



MPLAB® Harmony Help - MPLAB Harmony Driver Development Guide

MPLAB Harmony Integrated Software Framework v1.11

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);

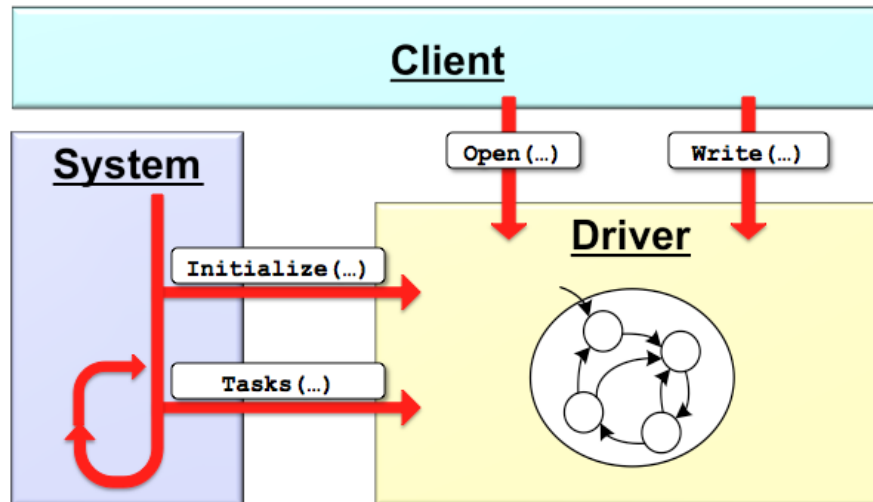
    while( true )
    {
        SYS_Tasks();
    }

    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 managers, 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_INITIALIZE;
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 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	<code>SYS_MODULE_POWER_OFF</code>	Module Power-off state code.
1	<code>SYS_MODULE_POWER_SLEEP</code>	Module Sleep state code.
2	<code>SYS_MODULE_POWER_IDLE_STOP</code>	Module Idle-Stop state code.
3	<code>SYS_MODULE_POWER_IDLE_RUN</code>	Module Idle-Run state code.
4 through 14	<Module-specific Definition>	Module-specific meaning.
15	<code>SYS_MODULE_POWER_RUN_FULL</code>	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_INITIALIZE;
```

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: <ul style="list-style-type: none"> • 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: <ul style="list-style-type: none"> • DRV_IO_INTENT_NON_BLOCKING – Routines return immediately • DRV_IO_INTENT_BLOCKING – Routines return after operation is complete One of: <ul style="list-style-type: none"> • 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.

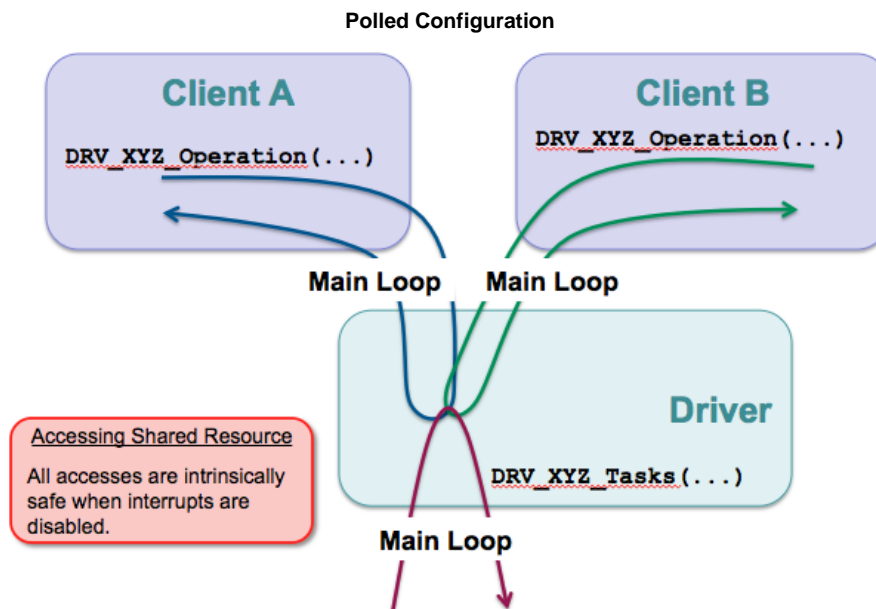
Atomicity

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.



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.

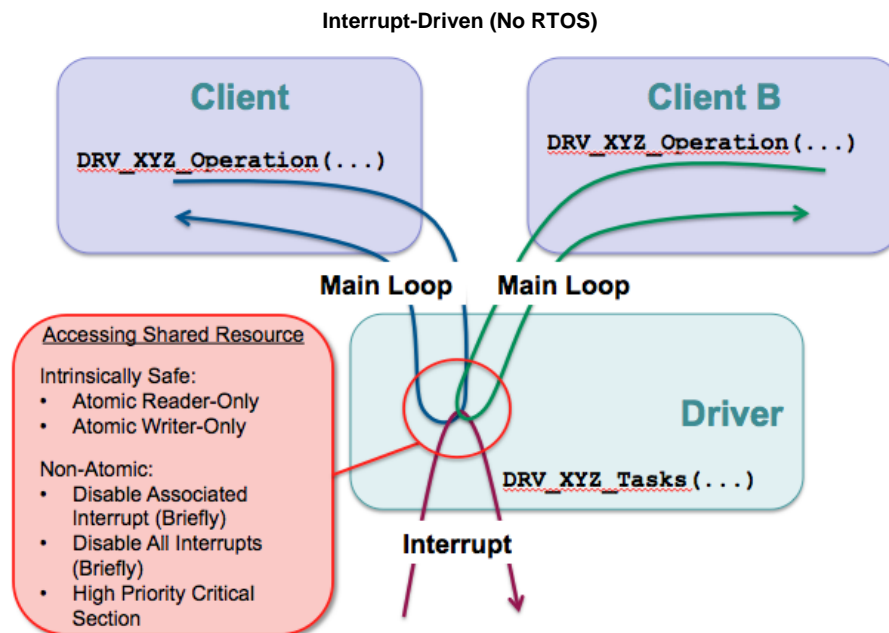
Even if a data type is atomic (only one word or less in size), it may still be necessary to protect a code sequence that performs a read-modify-write operation on it because almost all read-modify-write operations require execution of multiple instructions, which could be interrupted. This concern applies 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.



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.

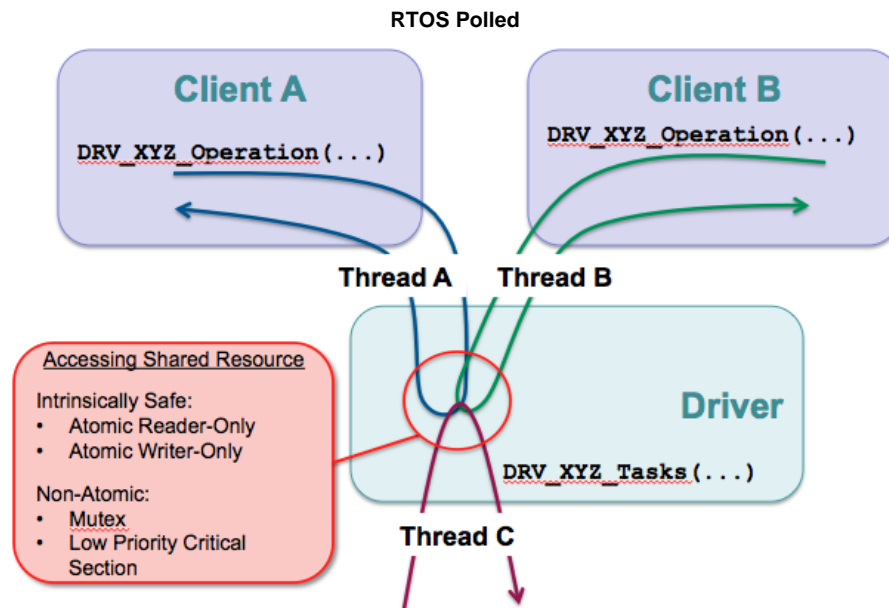
This method works to ensure safe access to shared resources without disabling interrupts globally when using a bare-metal configuration. Of course, if a section of code must be truly atomic (uninterruptible), interrupts can be globally disabled for a short period of time using either the Interrupt System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

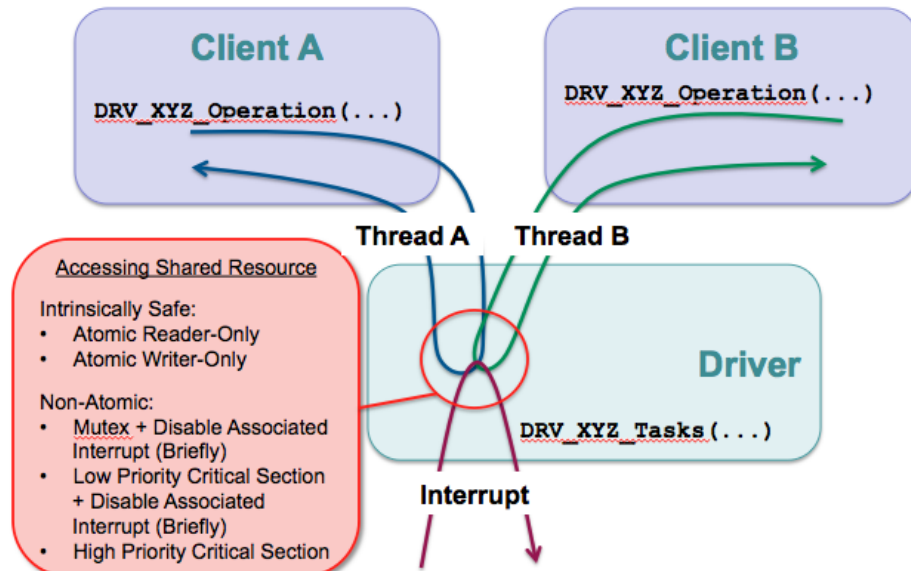
When utilizing a RTOS, it is possible that a driver and its clients may each run in its own RTOS thread. If the RTOS is preemptive, it is possible that the scheduler may interrupt any of these threads at any time and switch to another. If the other thread happens to also non-atomically access the same 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).



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.

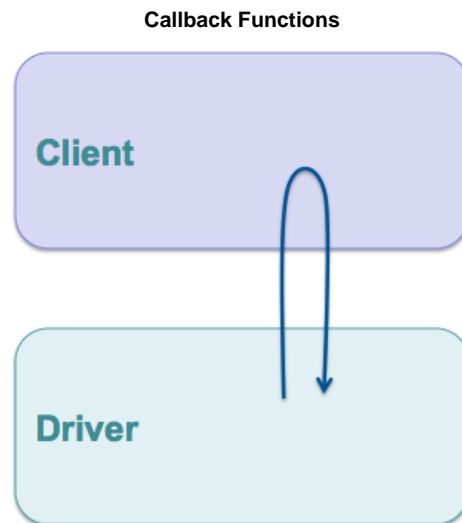
 **Note:** Tasks functions that do not manage an interrupt (such as tasks functions for applications or second-level tasks function) and that can only be polled should be treated like client interface functions. Non-atomic accesses to resources that can be accessed from other thread contexts and/or interrupts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

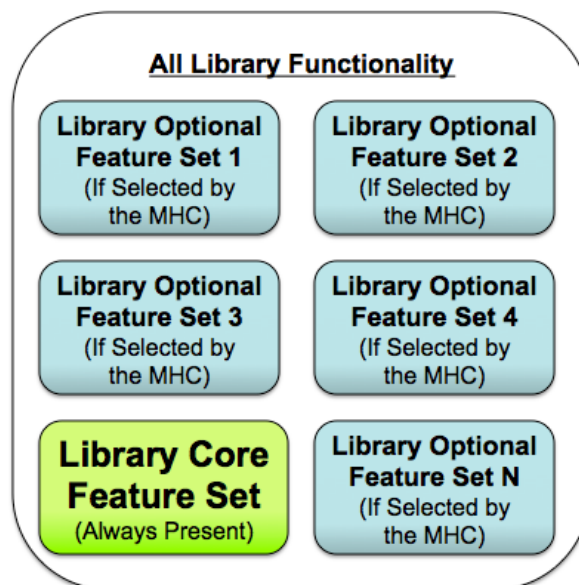
Optional Feature Sets

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_RETURN DRV_MYDEV_CoreFunction1 ( DRV_MYDEV_HANDLE handle, DRV_MYDEV_A *data )
{
    /* Implementation of core interface function 1 */
}

MY_RETURN 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_RETURN DRV_MYDEV_Option1Function1 ( DRV_MYDEV_HANDLE handle, DRV_MYDEV_A *data )
{
    /* Implementation of option 1 interface function 1 */
}

MY_RETURN 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      8000000
#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      8000000
#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 driver (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, an 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.

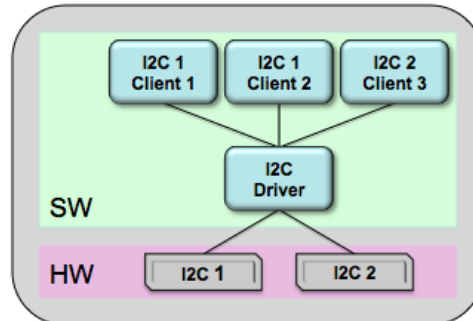
Refer to [Static Implementations](#) for details.

Implementing Multiple Client Drivers

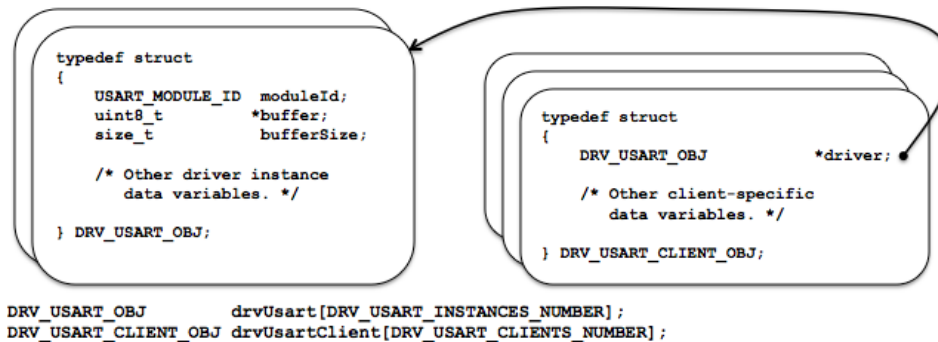
Describes how to implement multiple client drivers.

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 from 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 information 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
<input type="checkbox"/>	1	Define (and document) System interface functions.
<input type="checkbox"/>	2	Define (and document) Client interface functions.
<input type="checkbox"/>	2a	Define data transfer functions (following common models if appropriate).
<input type="checkbox"/>	2b	Define driver-specific functions.
<input type="checkbox"/>	3	Develop Hconfig and FreeMarker template support for initialization, tasks, and other system functions as needed to test and develop.
<input type="checkbox"/>	4	Develop and test initialize, tasks, and core functions.
<input type="checkbox"/>	4a	Start in a polled environment.
<input type="checkbox"/>	4b	Update & test for interrupt-driven environment (retest polled).
<input type="checkbox"/>	4c	Update & test OS support (retest polled & interrupt-driven).
<input type="checkbox"/>	5	Define (and document) MHC support for static configuration options.
<input type="checkbox"/>	6	Define (and document) optional feature sets and functions (one at a time).
<input type="checkbox"/>	6a	Implement & test in all three environments (polled, interrupt-driven, & OS-driven).
<input type="checkbox"/>	6b	Define MHC support for optional feature sets.
<input type="checkbox"/>	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.

Index

A

Atomicity 21

B

Buffer Queuing 18

Byte-by-Byte (Single Client) 16

C

Callback Functions 27

Checklist and Information 44

Client Interface 14

Close 16

Common Data Transfer Models 16

Configuration and Implementations 31

Configuration Options 33

D

Driver Client Interface Functions 14

Driver-Client Usage Model 14

F

File System Read/Write 17

G

General Guidelines 19

I

Implementing Multiple Client Drivers 36

Interrupt and Thread Safety 21

Interrupt Safety 22

Introduction 3

M

Module Deinitialize 11

Module Initialization 6

Module Reinitialize 12

Module Status 9

Module Tasks 8

MPLAB Harmony Driver Development Guide 2

Multiple Client Static Drivers 41

Multiple Implementations 34

O

Open 15

Optional Feature Sets 31

R

Review and Testing 43

RTOS Thread Safety 24

S

Static Implementations 38

Synchronization 29

System Interface 5

U

Using This Document 4