MPLAB® Harmony Help - MPLAB Harmony Compatibility Guide

MPLAB Harmony Integrated Software Framework v1.11

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.

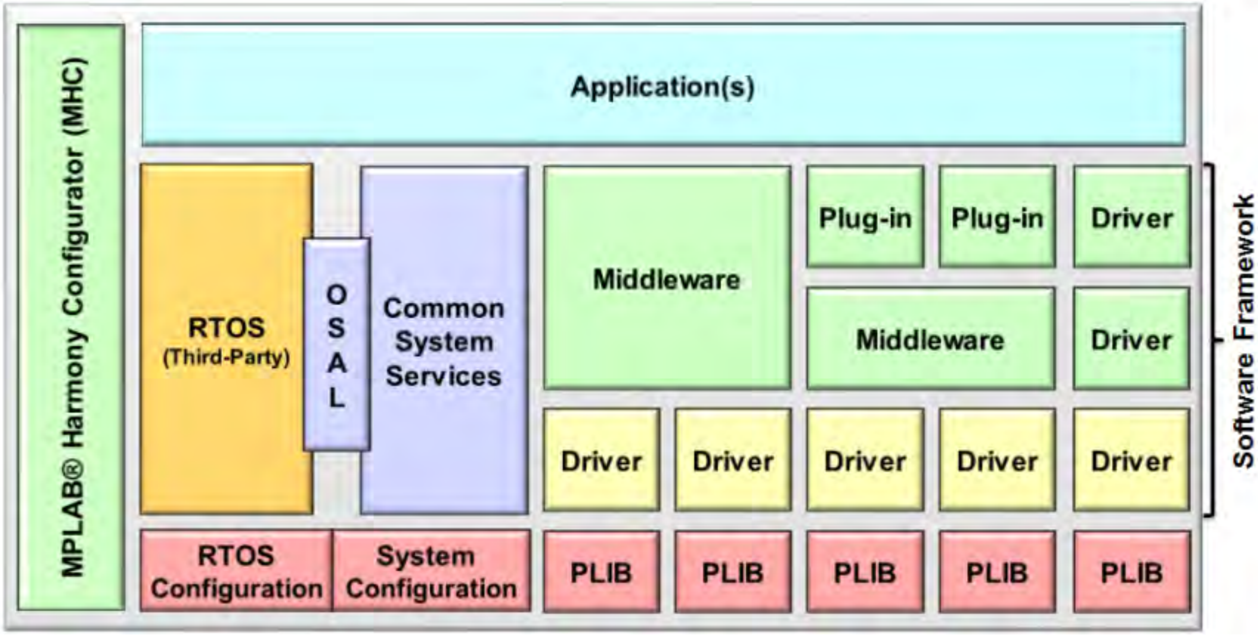
A range of compatibility and compliance to these guidelines is possible. One key example is that it is not necessary that a library support all possible execution models to be considered MPLAB Harmony compatible. However, it does affect the level of flexibility supported and restrictions must be clearly 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.

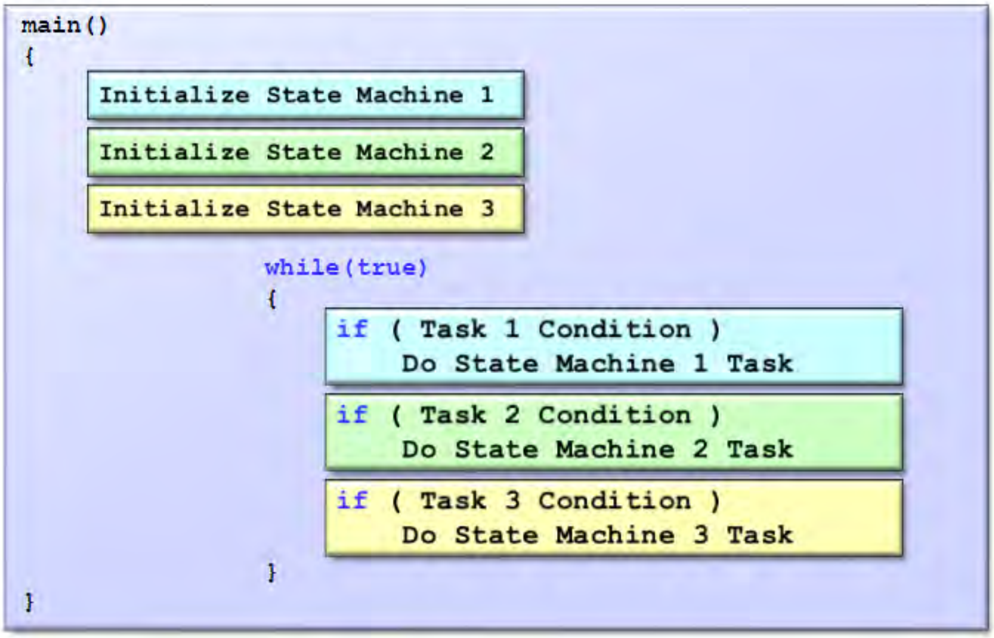
2.2: Execution Model

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



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.

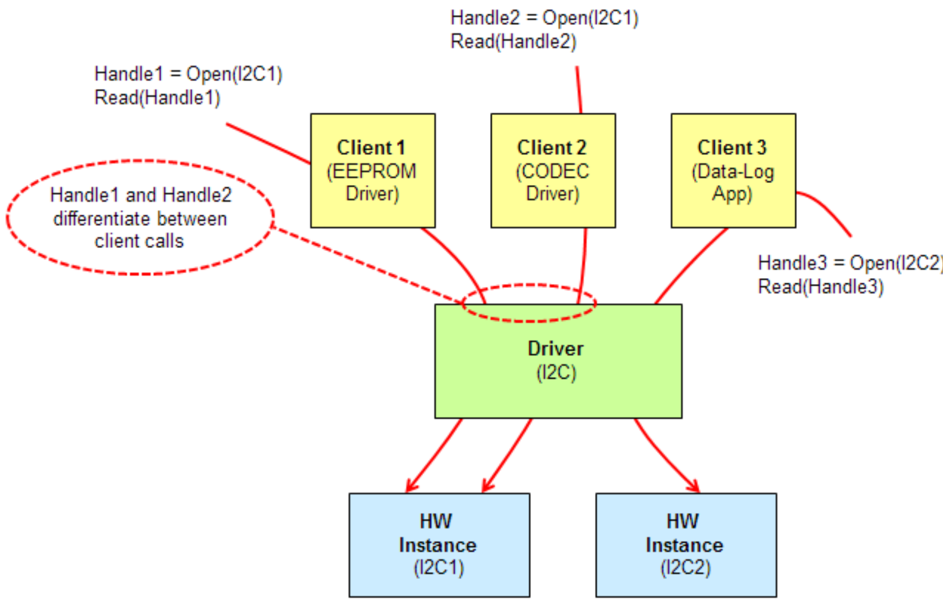
3.3.1: Protect Owned Resources From Accesses by Multiple Clients

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

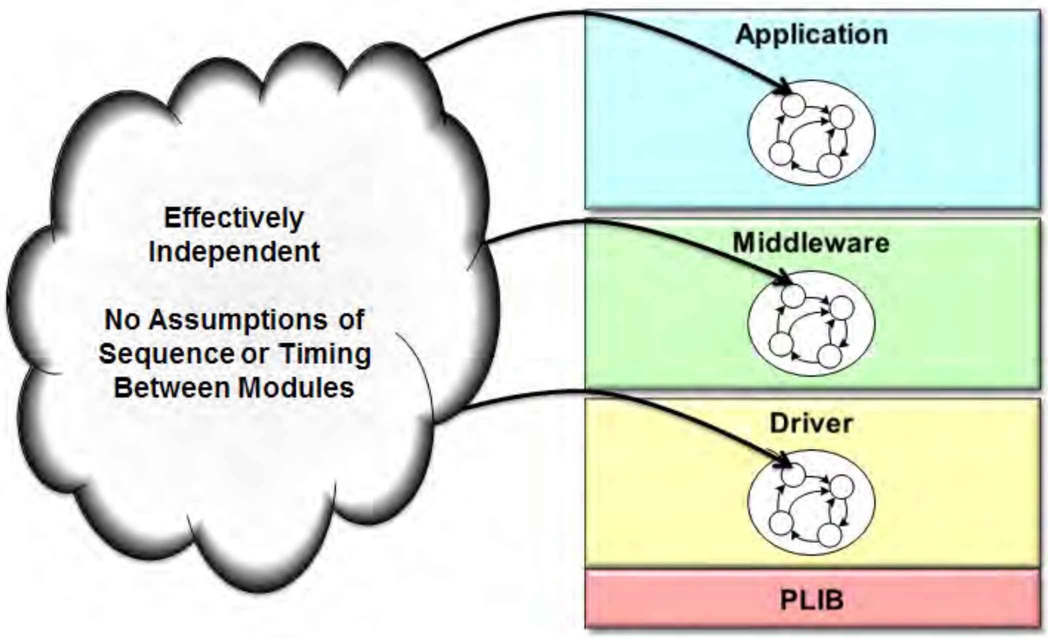
This describes the MPLAB Harmony system module model.

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.

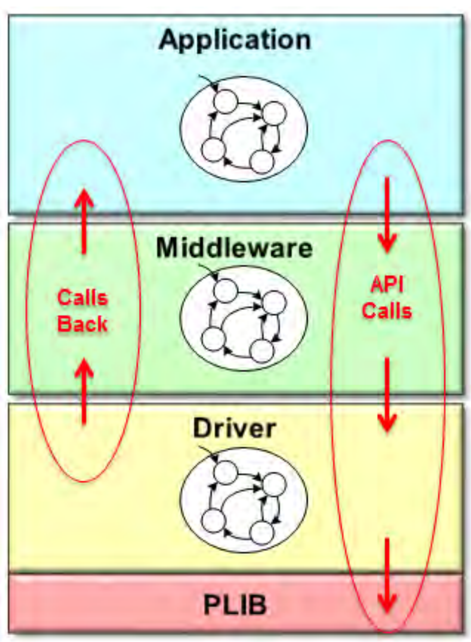
3.6: Drivers Use Driver-Client Model

This describes the Driver-Client model used by drivers.

Description

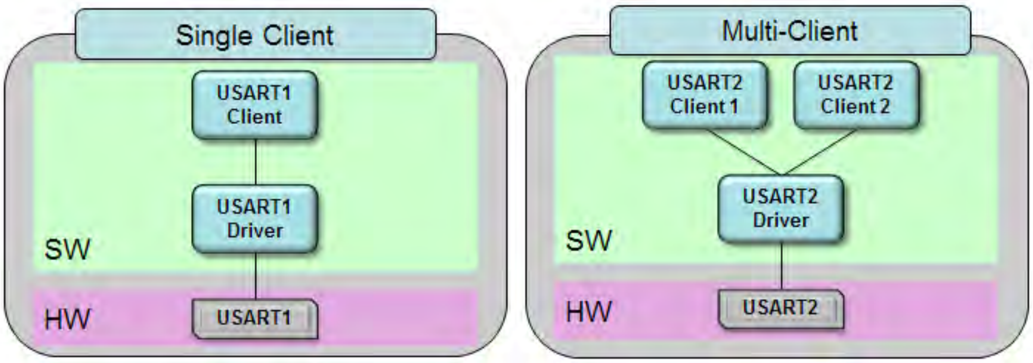
All libraries implement a vertical "client" interface through which clients of the library interact with it. This allows libraries, especially active modules in the system, to be "stacked" so that their state machines work together.

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.

3.8.4: Graphics Display Drivers Use the Graphics Display Driver Model

This topic describes the Graphics Display Driver model.

Description

Modules that implement a graphics display driver to be used with the MPLAB Harmony graphics library must provide a display driver interface as required by the graphics primitives layer. Refer to the `<install-dir>/framework/driver/gfx/drv_gfx_display.h` header file and the Graphics Driver Library 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.

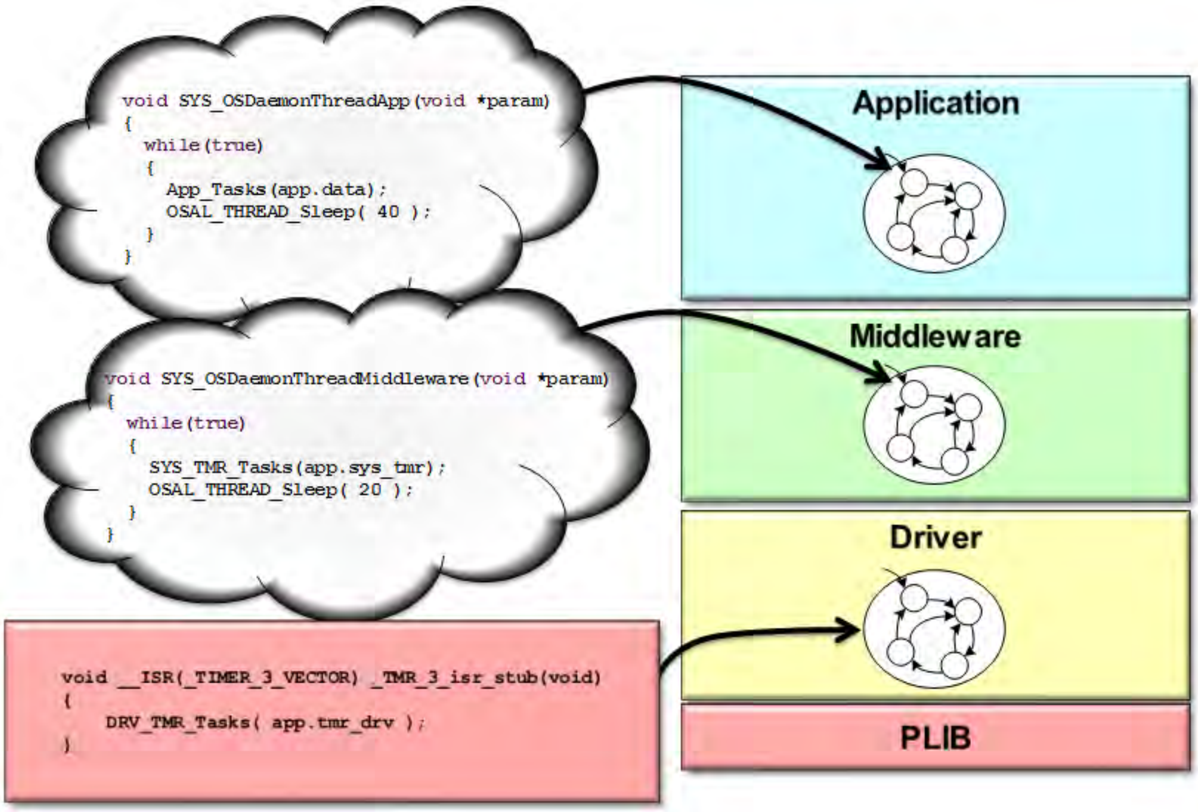
Description

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).

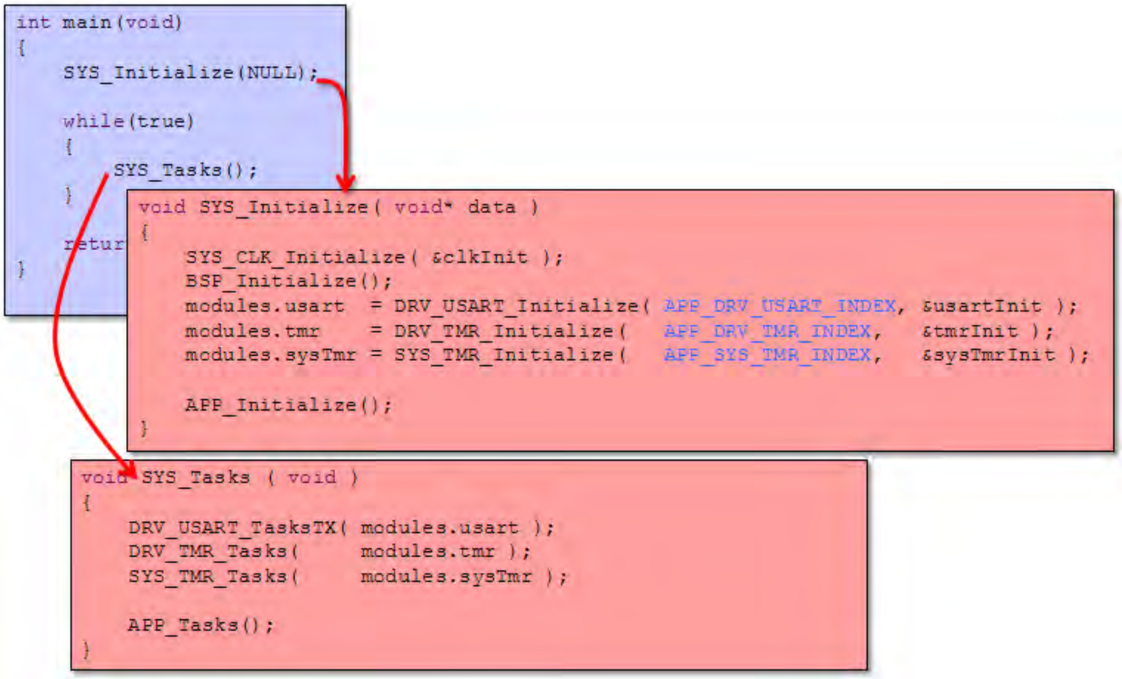
4.1.3: Polled Execution in a Super Loop With No RTOS

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



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.

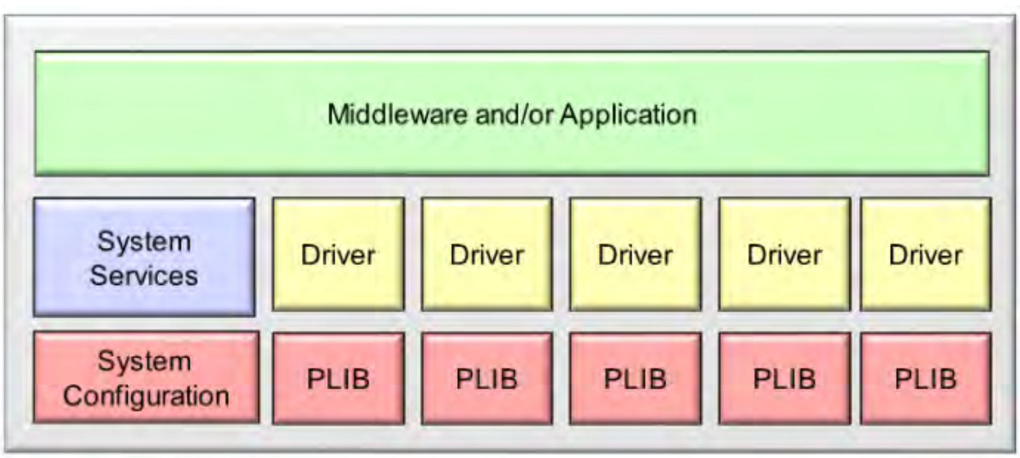
Description

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

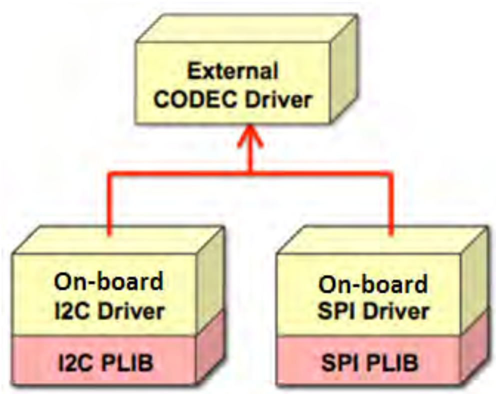


Note: Some driver or system services features may not be supported on specific microcontrollers as MPLAB Harmony libraries do not usually emulate hardware features in software.

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.

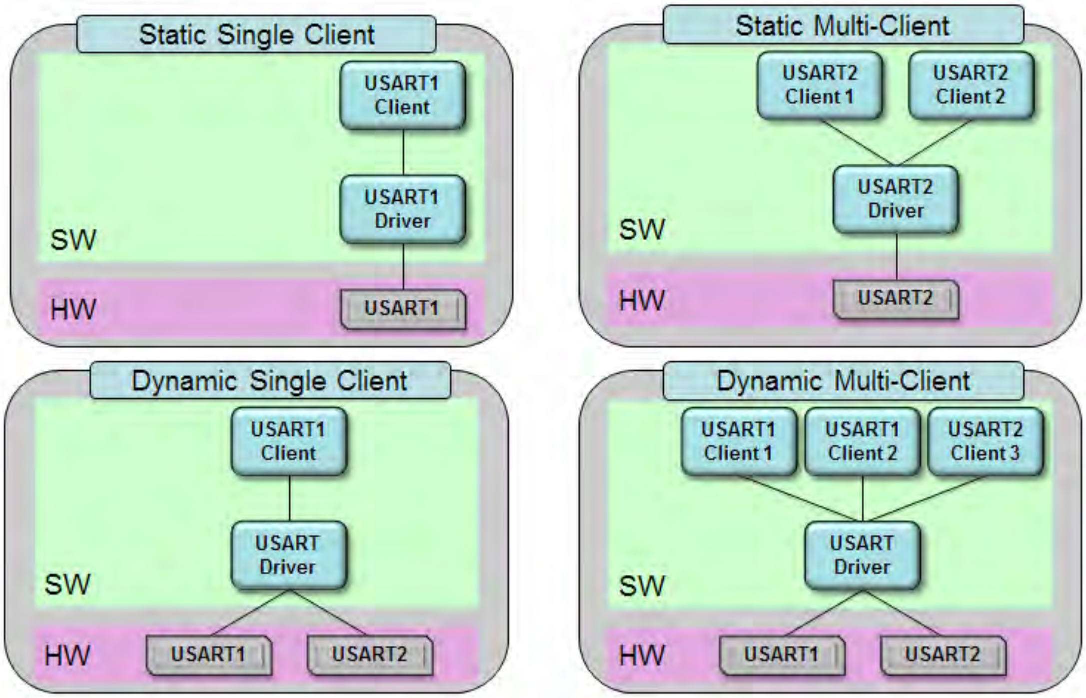
4.3: Support a Dynamic Interface

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:		

Index