the appropriate ISR or it is polled and the associated interrupt source is disabled. And, the previous method can be used to also make them thread safe. So, it provides all four combinations of ISR and thread safety and can be used to make fully configurable drivers for all four combinations of interrupt versus polled and RTOS versus non-RTOS configurations.

Note: Tasks functions that do not manage an interrupt (such as tasks functions for applications or second-level tasks function) and that can only be polled should be treated like client interface functions. Non-atomic accesses to resources that can be accessed from other thread contexts and/or interrupts must be protected using the same methods as used by client interface functions.

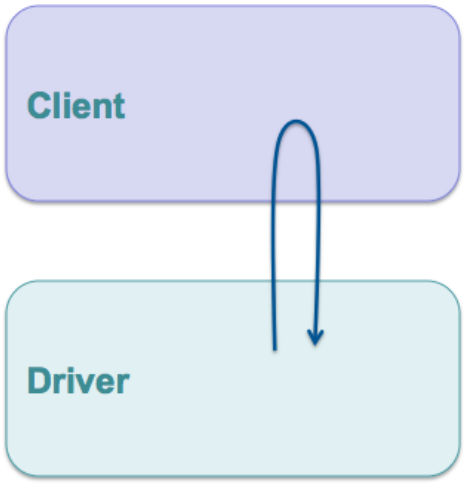
Callback Functions

Describes callback functions and the potential interrupt and thread-safety concerns they may cause.

Description

Normally, a client calls the driver's interface functions to interact with it. But, a callback function is a client function called by the driver back to the client, instead of the other way around, as shown in the following diagram.

Callback Functions



Callback functions are usually dynamically registered with a driver (or other server library). To do this, the driver provides an interface function that the client can call and pass in a pointer to the function it wants the driver to call back (the MyCallback function in the following example).

Example: Registering a Callback Function

```
#define PERIOD    1000
#define NO_REPEAT false

if (DRV_TMR_AlarmRegister(pObj->tmrHandle, PERIOD, NO_REPEAT, pObj, MyCallback))
{
    /* Successfully registered “MyCallback” function */
}
```

A callback is different from functions that are statically linked and called from the driver by name. Statically linked functions are considered dependencies. Dependencies are requirements of the driver. It will not build without an implementation of the dependency. Dependencies should be limited to only the things that the driver needs to use to do its job. They should not be part of the client interface. If they are, they force the driver to be static and single client. Dynamically registering a callback function (by a function pointer) instead of statically linking it to the driver allows the driver to be static or dynamic, or single client or multiple client.

Note: Sometimes two libraries may be mutually dependent on each other or a library may have dependencies upon functions defined in configuration files that are implemented as part of the system configuration. But, a library should not be dependent upon a client.

The callback function whose address is passed to the callback registration function probably needs to do something when it is called back. Likely it will need to set a flag (or semaphore) or capture some status value (see the following example).

Example: Callback Function

```
void MyCallback ( uintptr_t context, uint32_t alarmCount )
{
    MY_DATA_OBJECT *pObj = (MY_DATA_OBJECT *)context;

    pObj->alarmCount += alarmCount;
}
```

Caution should be taken when designing the usage model of a driver callback. This is because in an interrupt-driven configuration the callback function might be called from the driver's ISR or in a RTOS configuration the callback might be called from a different thread context. This places additional complexity on the client and on the driver, especially if the client then needs to call the driver's interface functions from the callback function.

Also, it is important to carefully document the context in which a callback function can be called because a client cannot disable an interrupt owned by a driver, or any other module, because each module manages its own interrupts. It would be a violation of the driver's abstraction. The client should not know what interrupts the driver uses or if it uses any interrupts at all. In this situation, a client may not be able to make non-atomic accesses to its own internal data structures if they are accessed by the callback and its own state machine or interface functions without globally disabling interrupts. For example, the `pObj->alarmCount += alarmCount` line in the previous example is a non-atomic (read-modify-write) access so the client using this driver would need to stop the timer callbacks from happening before it attempted to perform a non-atomic access to the `alarmCount` variable in its MY_DATA_OBJECT structure from any of its other functions, as shown in the following example.

Example: Temporarily Disabling a Callback

```
bool previousAlarmEnable;

alarmWasDisabled = DRV_TMR_AlarmDisable(pObj->myTimer);

if (pObj->alarmCount > MY_MAX_ALARM_COUNT)
{
    /* Reset my alarm count. */
    pObj->alarmCount = 0;
}
```

```
DRV_TMR_AlarmEnable(pObj->myTimer, previousAlarmEnable);
```

In the previous example, the `if` statement reads the `alarmCount` member of the current module's structure and, depending on its value, the assignment inside the `if` statement modifies and writes it. Since this takes more than one instruction, the caller cannot allow alarm callbacks to occur in this time. So, the Timer Driver provides client interface functions to conveniently disable and restore the callback functionality. If the driver did not provide these functions the client would have to deregister the callback and/or stop the timer to ensure that the `alarmCount` variable would not be corrupted by a simultaneous access by its own interface or tasks functions.

Another concern that the driver developer must take care to avoid when providing a callback function is a common race condition that can occur. Most callback functions are used as synchronization methods. (See Synchronization for details.) An interface call-in function will usually start some process that takes time and a callback function will notify the client when the process has completed. The danger is that, if the process completes too quickly, it may cause an interrupt to occur immediately and call the callback function before the call-in function returns. That can be a serious problem if the client needs to use the return value of the interface call-in function from within the callback function. Unfortunately, that is exactly what would happen when a transfer handle is returned from a data transfer function, as shown in the following example.

Example: Transfer Synchronization Callback

```
void MyBufferEventHandler ( DRV_USART_BUFFER_EVENT  event,
                            DRV_USART_BUFFER_HANDLE bufferHandle,
                            uintptr_t               context )
{
    MY_OBJ *pObj = (MY_OBJ *)context;

    if (pObj->myBufferHandle == bufferHandle)
    {
        switch(event)
        {
            case DRV_USART_BUFFER_EVENT_COMPLETE:
            {
                /* Clean up after my buffer transfer is complete. */
            }

            /* Handle other events for my buffer */
        }
    }
}
```

Example: Interface With an Intrinsic Race Condition

```
char buffer[] = “Hello World\n”;

pObj->bufferHandle = DRV_USART_BufferAddWrite(pObj->myUsart, buffer, strlen(buffer));
```

In the previous example, the DRV_USART_BufferAddWrite function adds the buffer containing the `Hello World\n` string to the USART driver's write buffer queue and returns a handle identifying the request to write that buffer. When the transfer completes, the driver will call the MyBufferEventHandler function and pass in the DRV_USART_BUFFER_EVENT_COMPLETE event, the buffer handle returned from the DRV_USART_BufferAddWrite function, and the context value passed in when the callback was registered. (The context is used to identify the instance of the caller that registered the callback. In most cases, a caller will pass in a pointer to its instance data structure so the callback can recover it, as shown in this example.)

In this example, as long as the DRV_USART_BufferAddWrite function returns the buffer handle before transfer finishes and the MyBufferEventHandler function is called back, this will work just fine. But, what happens if the write queue is empty, the buffer is only one byte in size and the baud rate really is high? It is possible that the transfer will finish and the callback will happen when the interrupt occurs before the DRV_USART_BufferAddWrite function has had time to return and the buffer handle value has been assigned to the `pObj->bufferHandle` variable. If that should happen, the value of the `bufferHandle` parameter passed into the MyBufferEventHandler callback function will not match the value in the value stored in the `pObj->bufferHandle` variable because it has not yet been stored there. When that occurs, the MyBufferEventHandler function will incorrectly decide that the event was not for buffer it was looking for and it will not correctly perform whatever clean-up logic it was designed to perform. So, whether or not this callback works correctly depends on who wins the race, the client or the driver.

To avoid this common race condition, it is necessary to put the timing of the assignment of the buffer handle into the hands of the driver where it can be managed successfully. A better driver interface design would make the transfer handle an output parameter instead of a return value by passing the address of the variable to receive the value of the buffer handle as a parameter, as shown in the following example.

Example: Interface Without an Intrinsic Race Condition

```
char buffer[] = “Hello World\n”;

DRV_USART_BufferAddWrite(pObj->myUsart, buffer, strlen(buffer), &pObj->bufferHandle);
```

In this example, the USART driver can now eliminate the potential race condition that was intrinsic in the previous DRV_USART_BufferAddWrite and callback interface. By passing the address of the `pObj->bufferHandle` variable into the DRV_USART_BufferAddWrite function, the driver can ensure that it assigns the correct `buffer` handle value to the variable before it starts transferring the buffer. This guarantees that, no matter how quickly the buffer transfer finishes and how quickly the callback occurs, the client's variable has the correct value so it will match the value passed in by the driver.

Callback functions can be a very useful mechanism for synchronizing between a driver and its client, but care must be taken make sure the client has a working usage model or the driver may not be useful in some configurations.

Synchronization

Describes how to use the OS Abstraction Layer (OSAL) to synchronize between threads and ISRs to manage blocking behavior.

Description

Some driver interface functions have an intrinsically blocking usage model. For example, most developers would expect the file system style read and write functions to block and not return until the entire transfer had completed. While this cannot be accomplished in a bare-metal environment, it can be accomplished in a RTOS configuration in a way that still allows usage in a non-RTOS environment by using the OSAL semaphore support.

Example: Blocking Function

```
size_t DRV_MYDEV_Write( DRV_HANDLE handle, void *buffer, size_t size)
{
    size_t        count;
    DRV_MYDEV_OBJ pObj  = (DRV_MYDEV_OBJ *)handle;

    count = 0;
    while(count < size)
    {
        count += PLIB_MYDEV_Transmit(pObj->devIndex, buffer, size-count);
        if (count < size)
        {
            if (OSAL_SEM_Pend(pObj->txSemaphore) == OSAL_RESULT_TRUE)
            {
                /* Exit loop if semaphore fails or if no RTOS */
                break;
            }
        }
    }

    return count;
}
```

In the previous example, the DRV_MYDEV_Write function attempts to loop, repeatedly filling the fictitious MYDEV device's transmit FIFO until it has sent all size bytes of data pointed to by the buffer parameter by calling the PLIB_MYDEV_Transmit function (assuming that is what this function does and that it returns the actual number of bytes copied to the transmitter FIFO). If the buffer passed in contains more data bytes than will fit into the device's FIFO, `count` will be less than `size` and the code will call the OSAL_SEM_Pend function. In a RTOS configuration, assuming that no other code has previously posted the `txSemaphore`, this will cause the thread that called the DRV_MYDEV_Write to block (be suspended by the OS scheduler) until some other thread or ISR posts the semaphore.

Presumably, this fictitious device will set an interrupt flag when its transmitter FIFO is empty (or as shown in a following section, a given watermark level). If that is the case, the driver's tasks function will need to call either the OSAL_SEM_Post or OSAL_SEM_PostISR function (depending upon whether or not it is configured for polled or interrupt-driven operation), passing in the `txSemaphore`, to signal that the transmitter is ready to accept more data. When that happens, the previous call to OSAL_SEM_Pend will return with an OSAL_RESULT_TRUE value. This causes the loop to continue until it has successfully transmitted all data (when `count` equals `size`) or until OSAL_SEM_Pend returns some other value.

If this code is built in a non-RTOS configuration, the OSAL_SEM_Pend function will instead return OSAL_RESULT_FALSE. When that occurs, the example will break out of the loop and the DRV_MYDEV_Write function will return the current count of how much data was successfully written to the device's transmitter FIFO. While not ideal (after all, the caller was hoping all of its data would be written), this behavior is still consistent with the expected behavior of the DRV_MYDEV_Write function and is completely safe so long as the caller appropriately checks the return value and adjusts accordingly.

When using a technique like this, blocking behavior becomes an optimization that is available when a RTOS is used, which is why the naturally blocking file system style read and write functions operate best in a RTOS environment and they are a bit inefficient (but still work) in a bare metal environment. Similar techniques can be used to synchronize between any interface functions and the associated tasks function(s) in any driver (or other library). Using this technique also illustrates that a driver is best designed to block in its interface functions and not in its tasks function(s). Using this technique best synchronizes clients, calling a driver's interface routines, with the operation of the driver's state machine.

Configuration and Implementations

Describes the different configuration capabilities that a MPLAB Harmony device driver developer must comprehend.

Description

MPLAB Harmony is very flexible and can support a number of ways in which a device driver (or other library) may be configured and customized to suit the individual needs of a specific system. These usually methods fall into one of the following categories.

- Optional Feature Sets
- Configuration Options
- Multiple Implementations

These different methods are described in the following sections. However, there are two important overriding concerns that the driver (or other library) developer must keep in mind when utilizing this flexibility.

First, all options should be managed and presented to the user for selection by the MPLAB Harmony Configurator (MHC). The MHC is the utility that manages MPLAB Harmony libraries and integrates them into the MPLAB X IDE development environment. While it is certainly possible to develop MPLAB Harmony-compatible drivers and add them to an application directly (in source code) with no MHC integration, doing so is missing out on the power and convenience of the MHC and will quickly and inevitably result in a need to manage changes in less effective ways. Adding MHC support for the libraries and their configuration options will dramatically simplify the tasks necessary to manage the addition, removal, and configuration of a driver (or any library) to one or more MPLAB Harmony projects and is will worth the effort.

Note: Please refer to the MPLAB Harmony Configurator Developer's Guide for information on how to support the MHC.

Second, the most important thing about any configuration or implementation of a MPLAB Harmony driver (or other library) is that its interface must stay consistent with the interface of all other configurations and implementations of the same driver (or library). This does not mean that every implementation of a driver must support every function defined in the driver's interface (see optional features). But, it does mean that if an implementation supports a feature then it must provide the same interface to that feature that all other implementations or configurations provide to that feature. An implementation of a driver that does not follow the interface defined in the help and by the driver's interface header is not an alternate implementation or configuration of the same driver. It is an entirely different driver or library.

These two considerations are important to all driver development. If they are not properly comprehended, the end result cannot be easily managed in the same way as other MPLAB Harmony drivers. The user will not be able to treat the driver as a building block module and he will likely need to either modify his application to reuse the driver or directly modify the driver itself. Properly managing these concerns is vital to developing highly reusable MPLAB Harmony drivers.

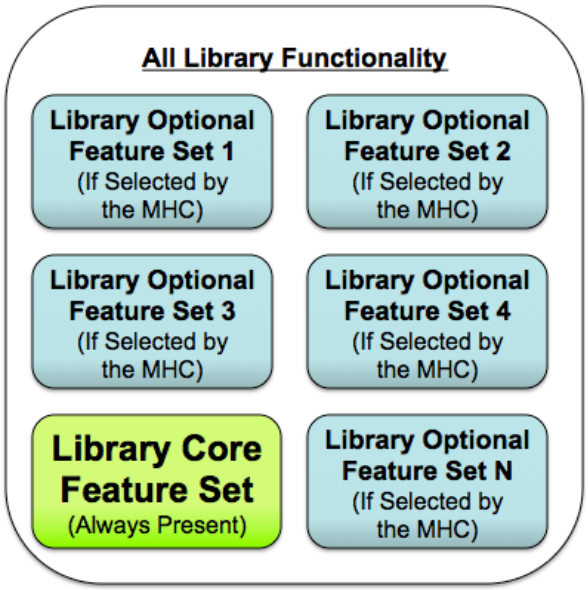
Optional Feature Sets

Describes how to group MPLAB Harmony driver features into sets and manage them as build configuration options.

Description

Any MPLAB Harmony library or driver should have a core feature set and interface to that feature set. This is the simplest and most basic functionality that is always provided by that driver, without which, there is no reason to use the driver in a system. All other features that may be provided by the driver can be considered optional and should be broken into sets, as shown in the following diagram.

Library Feature Sets



These feature sets should be identified and described in the **Configuring the Library** section of the driver's Help documentation, along with a description of how to select and configure each feature set. Any implementation of a driver that supports a feature set should support all features that are part of the set. Configuration options may affect how that feature set is supported (for example, buffer sizes used or minimums and maximums supported), but the inclusion or exclusion of that feature must happen as a single unit. Either all features in a set are included in the project when support for the feature is selected or they are all excluded from the project, based on a single MHC selection. Driver implementations that support *rogue* features that are undocumented or are not part of any defined feature set dramatically complicate the usage of the driver, make it difficult to represent its configuration in the MHC and prevent the user from treating different implementations of the driver as a simple building blocks.

In general, it is best to develop a driver so that it builds upon the core feature set in a clean and modular way. A good way to do this is to think of a feature set as a sub-module of its own and implement all code for that one feature set in a common source file. The source file for the core feature set will always be included in the project whenever the driver is included. The source file for an optional feature can then be included or excluded from the project, depending on whether or not the user selects the feature, as shown by the following examples.

Example: Core Feature (drv_mydev.c)

```
MY_RETVAL DRV_MYDEV_CoreFunction1 ( DRV_MYDEV_HANDLE handle, DRV_MYDEV_A *data )
{
    /* Implementation of core interface function 1 */
}

MY_RETVAL DRV_MYDEV_CoreFunction2 ( DRV_MYDEV_HANDLE handle, DRV_MYDEV_B *data )
{
    /* Implementation of core interface function 2 */
}

/* Additional core interface and internal functions... */

void __attribute__((weak)) _DRV_MYDEV_TasksOpt1 ( DRV_MYDEV_OBJ obj )
{
    return;
}

void DRV_MYDEV_Tasks ( DRV_MYDEV_OBJ obj )
{
    /* Implementation of core tasks state machine. */

    _DRV_MYDEV_TasksOpt1(obj);
}
```

Example: Optional Feature (drv_mydev_opt1.c)

```
MY_RETVAL DRV_MYDEV_Option1Function1 (DRV_MYDEV_HANDLE handle, DRV_MYDEV_A *data )
{
    /* Implementation of option 1 interface function 1 */
}

MY_RETVAL DRV_MYDEV_InterfaceFunction2 ( DRV_MYDEV_HANDLE handle, DRV_MYDEV_B *data )
{
    /* Implementation of option 1 interface function 2 */
}

/* Additional option 1 interface and internal functions... */

void _DRV_MYDEV_TasksOpt1 ( DRV_MYDEV_OBJ obj )
{
    /* Implementation of optional feature set 1 state machine. */
}
```

In the previous examples, the any interface or internal functions that are specific to the optional feature are implemented in a separate source file (`drv_mydev_opt1.c`) from the core feature set functions implemented in the driver's primary source file (`drv_mydev.c`). The core driver implementation file is always included in a project if the *mydev* driver is used. If the optional feature set is selected, the optional source file is also included. This provides implementations of the optional interface functions (DRV_MYDEV_InterfaceFunction1 and DRV_MYDEV_InterfaceFunction1, for example) as well as any internal and tasks or sub-tasks functions.

The _DRV_MYDEV_TasksOpt1 function shows the technique of using a weak function in the core feature set's implementation file to implement a sub-tasks state machine function. This can then be called from within the main state machine's tasks function. If the optional feature file (`drv_mydev_opt1.c`) is not included in the build, the weak implementation of the function will be called (and likely removed from the build, depending on the level of optimization chosen). However, if the optional feature set is selected and the implementation of the _DRV_MYDEV_TasksOpt1 function is built, the linker will override the weak definition in `drv_mydev.c` and link the call to the full non-weak implementation in the `drv_mydev_opt1.c` file calling the optional feature's state machine when the driver's core state machine is called.

This is the preferred method for implementing optional features (in separate files), but if this method is not feasible or if it adds more complexity than it saves, it is acceptable to use preprocessor macros to switch implementations of a function or short sequence of code based upon the selection of a configuration option, using the mapping method shown in the following example.

Example: Macro Mapping Function Implementations

```
#if defined(DRV_MYDEV_USE_OPT1)

    #define _DRV_MYDEV_TasksOpt1(obj)    _DRV_MYDEV_TasksOpt1Implementation(obj)

#else

    #define _DRV_MYDEV_TasksOpt1(obj)

#endif
```

This method allows the optional function's code to be removed even in builds with no optimizations, but it is more confusing and can be harder to debug. It is most useful an optional feature requires one or two short code sequences, when the first method is too much. This method can also be

used to wrap data allocations (see the OSAL_MUTEX_DECLARE and OSAL_SEM_DECLARE macros for examples).

In general, it is best to work to minimize the use of preprocessor `#if` directives as much as possible as they tend complicate the code and obfuscate the logic. If they must be used (as shown previously), it is best to group them into a single local mapping header included (for example the previous macros might be defined in a `drv_mydev_local_mapping.h` file). If they must be used directly in source code, it is best to indent them and the contents as if they were normal `if` statements. Do not force them to align at column 0. That is an old C-language standard that is no longer required and only serves to render code harder to read.

Configuration Options

Describes how to create and manage static build-time configuration options for MPLAB Harmony drivers.

Description

One of the primary ways in which MPLAB Harmony libraries are configured is by the definition and utilization of static configuration options defined at build-time. In most cases, these configuration options take the form of name-value pairs, defined in the system-wide `system_config.h` header using C preprocessor `#define` statements, as shown in the following example.

Example: Static Configuration Option Definitions

```
/* DRV USART Configuration Options */
#define DRV_USART_QUEUE_DEPTH_COMBINED              20
#define DRV_USART_CLIENTS_NUMBER                    6
#define DRV_USART_INSTANCES_NUMBER                  2

/* DRV USART 0 Initialization */
#define DRV_USART_PERIPHERAL_ID_IDX0                USART_ID_2
#define DRV_USART_BRG_CLOCK_IDX0                    80000000
#define DRV_USART_BAUD_RATE_IDX0                    9600

/* DRV USART 1 Initialization */
#define DRV_USART_PERIPHERAL_ID_IDX0                USART_ID_2
#define DRV_USART_BRG_CLOCK_IDX0                    80000000
#define DRV_USART_BAUD_RATE_IDX0                    9600
```

These macros are utilized in one of two ways.

- To define implementation-specific options
- To define instance-specific options

Implementation-specific macros are often used to control allocation of internal arrays or buffers and to control logic that manages them, as shown by the following example.

Example: Using Implementation-Specific Options

```
DRV_USART_BUFFER_OBJ gDrvUSARTBufferObj[DRV_USART_QUEUE_DEPTH_COMBINED];
unsigned int i;

/* Search the buffer pool for a free buffer object */
for(i = 0; i < DRV_USART_QUEUE_DEPTH_COMBINED; i++)
{
    if(!gDrvUSARTBufferObj[i].inUse)
    {
        /* Initialize buffer object. */
        gDrvUSARTBufferObj[i].inUse = true;
        break;
    }
}

if(i >= DRV_USART_QUEUE_DEPTH_COMBINED)
{
    /* Could not find a buffer. */
}
```

In the previous example, the USART driver keeps a common pool of buffer objects in an array, the size of which is determined by the DRV_USART_QUEUE_DEPTH_COMBINED configuration option. When a buffer object is needed, the driver logic searches through the array, looking for one that has not been allocated by checking its `inUse` flag. This option affects both the amount of RAM statically allocated by this driver and the code generated when it is built.

Note: This sequence of code is simplified for explanation. A real implementation would need to perform additional initializations and be protected for interrupt and thread safety.

Instance specific macros are used in `system_init.c` to initialize a dynamic driver's `init` data structure, as shown in the following example, or are built directly into instance-specific static driver implementation, using the same method shown previously.

Example: Using Instance-Specific Options

```
const DRV_USART_INIT drvUsart0InitData =
{
    .usartID  = DRV_USART_PERIPHERAL_ID_IDX0,
    .brgClock = DRV_USART_BRG_CLOCK_IDX0,
    .baud     = DRV_USART_BAUD_RATE_IDX0,

    /* Initialize other “init” data members. */
};

sysObj.drvUsart0 = DRV_USART_Initialize(DRV_USART_INDEX_0,
                                        (SYS_MODULE_INIT *)&drvUsart0InitData);
```

Since the user must provide the values of these options, the MHC must be made aware of them To make the MHC aware of an option, you must provide the appropriate *config* definitions in the Hconfig file hierarchy. And, to enable code generation based on these options, you must develop the necessary FreeMarker templates as described in the MPLAB Harmony Configurator Developer's Guide. Please refer to that document for details on developing Hconfig and FreeMarker template files.

Additionally, since static configuration options are defined in the `system_config.h` header, there are a few key guidelines governing this file to keep in mind.

Key `system_config.h` Guidelines:

- Any driver (or other source file) that uses any build-time configuration options supported by the MHC must include this header. The MHC always adds the path to this header to the compiler's include file search path. So, it is included without any path information (e.g., `#include "system_config.h"`).
- The `system_config.h` header must not contain any data type or function prototypes definitions. It must only define pre-processor name-value macros or header file include file dependency loops that cannot be resolved may occur.
- It is acceptable for configuration options to utilize symbol names that have not yet been defined because the macros it defines are not instantiated until used in a source file

Note: Type definitions used by system configuration code are defined in the `system_definitions.h` header file. This file should only be used by the system configuration code as it defines data types and external references for system configuration code.

While it is possible to define callable macros that emulate functions (or that map function call names and parameters to different selectable functions) in the `system_config.h`, it is best to implement such macros in the library code and select their implementations based upon name-value macro definition(s) defined by MHC, as shown in the following example. This simplifies the configuration files and template code and keeps the knowledge and complexity of the macro's implementation encapsulated in the library.

Example: Defining "Callable" Macros

```
#if (SAMPLE_MODULE_INTERRUPT_MODE == true)

    #define _InterruptDisable(s)    SYS_INT_SourceDisable(s)

#else

    #define _InterruptDisable(s)    false

#endif
```

The previous example shows how to map a local function used to disable the sample module's interrupt source to the appropriate system service when the library is configured for interrupt-driven operation (`SAMPLE_MODULE_INTERRUPT_MODE == true`) or to an implementation that provides an appropriate return constant when it is not. This technique allows the driver developer to capture the knowledge of the necessary implementation variants while providing the higher-level choice (of interrupt-driven or polled, in this example) to the user.

Multiple Implementations

Describes how to define multiple implementations of MPLAB Harmony drivers.

Description

As described in Interface vs. Implementation, the features and functionality provided by a driver are defined by its interface, not by any specific implementation of that driver. And, due to the flexibility and configurability provided by MPLAB Harmony and the MHC, it is possible to provide multiple implementations of the driver that are optimized for different hardware or different purposes. To support the selection and management of such implementations, the MHC provides the ability to select different source files, based upon configuration choices made by the user. (Refer to the MPLAB Harmony Configurator Developer's Guide for instructions on how to develop the Hconfig files to support this capability.)

Because different implementation variants provided the same interface (and thus, define the same interface functions), they are mutually exclusive. Only one implementation of a specific diver (or other library) can be included in the system at a time. Variant implementations of a MPLAB Harmony driver (or other library) are usually defined for one of three reasons:

- Targeted/optimized usage
- Integration of dependencies
- Static implementations

A targeted implementation is optimized for a specific usage or hardware selection. It can make use of hardware acceleration (for example built-in

DMA support) or simply eliminate functionality that is not required for a specific usage.

Integrated implementations may combine multiple libraries or driver *stacks* into a single driver, improving efficiency and reducing code size at the cost of the flexibility provided by a stack. For example, a SPI Codec driver would normally utilize the SPI driver to access its Codec over the SPI bus so that it can switch SPI bus drivers, if necessary. However, a integrated SPI Codec driver may directly utilize the SPI peripheral itself, integrating the SPI driver functionality to save code size and achieve better performance at the cost of being able to switch to a different type of SPI peripheral.

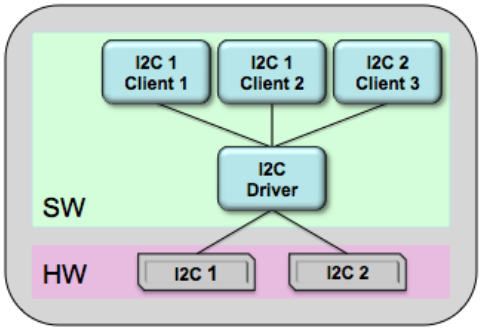
Refer to Static Implementations for details.

Implementing Multiple Client Drivers

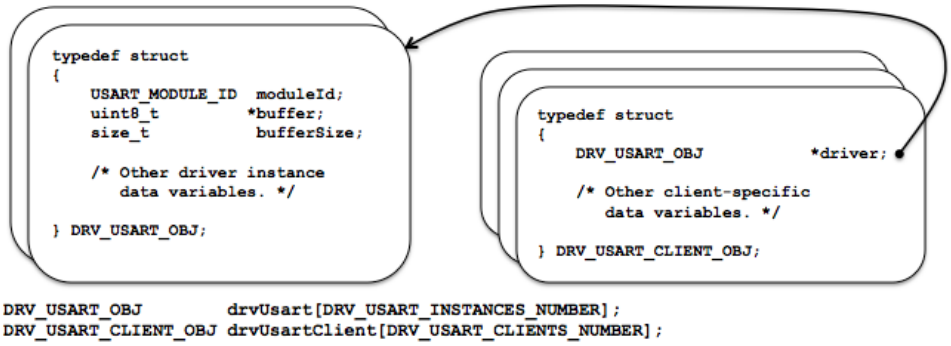
Describes how to implement multiple client drivers.

Description

When the concept of managing multiple clients is combined with the concept of managing multiple instances, the full picture of a MPLAB Harmony driver emerges, as shown in the following diagram. See Single Client vs. Multiple Client and Implementation vs. Instances for more information.



One method of implementing the ability to manage multiple instances of the peripheral hardware and multiple independent clients within the same driver is described in the following diagram, using the familiar USART as an example. Other methods may be possible, but this example illustrates the requirement.



The diagram shows the definitions of two data structures. The DRV_USART_OBJ structure is used to store all of the data required to manage a single instance of the peripheral hardware. (This structure is described in the Implementation vs. Instances section.) The DRV_USART_CLIENT_OBJ structure stores all of the data required to keep track of an individual client.

Since the driver manages multiple instances of the peripheral, there will be one instance of the DRV_USART_OBJ structure per peripheral instance, as defined by the DRV_USART_INSTANCES_NUMBER configuration parameter and allocated by the `drvUsart` array shown previously. Since the driver manages multiple clients, there will also be a number of client data structures, as defined by the DRV_USART_CLIENTS_NUMBER configuration parameter and allocated by the `drvUsartClient` array. One structure form the array will be assigned to each client that calls the driver's open function until they are all allocated.

One item of particular importance in the client object structure is the pointer that associates a client with the driver instance. This pointer (and usually other data) will be initialized when a client calls the driver's open function. The following example shows a possible implementation of this function (assuming the previous structure definitions).

Example: Driver Open Function

```
DRV_HANDLE DRV_USART_Open( const SYS_MODULE_INDEX index,
                           const DRV_IO_INTENT    ioIntent )
{
    int i;
    DRV_USART_CLIENT_OBJ pClient = (DRV_USART_CLIENT_OBJ *)DRV_HANDLE_INVALID;

    for (i=0; i < DRV_USART_CLIENTS_NUMBER; i++)
    {
        if (drvUsartClient[i].driver == NULL)
```

```
        {
            pClient = &drvUsartClient[i];
            pClient->driver = &drvUsart[index];
            break;
        }
    }

    return (DRV_HANDLE)pClient;
}
```

This implementation of the DRV_USART_Open function does a linear search through the array of client objects. The first one it finds with a NULL driver pointer is assumed to be unallocated and available for use. It then assigns the address of the driver object structure in the `drvUsart` array that is identified by the function's index parameter. Doing this simultaneously allocates that client object and associates it with the specified driver instance. An open function would normally store some additional data and maybe do some other preparation to get ready to service the client, but this simple example shows how a unique opened driver handle can be created that identifies a client, how a client object structure might provide a storage location for client-specific data, and how the driver can associate the handle with a specific instance of the driver and peripheral.

Note: This example is not RTOS safe. A RTOS safe implementation would protect the for loop with a mutex. Refer to the Interrupt and Thread Safety section for more in formation on RTOS safety.

Once the driver has been opened and the association between the client and the driver instance has been made, the driver handle can be returned to the client and then later be used by other client interface functions to interact with the peripheral safely, as shown in the following example.

Example: Client API Function Implementation

```
void DRV_USART_BufferAddRead ( const DRV_HANDLE handle,
                               DRV_USART_BUFFER_HANDLE * const bufferHandle,
                               void * buffer, const size_t size )
{
    DRV_USART_CLIENT_OBJ *client = (DRV_USART_CLIENT_OBJ *)handle;
    DRV_USART_OBJ        *driver = (DRV_USART_OBJ *)client->driver;

    if (driver->buffer == NULL)
    {
        driver->buffer       = buffer;
        driver->bufferSize   = size;
        bufferHandle         = buffer;

        /* Start the data transfer process. */

    }
    else
    {
        *bufferHandle = DRV_USART_BUFFER_HANDLE_INVALID;
    }

    return;
}
```

This example shows how a buffer queuing read might work. However, it is somewhat oversimplified because it only maintains a queue size of one. This is because the driver structure only keeps a single buffer pointer and `bufferSize` variable (instead of a queue of several of them). So, in this example, the driver checks to see if its buffer pointer is NULL (which, presumably, that is how it was initialized during the driver's initialize function and how it is reset whenever a transfer completes). However, if the buffer pointer is not NULL, it means that the driver is currently busy transferring a previous request. This causes the driver to return an invalid buffer handle value (DRV_USART_BUFFER_HANDLE_INVALID) to the caller. When this happens, the client calling will have to try again later because the queue (of one entry) is full. If the buffer pointer is NULL, it is not in use (i.e., the queue is not full) and the function will save the caller's buffer pointer and size information and do whatever is necessary to start the data transfer, likely interacting with the hardware through the peripheral library at that point.

While this example is incomplete and somewhat limited, it does demonstrate how a client API function might use the opened driver handle to identify the link to the peripheral instance and prevent conflicts between peripherals. In this case, a rather brute force method of only allowing a single transfer to occur at a time is used; but, that method is completely valid as regardless of how big the queue is, it can always become full. However, more sophisticated methods (such as implementing an actual transfer queue) would let the driver provide better throughput.

Static Implementations

Describes how to develop static MPLAB Harmony driver implementations.

Description

As described in Static vs. Dynamic, a dynamic driver implementation manages multiple instances of a particular type of peripheral and a static driver implementation only manages one. This allows a single static implementation of a driver to be smaller than the equivalent dynamic

implementation, saving code space by hard coding values that can be made constant. The following examples show the basic differences between the two types of implementations.

Example: Dynamic Implementation

```
SYS_MODULE_OBJ DRV_USART_Initialize ( const SYS_MODULE_INDEX  index,
                                      const SYS_MODULE_INIT * const init )
{
    DRV_USART_OBJ  *pObj  = (DRV_USART_OBJ *)&gDrvUsartObj[index];
    DRV_USART_INIT *pInit = (DRV_USART_INIT *)init;

    /* Initialize data for this instance */
    pObj->usartId               = pInit->usartId;
    pObj->interruptSourceTx     = pInit->interruptSourceTx;
    pObj->interruptSourceRx     = pInit->interruptSourceRx;
    pObj->interruptSourceErr    = pInit->interruptSourceErr;
    pObj->queueSizeCurrentRead  = 0;
    pObj->queueSizeCurrentWrite = 0;
    pObj->queueRead             = NULL;
    pObj->queueWrite            = NULL;

    /* Initialize USART Hardware */
    PLIB_USART_Disable(pObj->usartId);
    PLIB_USART_HandshakeModeSelect(pObj->usartId, pInit->handshake);
    PLIB_USART_BaudSetAndEnable(pObj->usartId, pInit->brgClock, pInit->baud);
    PLIB_USART_LineControlModeSelect(pObj->usartId, pInit->lineControl);

    /* Clear and enable the interrupts */
    SYS_INT_SourceStatusClear(pObj->interruptSourceTx);
    SYS_INT_SourceStatusClear(pObj->interruptSourceRx);
    SYS_INT_SourceStatusClear(pObj->interruptSourceErr);
    _InterruptSourceEnable(pObj->interruptSourceErr);

    /* Ready! */
    pObj->status = SYS_STATUS_READY;
    PLIB_USART_Enable(pObj->usartId);
    return (SYS_MODULE_OBJ)pObj;
}
```

The previous dynamic example shows that the driver's initialization function must capture any initialization data that could be different from one instance to another and that it must use a pointer (`pObj`) to the desired instance of its global data object/structure (`gDrvUsartObj[index]`) and clear the references that point to access of any global data in the object. It must also clear the reference to the `init` pointer (cast to `pInit`) to access any initialization data, whether or not it is stored in the global data object.

Note: The _InterruptSourceEnable function is a locally mapped function that switches implementations depending upon whether or not the driver was built in Interrupt-driven mode or Polled mode, as shown in Interrupt Safety.

The following equivalent static example shows how a static implementation saves both code and data space.

Example: Static Implementation

```
void DRV_USART0_Initialize ( void )
{
    /* Initialize data for this instance */
    gDrvUsart0Obj.queueSizeCurrentRead  = 0;
    gDrvUsart0Obj.queueSizeCurrentWrite = 0;
    gDrvUsart0Obj.queueRead             = NULL;
    gDrvUsart0Obj.queueWrite            = NULL;

    /* Initialize USART Hardware */
    PLIB_USART_Disable(DRV_USART_ID_IDX0);
    PLIB_USART_HandshakeModeSelect(DRV_USART_ID_IDX0,
                                   DRV_USART_HANDSHAKE_MODE_IDX0);
    PLIB_USART_BaudSetAndEnable(DRV_USART_ID_IDX0,
                                DRV_USART_BRG_CLOCK_IDX0,
                                DRV_USART_BAUD_RATE_IDX0);
    PLIB_USART_LineControlModeSelect(DRV_USART_ID_IDX0,
                                     DRV_USART_LINE_CNTRL_IDX0);

    /* Clear and enable the interrupts */
    SYS_INT_SourceStatusClear(DRV_USART_INT_SRC_TX_IDX0);
    SYS_INT_SourceStatusClear(DRV_USART_INT_SRC_RX_IDX0);
    SYS_INT_SourceStatusClear(DRV_USART_INT_SRC_ERR_IDX0);
```

```
    _InterruptSourceEnable(DRV_USART_INT_SRC_ERR_IDX0);

    /* Ready! */
    gDrvUsart0Obj.status = SYS_STATUS_READY;
    PLIB_USART_Enable(DRV_USART_ID_IDX0);
}
```

In the previous static example, the function would be implemented differently for different driver instances (DRV_USART0_Initialize, DRV_USART1_Initialize, etc.). So, any initialization data that is different from one instance to another can be defined by different configuration macros (such as DRV_USART_ID_IDX0) and hard-coded directly into the function's implementation. This reduces code and data size because it eliminates the need to store these items in the driver's global data structure instance and it eliminates the need to clear the reference to a pointer and access a variable when using these values.

And, using a constant instead of a variable greatly reduces the amount of code generated by PLIB functions because PLIB functions are implemented as C-language inline functions. When a constant is passed to an inline function, the compiler can optimize them by performing calculations before generating the object code instead of generating object code instructions to do the calculations. This eliminates a significant amount of object code, especially when each PLIB function would otherwise index to the appropriate SFRs for the instance of the peripheral passed in as a variable.

Note: The mapping functions shown in Static vs. Dynamic shows how the parameters are dropped and the return value is provided. Refer to this section for an explanation and example of mapping the dynamic driver interface functions to the static implementation functions.

Creating a static driver implementation requires development of a FreeMarker template. (Note that it does not require any additional Hconfig file development, since the dynamic and static implementations both utilize the same configuration options.) Creating a static implementation from a dynamic implementation is primarily a matter of removing the unnecessary code and marking up the dynamic implementation using FreeMarker syntax to parameterize the driver's source code and insert the appropriate values where necessary, as shown by the following example.

Example: FreeMarker Code for Static Implementation

```
<#macro make_drv_usart_initialize_function DRV_INSTANCE>
void DRV_USART${DRV_INSTANCE}_Initialize ( void )
{
    /* Initialize data for this instance */
    gDrvUsart${CONFIG_DRV_INSTANCE}Obj.queueSizeCurrentRead  = 0;
    gDrvUsart${DRV_INSTANCE}Obj.queueSizeCurrentWrite = 0;
    gDrvUsart${DRV_INSTANCE}Obj.queueRead             = NULL;
    gDrvUsart${DRV_INSTANCE}Obj.queueWrite            = NULL;

    /* Initialize USART Hardware */
    PLIB_USART_Disable(DRV_USART_ID_IDX${DRV_INSTANCE});
    PLIB_USART_HandshakeModeSelect(DRV_USART_ID_IDX${DRV_INSTANCE},
                                   DRV_USART_HANDSHAKE_MODE_IDX${DRV_INSTANCE});
    PLIB_USART_BaudSetAndEnable(DRV_USART_ID_IDX${DRV_INSTANCE},
                                DRV_USART_BRG_CLOCK_IDX${DRV_INSTANCE},
                                DRV_USART_BAUD_RATE_IDX${DRV_INSTANCE});
    PLIB_USART_LineControlModeSelect(DRV_USART_ID_IDX${DRV_INSTANCE},
                                     DRV_USART_LINE_CNTRL_IDX${DRV_INSTANCE});

    /* Clear and enable the interrupts */
    SYS_INT_SourceStatusClear(DRV_USART_INT_SRC_TX_IDX${DRV_INSTANCE});
    SYS_INT_SourceStatusClear(DRV_USART_INT_SRC_RX_IDX${DRV_INSTANCE});
    SYS_INT_SourceStatusClear(DRV_USART_INT_SRC_ERR_IDX${DRV_INSTANCE});
    _InterruptSourceEnable(DRV_USART_INT_SRC_ERR_IDX${DRV_INSTANCE});

    /* Ready! */
    gDrvUsart${DRV_INSTANCE}Obj.status = SYS_STATUS_READY;
    PLIB_USART_Enable(DRV_USART_ID_IDX${DRV_INSTANCE});
}

</#macro>
<#list index = 0..CONFIG_DRV_USART_INSTANCES_NUMBER>
  <@make_drv_usart_initialize_function DRV_INSTANCE=index/>
</#list>
```

The previous example defines a FreeMarker macro called make_drv_usart_initialize_function to define the static (instance specific) USART driver initialization functions. Within that function, it uses a local FreeMarker macro variable (DRV_INSTANCE) to indicate the driver's instance index. It places this variable within the function's source code wherever an instance index number is required. Then, it defines a list that increments from 0 to less than the CONFIG_DRV_USART_INSTANCES_NUMBER value defined by the MHC when the user selected how many USART driver instances he wanted. From within that list, it calls the macro, passing in the index value iterated by the list, generating as many static implementations of the USART driver's initialize function as desired.

The previous examples are simplified to aid understanding. Implementing an entire static driver may become more complicated, requiring some

skill with the FreeMarker language. However, the principles remain the same. All configuration variables defined by the MHC are available to the FreeMarker code and may be used as needed. Refer to MPLAB Harmony Configurator Developer's Guide for details on the Hconfig and FreeMarker languages necessary to develop static driver implementations.

Multiple Client Static Drivers

Describes multiple client static drivers.

Description

Supporting multiple clients also has an affect on how static drivers are implemented, particularly if mapping functions are used (see the Static vs. Dynamic section for a description of a static implementation). Other than the open function, all client interface functions require the use of an opened driver. For static implementations, this can be done the same way it is done in a dynamic implementation. The main difference is that the driver object instance is identified in the driver structure name, so the index parameter is ignored, as shown in the following example.

Example: Static Multiple Client Open Function

```
DRV_USART_OBJ        drvUsart0;
DRV_USART_CLIENT_OBJ drvUsartClient[DRV_USART_CLIENTS_NUMBER];


DRV_HANDLE DRV_USART0_Open( const SYS_MODULE_INDEX index,
                            const DRV_IO_INTENT ioIntent )
{
    int i;
    DRV_USART_CLIENT_OBJ pClient = (DRV_USART_CLIENT_OBJ *)DRV_HANDLE_INVALID;

    for (i=0; i < DRV_USART_CLIENTS_NUMBER; i++)
    {
        if (drvUsartClient[i].driver == NULL)
        {
            pClient = &drvUsartClient[i];
            pClient->driver = &drvUsart0;
            break;
        }
    }

    return (DRV_HANDLE)pClient;
}
```

If multiple static implementations are defined, the static driver's client API mapping functions can use the driver object pointer in the client object structure to identify driver instance. The following example shows how this can be done.

Example: Multiple Client Dynamic-to-Static Mapping Function

```
inline void DRV_USART_BufferAddRead ( const DRV_HANDLE handle,
                                      DRV_USART_BUFFER_HANDLE * const bufferHandle,
                                      void * buffer, const size_t size )
{
    DRV_USART_CLIENT_OBJ *client = (DRV_USART_CLIENT_OBJ *)handle;

    switch (client->driver)
    {
        case &drvUsart0:
        {
            DRV_USART0_BufferAddRead(handle, bufferHandle, buffer, size);
            break;
        }

        case &drvUsart1:
        {
            DRV_USART1_BufferAddRead(handle, bufferHandle, buffer, size);
            break;
        }

        default:
        /* invalid instance. */
    }

    return;
}
```

However, as stated in the Static vs. Dynamic section, it normally does not make sense to implement multiple static driver instances for the same system. (If you need multiple instances of a driver, using a dynamic implementation would be more efficient.) So, if only a single instance of a

static driver is implemented and if static and dynamic driver implementations are never used together in the same system (which they should not be because nothing is gained), the open function can be simplified by removing the driver object pointer altogether, as shown in the following example.

Example: Simplified Static Multiple Client Open Function

```
DRV_USART_OBJ        drvUsart0;
DRV_USART_CLIENT_OBJ drvUsartClient[DRV_USART_CLIENTS_NUMBER];


DRV_HANDLE DRV_USART_Open( const SYS_MODULE_INDEX index,
                           const DRV_IO_INTENT ioIntent )
{
    int i;
    DRV_USART_CLIENT_OBJ pClient = (DRV_USART_CLIENT_OBJ *)DRV_HANDLE_INVALID;

    for (i=0; i < DRV_USART_CLIENTS_NUMBER; i++)
    {
        if (drvUsartClient[i].driver == NULL)
        {
            pClient = &drvUsartClient[i];
            break;
        }
    }

    return (DRV_HANDLE)pClient;
}
```

Note: If this method is used, some method of marking a client object structure as assigned to a client, like an "in use" Boolean flag, must be used because you cannot rely on a NULL value for the driver pointer as the indicator.

Also, the mapping function is unnecessary because there is only one mapping. Instead, the client API functions can utilize the exact same names as the dynamic driver's client API functions (meaning both cannot be used together in the same system), as shown by the following example.

Example: Simplified Static Client API Function Implementation

```
void DRV_USART_BufferAddRead ( const DRV_HANDLE handle,
                               DRV_USART_BUFFER_HANDLE * const bufferHandle,
                               void * buffer, const size_t size )
{
    DRV_USART_CLIENT_OBJ *client = (DRV_USART_CLIENT_OBJ *)handle;

    if (drvUsart0.buffer == NULL)
    {
        drvUsart0.buffer     = buffer;
        drvUsart0.bufferSize = size;
        bufferHandle         = buffer;

        /* Start the data transfer process. */

    }
    else
    {
        *bufferHandle = DRV_USART_BUFFER_HANDLE_INVALID;
    }

    return;
}
```

Notice that in this example, the client API function does not need to retrieve the driver instance structure pointer from the client object. Since there is only one driver object structure (`drvUsart0`), it is always used. No other driver object instance is possible. When used throughout the client interface routines in a static implementation of the driver, the elimination of repeated cleared pointer references will reduce the size of the generated object code.

Review and Testing

Describes how to test MPLAB Harmony drivers and references the resources available to help.

Description

The MPLAB Harmony Driver Development Guide (this document) describes how to develop a MPLAB Harmony device driver. To do so, it explains specific the requirements of a MPLAB Harmony driver, including support for the system and client interfaces as well as key design concepts, interrupt and thread safety concerns, and support for multiple configurations and implementations. In addition to this guide, the MPLAB Harmony

Compatibility Guide describes general modularity and flexibility guidelines and provides a compatibility checklist worksheet, available in a separate fillable PDF form in the following documentation folder in the MPAB Harmony installation.

MPLAB Harmony Compatibility Checklist Worksheet Location

`<install-dir>/doc/harmony_compatibility_worksheet.pdf`

Any MPLAB Harmony driver developed should be reviewed, tested and evaluated against the rules described in this guide and checklist. Providing a completed copy of this worksheet along with the documentation of any MPLAB Harmony compatible library will help the user to determine the environments and configuration limits supported by the library.

Additionally, each driver implementation and superset configuration should be thoroughly tested in all supported execution environments (bare metal polled, bare metal interrupt-driven, RTOS multi-threaded polled, RTOS multi-threaded interrupt-driven) to ensure correct and robust operation in all supported usages. To facilitate such testing, MPLAB Harmony provides a Test Harness that is a useful tool for developing and iterating through test in a controlled way that allows easy capturing of failures using MPLAB Harmony debug capabilities. The Test Harness is a library that is included in the MPLAB Harmony installation.

MPLAB Harmony Test Harness Library Location

`<install-dir>/framework/test`

Refer to the MPLAB Harmony Test Harness User's Guide for information on how to utilize the test harness to validate your drivers and libraries.

A good way to provide transparency to the customer is to provide the test applications and test results generated by them with the driver implementation. If a customer can reproduce the test results, it provides great confidence in your libraries and improves the customer's understanding of how the library works and how to use it.

Checklist and Information

Provides a quick reference checklist and important reference information for developing MPLAB Harmony drivers.

Description

At a high-level, the process of developing a MPLAB Harmony device driver is fairly simple. The following checklist describes the basic work flow. The individual steps are described in detail in other sections in this guide.

MPLAB Harmony Development Checklist

Done	Step	Description
☐	1	Define (and document) System interface functions.
☐	2	Define (and document) Client interface functions.
☐	2a	Define data transfer functions (following common models if appropriate).
☐	2b	Define driver-specific functions.
☐	3	Develop Hconfig and FreeMarker template support for initialization, tasks, and other system functions as needed to test and develop.
☐	4	Develop and test initialize, tasks, and core functions.
☐	4a	Start in a polled environment.
☐	4b	Update & test for interrupt-driven environment (retest polled).
☐	4c	Update & test OS support (retest polled & interrupt-driven).
☐	5	Define (and document) MHC support for static configuration options.
☐	6	Define (and document) optional feature sets and functions (one at a time).
☐	6a	Implement & test in all three environments (polled, interrupt-driven, & OS-driven).
☐	6b	Define MHC support for optional feature sets.
☐	7	Develop alternate implementations (particularly static/optimized implementations).

When Implementing a MPLAB Harmony driver, adhere to the following folder layout and file/folder naming conventions.

MPLAB Harmony Drier File and Folder Layout Conventions

Path	Description
`<library>`	Root folder of the MPLAB Harmony driver library. Contains all library header, source files, configuration, and template files.
`<library>/<library>.h`	Library interface header file.
`<library>/<library>_mapping.h`	Library interface level (dynamic-to-static) mapping file.
`<library>/src`	Library C-language source code folder.
`<library>/src/*.c, *.h`	Library implementation files and headers.

<library>/config	Library configuration folder.
<library>/config/*.h	C-language configuration header example (primarily for documentation purposes).
<library>/config/*.hconfig	MHC Hconfig files defining all library configuration options.
<library>/config/*.hconfig.ftl	MHC Hconfig files that are preprocessed by FreeMarker before being integrated into the Hconfig tree of MHC.
<library>/templates/*.ftl	MHC FreeMarker template files for MHC code generation of static implementations and system configuration code necessary integrate the library into a MPLAB Harmony system.

MPLAB Harmony Test Harness User's Guide

This section describes how to effectively use the MPLAB Harmony Test Harness.

Description

This user's guide describes how to use the MPLAB Harmony Test Harness to manage tests to verify that a library functions as expected and is compatible with the different execution models supported by MPLAB Harmony.

Additional information on the Test Harness Library and example tests is available in the Test Libraries Help.

Introduction

Introduces the MPLAB Harmony Test Harness.

Description

The Test Harness can execute up to 16 tests, each with up to five state machine "Tasks" functions (see **Note**). Each test implements a single test. Each test can have any number of sub-tests. If all sub-tests pass, the test passes. If any sub-test fails, the test fails. If all test pass, the harness reports an overall passing result. If any test fails, the harness reports an overall failure result.

Note: These limitations are imposed by the MHC configuration support and can be exceeded by modifying the system configuration source code as necessary. However, doing so is beyond the scope of this documentation.

The library help and usage documentation defines the library's interface and expected behavior. A test that fulfills the requirements of the Test Harness, must be written to call the library's interface and to verify its expected behavior. Configurations of the Test Harness project must then be created to validate the library's behavior in each of the supported execution environments.

The Test Harness supports the following MPLAB Harmony execution environments:

- Bare Metal (No OS), Polled Execution
- Bare Metal (No OS), Interrupt-driven Execution
- RTOS-Based (Multi-threaded), Polled Execution
- RTOS-Based (Multi-threaded), Interrupt-driven Execution

Error messages, individual test results and other messages are given as they occur, provided that a SYS_DEBUG output method is available and has been configured. Otherwise, a debugger must be used to check the results stored in an internal data structure of the Test Harness once the Test Harness has gone into it's idle state (at which point it executes a hard-coded debug breakpoint when built with debug support).

Assumptions and Prerequisites

Provides information on prerequisites and outlines assumptions on items the user should be familiar with before using the Test Harness.

Description

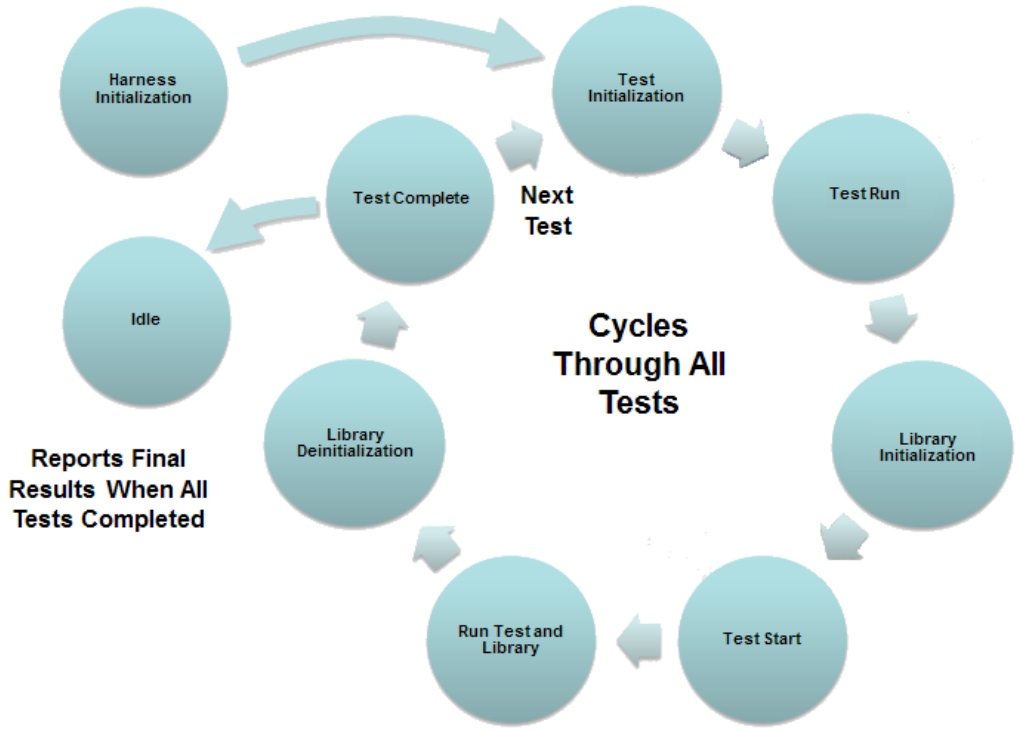
This user's guide assumes that you are familiar with the MPLAB X IDE, XC32 language tools, PIC32, C-language programming, and MPLAB Harmony, and that you have met all MPLAB Harmony development prerequisites. See the MPLAB Harmony Help Prerequisites section for additional information.

Test Harness State Machine

Describes the State Machine for the Test Harness Library.

Description

The basic logic of the Test Harness state machine is shown in the following figure.



The test harness is initialized by calling the TEST_Initialize function and passing in a pointer to a data structure that identifies how many tests to run and provides all of the information necessary to manage the tests and library to be tested by each. The test harness state machine then runs as the TEST_HarnessTasks function is called from repeatedly from within the appropriate system loop. After the harness has initialized itself, its state machine will drop into a cycle that repeats for each test in the test list until it runs out of tests. Tests are executed sequentially, one after another. Messages are "printed" using the SYS DEBUG message functions (assuming that they are appropriately configured) and the harness accumulates results. When the last test is complete, the harness will report over-all test results and go idle. In a debug configuration, the "idle"state Tasks execution contains a hard-coded debug breakpoint where the harness will stop execution and return control to the debugger for analysis of the results.

While somewhat simplified for this discussion, the basic cycle for each test is shown in the state machine above diagram. The harness starts off by entering a Test Initialization state that calls the "Initialize" function for the current test, passing it any necessary initialization data. The harness then transitions to a Test Run state where it allows the test's Tasks function(s) to be called from the appropriate system loop so that the test's state machine can run. It stays in this state until the test calls the TEST_LibraryInitialize function to indicate that it is ready for the harness to initialize the library under test. When that happens, the harness transitions to a the Library Initialization state where it calls the "Initialize" function for each instance (up to three) of the library, passing in the necessary initialization data. Then, the harness enters the Test Start state where it call the test's "Start" function to instruct it to begin active testing of the LUT, after which it transitions to the Run Test and Library state where it allows both the test and the LUT state machines to run freely.

The test's state machine, consisting of up to five Tasks functions (limited by the MHC configuration settings), interacts with the API functions provided by the LUT and any other resources required to verify that the LUT is operating as expected. The test reports intermediate results by calling the TEST_HasPassedSubtest function, passing in "true" to indicate success and "false" to indicate a failure. The test can also "print" error or informational messages unsign the SYS DEBUG service, if it is appropriately configured. This continues, with the harness counting sub-tests that pass and fail, until the test either calls the TEST_HasCompleted function, reports that it is idle when its "Status" function is called, or the test harness timeout occurs (which indicates a test failure). When one of these events occurs, the harness transitions to the Library Deinitialization state where it calls the LUT's "Deinitialize" function and checks its status to verify that it deinitialized correctly. Finally, the harness transitions to the Test Complete state where it reports the final results of the current test, advances to the next test in the list, and repeats the cycle.

For each test, if all sub-tests pass (and no timeout occurs), the test harness counts that test as having passed. For each test, the harness itself reports a sub-test pass result if the LUT reports a SYS_STATUS_READY status (or greater) after being initialized and a SYS_STATUS_UNINITIALIZED status after being deinitialized. These are in addition to any sub-test results reported by the test itself. After all tests are completed (or any test fails), the harness "prints" overall test pass/fail results and goes idle.

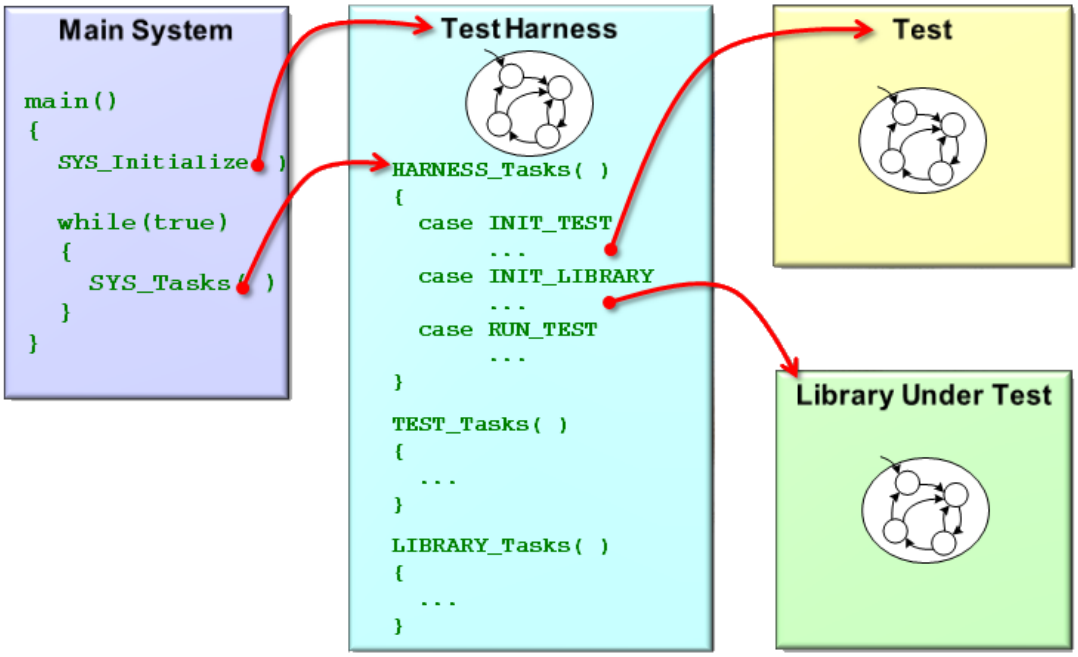
Test Execution Environments

Provides information on test execution environments

Description

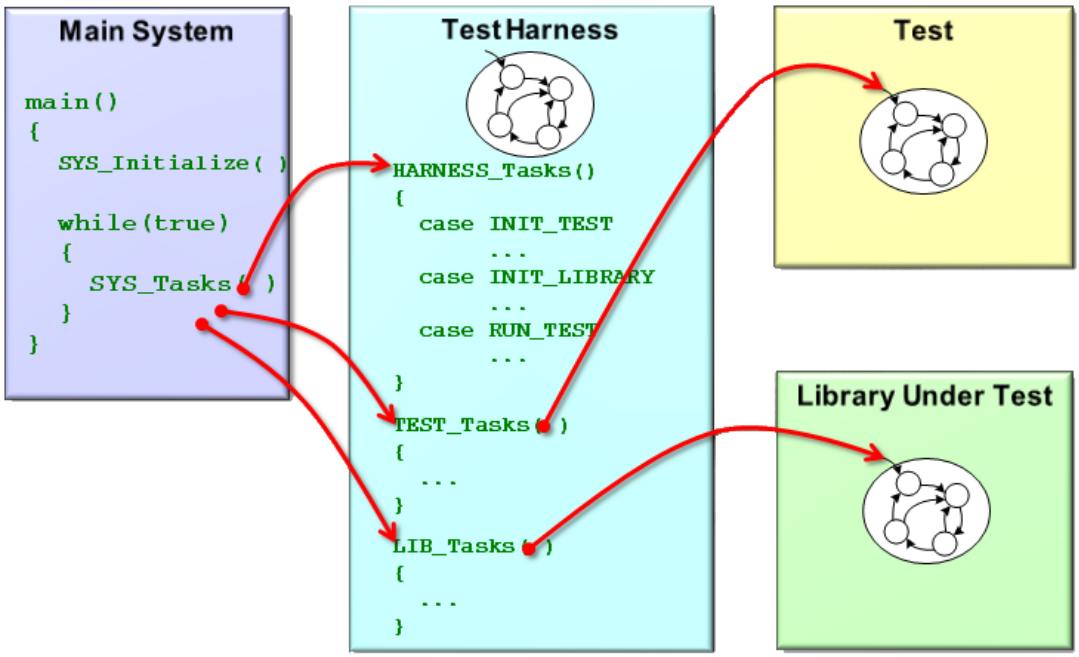
The Test Harness sits between the test and the normal MPLAB Harmony "main" function. In all configurations, the Test Harness controls initialization of the test application and the Library Under Test (LUT), as shown in the following diagram.

Initialization



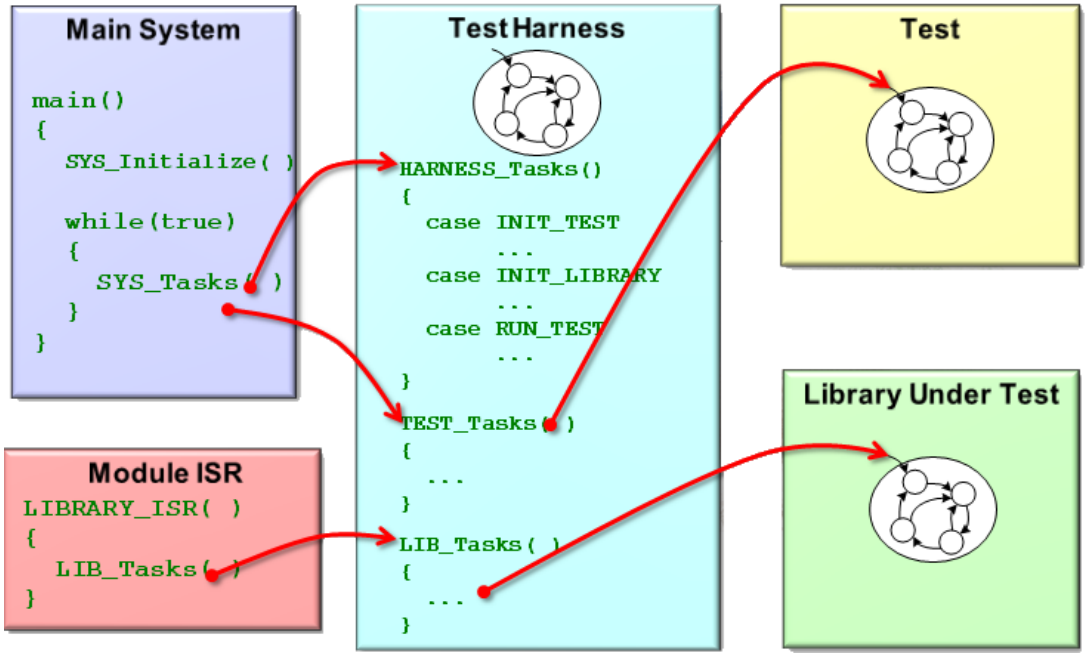
After initialization, the test harness state machine manages the execution of the test and the LUT. However, the test and LUT each execute within their own potentially different tasks environments, as described in the following sections.

Bare-Metal Polled Tests



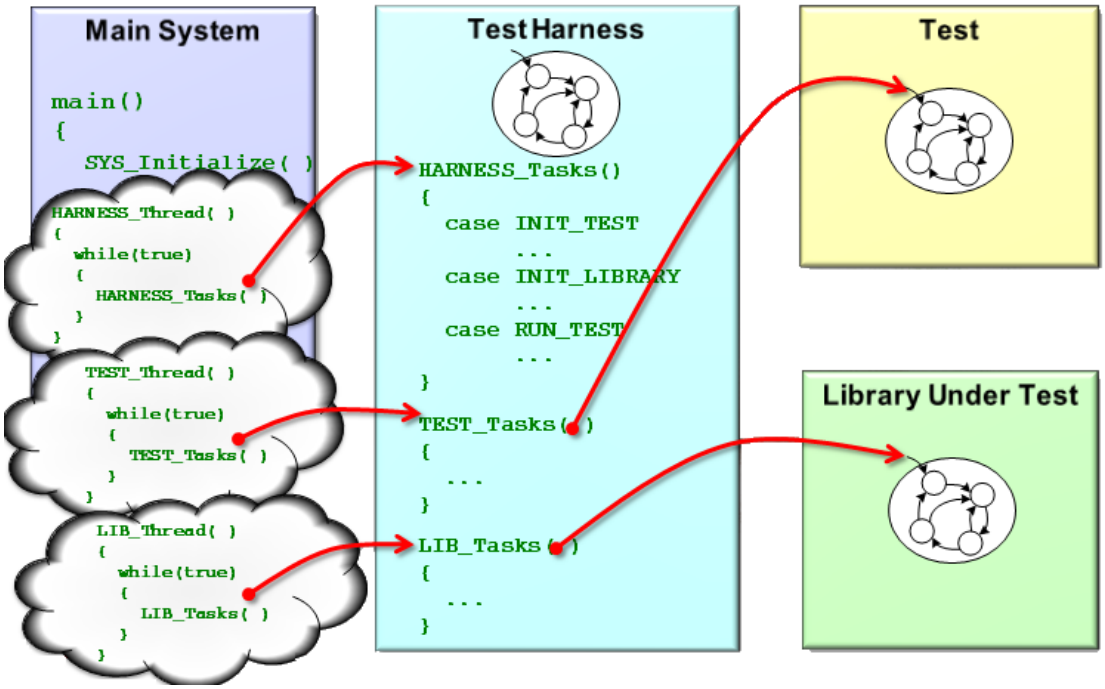
In a bare-metal polled configuration, all of the state machine tasks functions execute in the context of the main system loop. This includes the tasks function for the test harness, the test, and the LUT. However, the test and library tasks are only executed when the test harness state machine permits so that the harness can track and control them.

Bare-Metal Interrupt-Driven



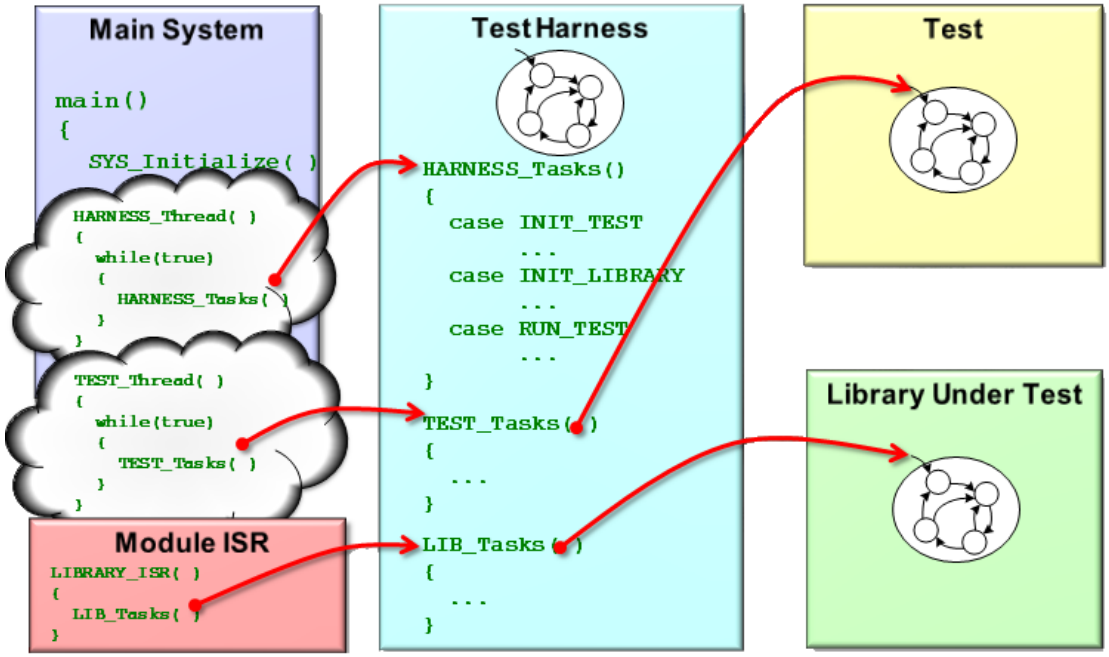
In an interrupt-driven test configuration in a bare-metal environment (with no OS), one or more of the tasks functions for the library under test may run within the context of an Interrupt Service Routine (ISR) that is defined for that peripheral module, depending on what is appropriate for the LUT. These ISR-ready Tasks functions are still called indirectly through the test harness so that it can capture spurious interrupts and manage the execution of the test and LUT.

RTOS-Based Polled



In an RTOS-based polled configuration, the Tasks functions for the test harness, the test, and the LUT are each called from its own RTOS thread loop. This supports testing of the OSAL mechanisms used to protect the shared resources used by the library when different clients call it from different thread contexts while it is running in its own thread context.

RTOS-Based Interrupt-Driven



In an interrupt-driven test configuration in an RTOS-based environment, one or more of the tasks functions for the library under test may run within the context of an Interrupt Service Routine (ISR) that is defined for that peripheral module, depending on what is appropriate for the LUT. As before (in the bare-metal, interrupt-driven environment), these ISR-ready Tasks functions are called indirectly through the test harness so that it can capture spurious interrupts and manage the execution of the test and LUT. But, in an RTOS environment, the test and the test harness Tasks functions are called from their own RTOS thread loops. Also, there may be RTOS-specific restrictions or requirements on the ISRs themselves.

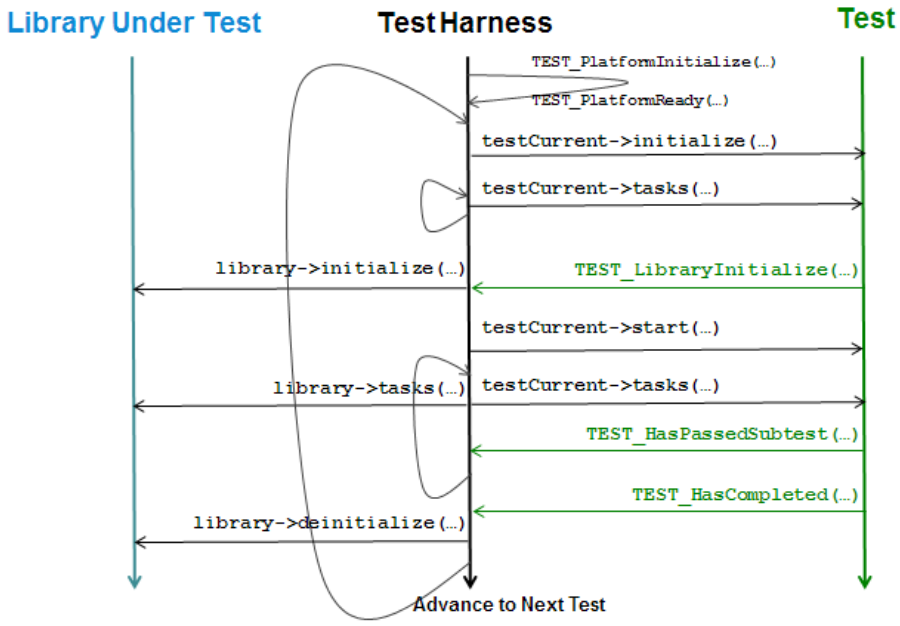
Harness, Test, and Module Interactions

Describes the interactions between the Test Harness, the test, and the LUT.

Description

There are a number of interactions between the Test Harness, the test application, and the LUT, as shown by the following diagram.

Test Harness Interactions



Before running any tests, the harness calls a configuration specific function: TEST_PlatformInitialize. If required, the system configuration can implement this function (usually in `system_init.c`) to initialize any hardware specific items such as timers or special I/O devices. If this process takes time to complete, the code that does so can call TEST_PlatformReady when it is finished. If there are no platform initialization requirements, then this process can be safely ignored as there is a weak implementation of the TEST_PlatformInitialize function built into the harness that simply returns "true" to indicate that the platform is already initialized.

Once the harness is ready, calls the current test's initialization function through a pointer provided in the TEST_DATA structure in the test list.

Then, it runs the test's state machine by calling its tasks function using a pointer from the same test data structure. When the test indicates it is ready for the harness to initialize the library (by calling TEST_ModuleUnderTestInitialize), the harness then calls the library's initialization function using a pointer to it provided in a LIBRARY_UNDER_TEST data structure. After successfully initializing the library, the harness starts the actual testing process by calling the test's start function (also using a pointer in the TEST_DATA structure).

After all the initializations are completed and the test has been started, the harness continues calling the tasks functions of both the test and the library (in potentially very different contexts, as described previously). While the test is running, it calls the TEST_HasPassedSubtest function, passing in a Boolean (“true” or “false”) to indicate the results of individual sub-tests. When the test is finished, it reports an idle status when its "Status" function is called (this is not shown in the diagram) or it can call the TEST_HasCompleted function to end the test prematurely. At that point, the harness deinitializes the library, stops running both the test and the library, and advances to the next test.

Configuring the Test Harness

Provides information on how to configure the Test Harness.

Description

The MPLAB Harmony Test Harness has a number of build options, primarily to configure the test time-out timer and OSAL idle time-out values. Normally, you will need to adjust the timer values to account for the selected clock frequency and for hardware resource conflicts if the default timer is used by something else in the system.

Additionally, you should configure the debug message service for the desired output method. If you do not have or do not want to use a debug output method, you should configure the SYS_DEBUG_PRINT and SYS_DEBUG_MESSAGE macros to “throw away” (ignore) the message parameter and hit a breakpoint if the debug error level parameter is greater or equal to SYS_ERROR_ERROR, as shown in the following code example.

```
// *****************************************************************************
/* Debug System Service Configuration Options

   Remarks:
    Configures debug print and message macros to execute a software breakpoint
    if the error level is equal or greater SYS_ERROR_ERROR.
*/

#define SYS_DEBUG_PRINT(level, format, ...)  \
        do{if ((level) <= SYS_ERROR_ERROR) SYS_DEBUG_BreakPoint();}while(0)
#define SYS_DEBUG_MESSAGE(level, message)    \
        do{if ((level) <= SYS_ERROR_ERROR) SYS_DEBUG_BreakPoint();}while(0)
```

For additional information on Test Harness configuration options, refer to the Test Harness Library.

Initializing the Test Harness

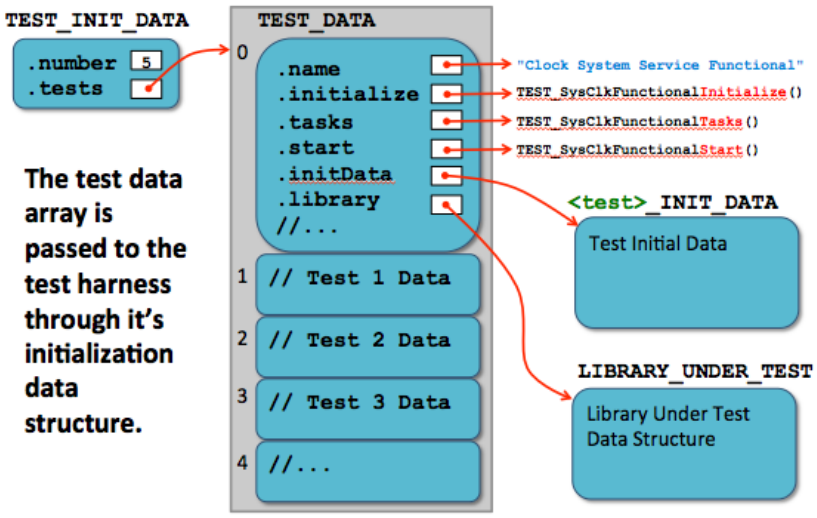
Provides information on initializing the Test Harness.

Description

The MPLAB Harmony Test Harness requires a number of properly initialized data structures to identify the tests it will run and the libraries they will test. The MHC will generate the necessary code to define these initialization structures in the `system_init.c` file of the current configuration. This section describes the relationships between these structures and the fundamental purposes of each.

Note: The Test Harness assumes a one-to-one relationship between a test and a LUT. When testing a stack of related libraries it is best to test them one at a time, starting at the bottom of the stack and working up to higher-level libraries. If testing a desired library requires other libraries to be running, then they will need to run outside of the control of the Test Harness in exactly the way they are normally configured by the MHC.

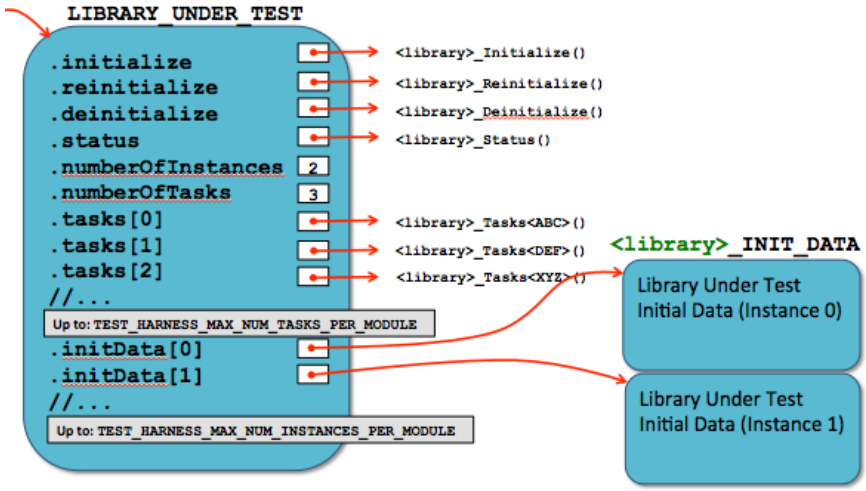
Test Harness Initialization Data



The test harness initialization data structure (TEST_INIT_DATA) contains the number of tests listed in the test data array and provides a pointer to the array. Each entry in the test data array is a TEST_DATA structure. This structure contains pointers to a display name string as well as the initialize, tasks, start, and status functions for an individual test instance. It also contains pointers to the test's own initial data structure (each test can have it's own "init" structure, just like any Harmony module) and to a "Library Under Test" data structure that identifies the library to be tested by that specific test. There are also few other data items in each structure in the array that are managed by the test harness itself to track the test's progress. So, this array must remain in RAM (i.e., it cannot be declared as "const").

The test's initialization data structure is a standard MPLAB Harmony system module "init" (SYS_MODULE_INIT) data structure that can be extended by the test itself to contain any required initialization data. The LUT data structure from the previous figure is shown in more detail in the following figure.

Module Under Test Data Structure



Each LIBRARY_UNDER_TEST structure contains enough information to initialize and run a single Library Under Test (LUT). It contains pointers to all system-interface functions of the LUT as well as pointers to each module instance's "Init" data structure. Since a module can have more than one tasks function, the LIBRARY_UNDER_TEST structure contains an array of pointers to the "Tasks" functions and an integer value identifying how many Tasks functions there are. Also, since a dynamic library can support multiple instances of itself, the LIBRARY_UNDER_TEST structure also contains an array of pointers to the LUT's "Init" data structure. Both of these array sizes are configurable in the source code, but they are values are effectively fixed by the requirements of the MHC templates.

While the MHC will generate the necessary code to initialize these structures, it is necessary to understand what they contain and what purpose each structure serves to effectively enter the appropriate data into the MHC . The data types are defined in the test harness (`<install-dir>/framework/test/test_harness.h`) header file, but they also use data types defined in other MPLAB Harmony system headers. The root of this data structure tree is the TEST_INIT_DATA structure, a pointer to which is passed into the TEST_Initialize function, which is called from the SYS_Initialize function.

It is generally best to start out with a single test add more as needed, later. The initialization of the number member of the TEST_INIT_DATA will adjust during the build if the MHC generated code is used. Be sure to give each test a meaningful name string in the test data structure so you can easily to identify each test in the debug output. The test harness uses the pointers to the initialize, tasks, start, and status functions in the sample test data structure to execute the test, so be sure that these match the names of the functions defined by your test. (**Note:** These can be given as NULL if you want to do an initial build test before you have implemented your test.)

During test development, leave the index member at zero (0) and define the initData member as NULL. You can initialize them later, once you

have implemented your test and defined its initialization data structure type. Of course, this structure will also need to be initialized in the `system_init.c` file before it is referenced by the TEST_DATA structure, test, the MHC has no knowledge of it so you will have to initialize it yourself manually.

If your test requires knowledge of the initial data passed to the library under test (and it will if it verifies the initial settings of the library), the LIBRARY_UNDER_TEST data structure will need pointers to the "init" data structures for each instance of the library. Of course, these will need to be cast to SYS_MODULE_INIT structure pointers; just like they are when they are passed to the library's "initialize" routine. Notice that the tasks member is an array of pointers to the LUT's tasks functions. It must have exactly as many initialized entries as specified in the numberOfTasks member. This number is limited to no more than the value defined by the TEST_HARNESS_MAX_NUM_TASKS_PER_MODULE setting. Unfortunately, this value is not configurable from within the MHC as it is restricted by MHC templates. However, it can be modified by hand in the source code.

After initializing these data structures, it is a good idea to do a test build and verify that the project has no errors or warnings. You can also debug the test harness state machine and verify that it recognizes and executes your test. If the harness is executed with no test list data, it will report an error and to into the idle (APP_TH_STATE_IDLE) state, providing a failure result (harness.result == false).

Creating a New Test Harness Project

Describes the process to create a new MPLAB Harmony Test Harness project.

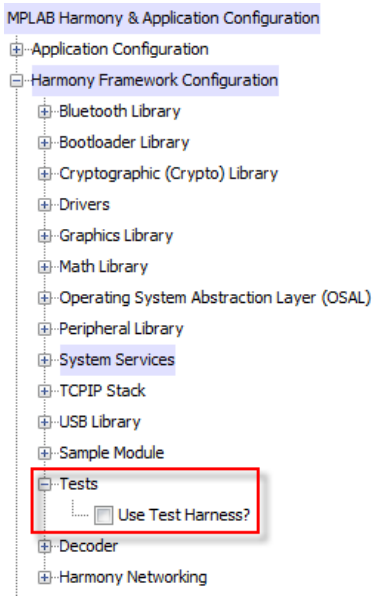
Description

A MPLAB Harmony test harness project can effectively integrate a number tests for multiple modules. However, it is not efficient to attempt to maximize the number of tests within a single project. It is better to create a comprehensive test plan that defines the appropriate test requirements and use it to identify a strategic set of tests, test projects, and project configurations to validate a specific library.

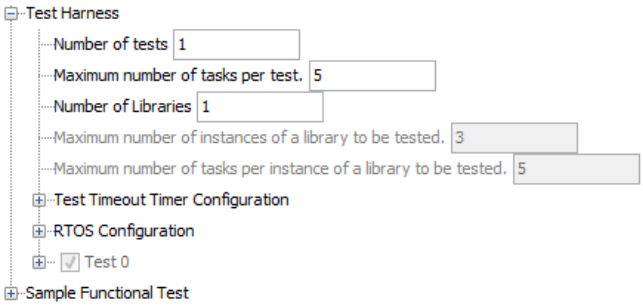
To create a new MPLAB Harmony test harness project, perform the following steps.

Note: These steps are listed sequentially, but it is usually best to work incrementally. Generate your initial configuration with a minimum of changes to the defaults and start developing your test. As you complete the test development, come back to the MHC and add the new test functions and information, regenerate, the configuration, then build, and debug your test.

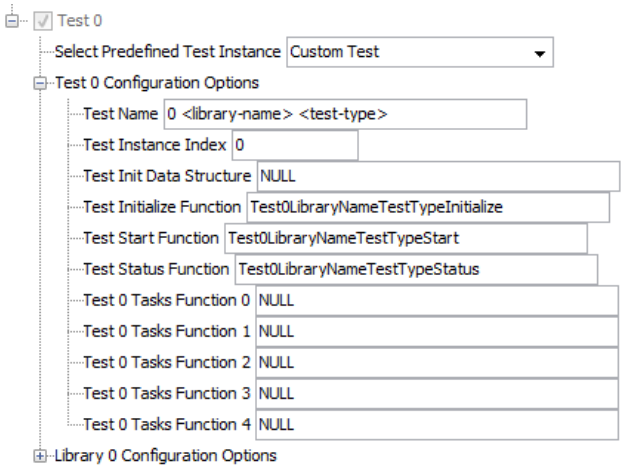
1. Using MPLAB X IDE, create a new MPLAB Harmony project.
   - Be sure to select the appropriate MPLAB Harmony installation and PIC32 processor
   - Rename the project and its initial (default) configuration to something appropriate for your test purposes
   - Be sure to set the new project as the "main" project
   - Set optimization level to zero (–O0) to facilitate debugging. Clear the “Enable recommended compiler optimizations” option in the "File Generation" dialog when you click **Generate** in the MHC.
   - It is usually easiest to start with a bare-metal, polled configuration as that is the easiest to debug
2. Open the MHC and use it to create the initial test configuration.
3. Add and configure the library under test and any required dependencies.
4. Select **Use Test Harness?** within the *Harmony Framework Configuration* > *Tests* options.



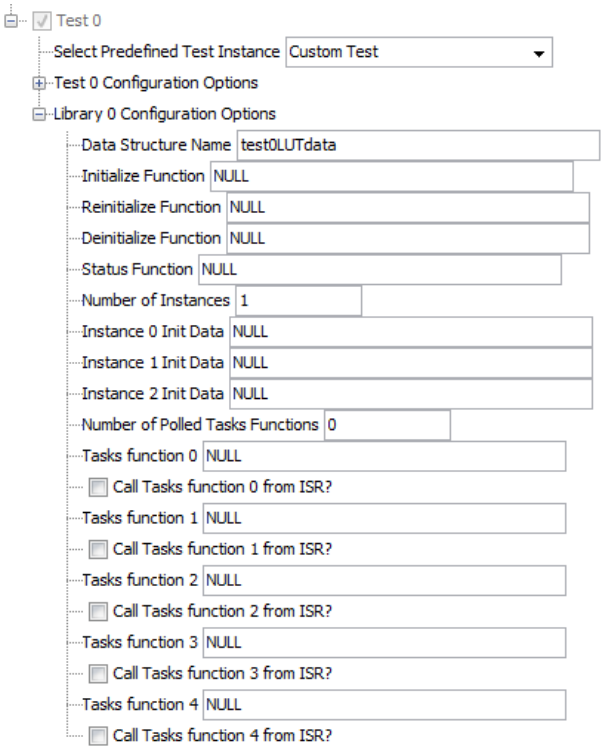
5. Configure the test harness. Refer to the online Help associated with each option for a description of what it means. The test harness directly utilizes a timer (via the Timer PLIB) to provide a timeout capability and OSAL functions, even if no RTOS is selected.



6. Configure each test, entering the names of the test, its required functions and data structure and other information. Note that the MPLAB Harmony installation does provide some predefined tests. These can be selected and used to test supported pre-existing libraries. However, you will normally need to configure a "Custom Test" and fill in all of the required information manually as no predefined test will be provided for a newly created library.



7. Enter the library configuration options for the library under test that is associated with each test. This will include the library's function names and initialization data structure name as well as information on the number of instances, polled and interrupt-driven tasks functions (and associated interrupt details).



8. Generate the configuration. This will insert the test harness library into your project and generate the necessary test list, initial data structures and other configuration code. You may need to make updates to these later as you develop your tests so that you can easily execute them under control of the test harness.

Example

```
/* Test List */
```

```
static TEST_DATA testList[] =
{
    {
        .name       = "Sample Module Functional Test 0",
        .initialize = TEST_SampleFunctionalInitialize,
        .tasks      = {
                        TEST_SampleFunctionalTasks,
                        TEST_SampleFunctionalTasks,
                        NULL,
                        NULL,
                        TEST_SampleFunctionalTasks,
                      },
        .start      = TEST_SampleFunctionalStart,
        .status     = TEST_SampleFunctionalStatus,
        .index      = 0,
        .initData   = (SYS_MODULE_INIT *)&testSampleInit,
        .library    = &testSample0LUTdata
    }
};

/* Test Harness Initialization Structure */
static const TEST_INIT_DATA testData =
{
    .numberOfTests  = sizeof(testList)/sizeof(TEST_DATA),
    .tests          = testList
};
```

9. Implement your test as required to test the desired library.
10. Create new project configurations for each new execution environment, hardware platform, or test configuration.

After each configuration change, you should do a test build. If you have entered names for functions or data structures that do not yet exist, you will see build errors for them. If you do not enter any new information, the MHC will place "NULL" pointers in those locations and the project will build, but of course, the harness will not call your test until it has the required configuration information.

Developing a Test

Provides information on developing a test application.

Description

A test should exercise the API of the library under test and verify that the expected results occur. Doing so may require significant development effort. It may also require development of specific supporting hardware. Each test can have any number of sub-tests. A sub-test is counted any time a sub-test result is reported to the test harness by calling the TEST_HasPassedSubtest function with a true or false parameter value. The test harness keeps a count of every time this function is called and a count of how many times it was called with a true parameter. If these two numbers are equal, then the harness counts that the test has passed once it has completed. To indicate that the test has completed, the test application can either call the TEST_HasCompleted function or report a SYS_STATUS_TEST_COMPLETED status and enter its idle state. Once either of these conditions is true, the harness will no longer call the test's tasks function and the test effectively stops running.

A test consists of the following primary functions and any supporting code to exercise and verify the operation of the Library Under Test (LUT).

- Test_<test-short-name>Initialize
- Test_<test-short-name>Tasks
- Test_<test-short-name>Start
- Test_<test-short-name>Status

The Initialize function places the test in its initial state and initializes any internal data, but it does not interact with the LUT. The Tasks function maintains the test's state machine and performs the actual test. And, the Start function tells the test when the LUT has been initialized and it is safe to interact with it.

It is easiest to develop a test by starting from an existing sample test and modifying it to suit your needs. Sample test files are in the MPLAB Harmony installation in the following locations.

- `<install-dir>/framework/test/test_sample_functional.h`
- `<install-dir>/framework/test/src/test_sample_functional.c`

Test Development Process

To get started developing a test application perform the following steps.

1. Copy the `test_sample_functional.c` and `test_sample_functional.h` files from the locations listed previously.
2. Rename these files to reflect the short name of your test and add them to your project.
3. Edit the test's `.c` file, as follows:
   - Rename any references to the sample module with references to the library under test

- Be sure to change the name of the test's header file in the `#include` statement
- Be sure to change the state enumeration names and local data type names
- Delete the *GIVE_DATA" and *GET_DATA" states and the functions that implement them. They are not needed and will be replaced by states that you will define. (Keep the *INIT, *INIT_WAIT, *INITIAL_DATA, *COMPLETED, and *IDLE states. Normally you will reuse these.)
- Delete all members of the *LIB_DATA structure, except the state and initData member. You will reuse them.
- Rename the testSampleFunc static global to variable to represent the short name of your test
- In the VerifyInitialData function, replace the internal logic of this function with a single call to TEST_HasPassedSubtest(false) and return the new <test_STATE_COMPLETED state. You will need to implement this function appropriately later.
- Delete the AbleToSendNewData and AbleToReadSentData functions. You will not need them.
- Rename the TestSampleFuncionalInitialize function to reflect the short name of your test and delete unneeded initialization coded
- Rename the TEST_SampleFunctionStart function to reflect the short name of your test
- Rename the TEST_SampleFunctionTasks function to reflect the short name of your test and delete code for unneeded states transitions

4. Edit the test's `.h` file.
   - Rename any references to the sample library with references to the library under test
   - Be sure to change the name of the _TEST_SAMPLE_FUNCTIONAL_H macro
   - Delete the inclusion of the `sample_module.h` file, including the API header for the library under test instead
   - Rename the test application's interface functions (both the prototype and the comment block) to match the new names given in the `.c` file
5. Rebuild the project and fix any build errors.
   - The sample function test was designed to test multiple instances of the sample library. If your test is also designed to test multiple instances of the LUT, then you can rename and use the <test>_LIB_INSTANCES_NUMBER configuration option. But, be sure to define it in the `system_config.h` file to set internal array sizes.
   - Fix any editing errors and change any references to the sample library to refer to the desired library under test
6. Once you have a clean build of a test application skeleton, you need to add it to the test application list as described in Initializing the Test Harness and begin developing the test to exercise and validate the module under test.

Reviewing Results

Provides information on reviewing results.

Description

The Test Harness accumulates results given to it by the tests. If SYS DEBUG output is supported and configured, then results will be displayed textually on the debug monitor terminal used. However, depending upon the testing needs or the hardware used, it may not be possible to support a debug output method. In this case, it is necessary to utilize the debugger to analyze the TEST_HARNESS_DATA structure to determine the results of the test. The following members of this structure give the over-all results once the test harness enters its idle state.

- testsCount - This is the total number of test applications executed
- testsPassed - This is the number of test applications that reported no sub-test failures, resulting in a passing result
- result - This is the final overall result, which is true (1) if all sub-tests in all test applications passed; otherwise, false (0) is returned

Note: Using debugger scripting, it is possible to automate the programming, execution, and results analysis process so that automated testing may occur.

Adding the Test Application to the Test List

Description

In order for the Test Harness to initialize and run a test application (and the module it tests), it must be added to the test list. To add a test application to the test list, you must create and initialize specific members of the TEST_DATA structure that makes up that test's entry in the test list array, as shown in the following example.

Example: Array of Tests

```
// *****************************************************************************
/* Array of Tests

   This is an array of structures that provide data necessary for the harness to
   run a test.  It lists all tests that the harness will run
*/

static TEST_DATA tests[] =
{
    {
        .name       = "USART Driver Functional",
        .initialize = TEST_DrvUsartFuncInitialize,
```

```
        .tasks      = TEST_DrvUsartFuncTasks,
        .start      = TEST_DrvUsartFuncStart,
        .index      = 0,
        .initData   = (SYS_MODULE_INIT *)&testDrvUsartFuncInitData,
        .library    = &testDrvUsartFuncData
    }
};
```

The name member is a pointer to a string that contains the display name of the test, when debut output is supported. The initialize, tasks, and start members are pointers to the initialize, tasks, and start functions (respectively) of the test application. The index member identifies to which instance of the test application this test list entry refers. (For now, this is normally zero.) The initData member is a pointer to a structure that holds any initialization data required by the test application. In the sample test this structure is used to obtain pointers to the initialization data passed to the LUT. And finally, the library member is a pointer to a LIBRARY_UNDER_TEST structure (described below) that contains information about the LUT.

The LIBRARY_UNDER_TEST structure must be initialized to provide all the necessary information about the system interface of the library under test so that the harness can initialize and run the library, as shown in the following example.

Example: Initializing the Library Under Test

```
/* USART Driver Module */
LIBRARY_UNDER_TEST lutDriverUsart =
{
    .initialize         = DRV_USART_Initialize,
    .reinitialize       = NULL,
    .deinitialize       = DRV_USART_Deinitialize,
    .status             = DRV_USART_Status,
    .numberOfInstances  = 2,
    .numberOfTasks      = 3,
    .tasks              = {
                            DRV_USART_TasksTransmit,
                            DRV_USART_TasksReceive,
                            DRV_USART_TasksError
                          },
    .initData           = {
                            (SYS_MODULE_INIT *)&drvUsart0InitData,
                            (SYS_MODULE_INIT *)&drvUsart1InitData
                          }
};
```

In the previous example, the LUT is the USART driver. It has initialize, deinitialize, and status functions as well as three tasks functions (DRV_USART_TasksTransmit, DRV_USART_TasksReceive, and DRV_USART_TasksError). Pointers to these functions are given as initialization values for the initialize, reinitialize, deinitialize, and tasks array members of this structure. The number of entries in the tasks array are configurable by defining the TEST_HARNESS_MAX_NUM_TASKS_PER_MODULE option and the number of tasks functions that must be called to support the LUT must be identified by initializing the numberOfTasks member, as shown above. Lastly, the number of instances of the LUT to be tested must be identified by initializing the numberOfInstances member and pointers to the initialization data of each instance (in order, starting from instance 0) must be given as initialization values of the initData array (also starting from entry 0 and going in order). The number of possible entries in the initData array is defined by the TEST_HARNESS_MAX_NUM_INSTANCES_PER_MODULE configuration option.

Finally, a pointer to the test list array is then passed into the initialization structure (TEST_INIT_DATA) of the Test Harness in the `system_init.c` file to make all of this information available to the Test Harness, which then passes it to the test application and the LUT.

Once the test list and all associated data structures are initialized properly, the Test Harness can properly manage the test application and its associated module under test. To add additional test applications and LUTs, add additional entries to the test list and initialize them similarly. If you use the method shown in the Test Harness examples to calculate the size of the test list array the compiler will automatically calculate the number of entries in the array.

Note: The Test Harness assumes a one-to-one relationship between a test and a library under test. When testing a stack of related libraries it is best to test them one at a time, starting at the bottom of the stack and working up to higher-level libraries. If testing a desired library requires other libraries to be running, these libraries will need to run outside of the control of the Test Harness in exactly the way they are normally configured by the MHC.

Adding Additional Configurations

Provides information on adding additional configurations.

Description

It is usually easiest to create a bare-metal polled configuration first because you can more easily debug the harness, test, and library state machines. Once you have a good functional test that passes when running polled, add an interrupt driven configuration and ensure that the library works as expected when interrupt driven. If the library under test does not have an associated interrupt, a timer interrupt can be used to simulate an interrupt driven mode as is done in the “sample” module. While this is not strictly necessary for libraries that are not intended to run in an interrupt-driven mode, it can help make the library more robust.

Once bare metal testing is completed, create RTOS-based configurations and test the library in a polled, multi-threaded environment. Verify thread safety and proper blocking operation and experiment with different relative thread priorities between the test and library threads. Then, add another configuration to test the library running interrupt-driven in an RTOS environment. There are unique issues possible in each of these environments and testing in all of then will create the most reliable library. Also, while use of the MPLAB Harmony OSAL will help make your library RTOS-independent, but it is best to create test configurations for every supported RTOS to verify that there are no resource conflicts or unique issues between the library and a specific supported RTOS.

Refer to Tips and Tricks for the process to add additional configurations.

Test Requirements

Describes test requirements.

Description

While this user's guide does describe how to use the MPLAB Harmony Test Harness to control and execute a test within the different execution environments supported by MPLAB Harmony, it does not describe the requirements of that test.

The MPLAB Harmony compatibility requirements are described in the MPLAB Harmony Compatibility Guide, in the What is MPLAB Harmony? section of the help documentation in the `<install-dir>/doc` folder included in the MPLAB Harmony installation. In that same folder is a fillable PDF form (`harmony_compatibility_worksheet.pdf`) that contains a compatibility checklist worksheet that you can fill out to track and report testing methods and results.

Tips and Tricks

This topic provides a few helpful tips and tricks to keep in mind while developing your MPLAB Harmony projects.

Description

Tip 1: Using Example and Demonstration Projects

You can use MPLAB Harmony example and demonstration projects to put together a "first approximation" of your application:

1. Copy the source files from the various example or demonstration applications to your project.
2. Rename the APP_Initialize and APP_Tasks functions so that they are all different and will build in your application by simply appending a descriptive name to the end of the existing function names.


Note: You may also need to rename any global data "objects" that conflict with each other from the examples.

3. Call the individual APP_Initialize<new_name> and APP_Tasks<new_name> functions from the SYS_Initialize and SYS_Tasks functions in your application's configuration files.
4. Use the MHC to add any required drivers, services, or middleware, as described in **Tip 2**.
5. At this point, your application consists of several independent state machines. Now, you can begin to develop your own custom application code and modify any of the code copied from the provided applications to behave as you desire.

Tip 2: Adding Additional Configurations

To most easily create a new configuration, copy an existing one using these steps.

1. Duplicate the configuration using the normal MPLAB X IDE method.
   - Right-click the project and choose **Properties**
   - Click **Manage Configurations** to open a dialog box that allows you do duplicate configurations
2. Rename the new configuration copy and set it as the active configuration.
3. Open the MHC, click **Configuration** and save the current configuration using the default name.
4. Click **Configuration** again and browse to and open the previous configuration's .mhc file.
5. Using the MHC, select any new configuration items that should be changed for the new configuration.
6. Save the modified configuration to the new configuration's .mhc file. You will need to issue a **Save As** command because the MHC remembers the file you opened last. Be careful to not overwrite the original configuration file.
7. Generate the new configuration.
8. Add any new files required and/or use a "diff/merge" utility to merge any required manually-edited code from the previous configuration to the new one.
9. Be sure to update the compiler's include file search paths to use the new configuration folder before building the new configuration or the build will use the previous configuration header files and cause errors or incorrect behavior.

Individual configurations have to be built, debugged, executed, and evaluated individually.

Tip 3: Operating System Abstraction Layer (OSAL)

By design, the OSAL is a minimal API intended only to enable thread-safe operation for MPLAB Harmony libraries. It only exposes a very small subset of the capabilities of an operating system so that MPLAB Harmony libraries can use semaphores, mutexes, and critical sections (and a few other things) necessary to protect shared resources (data structures, peripheral registers, and other memory objects) from corruption by unsynchronized access by multiple threads. This is done to allow MPLAB Harmony libraries to be made compatible with the largest variety of operating systems, by using a minimal subset of the some of the most common OS features. The OSAL is not intended to provide a complete abstraction of an RTOS, which is what you will require to implement a complete application. Abstracting an entire operating system is a much more complex task that is roughly equivalent to defining your own RTOS. It is important to understand that the OSAL has not done that.

Index