



MPLAB® Harmony Help - System Service Libraries

MPLAB Harmony Integrated Software Framework v1.11

System Service Libraries Help

This section provides descriptions of the System Service libraries that are available in MPLAB Harmony.

System Service Overview

This section provides an overview of the System Service libraries that are available in MPLAB Harmony.

Introduction

Describes system services provided by MPLAB Harmony.

Description

MPLAB Harmony provides system service libraries to support common functionality and manage resources that are shared by multiple drivers, libraries, and other modules.

A system service encapsulates code that manages a shared resource or implements a common capability in a single location so that it does not need to be replicated by individual drivers and libraries. This feature eliminates duplicated code and creates consistency across all modules, and also helps to eliminate potential conflicts and complex configuration issues and runtime interactions, resulting in a smaller and simpler overall solution.

System services may directly manage one or more peripherals or core processor capabilities by utilizing peripheral libraries, special function registers, special CPU instructions, or coprocessor registers. Some system services may utilize drivers, other system services, or even entire middleware stacks to share or emulate a common resource.

System services may be implemented statically (possibly generated by the MPLAB Harmony Configurator (MHC)) or dynamically to support multiple channels and instances like a driver. However, system services will not normally provide common *Open* and *Close* functions like a device driver.

In general, the distinguishing feature of a system service is that it implements a common capability that would otherwise "cut horizontally" across multiple modules in a system, causing interoperability and compatibility conflicts if the capability were implemented by other libraries.

System service functions use the following naming convention:

`SYS_<module-abbreviation>_[<feature-short-name>]<operation>`

Where,

- `SYS_` indicates that this is a system service function
- `<module-abbreviation>` is the abbreviated name of the system service module to which this function belongs
- `[<feature-short-name>]` is an optional short (or shortened) name that identifies the feature of the associated system service module to which this function refers. The feature short name will appear in the name of all functions that interact with or provide access to that particular feature.
- `<operation>` is a verb that identifies the action taken by the function

For example, `SYS_TMR_AlarmSet`, where:

- `<module-abbreviation>` = TMR, which indicates that this is a Timer System Service function
- `<feature-short-name>` = Alarm, which indicates that this function controls the alarm feature of the Timer System Service
- `<operation>` = Set, which indicates that this function sets the value of the alarm feature of the Timer System Service, as indicated by the function's parameters (not shown above).

System State Machine

Describes the MPLAB Harmony main function and system-wide state machine.

Description

In its most basic configuration, a MPLAB Harmony system operates in a single polled *super* loop that is implemented in the project's *main* function (in the `<project-name>/firmware/src/main.c` file) that is generated by the MHC, as shown in the following example. The *main* function calls two system-wide state machine functions, [SYS_Initialize](#) and [SYS_Tasks](#) that are also generated by the MHC to initialize and run the system.

MPLAB Harmony "main" Function

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

    while(true)
    {
        SYS_Tasks();
    }

    return(EXIT_FAILURE);
}
```

The [SYS_Initialize](#) function calls the initialization functions for all library and application modules that are used in the system to place them in their

initial states. Each module's initialization function must prepare the module so it is safe to call its *Tasks* function(s). The [SYS_Initialize](#) function's implementation (and necessary support code) is generated by the MHC in the

<project-name>/firmware/src/system_config/<configuration-name>/system_init.c file.

After initializing all modules, the *main* function contains the system-wide super loop that executes continuously until the system is powered off or reset. Inside the super loop, the *main* function calls the [SYS_Tasks](#) function. This function in turn calls the *Tasks* functions for any library or application modules whose state machines operate in a polled manner in the system. Of course, those state machines must first have been initialized by a prior call to their initialization functions by the [SYS_Initialize](#) function. The *SYS_Task* function's implementation is also generated by the MHC, but it is contained in the <project-name>/firmware/src/system_config/<configuration-name>/system_tasks.c file.

The polled super loop configuration described above is the most basic configuration of a MPLAB Harmony project. However, a single MPLAB Harmony project can have multiple configurations and different configurations may change modules so that they operate either polled or interrupt driven or in an RTOS thread. However, regardless of the configuration selected, the *main* function does not normally change.

MPLAB Harmony Documentation Volumes

For further information on MPLAB Harmony configurations and execution models, refer to the following MPLAB Harmony documentation.

- Volume I: Getting Started with MPLAB Harmony
- Volume II: MPLAB Harmony Configurator (MHC)
- Volume III: MPLAB Harmony Development



Note: Although the MPLAB Harmony *main* function and system state machine functions (and other system configuration code) are generated as part of a MPLAB Harmony project for your convenience, these files are unique to your project and can be modified or implemented by other means if so desired, or even removed from the project to create a binary library (.a file) containing all configured MPLAB Harmony libraries. Refer to the "MPLAB® X IDE User's Guide" (DS50002027) and the "MPLAB® XC32 C/C++ Compiler User's Guide" (DS50001686) for information on creating library projects.

MPLAB Harmony Module System Interface

Describes the MPLAB Harmony module system interface and provides usage examples.

Description

To support the system-wide state machine (see [System State Machine](#)), an MPLAB Harmony module must provide an initialization function and a *Tasks* function. In addition, a MPLAB Harmony module may (optionally) provide deinitialization, reinitialization, and status functions. This set of functions is considered the module's "system interface". The system state machine, system scheduler, or any other system-management code uses a module's system interface to initialize, run, and otherwise control the execution of a module in the MPLAB Harmony system.

To define the calling signature of these system interface functions, the [sys_module.h](#) header defines function pointer data types for each. These data types could be used to develop a dynamic system with capabilities beyond the basic system state machine, such as dynamic task registration, power management, advanced schedulers, or even your own operating system.

The following examples show how the system interface could be used to create a simple dynamic polled system tasks scheduler.

Example Dynamic System Data Structures

```
typedef struct _system_module_interface
{
    SYS_MODULE_INITIALIZE_ROUTINE    initialize;
    SYS_MODULE_TASKS_ROUTINE         tasks;
    SYS_MODULE_REINITIALIZE_ROUTINE  reinitialize;
    SYS_MODULE_DEINITIALIZE_ROUTINE  deinitialize;
    SYS_MODULE_STATUS_ROUTINE        status;
} SYSTEM_MODULE_INTERFACE;

typedef struct _system_module_data
{
    SYSTEM_MODULE_INTERFACE function;
    SYS_MODULE_INDEX         index;
    SYS_MODULE_INIT          *initData;
    SYS_MODULE_OBJ           obj;
    SYS_STATUS               status;
    uint8_t                  powerState;
} SYSTEM_MODULE_DATA;

SYSTEM_MODULE_DATA gModules[CONFIG_NUMBER_OF_MODULES];
```

In the previous example code, the *SYSTEM_MODULE_INTERFACE* structure contains pointers to all of a module's system interface functions. This structure could be filled in with pointers to a module's system interface functions and a pointer to the structure passed into a dynamic module

registration function, along with the module index number and a pointer to any initialization data required by the module. The following example shows how this dynamic registration function might appear.

Example Dynamic Module Registration Function

```
bool SYS_ModuleRegister ( SYSTEM_MODULE_DATA *module,
                        SYS_MODULE_INDEX   index,
                        SYS_MODULE_INIT    *iniData )
{
    SYSTEM_MODULE_DATA module;
    int                i;
    bool               success = false;

    for (i=0; i < CONFIG_NUMBER_OF_MODULES; i++)
    {
        if (gModules[i].function.initialize != NULL)
        {
            module = &gModules[i];
        }
    }

    if (i < CONFIG_NUMBER_OF_MODULES)
    {
        module->function.initialize      = module->initialize;
        module->function.tasks           = module->tasks;
        module->function.reinitialize    = module->reinitialize;
        module->function.deinitialize    = module->deinitialize;
        module->function.status          = module->status;

        module->index                    = index;
        module->initData                  = iniData;
        module->obj                       = SYS_MODULE_OBJ_INVALID;
        module->status                    = SYS_STATUS_UNINITIALIZED;
        module->powerState                 = SYS_MODULE_POWER_RUN_FULL;

        success = true;
    }

    return success;
}
```

The SYS_ModuleRegister function could then scan a system-global array (gModules, from the previous system data structures example) to find an empty SYSTEM_MODULE_DATA structure (using the initialization function pointer as a "flag" to indicate if the structure is in use or not), copy the newly registered module's interface and other data into the structure, and initialize the other members of the structure.

Once all modules to be used have been similarly registered, the entire system could be initialized when desired by calling a SYS_InitializeDynamic function implementation, similar to the following example.

Example Dynamic System Initialization Function

```
void SYS_InitializeDynamic ( void *data )
{
    SYSTEM_MODULE_DATA module = (SYSTEM_MODULE_DATA *)data;
    int                i;

    for (i=0; i < CONFIG_NUMBER_OF_MODULES; i++)
    {
        if (module->function.initialize != NULL)
        {
            module->obj = module->function.initialize(module->index,
                                                    module->initData);

            module->status      = SYS_STATUS_BUSY;
            module->powerState = SYS_MODULE_POWER_RUN_FULL;

            if (module->obj == SYS_MODULE_OBJ_INVALID)
            {
                module->function.initialize = NULL;
            }
        }
    }
}
```

The previous SYS_InitializeDynamic example function iterates through the global array of module data structures, calling the initialization functions

for any modules that have been registered and skipping over any structures that have no module registered (again using the required initialization function pointer as a sort of flag). If a module's initialization function successfully initializes the module it returns a valid module object handle, which the dynamic system initialization function captures in the module's data structure so it can call the module's other system interface routines. If the module is not successfully initialized, the object handle reported will be invalid (SYS_MODULE_OBJECT_INVALID) and the module is deregistered by nulling the pointer to the initialization function.

Once the system has been initialized, a dynamic tasks function like the following example would also iterate through the module data array and call the tasks functions for each module registered.

Example Dynamic System Tasks Function

```
void SYS_TasksDynamic ( void *data )
{
    SYSTEM_MODULE_DATA module = (SYSTEM_MODULE_DATA *)data;
    int i;

    for (i=0; i < CONFIG_NUMBER_OF_MODULES; i++)
    {
        if (module->function.initialize != NULL &&
            module->function.tasks      != NULL )
        {
            if (module->status(module->obj) >= SYS_STATUS_UNINITIALIZED)
            {
                module->function.tasks(module->obj);
            }
            else
            {
                module->function.deinitialize(module->obj);
                module->function.initialize = NULL;
            }
        }
    }
}
```

After calling a module's *Tasks* function, the previous example checks the module's status by calling its *Status* function. If the module reports an error status (any status less than SYS_STATUS_UNINITIALIZED), the module is deinitialized by calling its *Deinitialize* function and deregistered by nulling out the *Initialize* function pointer.

Please note that the example code provided is for informational purposes only, and is used to describe the purpose and usage of the MPLAB Harmony module system interface. A real dynamic tasks scheduler would need to deal with additional complexities. For example, MPLAB Harmony modules may have zero or more *Tasks* functions (only the initialization function is absolutely required). Therefore, a single tasks-function pointer would not be sufficient. The previous example demonstrates usage of the system status and power state data, but it does not actually update or manage these items. Also, this example does not demonstrate the usage of a module's reinitialization function. This function allows a module to provide a way to change its initial parameters while the module is active (after it has been initialized) without disrupting active clients of the module. *However, at the time of this writing, most MPLAB Harmony modules do not implement this function, so this capability is not usually available.*

Using the SYS_ASSERT Macro

Describes the purpose and usage of the system assertion macro.

Description

The **SYS_ASSERT** macro is a testing and debugging tool used throughout MPLAB Harmony libraries to verify (or assert) critical assumptions before any action is taken on them. However, it is not usually desirable to have these tests in production code because system assertion failures are normally fatal events that stop current execution, either hanging or resetting the system because it is not safe to continue. Also, even if the tests pass, they would add significant overhead, affecting code size and execution time. To avoid these issues, the default implementation of this macro is empty to eliminate the assertion tests and failure messages from the build, as shown in the following example.

Default SYS_ASSERT Macro Definition

```
#define SYS_ASSERT(test,message)
```

Conversely, when developing, debugging, and testing a project, having **SYS_ASSERT** statements in the code (particularly in library code) can be very helpful. This macro is used to check key parameter values and intermediate results and provide messages that explain the consequences that occur when they don't match what was expected when the library was implemented. A **SYS_ASSERT** failure during testing may save significant time spent debugging a MPLAB Harmony library only to find that a configuration setting or value passed into a library was incorrect or unexpected.

The **SYS_ASSERT** macro provides a convenient way to obtain the desired behavior when an assertion fails. In production code, simply accept the default definition that disposes of the assertion test code and the failure message. If things go wrong during testing, either enable a predefined implementation of this macro or define it in any way that is convenient for the current application.

One particularly useful definition of the **SYS_ASSERT** macro is to execute a hard-coded breakpoint instruction if the assertion fails, as shown in the following example.

Example SYS_ASSERT Macro Breakpoint Definition

```
#define SYS_ASSERT(test, message) \
    do{ if(!(test)) SYS_DEBUG_Breakpoint(); }while(true)
```

When using the previous definition, if the assertion test fails, the processor will reach a hard-coded breakpoint and execution will stop in the debugger at the `SYS_ASSERT` call. This behavior identifies the point in the source code where the assertion call failed. Then, the assertion call conveniently provides an explanation of what the failure means in the associated message string. A system assertion failure is a catastrophic failure, so the `do... while` loop hangs the system to prevent execution of code under an invalid assumption that may cause incorrect behavior or even damage to the system. It is also a way to guarantee that the `if` statement is correctly interpreted by the compiler and not accidentally associated with a following `else` statement.

Another useful implementation of the `SYS_ASSERT` macro might use the Debug System Service to display the message, as shown in the following example.

Example `SYS_ASSERT` Macro Definition Using Debug Message Service

```
#define SYS_ASSERT(test, message) \
    do{ if(!(test)) \
        SYS_DEBUG_MESSAGE(SYS_ERROR_FATAL, (message)); \
    }while(false)
```

The previous definition will display the assertion failure message using whatever method the `SYS_DEBUG` service is configured to use. However, it has the drawback that it does not hang the system (because the message service may need to continue running), so incorrect or unsafe behavior may result.

It is also possible to combine these two example definitions, which is particularly useful if the debug message service is configured to use the debugger's output window, or to create any another definition that is more useful for a given situation.

Obtaining System Version Information

Describes the purpose and usage of the system version functions.

Description

It is possible to programmatically obtain the version information for the release of MPLAB Harmony using the `SYS_Version` group of functions. For example, once the Debug System Service has been initialized and is running, the following code would retrieve the current version number and display it on the configured debug console if it were lower than a given value.

Example Version Function Usage

```
#define REQUIRED_VERSION 10700
```

```
if (SYS_VersionGet() < REQUIRED_VERSION)
{
    SYS_DEBUG_MESSAGE( SYS_ERROR_WARNING, "Version %s is too low",
        SYS_VersionStrGet() );
}
```

In the previous example, the `SYS_VersionGet` function is used to get the numeric representation of the version number so that it can be easily compared to a known value and the `SYS_VersionStrGet` is used to get a string representation of it for displaying on the debug console.

Library Interface

Data Types and Constants

	Name	Description
	<code>MAIN_RETURN_CODES</code>	Defines return codes for "main".
	<code>SYS_MODULE_DEINITIALIZE_ROUTINE</code>	Pointer to a routine that deinitializes a system module (driver, library, or system-maintained application).
	<code>SYS_MODULE_INDEX</code>	Identifies which instance of a system module should be initialized or opened.
	<code>SYS_MODULE_INIT</code>	Initializes a module (including device drivers) in a current power status as requested by the system or power manager.
	<code>SYS_MODULE_INITIALIZE_ROUTINE</code>	Pointer to a routine that initializes a system module (driver, library, or system-maintained application).
	<code>SYS_MODULE_OBJ</code>	Handle to an instance of a system module.
	<code>SYS_MODULE_REINITIALIZE_ROUTINE</code>	Pointer to a routine that reinitializes a system module (driver, library, or system-maintained application).
	<code>SYS_MODULE_STATUS_ROUTINE</code>	Pointer to a routine that gets the current status of a system module (driver, library, or system-maintained application).
	<code>SYS_MODULE_TASKS_ROUTINE</code>	Pointer to a routine that performs the tasks necessary to maintain a state machine in a module system module (driver, library, or system-maintained application).
	<code>SYS_STATUS</code>	Identifies the current status/state of a system module (including device drivers).
	<code>MAIN_RETURN_CODE</code>	Casts the given value to the correct type for the return code from "main".

	SYS_MODULE_OBJ_INVALID	Object handle value returned if unable to initialize the requested instance of a system module.
	SYS_MODULE_OBJ_STATIC	Object handle value returned by static modules.
	SYS_MODULE_POWER_IDLE_RUN	Module power-state idle-run state code.
	SYS_MODULE_POWER_IDLE_STOP	Module power-state idle-stop state code.
	SYS_MODULE_POWER_OFF	Module power-state power off state code.
	SYS_MODULE_POWER_RUN_FULL	Module power-state run-full state code.
	SYS_MODULE_POWER_SLEEP	Module power-state sleep state code.



Main Function Support Macro

	Name	Description
	MAIN_RETURN	Defines the correct return type for the "main" routine.

System Assert Macro

	Name	Description
	SYS_ASSERT	Implements default system assert routine, asserts that "test" is true.

System State Machine Functions

	Name	Description
	SYS_Tasks	Function that performs all polled system tasks.
	SYS_Initialize	Initializes the board, services, drivers, and other modules

Version Functions

	Name	Description
	SYS_VersionGet	Gets SYS_COMMON version in numerical format.
	SYS_VersionStrGet	Gets SYS_COMMON version in string format.

Description

This section describes the APIs of the System Service Library.

Refer to each section for a detailed description.

Main Function Support Macro

MAIN_RETURN Macro

Defines the correct return type for the "main" routine.

File

[sys_common.h](#)

C

```
#define MAIN_RETURN void
```

Description

Main Function Return Type

This macro defines the correct return type for the "main" routine for the selected Microchip microcontroller family.

Remarks

The main function return type may change, depending upon which family of Microchip microcontrollers is chosen. Refer to the user documentation for the C-language compiler in use for more information.

Example

```
MAIN_RETURN main ( void )
{
    // Initialize the system
    SYS_Initialize(...);

    // Main Loop
    while(true)
```



```
{  
    SYS_Tasks();  
}  
  
return MAIN_RETURN_CODE(MAIN_RETURN_SUCCESS);  
}
```

System Assert Macro

SYS_ASSERT Macro

Implements default system assert routine, asserts that "test" is true.

File

[sys_common.h](#)

C

```
#define SYS_ASSERT(test,message)
```

Returns

None. Normally hangs in a loop, depending on the implementation to which it is mapped.

Description

This macro implements the default system assert routine that asserts that the provided boolean test is true.

Remarks

Can be overridden as desired by defining your own SYS_ASSERT macro before including [system.h](#).

The default definition removes this macro from all code because it adds significant size to all projects. The most efficient use is to enable it in individual libraries and build them separately for debugging.

Preconditions

None, depending on the implementation to which this macro is mapped.

Example

```
void MyFunc ( int *pointer )  
{  
    SYS_ASSERT(NULL != pointer, "NULL Pointer passed to MyFunc");  
  
    Do something with pointer.  
}
```

Parameters

Parameters	Description
test	This is an expression that resolves to a boolean value (zero=false, non-zero=true)
message	This is a NULL-terminated ASCII character string that can be displayed on a debug output terminal if "test" is false (if supported).

Function

```
void SYS_ASSERT ( bool test, char *message )
```

System State Machine Functions

SYS_Tasks Function

Function that performs all polled system tasks.

File

[sys_module.h](#)

C

```
void SYS_Tasks();
```

Returns

None.

Description

System Tasks Function

This function performs all polled system tasks by calling the state machine "tasks" functions for all polled modules in the system, including drivers, services, middleware and applications.

Remarks

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

Preconditions

The [SYS_Initialize](#) function must have been called and completed.

Example

```
SYS_Initialize ( NULL );

while ( true )
{
    SYS_Tasks ( );
}
```

Function

```
void SYS_Tasks ( void );
```

SYS_Initialize Function

Initializes the board, services, drivers, and other modules

File

[sys_init.h](#)

C

```
void SYS_Initialize(void * data);
```

Returns

None.

Description

This function initializes the board, services, drivers, and other modules as configured at build time. It should be called almost immediately after entering the "main" function.

Remarks

Basic System Initialization Sequence:

1. Initialize core processor services.
2. Initialize board support package.
3. Initialize RTOS (if utilized).
4. Initialize drivers.
5. Initialize system services.
6. Initialize middleware.
7. Initialize application(s).

This function may be overridden and implemented as desired as long as it appropriately initializes all necessary board, services, and modules.

Most MPLAB Harmony libraries are designed so that the order in which they are initialized is not important. However, core processor services and board support package initialization should be completed before any other initialization takes place and RTOS initialization (if utilized) should take place before drivers, system services and middleware are initialized. Applications should be initialized last.

Preconditions

The C-language run-time environment and stack must have been initialized.

Example

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

    while (true)
    {
        SYS_Tasks();
    }
}
```

Parameters

Parameters	Description
data	Pointer to any system initialization data required. Normally passed as NULL for static system implementations.

Function

void SYS_Initialize (void *data)

Version Functions

SYS_VersionGet Macro

Gets SYS_COMMON version in numerical format.

File

[sys_common.h](#)

C

```
#define SYS_VersionGet( void ) SYS_VERSION
```

Returns

Current driver version in numerical format.

Description

This routine gets the SYS_COMMON version. The version is encoded as major * 10000 + minor * 100 + patch. The string version can be obtained using [SYS_VersionStrGet\(\)](#)

Function

SYS_VersionGet(void)

SYS_VersionStrGet Macro

Gets SYS_COMMON version in string format.

File

[sys_common.h](#)

C

```
#define SYS_VersionStrGet( void ) SYS_VERSION_STR
```

Returns

Current SYS_COMMON version in the string format.

Description

Macro: char * SYS_VersionStrGet (void)

This routine gets the SYS_COMMON version in string format. The version is returned as major.minor.patch[type], where type is optional. The

numerical version can be obtained using [SYS_VersionGet\(\)](#)

Remarks

None.

Data Types and Constants

MAIN_RETURN_CODES Enumeration

Defines return codes for "main".

File

[sys_common.h](#)

C

```
typedef enum {  
    MAIN_RETURN_FAILURE = -1,  
    MAIN_RETURN_SUCCESS = 0  
} MAIN_RETURN_CODES;
```

Description

Main Routine Codes Enumeration

This enumeration provides a predefined list of return codes for the main function. These codes can be passed into the [MAIN_RETURN_CODE](#) macro to convert them to the appropriate type (or discard them if not needed) for the Microchip C-language compiler in use.

Remarks

The main function return type may change, depending upon which family of Microchip microcontrollers is chosen. Refer to the user documentation for the C-language compiler in use for more information.

Example

```
MAIN_RETURN main ( void )  
{  
    SYS_Initialize(...);  
  
    while(true)  
    {  
        SYS_Tasks();  
    }  
  
    return MAIN_RETURN_CODE(MAIN_RETURN_SUCCESS);  
}
```

SYS_MODULE_DEINITIALIZE_ROUTINE Type

Pointer to a routine that deinitializes a system module (driver, library, or system-maintained application).

File

[sys_module.h](#)

C

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

Returns

None.

Description

System Module Deinitialization Routine Pointer. This data type is a pointer to a routine that deinitializes a system module (driver, library, or system-maintained application).

Remarks

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

Preconditions

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

Example

None.

Parameters

Parameters	Description
object	Handle to the module instance

Function

void (* SYS_MODULE_DEINITIALIZE_ROUTINE) ([SYS_MODULE_OBJ](#) object)

SYS_MODULE_INDEX Type

Identifies which instance of a system module should be initialized or opened.

File

[sys_module.h](#)

C

```
typedef unsigned short int SYS_MODULE_INDEX;
```

Description

System Module Index

This data type identifies to which instance of a system module a call to that module's "Initialize" and "Open" routines refers.

Remarks

Each individual module will usually define macro names for the index values it supports (e.g., DRV_TMR_INDEX_1, DRV_TMR_INDEX_2, ...).

SYS_MODULE_INIT Union

Initializes a module (including device drivers) in a current power status as requested by the system or power manager.

File

[sys_module.h](#)

C

```
typedef union {
    uint8_t value;
    struct {
        uint8_t powerState : 4;
        uint8_t reserved : 4;
    } sys;
} SYS_MODULE_INIT;
```

Members

Members	Description
uint8_t powerState : 4;	Requested power state
uint8_t reserved : 4;	Module-definable field, module-specific usage

Description

System Module Init

This structure provides the necessary data to initialize or reinitialize a module (including device drivers) into a requested power state. The structure can be extended in a module specific way as to carry module specific initialization data.

Remarks

This structure is used in the device driver routines DRV__Initialize and DRV__Reinitialize that are defined by each device driver.

The "powerState" member has several predefined values (shown below). All other available values (within the 4-bit field) are available for

module-specific meaning.

Predefined powerState Values:

- 0. [SYS_MODULE_POWER_OFF](#) - Module power-state power off state code
- 1. [SYS_MODULE_POWER_SLEEP](#) - Module power-state sleep state code
- 2. [SYS_MODULE_POWER_IDLE_STOP](#) - Module power-state idle-stop state code
- 3. [SYS_MODULE_POWER_IDLE_RUN](#) - Module power-state idle-run state code
- 4. through 14. - Module-specific meaning
- 15. [SYS_MODULE_POWER_RUN_FULL](#) - Module power-state run-full state code

SYS_MODULE_INITIALIZE_ROUTINE Type

Pointer to a routine that initializes a system module (driver, library, or system-maintained application).

File

[sys_module.h](#)

C

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

Returns

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

Description

System Module Initialization Routine Pointer

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

Remarks

This function will only be called once during system initialization.

Preconditions

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

Parameters

Parameters	Description
index	Identifier for 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 and default initialization is to be used.

Function

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

SYS_MODULE_OBJ Type

Handle to an instance of a system module.

File

[sys_module.h](#)

C

```
typedef uintptr_t SYS_MODULE_OBJ;
```

Description

System Module Object

This data type is a handle to a specific instance of a system module (such as a device driver).

Remarks

Code outside of a specific module should consider this as an opaque type (much like a void *). Do not make any assumptions about base type as it may change in the future or about the value stored in a variable of this type.

SYS_MODULE_REINITIALIZE_ROUTINE Type

Pointer to a routine that reinitializes a system module (driver, library, or system-maintained application)

File

[sys_module.h](#)

C

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

Returns

None.

Description

System Module Reinitialization Routine Pointer

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

Remarks

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

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

This operation can also be used to refresh the hardware state as defined by the initialization data, thus it must guarantee that all hardware state has been refreshed.

This function can be called multiple times to reinitialize the module.

Preconditions

The low-level board initialization must have (and will be) completed and the module's initialization routine will have been called before the system will call the reinitialization routine for any modules.

Example

None.

Parameters

Parameters	Description
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.

Function

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

SYS_MODULE_STATUS_ROUTINE Type

Pointer to a routine that gets the current status of a system module (driver, library, or system-maintained application).

File

[sys_module.h](#)

C

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

Returns

One of the possible status codes from [SYS_STATUS](#)

Description

System Module Status Routine Pointer

This data type is a pointer to a routine that gets the current status of a system module (driver, library, or system-maintained application).

Remarks

A module's status operation can be used to determine when any of the other module level operations has completed as well as to obtain general status of the module. The value returned by the status routine will be checked after calling any of the module operations to find out when they have completed.

If the status operation returns `SYS_STATUS_BUSY`, the previous operation has not yet completed. Once the status operation 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 operation returns an error value, the error may be cleared by calling the reinitialize operation. If that fails, the deinitialize operation will need to be called, followed by the initialize operation to return to normal operations.

Preconditions

The low-level board initialization must have (and will be) completed and the module's initialization routine will have been called before the system will call the status routine for any modules.

Example

None.

Parameters

Parameters	Description
object	Handle to the module instance

Function

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

SYS_MODULE_TASKS_ROUTINE Type

Pointer to a routine that performs the tasks necessary to maintain a state machine in a module system module (driver, library, or system-maintained application).

File

[sys_module.h](#)

C

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

Returns

None.

Description

System Module Tasks Routine Pointer

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

Remarks

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

Preconditions

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

Example

None.

Parameters

Parameters	Description
object	Handle to the module instance

Function

void (* SYS_MODULE_TASKS_ROUTINE) ([SYS_MODULE_OBJ](#) object)

SYS_STATUS Enumeration

Identifies the current status/state of a system module (including device drivers).

File

[sys_module.h](#)

C

```
typedef enum {
    SYS_STATUS_ERROR_EXTENDED = -10,
    SYS_STATUS_ERROR = -1,
    SYS_STATUS_UNINITIALIZED = 0,
    SYS_STATUS_BUSY = 1,
    SYS_STATUS_READY = 2,
    SYS_STATUS_READY_EXTENDED = 10
} SYS_STATUS;
```

Members

Members	Description
SYS_STATUS_ERROR_EXTENDED = -10	Indicates that a non-system defined error has occurred. The caller must call the extended status routine for the module in question to identify the error.
SYS_STATUS_ERROR = -1	An unspecified error has occurred.
SYS_STATUS_UNINITIALIZED = 0	The module has not yet been initialized
SYS_STATUS_BUSY = 1	An operation is currently in progress
SYS_STATUS_READY = 2	Any previous operations have succeeded and the module is ready for additional operations
SYS_STATUS_READY_EXTENDED = 10	Indicates that the module is in a non-system defined ready/run state. The caller must call the extended status routine for the module in question to identify the state.

Description

System Module Status

This enumeration identifies the current status/state of a system module (including device drivers).

Remarks

This enumeration is the return type for the system-level status routine defined by each device driver or system module (for example, DRV_I2C_Status).

MAIN_RETURN_CODE Macro

Casts the given value to the correct type for the return code from "main".

File

[sys_common.h](#)

C

```
#define MAIN_RETURN_CODE(c)
```

Description

Main Routine Code Macro

This macro casts the given value to the correct type for the return code from the main function.

Remarks

The main function return type may change, depending upon which family of Microchip microcontrollers is chosen. Refer to the user documentation for the C-language compiler in use for more information.

Example

```
MAIN_RETURN main ( void )
{
    // Initialize the system
    SYS_Initialize(...);
}
```

```
// Main Loop
while(true)
{
    SYS_Tasks();
}

return MAIN_RETURN_CODE(MAIN_RETURN_SUCCESS);
}
```

SYS_MODULE_OBJ_INVALID Macro

Object handle value returned if unable to initialize the requested instance of a system module.

File

[sys_module.h](#)

C

```
#define SYS_MODULE_OBJ_INVALID ((SYS_MODULE_OBJ) -1 )
```

Description

System Module Object Invalid

This is the object handle value returned if unable to initialize the requested instance of a system module.

Remarks

Do not rely on the actual value of this constant. It may change in future implementations.

SYS_MODULE_OBJ_STATIC Macro

Object handle value returned by static modules.

File

[sys_module.h](#)

C

```
#define SYS_MODULE_OBJ_STATIC ((SYS_MODULE_OBJ) 0 )
```

Description

System Module Object Static

This is the object handle value returned by static system modules.

Remarks

Do not rely on the actual value of this constant. It may change in future implementations.

SYS_MODULE_POWER_IDLE_RUN Macro

Module power-state idle-run state code.

File

[sys_module.h](#)

C

```
#define SYS_MODULE_POWER_IDLE_RUN 3
```

Description

System Module Power Idle-Run State

This value identifies the current power status/state of a system module (including device drivers). It is used to indicate that the module should prepare to enter an idle-run state.

Remarks

An idle-run state indicates that the core CPU clock may be stopped, but the module's peripheral clock may continue running and peripheral operations may continue as long as no code needs to be executed. If code needs to execute, the module must cause an interrupt.

This value is passed in the powerState field of the [SYS_MODULE_INIT](#) structure that takes part in all modules initialization and reinitialization. The power state codes between [SYS_MODULE_POWER_IDLE_RUN](#) (with a value of 3) and [SYS_MODULE_POWER_RUN_FULL](#) (with a value of 15) are available for module-specific definition and usage.

SYS_MODULE_POWER_IDLE_STOP Macro

Module power-state idle-stop state code.

File

[sys_module.h](#)

C

```
#define SYS_MODULE_POWER_IDLE_STOP 2
```

Description

System Module Power Idle-Stop State

This value identifies the current power status/state of a system module (including device drivers). It is used to indicate that the module should prepare to enter an idle-stop state.

Remarks

An idle-stop state indicates that the core CPU clock may be stopped, but the module's peripheral clock may continue running. However, the peripheral should prepare to stop operations when the idle state is entered.

This value is passed in the powerState field of the [SYS_MODULE_INIT](#) structure that takes part in all modules initialization and reinitialization.

The power state codes between [SYS_MODULE_POWER_IDLE_RUN](#) (with a value of 3) and [SYS_MODULE_POWER_RUN_FULL](#) (with a value of 15) are available for module-specific definition and usage.

SYS_MODULE_POWER_OFF Macro

Module power-state power off state code.

File

[sys_module.h](#)

C

```
#define SYS_MODULE_POWER_OFF 0
```

Description

System Module Power Off State

This value identifies the current power status/state of a system module (including device drivers). It is used to indicate that the module should prepare to enter a full power-off state.

Remarks

A power off state indicates that power may be completely removed (0 Volts).

This value is passed in the powerState field of the [SYS_MODULE_INIT](#) structure that takes part in all modules initialization and reinitialization.

The power state codes between [SYS_MODULE_POWER_IDLE_RUN](#) (with a value of 3) and [SYS_MODULE_POWER_RUN_FULL](#) (with a value of 15) are available for module-specific definition and usage.

SYS_MODULE_POWER_RUN_FULL Macro

Module power-state run-full state code.

File

[sys_module.h](#)

C

```
#define SYS_MODULE_POWER_RUN_FULL 15
```

Description

System Module Power Run-Full State. This value identifies the current power status/state of a system module (including device drivers). It is used to indicate that the module should prepare to enter an run-full state.

Remarks

An run-full state indicates that the core CPU and peripheral clocks are operational at their normal configured speed and the module should be ready for normal operation.

This value is passed in the powerState field of the [SYS_MODULE_INIT](#) structure that takes part in all modules initialization and reinitialization.

The power state codes between [SYS_MODULE_POWER_IDLE_RUN](#) (with a value of 3) and [SYS_MODULE_POWER_RUN_FULL](#) (with a value of 15) are available for module-specific definition and usage.

SYS_MODULE_POWER_SLEEP Macro

Module power-state sleep state code.

File

[sys_module.h](#)

C

```
#define SYS_MODULE_POWER_SLEEP 1
```

Description

System Module Power Sleep State

This value identifies the current power status/state of a system module (including device drivers). It is used to indicate that the module should prepare to enter a sleep state.

Remarks

A Sleep state indicates that the core CPU and peripheral clocks may be stopped and no code will execute and any module hardware will be stopped.

This value is passed in the powerState field of the [SYS_MODULE_INIT](#) structure that takes part in all modules initialization and reinitialization.

The power state codes between [SYS_MODULE_POWER_IDLE_RUN](#) (with a value of 3) and [SYS_MODULE_POWER_RUN_FULL](#) (with a value of 15) are available for module-specific definition and usage.

Files

Files

Name	Description
sys_common.h	Common System Services definitions and declarations.
sys_module.h	Defines definitions and declarations related to system modules.
system.h	Top level common system services library interface header.
sys_init.h	Initialization System Service Library definitions and declarations.

Description

This section lists the source and header files used by the library.

sys_common.h

Common System Services definitions and declarations.

Enumerations

	Name	Description
	MAIN_RETURN_CODES	Defines return codes for "main".

Macros

	Name	Description
	MAIN_RETURN	Defines the correct return type for the "main" routine.
	MAIN_RETURN_CODE	Casts the given value to the correct type for the return code from "main".
	SYS_ASSERT	Implements default system assert routine, asserts that "test" is true.
	SYS_VersionGet	Gets SYS_COMMON version in numerical format.
	SYS_VersionStrGet	Gets SYS_COMMON version in string format.

Description

System Services Common Library Header

This file provides commonsystem services definitions and declarations.

Remarks

This file is included by "[system.h](#)" and does not need to be included directly.

File Name

sys_common.h

Company

Microchip Technology Inc.


sys_module.h

Defines definitions and declarations related to system modules.

Enumerations

	Name	Description
	SYS_STATUS	Identifies the current status/state of a system module (including device drivers).

Functions

	Name	Description
	SYS_Tasks	Function that performs all polled system tasks.

Macros

	Name	Description
	SYS_MODULE_OBJ_INVALID	Object handle value returned if unable to initialize the requested instance of a system module.
	SYS_MODULE_OBJ_STATIC	Object handle value returned by static modules.
	SYS_MODULE_POWER_IDLE_RUN	Module power-state idle-run state code.
	SYS_MODULE_POWER_IDLE_STOP	Module power-state idle-stop state code.
	SYS_MODULE_POWER_OFF	Module power-state power off state code.
	SYS_MODULE_POWER_RUN_FULL	Module power-state run-full state code.
	SYS_MODULE_POWER_SLEEP	Module power-state sleep state code.

Types

	Name	Description
	SYS_MODULE_DEINITIALIZE_ROUTINE	Pointer to a routine that deinitializes a system module (driver, library, or system-maintained application).
	SYS_MODULE_INDEX	Identifies which instance of a system module should be initialized or opened.
	SYS_MODULE_INITIALIZE_ROUTINE	Pointer to a routine that initializes a system module (driver, library, or system-maintained application).
	SYS_MODULE_OBJ	Handle to an instance of a system module.
	SYS_MODULE_REINITIALIZE_ROUTINE	Pointer to a routine that reinitializes a system module (driver, library, or system-maintained application).
	SYS_MODULE_STATUS_ROUTINE	Pointer to a routine that gets the current status of a system module (driver, library, or system-maintained application).
	SYS_MODULE_TASKS_ROUTINE	Pointer to a routine that performs the tasks necessary to maintain a state machine in a module system module (driver, library, or system-maintained application).

Unions

	Name	Description
	SYS_MODULE_INIT	Initializes a module (including device drivers) in a current power status as requested by the system or power manager.

Description

System Module Header

This file defines definitions and interfaces related to system modules.

Remarks

This file is included via "[system.h](#)" and does not normally need to be included directly.

File Name

sys_module.h

Company

Microchip Technology Inc.

system.h

Top level common system services library interface header.

Description

System Services Library Interface Header

This file is the top level common system services library interface header. It defines (or includes files that define) the common system service types, prototypes, and other definitions that are commonly used by MPLAB Harmony libraries and system services.

System services provide common functionality that would otherwise need to be duplicated by multiple other modules or that would force them to interact in complex and hard to manage ways. System services eliminate conflicts by controlling access shared resources.

Remarks

The parent directory to the "system" directory should be added to the compiler's search path for header files such that the following include statment will successfully include this file.

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

File Name

system.h


Company

Microchip Technology Inc.

sys_init.h

Initialization System Service Library definitions and declarations.

Functions

	Name	Description
	SYS_Initialize	Initializes the board, services, drivers, and other modules

Description

Initialization System Service Library Definitions

This file contains Initialization System Service Library definitions and declarations.

Remarks

This file is included by "[system.h](#)" and need not be included directly.

File Name

sys_init.h

Company

Microchip Technology Inc.

Clock System Service Library

This section describes the Clock System Service Library.

Introduction

This library provides an interface to manage the Oscillator module on the Microchip family of microcontrollers during Execution time using the Clock System Service Library.

Description

The Clock System Service can be configured to run in either Static mode or Dynamic mode via the MPLAB Harmony Configurator. In Static mode, the service will configure the Oscillator module to the user-desired system clock frequency, peripheral bus clock frequency, reference bus clock frequency, and USB PLL clock frequency. These settings are set at initialization and are designed to remain the same for the duration of run-time. In Dynamic mode, the service will also set all of these frequencies at initialization; however it offers the function support to allow users to change the system clock frequency, peripheral bus frequency, and if the device supports it, the reference clock frequency.

Design Recommendations

If performance and small code size are critical, and there is no requirement to change the clock speed, the recommendation is to use Static mode. If there is a need to change the system, peripheral, or reference clock frequencies during run-time, the recommendation is to use Dynamic mode.

Configuration

The Oscillator is the heart of the microcontroller, which provides the clock on which the core and the peripherals run. For all of the oscillators present on a Microchip microcontroller, two types of configurations exist:

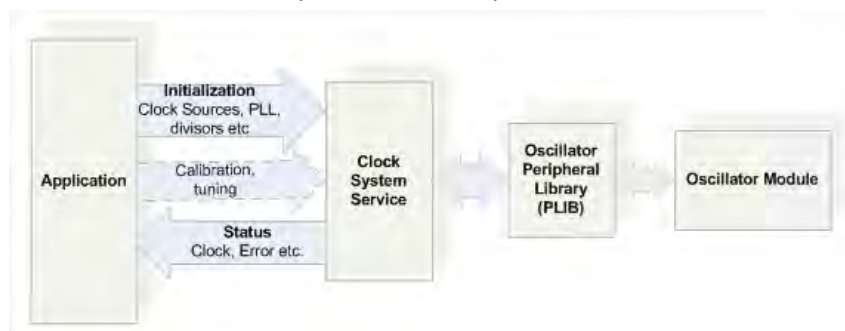
- Through Configuration bits
- At Execution time

Configuring the oscillator through the Configuration bits is a one-time process that it is done during the programming of the device. This one-time configuration is programmed in the code memory. The Execution time configuration provides features that are allowed to be changed during code execution. The Clock System Service Library provides functions the Execution time configurable features of the Oscillator module.

The Oscillator module supports the CPU and other peripherals by providing the clock. Therefore, there is more initialization and little or no run-time operations. The run-time operations set the system clock or any other output clocks. If the application needs to make any initialization changes, they can be done at run-time.

The Oscillator module as a whole is a group of oscillators. These are provided for the application to choose the correct oscillator for the application. It is important to wisely choose the oscillator and the frequency at which the CPU and the other peripherals should run in power-critical environments.

The following figure illustrates the interaction of the Clock System Service Library.



Using the Library

This topic describes the basic architecture of the Clock System Service and provides information and examples on its use.

Description

Interface Header File: `sys_osc.h`

The interface to the Clock System Service is defined in the `sys_osc.h` header file, which is included by the `sys.h` header file.

Any C language source (`.c`) file that uses the Clock System Service must include `sys.h`.

Abstraction Model

This library provides a low-level abstraction of the Clock System Service Library. This topic describes how that abstraction is modeled in software

and introduces the library's interface.

Description

To understand the Oscillator module and how each of its feature is mapped in this library, it is important to understand the following terminologies.

Oscillators/Clock Sources

A clock source is hardware that generates oscillations. This may be internal or external.

Divisor and Multiplier/PLL

These are hardware modules that can scale the clock. The rate at which the scaling is done may be fixed or configurable.

Clocks

Clock outputs are output lines from the Oscillator module, which may route to different modules of the device or to the CPU (the system clock).

The following diagram provides a simplified explanation and the relationship between the previously mentioned terms. In most cases, there are multiple clock source options available for each of the clock outputs. However, not all clock sources are available for all output clocks. Scaling is an optional feature in most cases.



Library Overview

Please refer to the [System Service Introduction](#) for a detailed description of MPLAB Harmony system services.

The library interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the Clock System Service module.

Library Interface Section	Description
System Clock Functions	Provides functions to set and get the system clock frequency.
Peripheral Clock Functions	Provides functions to set and get the peripheral clock frequency.
Reference Clock Functions	Provides functions to set up the reference clock, and set and get the reference clock frequency.
Clock Failure Functions	Provides functions to allow registration of a callback function that will be triggered on a clock failure and to inform the user on a clock failure by invoking the registered call back function.
Initialization Functions	Provides an initialization routine to configure the module at the start-up of the system and reinitialization if the application wants to change any of the initialization settings.

How the Library Works

The following diagram shows the flow of an application call to the Clock System Service Library.

Refer to the [System Services Introduction](#) section for information on how the system services operate in a system.

Configuration
<ul style="list-style-type: none"> - Configuration bit divisors - Configuration bit multipliers - Primary oscillator clock input - Secondary oscillator clock input
Initialization
<ul style="list-style-type: none"> - General <ul style="list-style-type: none"> - Action on Wait instruction - Enable Secondary - Clock Sources <ul style="list-style-type: none"> - System Clock - USB Clock - Graphics Clock - Reference clock - PLL and Clock divisors - Doze Mode
Oscillator Tuning
<ul style="list-style-type: none"> - Internal FRC oscillator Tuning
Status routines
<ul style="list-style-type: none"> - Get clock - Clock Fail Status

Initialization

Provides information and code examples for initializing the Oscillator.

Description

The Oscillator is a module with little or no run-time operations. Therefore, the application must do most of the configurations at the time of system initialization even though it is allowed to change the clock frequency at run-time.

The Oscillator needs some time to settle. Therefore, it is recommended to perform the oscillator initialization before initializing other modules. Check for the oscillator status once all of the initializations are complete and before the application task routine takes control.



Note: Not all devices require all of the initialization parameters of the structure. Refer to the specific device data sheet to determine which features are available for your device.

Clock is Set Through Configuration bits

In most cases, setting the clock is done through Configuration bits. Therefore, what the application expects from the Clock System Service Library is to provide all of the clock frequency values on a function call. To use the Clock System Service in this way, call the [SYS_CLK_Initialize](#) function by passing a NULL to it, as shown in the following code example. The function reads the registers and will initialize its internal data structure.

```
SYS_CLK_Initialize ( NULL );
//As per my setting, the system clock should be 80 MHz
If ( SYS_CLK_SystemFrequencyGet ( ) != 80000000 )
{
    //Something is wrong, check the Configuration bit settings
}
```

Alter the Configuration bit Setting

To alter the settings made in the Configuration bit, pass the structure to the [SYS_CLK_Initialize](#) function.

```
SYS_CLK_INIT generalInit;

// Populate the oscillator initialization structure
generalInit.systemClockSource = SYS_CLK_SOURCE_FRC_SYSPLL;
generalInit.systemClockFrequencyHz = 30000000;
generalInit.onWaitInstruction = OSC_ON_WAIT_SLEEP;
generalInit.secondaryOscKeepEnabled = true;
SYS_CLK_Initialize (&generalInit);
```

Changing the Clock

Provides information on changing the clock.

Description

The module initializes only the system clock as part of the [SYS_CLK_Initialize](#) function. All other clocks including the peripheral clock must be set by calling specific functions.

Changing the System Clock

Changing the system clock is done using the [SYS_CLK_SystemFrequencySet](#) function, which expects the clock source and the frequency as arguments. If the clock source is a PLL-based source, a flag is provided so that a user can choose whether to wait until the operation is complete or just trigger the change and exit. The function will return zero if the requested clock is not achievable.

Changing the Peripheral Clock

Changing the Peripheral clock is done using the [SYS_CLK_PeripheralFrequencySet](#) function, which expects the clock source and the frequency as arguments. The function will return zero if the requested clock is not achievable. When this occurs, call the [SYS_CLK_PeripheralClosestFrequencyGet](#) function by passing the bus number to it, to determine the closest possible clock.

```
switch ( clockState )
{
case 1:
SYS_CLK_PeripheralFrequencySet ( CLK_BUS_PERIPHERAL_1, CLK_SOURCE_PERIPHERAL_SYSTEMCLK, 8000000, true );

clockState++;

break;

case 2:
if ( SYS_CLK_PeripheralOperationStatus ( CLK_BUS_PERIPHERAL_1 ) == SYS_CLK_OPERATION_COMPLETE )
{
//Peripheral clock setting is completed
clockState++;
}
break;
}
```

Changing/Setting the Reference Clock

Changing or setting the reference clock is done using two functions. Set the module's response to Sleep and Idle mode using the function [SYS_CLK_ReferenceClockSetup](#). Use the [SYS_CLK_ReferenceFrequencySet](#) function to set the clock out, which expects the clock source and the frequency as arguments. The function will return zero if the requested clock is not achievable. When this occurs, call the [SYS_CLK_ReferenceClosestFrequencyGet](#) function by passing the bus number to it, to determine the closest possible clock.

```
uint32_t achievedFrequencyHz;

//I want 200 kHz as output
achievedFrequencyHz = SYS_CLK_ReferenceFrequencySet ( CLK_BUS_REFERENCE_3, CLK_SOURCE_REF_FRC, 200000, true );

if ( achievedFrequencyHz != 0 )
{
    //Frequency successfully set
}
```

Using Status Functions

Provides a code example to check the clock output.

Description

Checking the Clock Output

The following code provides an example of checking the clock output.

```
uint32_t usbClockOutputHz;
unsigned long sysClockOutputHz;
unsigned long peripheralClockHz;
uint32_t sysClockHz;

sysClockHz = SYS_CLK_SystemFrequencyGet ( );

peripheralClockHz = SYS_CLK_PeripheralFrequencyGet ( CLK_BUS_PERIPHERAL_1 );
usbClockOutputHz = SYS_CLK_USBClockFrequencyGet ( CLK_BUS_USB_1 );

sysClockOutputHz = SYS_CLK_ReferenceFrequencyGet ( CLK_BUS_REFERENCE_1 );
```

Oscillator Tuning

Provides information on tuning the Oscillator.

Description

Oscillator tuning will help compensate for temperature effects on the FRC frequency over a wide range of temperatures. The tuning step size is an approximation; the application is supposed to try different values to achieve the best result. In some devices, there are different tuning modes available.

Direct Number Method

```
//Software must unlock before the start of the tuning.
SYS_OSC_FRCTUNING_DATA tuningInput;
OSC_FRC_TUNING_VALUE tuningValue = OSC_TUNE_TO_CENTRAL_FREQ_PLUS_0_86;

tuningInput.tuningMode = OSC_TUNING_USING_NUMBER;
tuningInput.tuningData = &tuningValue;
SYS_OSC_FRCTuningSet(&tuningInput);
```

Sequential Dithering

To get the Sequential Dithering working, the application is supposed to set the value in seven sequencers and also in the tuning register. Next, the PWM module is configured and the period and pulse width are set. The Oscillator module generates frequencies corresponding to the value specified in these registers in every eighth PWM cycle.

```
//Software must unlock before the start of the tuning.
SYS_OSC_FRCTUNING_DATA tuningInput;
OSC_FRC_TUNING_VALUE tuningValue[8];

//Initialize with tuning values
tuningValue[0] = OSC_TUNE_TO_CENTRAL_MINUS_2_25_PERC;
tuningValue[1] = OSC_TUNE_TO_CENTRAL_MINUS_1_5_PERC;
tuningValue[2] = OSC_TUNE_TO_CENTRAL_MINUS_0_375_PERC;
tuningValue[3] = OSC_TUNE_TO_CENTRAL_PLUS_0_43_PERC;
tuningValue[4] = OSC_TUNE_TO_CENTRAL_PLUS_1_29_PERC;
tuningValue[5] = OSC_TUNE_TO_CENTRAL_PLUS_2_54_PERC;
tuningValue[6] = OSC_TUNE_TO_CENTRAL_MINUS_3_PERC;
tuningValue[7] = OSC_TUNE_TO_CENTRAL_MINUS_3_PERC;

tuningInput.tuningMode = OSC_TUNING_SEQ_DITHER;
tuningInput.tuningData = tuningValue;
SYS_OSC_FRCTuningSet(&tuningInput);
```

Pseudo-Random Number

In this method, select the tuning mode. Next, configure the PWM module and set the period and pulse width. The Oscillator system generates a 4-bit number based on a pseudo-random number generation algorithm. The Oscillator system then uses this value to tune the FRC oscillator. The module will generate different frequencies corresponding to the generated pseudo-random numbers every eighth PWM cycle.

```
//Software must unlock before the start of the tuning.
SYS_OSC_FRCTUNING_DATA tuningInput;
OSC_FRC_TUNING_VALUE tuningValue = 0x7FFF;

tuningInput.tuningMode = OSC_TUNING_PSEUDO_RANDOM;
tuningInput.tuningData = &tuningValue;
SYS_OSC_FRCTuningSet(&tuningInput);
```

Configuring the Library

Macros

	Name	Description
	SYS_CLK_CONFIG_FREQ_ERROR_LIMIT	Sets the allowed frequency configuration error Hertz.
	SYS_CLK_CONFIG_PRIMARY_XTAL	Gives the Primary oscillator input frequency. This is from the hardware and not the output required frequency.
	SYS_CLK_CONFIG_SECONDARY_XTAL	Gives the Secondary oscillator input frequency. This is from the hardware and not the output required frequency.

SYS_CLK_CONFIG_SYSPLL_INP_DIVISOR	Gives the System PLL input divisor configured through configuration bits. For some of the devices, the system PLL input divisor is not accessible through software. So use this macro to inform the System service about the input divisor.
SYS_CLK_CONFIGBIT_USBPLL_DIVISOR	Gives the Oscillator USB clock divisor.
SYS_CLK_CONFIGBIT_USBPLL_ENABLE	Defines the status of the USB PLL.
SYS_CLK_CONFIG_EXTERNAL_CLOCK	Gives the clock that is provided on the Reference clock output pin.
SYS_CLK_UPLL_BEFORE_DIV2_FREQ	Defines the value of the USB clock for the devices which does not have a software controllable USB clock.

Description

The configuration of the Clock System Service Library is based on the file `system_config.h`.

This header file contains the configuration selection for the Clock System Service Library. Based on the selections made, the Clock System Service Library may support the selected features. These configuration settings will apply to all instances of the Clock System Service Library.

This header can be placed anywhere, the path of this header needs to be present in the include search path for a successful build. Refer to the Applications Help section for more details.

SYS_CLK_CONFIG_FREQ_ERROR_LIMIT Macro

Sets the allowed frequency configuration error Hertz.

File

[sys_clk_config_template.h](#)

C

```
#define SYS_CLK_CONFIG_FREQ_ERROR_LIMIT 10
```

Description

Frequency Configuration error

This macro sets the allowed frequency error in Hertz.

Remarks

None.

SYS_CLK_CONFIG_PRIMARY_XTAL Macro

Gives the Primary oscillator input frequency. This is from the hardware and not the output required frequency.

File

[sys_clk_config_template.h](#)

C

```
#define SYS_CLK_CONFIG_PRIMARY_XTAL 8000000L
```

Description

Primary Oscillator clock input frequency

This macro gives the Primary Oscillator clock input frequency.

Remarks

Find this out by checking the hardware. examine the crystal connected to the Primary Oscillator input pin.

SYS_CLK_CONFIG_SECONDARY_XTAL Macro

Gives the Secondary oscillator input frequency. This is from the hardware and not the output required frequency.

File

[sys_clk_config_template.h](#)

C

```
#define SYS_CLK_CONFIG_SECONDARY_XTAL 8000000
```

Description

Secondary Oscillator clock input frequency

This macro gives the Secondary Oscillator clock input frequency.

Remarks

Find this out by checking the hardware. examine the crystal connected to the Secondary Oscillator input pin.

SYS_CLK_CONFIG_SYSPLL_INP_DIVISOR Macro

Gives the System PLL input divisor configured through configuration bits. For some of the devices, the system PLL input divisor is not accessible through software. So use this macro to inform the System service about the input divisor.

File

[sys_clk_config_template.h](#)

C

```
#define SYS_CLK_CONFIG_SYSPLL_INP_DIVISOR 2
```

Description

System PLL input divisor configured through configuration bits

Gives the System PLL input divisor configured through configuration bits. For some of the devices, the system PLL input divisor is not accessible through software. So use this macro to inform the System service about the input divisor. In case of some devices PLL expects its input frequency to be in a specific range. So it is important to set the input divisor get the right input frequency.

Remarks

This must be changed every time you change the System PLL input divisor through the configuration bits.

SYS_CLK_CONFIGBIT_USBPLL_DIVISOR Macro

Gives the Oscillator USB clock divisor.

File

[sys_clk_config_template.h](#)

C

```
#define SYS_CLK_CONFIGBIT_USBPLL_DIVISOR 2
```

Description

Oscillator USB clock divisor configured through configuration bits

This macro sets up the Oscillator USB clock divisor. Use OSC_CONFIGBITS_PLLIDIV to select the configuration.

Remarks

This must be changed every time you change the USB PLL divisor through the configuration bits.

SYS_CLK_CONFIGBIT_USBPLL_ENABLE Macro

Defines the status of the USB PLL.

File

[sys_clk_config_template.h](#)

C

```
#define SYS_CLK_CONFIGBIT_USBPLL_ENABLE true
```

Description

System clock PLL enable configuration

Setting this macro to 'true' states that USB PLL is enabled through the Configuration bits.

Remarks

This must be changed every time you enable/disable the USB PLL through the configuration bits.

SYS_CLK_CONFIG_EXTERNAL_CLOCK Macro

Gives the clock that is provided on the Reference clock output pin.

File

[sys_clk_config_template.h](#)

C

```
#define SYS_CLK_CONFIG_EXTERNAL_CLOCK 4000000
```

Description

External clock frequency

This macro gives the clock that is provided on the Reference clock output pin.

Remarks

None.

SYS_CLK_UPLL_BEFORE_DIV2_FREQ Macro

Defines the value of the USB clock for the devices which does not have a software controllable USB clock.

File

[sys_clk_config_template.h](#)

C

```
#define SYS_CLK_UPLL_BEFORE_DIV2_FREQ 48000000
```

Description

USB clock frequency

Defines the value of the USB clock for the devices which does not have a software controllable USB clock(Fully controlled through configuration bits).

Remarks

None.

Building the Library

This section lists the files that are available in the Clock System Service Library.

Description

The following three tables list and describe the header (.h) and source (.c) files that implement this library. The parent folder for these files is <install-dir>/framework/system/clock.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using #include) by any code that uses this library.

Source File Name	Description
sys_clk.h	Clock System Service Library API header file.

Required File(s)



All of the required files listed in the following table are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

Source File Name	Description
/src/sys_clk.c	Clock System Service Library implementation.
/src/sys_clk_pic32mx.c	System clock implementation specific to PIC32MX devices.
/src/sys_clk_pic32mx.h	

```
/src/sys_clk_pic32mz.c
/src/sys_clk_pic32mz.h
```

System clock implementation specific to PIC32MZ devices.

Optional File(s)



This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
N/A	No optional files are available for this library.



Module Dependencies

The Clock System Service is not dependent upon other modules.




Library Interface**a) System Clock Functions**

	Name	Description
	SYS_CLK_SystemFrequencyGet	Gets the system clock frequency in Hertz. Implementation: Static/Dynamic
	SYS_CLK_SystemFrequencySet	Configures the System clock of the device to the value specified. Implementation: Dynamic



b) Peripheral Clock Functions

	Name	Description
	SYS_CLK_PeripheralFrequencyGet	Gets the selected clock peripheral bus frequency in Hertz. Implementation: Static/Dynamic
	SYS_CLK_PeripheralFrequencySet	Configures the peripheral clock of the device to the value specified. Implementation: Dynamic


c) Reference Clock Functions

	Name	Description
	SYS_CLK_ReferenceClockSetup	Configures the reference clock of the device to the value specified. Implementation: Dynamic
	SYS_CLK_ReferenceFrequencySet	Configures the reference clock of the device to the value specified. Implementation: Dynamic
	SYS_CLK_ReferenceFrequencyGet	Gets the selected Reference clock bus frequency in Hertz. Implementation: Static/Dynamic




d) Clock Failure Functions

	Name	Description
	SYS_CLK_ClockFailureCallbackRegister	Allows registration of a call back function that will be triggered on a clock failure. Implementation: Dynamic
	SYS_CLK_TaskError	Informs the user on a clock failure by invoking the registered call back function. Implementation: Dynamic


e) Initialization Functions

	Name	Description
	SYS_CLK_Initialize	Initializes hardware and internal data structure of the System Clock. Implementation: Static/Dynamic

f) Secondary Oscillator Functions

	Name	Description
	SYS_CLK_SecondaryOscillatorDisable	Disables the secondary oscillator. Implementation: Static/Dynamic
	SYS_CLK_SecondaryOscillatorEnable	Enables the secondary oscillator. Implementation: Static/Dynamic
	SYS_CLK_SecondaryOscillatorIsEnabled	Identifies whether secondary oscillator is enabled or disabled. Implementation: Static/Dynamic

g) Data Types and Constants

	Name	Description
	SYS_CLK_INIT	Defines the data required to initialize the Oscillator for the Clock System Service.
	CLK_BUSES_PERIPHERAL	Lists the available peripheral clock buses.
	CLK_BUSES_REFERENCE	Lists the available reference clock buses.
	CLK_SOURCES_PERIPHERAL	Lists the available clock sources for the peripheral clock.
	CLK_SOURCES_REFERENCE	Lists the available clock sources for the Reference clock.
	CLK_SOURCES_SYSTEM	Lists the available clock sources for the system clock.
	SYS_CLK_ERROR_HANDLER	Pointer to a CLK System service SYS_CLK_ClockFailureCallbackRegister function.
	SYS_CLK_REFERENCE_SETUP	Defines the data required to initialize the Oscillator for the Clock System Service.
	SYS_CLK_STATUS	Lists all the possible status of a system clock operation.
	SYS_CLK_FRCTune	This function is used for direct value based FRC oscillator tuning. Implementation: Dynamic
	SYS_CLK_FRC_TUNING_TYPE	Defines the data type for tuning the FRC Oscillator.

Description

This section describes the APIs of the Clock System Service Library.

Refer to each section for a detailed description.

a) System Clock Functions

SYS_CLK_SystemFrequencyGet Function

Gets the system clock frequency in Hertz.

Implementation: Static/Dynamic

File

[sys_clk.h](#)

C

```
uint32_t SYS_CLK_SystemFrequencyGet();
```

Returns

System clock frequency in Hertz.

Description

This function gets the System clock frequency in Hertz.

Preconditions

The [SYS_CLK_Initialize](#) function should be called before calling this API.

Example

```
uint32_t sysClockHz;

sysClockHz = SYS_CLK_SystemFrequencyGet ( );
```

Function

```
uint32_t SYS_CLK_SystemFrequencyGet ( void )
```

SYS_CLK_SystemFrequencySet Function

Configures the System clock of the device to the value specified.

Implementation: Dynamic

File

[sys_clk.h](#)

C

```
uint32_t SYS_CLK_SystemFrequencySet(CLK_SOURCES_SYSTEM systemSource, uint32_t systemClockHz, bool
waitUntilComplete);
```

Returns

'Zero' on an unsuccessful operation. Configured system clock frequency on successful operation.

Description

This function configures the clock multipliers and divisors to achieve requested System clock frequency. Initially it checks the difference between the requested value and possible value. If it is not within '[SYS_CLK_CONFIG_FREQ_ERROR_LIMIT](#)', the registers values will not be changed and a value '0' will be returned to let user know that the operation was unsuccessful. If the value requested is acceptable, then it writes to the oscillator registers and return with the newly set frequency.

Remarks

Getting the new frequency doesn't mean that the operation is complete if the API is not called to 'wait until' the operation is complete. The API will return with the possible value immediately. The actual hardware switching may take longer. Use [SYS_CLK_SystemClockStatus](#) function to get the status of completion.

Calling this API is only necessary to change the system clock. Use [SYS_CLK_Initialize](#) function to set up the system clock initially.

Preconditions

The [SYS_CLK_Initialize](#) function should be called before calling this API. To change the clock source, clock switching must be enabled through 'configuration bits'.

Example

```
uint32_t achievedFrequencyHz;

// ***** Blocking call*****
achievedFrequencyHz = SYS_CLK_SystemFrequencySet (
    SYS_CLK_SOURCE_PRIMARY, 8000000, true );

if ( achievedFrequencyHz != 0 )
{
    //Clock setting complete
}
```

Parameters

Parameters	Description
systemSource	Clock source means the path of the clock signal. One of the possible value from CLK_SOURCES_SYSTEM enum. See the block diagram in the data sheet to get an idea.
systemClockHz	Required System Clock frequency in Hertz.
waitUntilComplete	Passing this parameter as 'true' will make until the actual hardware operation is complete.

Function

```
uint32_t SYS_CLK_SystemFrequencySet ( CLK_SOURCES_SYSTEM systemSource,
uint32_t systemClockHz, bool waitUntilComplete )
```

b) Peripheral Clock Functions**SYS_CLK_PeripheralFrequencyGet Function**

Gets the selected clock peripheral bus frequency in Hertz.

Implementation: Static/Dynamic

File

[sys_clk.h](#)

C

```
uint32_t SYS_CLK_PeripheralFrequencyGet(CLK_BUSES_PERIPHERAL peripheralBus);
```

Returns

Clock frequency in Hertz.

Description

This function gets the selected peripheral bus clock frequency in Hertz.

Remarks

Most devices do not have multiple peripheral clock buses. In those instances, pass CLK_USB_PERIPHERAL_1 as the bus number.

Preconditions

The [SYS_CLK_Initialize](#) function should be called before calling this API.

Example

```
unsigned long peripheralClockHz;

peripheralClockHz = SYS_CLK_PeripheralFrequencyGet ( CLK_BUS_PERIPHERAL_5 );
```

Parameters

Parameters	Description
peripheralBus	Reference clock bus selection. One of the possible value from CLK_BUSES_PERIPHERAL enum. For devices that do not have multiple clock channels for Reference clock, CLK_BUS_PERIPHERAL_1 should be the selection.

Function

```
uint32_t SYS_CLK_PeripheralFrequencyGet ( CLK_BUSES_PERIPHERAL peripheralBus )
```

SYS_CLK_PeripheralFrequencySet Function

Configures the peripheral clock of the device to the value specified.

Implementation: Dynamic

File

[sys_clk.h](#)

C

```
uint32_t SYS_CLK_PeripheralFrequencySet(CLK_BUSES_PERIPHERAL peripheralBus, CLK_SOURCES_PERIPHERAL
peripheralSource, uint32_t peripheralClockHz, bool waitUntilComplete);
```

Returns

None.

Description

This function configures the clock multipliers and divisors to achieve requested Peripheral clock frequency. Initially it checks the difference between the requested value and possible value. If it is not within '[SYS_CLK_CONFIG_FREQ_ERROR_LIMIT](#)', the registers values will not be changed and a value '0' will be returned to let user know that the operation was unsuccessful. If the value requested is acceptable, then it writes to the oscillator registers and return with the newly set frequency. If the operation is a failure, [SYS_CLK_PeripheralClosestFrequencyGet](#) function will give the closest possible frequency. If the closest possible value is acceptable, user can reconfigure with that value.

Remarks

Most devices do not have multiple Peripheral clock buses. In those instances, pass CLK_BUS_PERIPHERAL_1 as the bus number.

Preconditions

The [SYS_CLK_Initialize](#) function should be called before calling this API.

Example

```
uint32_t achievedFrequencyHz;

//Example for MX, single peripheral bus
achievedFrequencyHz = SYS_CLK_PeripheralFrequencySet ( CLK_BUS_PERIPHERAL_1,
CLK_SOURCE_PERIPHERAL_SYSTEMCLK, 8000000, true );

if ( achievedFrequencyHz != 0 )
{
    //Frequency successfully set
}
```

```
//Example for MZ, multiple peripheral buses
achievedFrequencyHz = SYS_CLK_PeripheralFrequencySet ( CLK_BUS_PERIPHERAL_4,
    CLK_SOURCE_PERIPHERAL_SYSTEMCLK, 8000000, true );

if ( achievedFrequencyHz != 0 )
{
    //Frequency successfully set
}
```

Parameters

Parameters	Description
peripheralBus	Peripheral bus selection. One of the possible value from CLK_BUSES_PERIPHERAL enum. For devices that do not have multiple clock channels for Peripheral clock, CLK_BUS_PERIPHERAL_1 should be the selection.
peripheralSource	Clock source means the path of the clock signal. One of the possible value from CLK_SOURCES_PERIPHERAL enum. See the block diagram in the data sheet to get an idea.
peripheralClockHz	Clock frequency in Hertz.
waitUntilComplete	Passing this parameter as 'true' will make until the actual hardware operation is complete.

Function

```
uint32_t SYS_CLK_PeripheralFrequencySet ( CLK\_BUSES\_PERIPHERAL peripheralBus,
    CLK\_SOURCES\_PERIPHERAL peripheralSource, uint32_t peripheralClockHz,
    bool waitUntilComplete )
```

c) Reference Clock Functions

SYS_CLK_ReferenceClockSetup Function

Configures the reference clock of the device to the value specified.

Implementation: Dynamic

File

[sys_clk.h](#)

C

```
void SYS_CLK_ReferenceClockSetup(CLK_BUSES_REFERENCE referenceBus, SYS_CLK_REFERENCE_SETUP * refSetup);
```

Returns

None.

Description

This function configures the clock multipliers and divisors to achieve requested Reference clock frequency. Initially it checks the difference between the requested value and possible value. If it is not within '[SYS_CLK_CONFIG_FREQ_ERROR_LIMIT](#)', the registers values will not be changed and a value '0' will be returned to let user know that the operation was unsuccessful. If the value requested is acceptable, then it writes to the oscillator registers and return with the newly set frequency. If the operation is a failure, [SYS_CLK_ReferenceClosestFrequencyGet](#) function will give the closest possible frequency. If the closest possible value is acceptable, user can reconfigure with that value.

Remarks

Most devices do not have multiple reference clock buses. In those instances, pass CLK_BUS_REFERENCE_1 as the bus number.

Preconditions

The [SYS_CLK_Initialize](#) function should be called before calling this API.

Example

```
SYS_CLK_REFERENCE_SETUP refSetup;

refSetup.suspendInSleep = true;
refSetup.stopInIdle = true;

SYS_CLK_ReferenceClockSetup ( ( CLK_BUS_REFERENCE_3, &refSetup );
```

Parameters

Parameters	Description
referenceBus	Reference clock bus selection. One of the possible value from CLK_BUSES_REFERENCE enum. For devices that do not have multiple clock channels for Reference clock, CLK_BUS_REFERENCE_1 should be the selection.
refSetup	A structure which holds the reference oscillator configuration.

Function

```
void SYS_CLK_ReferenceClockSetup ( CLK\_BUSES\_REFERENCE referenceBus,
                                   SYS\_CLK\_REFERENCE\_SETUP refSetup )
```

SYS_CLK_ReferenceFrequencySet Function

Configures the reference clock of the device to the value specified.

Implementation: Dynamic

File

[sys_clk.h](#)

C

```
uint32_t SYS_CLK_ReferenceFrequencySet(CLK_BUSES_REFERENCE referenceBus, CLK_SOURCES_REFERENCE
referenceSource, uint32_t referenceClockHz, bool waitUntilComplete);
```

Returns

None.

Description

This function configures the clock multipliers and divisors to achieve requested Reference clock frequency. Initially it checks the difference between the requested value and possible value. If it is not within '[SYS_CLK_CONFIG_FREQ_ERROR_LIMIT](#)', the registers values will not be changed and a value '0' will be returned to let user know that the operation was unsuccessful. If the value requested is acceptable, then it writes to the oscillator registers and return with the newly set frequency. If the operation is a failure, SYS_CLK_ReferenceClosestFrequencyGet function will give the closest possible frequency. If the closest possible value is acceptable, user can reconfigure with that value.

Remarks

Most devices do not have multiple reference clock buses. In those instances, pass CLK_BUS_REFERENCE_1 as the bus number.

Preconditions

The [SYS_CLK_Initialize](#) function should be called before calling this API. To change the clock source clock switching must be enabled through 'configuration bits'.

Example

```
uint32_t achievedFrequencyHz;

//I want 200 KHz as output
achievedFrequencyHz = SYS_CLK_ReferenceFrequencySet ( CLK_BUS_REFERENCE_3,
  CLK_SOURCE_REF_FRC, 200000, true );

if ( achievedFrequencyHz != 0 )
{
    //Frequency successfully set
}
```

Parameters

Parameters	Description
referenceBus	Reference clock bus selection. One of the possible value from CLK_BUSES_REFERENCE enum. For devices that do not have multiple clock channels for Reference clock, CLK_BUS_REFERENCE_1 should be the selection.
referenceSource	Clock source means the path of the clock signal. One of the possible value from CLK_SOURCES_REFERENCE enum. See the block diagram in the data sheet to get an idea.
referenceClockHz	Clock frequency in Hertz.
waitUntilComplete	Passing this parameter as 'true' will make until the actual hardware operation is complete.

Function

```
uint32_t SYS_CLK_ReferenceFrequencySet ( CLK\_BUSES\_REFERENCE referenceBus,  
                                         CLK\_SOURCES\_REFERENCE referenceSource, uint32_t referenceClockHz,  
                                         bool waitUntilComplete )
```

SYS_CLK_ReferenceFrequencyGet Function

Gets the selected Reference clock bus frequency in Hertz.

Implementation: Static/Dynamic

File

[sys_clk.h](#)

C

```
uint32_t SYS_CLK_ReferenceFrequencyGet(CLK\_BUSES\_REFERENCE referenceBus);
```

Returns

Clock frequency in Hertz.

Description

This function gets frequency of the selected Reference clock bus in Hertz.

Remarks

None.

Preconditions

The [SYS_CLK_Initialize](#) function should be called before calling this API.

Example

```
unsigned long sysClockOutputHz;  
  
sysClockOutputHz = SYS_CLK_ReferenceFrequencyGet ( CLK\_BUS\_REFERENCE\_3 );
```

Parameters

Parameters	Description
peripheralBus	Reference clock bus selection. One of the possible value from CLK_BUSES_REFERENCE enum. For devices that do not have multiple clock channels for Reference clock, CLK_BUS_REFERENCE_1 should be the selection.

Function

```
uint32_t SYS_CLK_ReferenceFrequencyGet ( CLK\_BUSES\_REFERENCE referenceBus )
```

d) Clock Failure Functions

SYS_CLK_ClockFailureCallbackRegister Function

Allows registration of a call back function that will be triggered on a clock failure.

Implementation: Dynamic

File

[sys_clk.h](#)

C

```
void SYS_CLK_ClockFailureCallbackRegister(SYS\_CLK\_ERROR\_HANDLER callback);
```

Returns

None.

Description

This function allows registration of a callback function that will be triggered on a clock failure.

Remarks

This function is not available on all devices. Please refer to the specific device data sheet for availability.

Preconditions

None.

Example

```
SYS_CLK_ClockFailureCallbackRegister ( &ErrorHandle );

void ErrorHandle (void)
{
    //Handle the error.
}
```

Parameters

Parameters	Description
callback	Address of a callback function of type ' SYS_CLK_ERROR_HANDLER '.

Function

void SYS_CLK_ClockFailureCallbackRegister ([SYS_CLK_ERROR_HANDLER](#) callback)

SYS_CLK_TaskError Function

Informs the user on a clock failure by invoking the registered call back function.

Implementation: Dynamic

File

[sys_clk.h](#)

C

```
void SYS_CLK_TaskError();
```

Returns

None.

Description

This function informs the user on a clock failure by invoking the registered call back function. This must be called from the Fail Safe Clock Monitor (FSCM) interrupt service routine.

Remarks

None.

Preconditions

None.

Example

```
void clockMonitorISR ( void )
{
    SYS_CLK_TaskError ( void );
}
```

Parameters

Parameters	Description
index	Identifies the desired System Clock

Function

void SYS_CLK_TaskError (void)

e) Initialization Functions

SYS_CLK_Initialize Function

Initializes hardware and internal data structure of the System Clock.

Implementation: Static/Dynamic

File

[sys_clk.h](#)

C

```
void SYS_CLK_Initialize(const SYS_CLK_INIT * clkInit);
```

Returns

None.

Description

This function initializes the hardware and internal data structure of System Clock Service.

Remarks

This is API must be called at the time of system initialization to initialize the oscillator. Call this again to change any of the initialization settings.

If the frequency settings are already done through the 'configuration bits', call this API by passing 'NULL' to it. In any case calling this API is necessary get the internal data structures updated.

To change the clock source clock switching must be enabled through 'configuration bits'.

Preconditions

None.

Example

```
//Example 1: Do not alter the configuration bit settings
SYS_CLK_Initialize ( NULL );

//Example 2: Want to change the configuration bit settings
SYS_CLK_INIT generalInit;

// Populate the oscillator initialization structure
generalInit.systemClockSource = SYS_CLK_SOURCE_FRC_SYSPLL;
generalInit.systemClockFrequencyHz = 30000000;
generalInit.onWaitInstruction = OSC_ON_WAIT_SLEEP;
generalInit.secondaryOscKeepEnabled = true;
SYS_CLK_Initialize (&generalInit);

//Wait until complete
while ( SYS_CLK_SystemOperationStatus ( ) != SYS_CLK_OPERATION_COMPLETE );

//Configuration success
```

Parameters

Parameters	Description
clkInit	Pointer to a data structure containing any data necessary to initialize the System Clock. This pointer can be NULL if no data is required as static overrides have been provided.

Function

```
void SYS_CLK_Initialize ( const SYS_CLK_INIT * clkInit )
```

f) Secondary Oscillator Functions

SYS_CLK_SecondaryOscillatorDisable Function

Disables the secondary oscillator.

Implementation: Static/Dynamic

File

[sys_clk.h](#)

C

```
void SYS_CLK_SecondaryOscillatorDisable();
```

Returns

None.

Description

This function disables the secondary oscillator.

Remarks

None.

Preconditions

The [SYS_CLK_Initialize](#) function should be called before calling this API.

Example

```
SYS_CLK_SecondaryOscillatorDisable ( );
```

Function

```
void SYS_CLK_SecondaryOscillatorDisable ( void )
```

SYS_CLK_SecondaryOscillatorEnable Function

Enables the secondary oscillator.

Implementation: Static/Dynamic

File

[sys_clk.h](#)

C

```
void SYS_CLK_SecondaryOscillatorEnable();
```

Returns

None.

Description

This function enables secondary oscillator which can be used as a clock source for peripherals like RTCC, Timer etc.. The SOSOC requires a warm-up period of 1024 before it can be used as a clock source.

Remarks

None.

Preconditions

The [SYS_CLK_Initialize](#) function should be called before calling this API.

Example

```
SYS_CLK_SecondaryOscillatorEnable ( );
```

Function

```
void SYS_CLK_SecondaryOscillatorEnable ( void )
```


SYS_CLK_SecondaryOscillatorIsEnabled Function

Identifies whether secondary oscillator is enabled or disabled.

Implementation: Static/Dynamic

File

[sys_clk.h](#)

C

```
bool SYS_CLK_SecondaryOscillatorIsEnabled();
```

Returns

true - If Secondary oscillator is enabled. false - If Secondary oscillator is disabled.

Description

This function identifies whether the secondary oscillator is enabled or disabled.

Remarks

None.

Preconditions

The [SYS_CLK_Initialize](#) function should be called before calling this API.

Example

```
bool status;

status = SYS_CLK_SecondaryOscillatorIsEnabled ( );
```

Function

```
bool SYS_CLK_SecondaryOscillatorIsEnabled ( void )
```

g) Data Types and Constants

SYS_CLK_INIT Structure

Defines the data required to initialize the Oscillator for the Clock System Service.

File

[sys_clk.h](#)

C

```
typedef struct {
    SYS_MODULE_INIT moduleInit;
    CLK_SOURCES_SYSTEM systemClockSource;
    uint32_t systemClockFrequencyHz;
    bool waitTillComplete;
    bool secondaryOscKeepEnabled;
    OSC_OPERATION_ON_WAIT onWaitInstruction;
} SYS_CLK_INIT;
```

Members

Members	Description
SYS_MODULE_INIT moduleInit;	System module initialization
CLK_SOURCES_SYSTEM systemClockSource;	Initializations for System clock
uint32_t systemClockFrequencyHz;	Set the System clock frequency
bool waitTillComplete;	Wait until the clock switch operation is completed
bool secondaryOscKeepEnabled;	Even though the secondary oscillator is not used, keeping the oscillator running, allows a fast switch to the lower system clock for low-power operation
OSC_OPERATION_ON_WAIT onWaitInstruction;	System action on a 'Wait' instruction

Description

Clock System Service Reference Oscillator initialization data

This structure defines the data required to initialize the Oscillator for the Clock System Service.

Remarks

This structure only includes the necessary settings for the clock module. Other features like USB clock and reference clock are considered to be optional and not every system will use it. There are dedicated APIs for configuring those.

CLK_BUSES_PERIPHERAL Enumeration

Lists the available peripheral clock buses.

File

[sys_clk.h](#)

C

```
typedef enum {
    CLK_BUS_PERIPHERAL_1 = 0x00,
    CLK_BUS_PERIPHERAL_2 = 0x01,
    CLK_BUS_PERIPHERAL_3 = 0x02,
    CLK_BUS_PERIPHERAL_4 = 0x03,
    CLK_BUS_PERIPHERAL_5 = 0x04,
    CLK_BUS_PERIPHERAL_6 = 0x05,
    CLK_BUS_PERIPHERAL_7 = 0x06,
    CLK_BUS_PERIPHERAL_8 = 0x07
} CLK_BUSES_PERIPHERAL;
```

Members

Members	Description
CLK_BUS_PERIPHERAL_1 = 0x00	Select peripheral bus 1. This should be the selection for the devices which doesn't have multiple peripheral buses
CLK_BUS_PERIPHERAL_2 = 0x01	Select peripheral bus 2
CLK_BUS_PERIPHERAL_3 = 0x02	Select peripheral bus 3
CLK_BUS_PERIPHERAL_4 = 0x03	Select peripheral bus 4
CLK_BUS_PERIPHERAL_5 = 0x04	Select peripheral bus 5
CLK_BUS_PERIPHERAL_6 = 0x05	Select peripheral bus 6
CLK_BUS_PERIPHERAL_7 = 0x06	Select peripheral bus 7
CLK_BUS_PERIPHERAL_8 = 0x07	Select peripheral bus 8

Description

Peripherals Clock Buses enumeration

This enumeration lists all of the available peripheral clock buses. This is used by the [SYS_CLK_PeripheralFrequencyGet](#) and [SYS_CLK_PeripheralFrequencySet](#) functions.

Remarks

The system service covers a wide range of devices. Not all the elements listed in here will be relevant for the your device.

CLK_BUSES_REFERENCE Enumeration

Lists the available reference clock buses.

File

[sys_clk.h](#)

C

```
typedef enum {
    CLK_BUS_REFERENCE_1 = 0x00,
    CLK_BUS_REFERENCE_2 = 0x01,
    CLK_BUS_REFERENCE_3 = 0x02,
    CLK_BUS_REFERENCE_4 = 0x03,
    CLK_BUS_REFERENCE_5 = 0x04
} CLK_BUSES_REFERENCE;
```

Members

Members	Description
CLK_BUS_REFERENCE_1 = 0x00	Select Reference bus 1. This should be the selection for the devices which doesn't have multiple Reference buses
CLK_BUS_REFERENCE_2 = 0x01	Select Reference bus 2
CLK_BUS_REFERENCE_3 = 0x02	Select Reference bus 3
CLK_BUS_REFERENCE_4 = 0x03	Select Reference bus 4
CLK_BUS_REFERENCE_5 = 0x04	Select Reference bus 5

Description

Reference Clock Buses enumeration

This enumeration lists all of the available Reference clock buses. This is used by the [SYS_CLK_ReferenceFrequencyGet](#) and [SYS_CLK_ReferenceFrequencySet](#) functions.

Remarks

The system service covers a wide range of devices. Not all the elements listed in here will be relevant for the your device.

CLK_SOURCES_PERIPHERAL Enumeration

Lists the available clock sources for the peripheral clock.

File

[sys_clk.h](#)

C

```
typedef enum {
    CLK_SOURCE_PERIPHERAL_SYSTEMCLK
} CLK_SOURCES_PERIPHERAL;
```

Members

Members	Description
CLK_SOURCE_PERIPHERAL_SYSTEMCLK	Source of clock is system clock

Description

Peripheral clock sources enumeration

This enumeration lists all the available peripheral clock sources. This is used by the [SYS_CLK_PeripheralFrequencySet](#) function.

Remarks

None.

CLK_SOURCES_REFERENCE Enumeration

Lists the available clock sources for the Reference clock.

File

[sys_clk.h](#)

C

```
typedef enum {
    CLK_SOURCE_REF_SYSTEMCLK,
    CLK_SOURCE_REF_PBCLK_BUS1,
    CLK_SOURCE_REF_PRIMARY,
    CLK_SOURCE_REF_FRC,
    CLK_SOURCE_REF_LPRC,
    CLK_SOURCE_REF_SECONDARY,
    CLK_SOURCE_REF_USBPLL_OUT,
    CLK_SOURCE_REF_SYSPLL_OUT,
    CLK_SOURCE_REF_EXTERNAL,
    CLK_SOURCE_REF_BFRC
} CLK_SOURCES_REFERENCE;
```

Members

Members	Description
CLK_SOURCE_REF_SYSTEMCLK	Source of Reference clock is System clock
CLK_SOURCE_REF_PBCLK_BUS1	Source of Reference clock is Peripheral clock
CLK_SOURCE_REF_PRIMARY	Source of Reference clock is Primary oscillator
CLK_SOURCE_REF_FRC	Source of Reference clock is Fast RC oscillator
CLK_SOURCE_REF_LPRC	Source of Reference clock is Low power RC oscillator
CLK_SOURCE_REF_SECONDARY	Source of Reference clock is Secondary oscillator
CLK_SOURCE_REF_USBPll_OUT	Source of clock is output of USB PLL This is used only for the Reference clock.
CLK_SOURCE_REF_SYSPll_OUT	Source of clock is the output of System PLL. Input to the system PLL may be Primary or FRC
CLK_SOURCE_REF_EXTERNAL	Source of clock is external(from the pin)
CLK_SOURCE_REF_BFRC	Source of Reference clock is backup Fast RC oscillator

Description

Reference clock sources enumeration

This enumeration lists all the available peripheral clock sources. This is used by the [SYS_CLK_ReferenceFrequencySet](#) function.

Remarks

None.

CLK_SOURCES_SYSTEM Enumeration

Lists the available clock sources for the system clock.

File

[sys_clk.h](#)

C

```
typedef enum {
    SYS_CLK_SOURCE_FRC,
    SYS_CLK_SOURCE_FRC_SYSPll,
    SYS_CLK_SOURCE_PRIMARY,
    SYS_CLK_SOURCE_PRIMARY_SYSPll,
    SYS_CLK_SOURCE_SECONDARY,
    SYS_CLK_SOURCE_LPRC,
    SYS_CLK_SOURCE_FRC_BY_16,
    SYS_CLK_SOURCE_FRC_BY_DIV
} CLK_SOURCES_SYSTEM;
```

Members

Members	Description
SYS_CLK_SOURCE_FRC	Source of clock is internal fast RC
SYS_CLK_SOURCE_FRC_SYSPll	Source of clock is internal fast RC multiplied by system PLL
SYS_CLK_SOURCE_PRIMARY	Source of clock is primary oscillator
SYS_CLK_SOURCE_PRIMARY_SYSPll	Source of clock is primary oscillator multiplied by the System PLL value and divided by the divisor configured by software
SYS_CLK_SOURCE_SECONDARY	Source of clock is secondary oscillator
SYS_CLK_SOURCE_LPRC	Source of clock is internal low power RC
SYS_CLK_SOURCE_FRC_BY_16	Source of clock is internal fast RC divided by the divisor configured in software
SYS_CLK_SOURCE_FRC_BY_DIV	Source of clock is internal fast RC divided by the divisor configured in software

Description

System Clock sources enumeration

This enumeration lists all the available clock sources for the system clock. This is used by the [SYS_CLK_INIT](#) structure.

Remarks

None.

SYS_CLK_ERROR_HANDLER Type

Pointer to a CLK System service [SYS_CLK_ClockFailureCallbackRegister](#) function.

File

[sys_clk.h](#)

C

```
typedef void (* SYS_CLK_ERROR_HANDLER)(CLK_SOURCES_SYSTEM systemSource, uint32_t systemFrequencyHz);
```

Returns

None.

Description

CLK System Service Error Handler Function Pointer

This data type defines the type of the CLK System Service Error Handler callback function. The parameters are described here and a partial example implementation is provided. On a clock failure, the device will switch its clock source to the FRC clock. By registering the callback function user will get the current System clock source and the clock frequency. User can use this information to try switching back.

Remarks

None.

Example

```
void SYS_CLK_ClockFailureCallbackRegister( errorHandler );

errorHandler ( CLK_SOURCES_SYSTEM systemSource, uint32_t systemFrequencyHz )
{
    //Log a warning
    //Try switching back to the old source
}
```

Parameters

Parameters	Description
systemSource	System clock source.
systemFrequencyHz	System clock frequency in hertz.

SYS_CLK_REFERENCE_SETUP Structure

Defines the data required to initialize the Oscillator for the Clock System Service.

File

[sys_clk.h](#)

C

```
typedef struct {
    bool suspendInSleep;
    bool stopInIdle;
} SYS_CLK_REFERENCE_SETUP;
```

Members

Members	Description
bool suspendInSleep;	Reference clock module should be stop its operation in 'Sleep' mode
bool stopInIdle;	Reference clock module should be stop its operation in 'Idle' mode

Description

Clock System Service Reference Oscillator initialization data

This structure defines the data required to initialize the Oscillator for the Clock System Service.

Remarks

This structure only includes the necessary settings for the clock module. Other features like USB clock and reference clock are considered to be optional and not every system will use it. There are dedicated APIs for configuring those.

SYS_CLK_STATUS Enumeration

Lists all the possible status of a system clock operation.

File

[sys_clk.h](#)

C

```
typedef enum {
    SYS_CLK_OPERATION_COMPLETE,
    SYS_CLK_OPERATION_IN_PROGRESS,
    SYS_CLK_OPERATION_FAIL_SWITCHING_DISABLED,
    SYS_CLK_OPERATION_FAIL_NOT_ACHIEVABLE,
    SYS_CLK_OPERATION_FAIL_REGISTER_NOT_READY,
    SYS_CLK_OPERATION_FAIL_USBPLL_NOT_ENABLED,
    SYS_CLK_OPERATION_FAIL_PLL_LOCK_TIMEOUT,
    SYS_CLK_PLL_INVALID_INP_FREQUENCY
} SYS_CLK_STATUS;
```

Members

Members	Description
SYS_CLK_OPERATION_COMPLETE	The last operation was successful
SYS_CLK_OPERATION_IN_PROGRESS	Operation started, may be PLL is not yet locked
SYS_CLK_OPERATION_FAIL_SWITCHING_DISABLED	The last operation was not successful, Clock switching is not enabled through configuration bits
SYS_CLK_OPERATION_FAIL_NOT_ACHIEVABLE	The last operation was not successful, requested frequency is not achievable
SYS_CLK_OPERATION_FAIL_REGISTER_NOT_READY	The last operation was not successful, register is not ready to take the new value
SYS_CLK_OPERATION_FAIL_USBPLL_NOT_ENABLED	The last operation was not successful, USB PLL is not enabled in through the 'configuration bits'. The same must be update in system_config.h
SYS_CLK_OPERATION_FAIL_PLL_LOCK_TIMEOUT	PLL Lock time out. A restart is the only option
SYS_CLK_PLL_INVALID_INP_FREQUENCY	PLL cannot work with the current input clock frequency, adjust the clock input or the input divisor. Check the device data sheet to know the PLL input requirement

Description

System clock operation status

This enumeration lists all the possible status of a system clock operation.

Remarks

None.

SYS_CLK_FRCTune Function

This function is used for direct value based FRC oscillator tuning.

Implementation: Dynamic

File

[sys_clk.h](#)

C

```
void SYS_CLK_FRCTune(SYS_CLK_FRC_TUNING_TYPE tuningData);
```

Returns

None.

Description

This function tunes the FRC as per the given value. FRC tuning functionality has been provided to help customers compensate for temperature effects on the FRC frequency over a wide range of temperatures.

Remarks

The tuning step size is an approximation, and is neither characterized, nor tested.

This API can only be used with devices that support direct value based FRC tuning. Refer to the specific device data sheet to determine whether this feature exists for your device.

Preconditions

The device selected must support the oscillator tuning feature.

Example

```
SYS_CLK_FRCTune ( 0x30 );
```

Parameters

Parameters	Description
tuningData	One of the possible value of Tuning. Refer to the specific device data sheet for possible values.

Function

```
void SYS_CLK_FRCTune ( SYS_CLK_FRC_TUNING_TYPE tuningData )
```

SYS_CLK_FRC_TUNING_TYPE Type

Defines the data type for tuning the FRC Oscillator.

File

[sys_clk.h](#)

C

```
typedef unsigned int SYS_CLK_FRC_TUNING_TYPE;
```

Description

Clock module FRC tuning type

This structure defines the data type to tune the FRC Oscillator.

Remarks

None.

Files

Files

Name	Description
sys_clk.h	Clock system service interface header.
sys_clk_config_template.h	Clock System Service configuration templates.

Description

This section lists the source and header files used by the library.

sys_clk.h














Clock system service interface header.

Enumerations

	Name	Description
	CLK_BUSES_PERIPHERAL	Lists the available peripheral clock buses.
	CLK_BUSES_REFERENCE	Lists the available reference clock buses.
	CLK_SOURCES_PERIPHERAL	Lists the available clock sources for the peripheral clock.
	CLK_SOURCES_REFERENCE	Lists the available clock sources for the Reference clock.
	CLK_SOURCES_SYSTEM	Lists the available clock sources for the system clock.
	SYS_CLK_STATUS	Lists all the possible status of a system clock operation.

Functions

	Name	Description
	SYS_CLK_ClockFailureCallbackRegister	Allows registration of a call back function that will be triggered on a clock failure. Implementation: Dynamic

	SYS_CLK_FRCTune	This function is used for direct value based FRC oscillator tuning. Implementation: Dynamic
	SYS_CLK_Initialize	Initializes hardware and internal data structure of the System Clock. Implementation: Static/Dynamic
	SYS_CLK_PeripheralFrequencyGet	Gets the selected clock peripheral bus frequency in Hertz. Implementation: Static/Dynamic
	SYS_CLK_PeripheralFrequencySet	Configures the peripheral clock of the device to the value specified. Implementation: Dynamic
	SYS_CLK_ReferenceClockSetup	Configures the reference clock of the device to the value specified. Implementation: Dynamic
	SYS_CLK_ReferenceFrequencyGet	Gets the selected Reference clock bus frequency in Hertz. Implementation: Static/Dynamic
	SYS_CLK_ReferenceFrequencySet	Configures the reference clock of the device to the value specified. Implementation: Dynamic
	SYS_CLK_SecondaryOscillatorDisable	Disables the secondary oscillator. Implementation: Static/Dynamic
	SYS_CLK_SecondaryOscillatorEnable	Enables the secondary oscillator. Implementation: Static/Dynamic
	SYS_CLK_SecondaryOscillatorIsEnabled	Identifies whether secondary oscillator is enabled or disabled. Implementation: Static/Dynamic
	SYS_CLK_SystemFrequencyGet	Gets the system clock frequency in Hertz. Implementation: Static/Dynamic
	SYS_CLK_SystemFrequencySet	Configures the System clock of the device to the value specified. Implementation: Dynamic
	SYS_CLK_TaskError	Informs the user on a clock failure by invoking the registered call back function. Implementation: Dynamic

Structures

	Name	Description
	SYS_CLK_INIT	Defines the data required to initialize the Oscillator for the Clock System Service.
	SYS_CLK_REFERENCE_SETUP	Defines the data required to initialize the Oscillator for the Clock System Service.

Types

	Name	Description
	SYS_CLK_ERROR_HANDLER	Pointer to a CLK System service SYS_CLK_ClockFailureCallbackRegister function.
	SYS_CLK_FRC_TUNING_TYPE	Defines the data type for tuning the FRC Oscillator.

Description

Clock System Service Interface Definition

This file contains the interface definition for the Clock System Service. It provides a way to interact with the Clock subsystem to manage the timing requests supported by the system.

File Name

sys_clk.h

Company

Microchip Technology Inc.

sys_clk_config_template.h

Clock System Service configuration templates.

Macros

	Name	Description
	SYS_CLK_CONFIG_EXTERNAL_CLOCK	Gives the clock that is provided on the Reference clock output pin.
	SYS_CLK_CONFIG_FREQ_ERROR_LIMIT	Sets the allowed frequency configuration error Hertz.
	SYS_CLK_CONFIG_PRIMARY_XTAL	Gives the Primary oscillator input frequency. This is from the hardware and not the output required frequency.

	SYS_CLK_CONFIG_SECONDARY_XTAL	Gives the Secondary oscillator input frequency. This is from the hardware and not the output required frequency.
	SYS_CLK_CONFIG_SYSPLL_INP_DIVISOR	Gives the System PLL input divisor configured through configuration bits. For some of the devices, the system PLL input divisor is not accessible through software. So use this macro to inform the System service about the input divisor.
	SYS_CLK_CONFIGBIT_USBPLL_DIVISOR	Gives the Oscillator USB clock divisor.
	SYS_CLK_CONFIGBIT_USBPLL_ENABLE	Defines the status of the USB PLL.
	SYS_CLK_UPLL_BEFORE_DIV2_FREQ	Defines the value of the USB clock for the devices which does not have a software controllable USB clock.

Description

Clock System Service Configuration Templates

This file contains constants to configure the Clock System Service.

File Name

sys_clk_config_template.h

Company

Microchip Technology Inc.

Command Processor System Service Library

This section describes the Command Processor System Service Library.

Introduction

This library provides an abstraction of the Command Processor System Service Library that is available on the Microchip family of PIC32 microcontrollers with a convenient C language interface. It provides the framework for a command console that can support commands from multiple client sources.

Description

The Command Processor System Service provides the developer with simple APIs to implement a command console. The console may support commands from one or more client software modules.

Using the Library

This topic describes the basic architecture of the Command Processor System Service Library and provides information and examples on its use.

Description

Interface Header File: [sys_command.h](#)

The interface to the Command Processor System Service library is defined in the [sys_command.h](#) header file.

Please refer to the What is MPLAB Harmony? section for how the Command Processor System Service interacts with the framework.

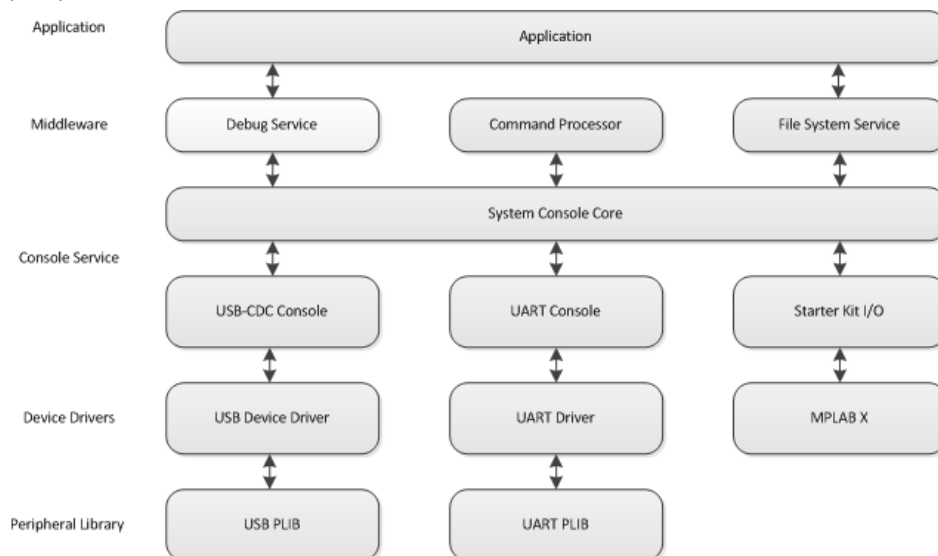
Abstraction Model

This topic provides a description of the software abstraction for the Command Processor System Service.

Description

The Command Processor System Service Library is a collection of operations specific to supporting user input commands. The commands can be uniquely native to each client service and can be dynamically added. The library can support multiple client services at once.

The Command Processor System Service is a module that works closely with the Console System Service to present a user interface command prompt.



Library Overview

Refer to the section [System Services Introduction](#) for how the system services operates in a system.

The library interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the Device Control module.

Library Interface Section	Description
System Interaction Functions	Provides system module APIs. Service initialization, deinitialization, reinitialization and status functions.
Core Functionality Functions	State Machine and control functions.
Data Types and Constants	Enumerations, structures, and constants.

How the Library Works

System Interaction

Initialization and Reinitialization

Initialization of the Command Processor System Service initializes the status of the module and sets the state of the internal state machine. The Command Processor Status and Tasks routines are required for the normal operation of the service.

Core Functionality

Adding Commands

The Command Processor System Service will accept commands dynamically during run-time through the follow interface:

```
SYS_COMMAND_ADDGRP(const _SYS_CMD_DCPT* pCmdTbl, int nCmds, const char*  
                    groupName, const char* menuStr)
```

Command Prompt

In conjunction with the Console System Service, the Command Processor System Service provides the user with an ASCII command prompt. It will interpret the commands entered at the prompt and process accordingly. It also supports command history, as well as command help.

Configuring the Library

The configuration of the Command Processor System Service is based on the file `system_config.h`.

This header file contains the configuration selection for the Device Control system service. Based on the selections made, the Device Control System Service Library may support the selected features. These configuration settings will apply to all instances of the Device Control System Service.

This header can be placed anywhere; however, the path of this header needs to be present in the include search path for a successful build. Refer to the Applications Help section for more details.

Building the Library

This section lists the files that are available in the Command Processor System Service Library.

Description

The following three tables list and describe the header (.h) and source (.c) files that implement this library. The parent folder for these files is `<install-dir>/framework/system/command`.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using `#include`) by any code that uses this library.

Source File Name	Description
<code>sys_command.h</code>	Command Processor System Service Library API header file.

Required File(s)



All of the required files listed in the following table are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

Source File Name	Description
/src/sys_command.c	Command Processor System Service Library implementation.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
N/A	No optional files are available for this library.










Module Dependencies

The Command Processor System Service is dependent on the following library:



- [Console System Service Library](#)

Library Interface


a) System Interaction Functions

	Name	Description
	SYS_CMD_Initialize	Initializes data for the instance of the Command Processor module.
	SYS_CMD_DELETE	Removes CMDIO parameters from the Command Processor System Service console.
	SYS_CMD_ADDGRP	Allows clients to add command process to the Command Processor System Service.
	SYS_CMD_MESSAGE	Outputs a message to the Command Processor System Service console.
	SYS_CMD_PRINT	Outputs a printout to the Command Processor System Service console.
	SYS_CMD_READY_TO_READ	Allows upper layer application to confirm that the command module is ready to accept command input
	SYS_CMD_READY_TO_WRITE	Allows upper layer application to confirm that the command module is ready to write output to the Console System Service.
	SYS_CMD_Tasks	Maintains the Command Processor System Service's internal state machine.
	SYS_CMD_RegisterCallback	Registers a callback function with the command service that will be executed when the lower layer read or write queue is empty.

b) Command I/O Functions

	Name	Description
	SYS_CMDIO_GET_HANDLE	Gets the CMDIO Handle base via index.
	SYS_CMDIO_ADD	Adds CMDIO parameters to the Command Processor System Service console.

c) Data Types and Constants

	Name	Description
	_promptStr	prompt string
	COMMAND_HISTORY_DEPTH	Command Processor System Service Maximum Depth of Command History.
	ESC_SEQ_SIZE	standard VT100 escape sequences
	LINE_TERM	line terminator
	MAX_CMD_ARGS	Command Processor System Service Maximum Number of Argument definitions.
	MAX_CMD_GROUP	Command Processor System Service Maximum Number of Command Group definitions.
	SYS_CMD_MAX_LENGTH	Command Processor System Service Command Buffer Maximum Length definition.
	SYS_CMD_READ_BUFFER_SIZE	Command Processor System Service Read Buffer Size definition.
	SYS_CMD_DEVICE_NODE	Defines the data structure to store each command instance.
	SYS_CMD_INIT	Identifies the system command initialize structure.
	SYS_CMD_API	Identifies the Command API structure.
	SYS_CMD_DATA_RDY_FNC	Identifies a data available function API.
	SYS_CMD_DESCRIPTOR	a simple command descriptor
	SYS_CMD_DESCRIPTOR_TABLE	table containing the supported commands
	SYS_CMD_DEVICE_LIST	Defines the list structure to store a list of command instances.
	SYS_CMD_FNC	Identifies the command process function API.
	SYS_CMD_GETC_FNC	Identifies a get single character function API.
	SYS_CMD_HANDLE	Identifies a particular Command I/O instance.

SYS_CMD_INIT_DATA	Defines the data structure to store each command.
SYS_CMD_MSG_FNC	Identifies a message function API.
SYS_CMD_PRINT_FNC	Identifies a print function API.
SYS_CMD_PUTC_FNC	Identifies a single character print function API.
SYS_CMD_READC_FNC	Identifies a read single character function API.
SYS_CMD_STATE	Defines the various states that can be achieved by a Command instance.
SYS_CMD_CONSOLE_IO_PARAM	Defines whether the command module is single character or full command read
SYS_CMD_BUFFER_DMA_READY	Define this for MX #define SYS_CMD_BUFFER_DMA_READY __attribute__((coherent)) __attribute__((aligned(4))) //Define this for MZ
SYS_CONSOLE_MESSAGE	ifdef SYS_CONSOLE_REMOVE_APIS
SYS_CONSOLE_PRINT	This is macro SYS_CONSOLE_PRINT.
SYS_CMD_DEVICE_MAX_INSTANCES	This is macro SYS_CMD_DEVICE_MAX_INSTANCES.
SYS_CMD_MESSAGE	This is macro SYS_CMD_MESSAGE.
SYS_CMD_PRINT	This is macro SYS_CMD_PRINT.
SYS_CMD_CallbackFunction	Command Callback Function Handle.
SYS_CMD_EVENT	Identifies the Command Event Types

Description

This section describes the APIs of the Command Processor System Service Library.
Refer to each section for a detailed description.

a) System Interaction Functions

SYS_CMD_Initialize Function

Initializes data for the instance of the Command Processor module.

File

[sys_command.h](#)

C

```
bool SYS_CMD_Initialize(const SYS_MODULE_INIT * const init);
```

Returns

- true - Indicates success
- false - Indicates an error occurred

Description

This function initializes the Command Processor module. It also initializes any internal data structures.

Remarks

This routine should only be called once during system initialization.

Preconditions

None.

Parameters

Parameters	Description
init	Pointer to a data structure containing any data necessary to initialize the sys command. This pointer may be null if no data is required because static overrides have been provided.

Function

```
bool SYS_CMD_Initialize( const SYS\_MODULE\_INIT * const init )
```

SYS_CMD_DELETE Function

Removes CMDIO parameters from the Command Processor System Service console.

File

[sys_command.h](#)

C

```
bool SYS_CMD_DELETE(SYS_CMD_DEVICE_NODE* pDevNode);
```

Returns

None.

Description

This function removes CMDIO parameters from the Command Processor System Service console.

Remarks

None.

Preconditions

[SYS_CMD_Initialize](#) was successfully run once.

Function

```
bool SYS_CMD_DELETE( SYS\_CMD\_DEVICE\_NODE* pDevNode);
```

SYS_CMD_ADDGRP Function

Allows clients to add command process to the Command Processor System Service.

File

[sys_command.h](#)

C

```
bool SYS_CMD_ADDGRP(const SYS\_CMD\_DESCRIPTOR* pCmdTbl, int nCmds, const char* groupName, const char* menuStr);
```

Returns

- true - Indicates success
- false - Indicates an error occurred

Description

Client modules may call this function to add command process to the Command Processor System Service.

Remarks

None.

Preconditions

[SYS_CMD_Initialize](#) was successfully run once.

Example

```
static const SYS\_CMD\_DESCRIPTOR cmdTbl[] = { {"command_as_typed_at_the_prompt", _Function_Name_That_Supports_The_Command, "Helpful description of the command for the user"}, };  
bool APP_AddCommandFunction() { if (!SYS_CMD_ADDGRP(cmdTbl, sizeof(cmdTbl)/sizeof(*cmdTbl), "Command Group Name", ": Command Group Description")) { return false; } return true; }
```

Function

```
bool SYS_CMD_ADDGRP(const SYS\_CMD\_DESCRIPTOR* pCmdTbl, int nCmds,  
const char* groupName,  
const char* menuStr)
```

SYS_CMD_MESSAGE Function

Outputs a message to the Command Processor System Service console.

File

[sys_command.h](#)

C

```
void SYS_CMD_MESSAGE(const char* message);
```

Returns

None.

Description

This function outputs a message to the Command Processor System Service console.

Remarks

None.

Preconditions

[SYS_CMD_Initialize](#) was successfully run once.

Function

```
void SYS_CMD_MESSAGE (const char* message)
```

SYS_CMD_PRINT Function

Outputs a printout to the Command Processor System Service console.

File

[sys_command.h](#)

C

```
void SYS_CMD_PRINT(const char * format, ...);
```

Returns

None.

Description

This function outputs a printout to the Command Processor System Service console.

Remarks

None.

Preconditions

[SYS_CMD_Initialize](#) was successfully run once.

Function

```
void SYS_CMD_PRINT(const char *format, ...)
```

SYS_CMD_READY_TO_READ Function

Allows upper layer application to confirm that the command module is ready to accept command input

File

[sys_command.h](#)

C

```
bool SYS_CMD_READY_TO_READ( );
```

Returns

- true - Indicates command module is ready
- false - Indicates command module is not ready

Description

This function allows upper layer application to confirm that the command module is ready to accept command input

Remarks

None.

Preconditions

[SYS_CMD_Initialize](#) was successfully run once.

Function

```
bool SYS_CMD_READY_TO_READ( void )
```

SYS_CMD_READY_TO_WRITE Function

Allows upper layer application to confirm that the command module is ready to write output to the Console System Service.

File

[sys_command.h](#)

C

```
bool SYS_CMD_READY_TO_WRITE ( ) ;
```

Returns

- true - Indicates command module is ready
- false - Indicates command module is not ready

Description

This function allows upper layer application to confirm that the command module is ready to write output to the Console System Service.

Remarks

None.

Preconditions

[SYS_CMD_Initialize](#) was successfully run once.

Function

```
bool SYS_CMD_READY_TO_WRITE( void )
```

SYS_CMD_Tasks Function

Maintains the Command Processor System Service's internal state machine.

File

[sys_command.h](#)

C

```
bool SYS_CMD_Tasks ( ) ;
```

Returns

- true - Indicates success
- false - Indicates an error occurred

Description

This function is used to maintain the Command Processor System Service internal state machine.

Remarks

None.

Preconditions

[SYS_CMD_Initialize](#) was successfully run once.

Function

```
bool SYS_CMD_Tasks( void )
```

SYS_CMD_RegisterCallback Function

Registers a callback function with the command service that will be executed when the lower layer read or write queue is empty.

File

[sys_command.h](#)

C

```
void SYS_CMD_RegisterCallback(SYS_CMD_CallbackFunction cbFunc, SYS_CMD_EVENT event);
```

Returns

None.

Description

This function is used by an application to register a callback function with the command service. The callback function is called in response to an event. Separate callback functions are required for each event.

Remarks

None.

Preconditions

None.

Parameters

Parameters	Description
cbFunc	The name of the callback function
event	Enumerated list of events that can trigger a callback

Function

```
void SYS_CMD_RegisterCallback( SYS_CMD_CallbackFunction cbFunc, SYS_CMD_EVENT event)
```

b) Command I/O Functions

SYS_CMDIO_GET_HANDLE Function

Gets the CMDIO Handle base via index.

File

[sys_command.h](#)

C

```
SYS_CMD_DEVICE_NODE* SYS_CMDIO_GET_HANDLE( short num );
```

Returns

- [SYS_CMD_DEVICE_NODE](#) Handle for the CMDIO - Indicates success
- NULL - Indicates not successful

Description

This function returns the handle for the CMDIO when supplied with an index.

Remarks

None.

Preconditions

[SYS_CMD_Initialize](#) was successfully run once.

Function

`SYS_CMD_DEVICE_NODE* SYS_CMDIO_GET_HANDLE(short num)`

SYS_CMDIO_ADD Function

Adds CMDIO parameters to the Command Processor System Service console.

File

`sys_command.h`

C

```
SYS_CMD_DEVICE_NODE* SYS_CMDIO_ADD(const SYS_CMD_API* opApi, const void* cmdIoParam, const
SYS_CMD_CONSOLE_IO_PARAM cmdIoType);
```

Returns

None.

Description

This function adds CMDIO Parameters to the Command Processor System Service console.

Remarks

None.

Preconditions

`SYS_CMD_Initialize` was successfully run once.

Function

```
SYS_CMD_DEVICE_NODE* SYS_CMDIO_ADD(const SYS_CMD_API* opApi, const void* cmdIoParam,
const SYS_CMD_CONSOLE_IO_PARAM cmdIoType)
```

c) Data Types and Constants**_promptStr Macro****File**

`sys_command.h`

C

```
#define _promptStr ">" // prompt string
```

Description

prompt string

COMMAND_HISTORY_DEPTH Macro

Command Processor System Service Maximum Depth of Command History.

File

`sys_command.h`

C

```
#define COMMAND_HISTORY_DEPTH 3
```

Description

SYS CMD Processor Command History Depth

This macro defines the maximum depth of the command history.

Remarks

None.

ESC_SEQ_SIZE Macro

File

[sys_command.h](#)

C

```
#define ESC_SEQ_SIZE 2           // standard VT100 escape sequences
```

Description

standard VT100 escape sequences

LINE_TERM Macro

File

[sys_command.h](#)

C

```
#define LINE_TERM "\r\n"        // line terminator
```

Description

line terminator

MAX_CMD_ARGS Macro

Command Processor System Service Maximum Number of Argument definitions.

File

[sys_command.h](#)

C

```
#define MAX_CMD_ARGS 15
```

Description

SYS CMD Processor Maximum Number of Command Arguments

This macro defines the maximum number of arguments per command.

Remarks

None.

MAX_CMD_GROUP Macro

Command Processor System Service Maximum Number of Command Group definitions.

File

[sys_command.h](#)

C

```
#define MAX_CMD_GROUP 8
```

Description

SYS CMD Processor Maximum Number of Command Group

This macro defines the maximum number of command groups.

Remarks

None.

SYS_CMD_MAX_LENGTH Macro

Command Processor System Service Command Buffer Maximum Length definition.

File

[sys_command.h](#)

C

```
#define SYS_CMD_MAX_LENGTH 128
```

Description

SYS CMD Processor Buffer Maximum Length
This macro defines the maximum length of the command buffer.

Remarks

None.

SYS_CMD_READ_BUFFER_SIZE Macro

Command Processor System Service Read Buffer Size definition.

File

[sys_command.h](#)

C

```
#define SYS_CMD_READ_BUFFER_SIZE 128
```

Description

SYS CMD Processor Read Buffer Size
This macro defines the maximum size of the command buffer.

Remarks

None.

SYS_CMD_DEVICE_NODE Structure

Defines the data structure to store each command instance.

File

[sys_command.h](#)

C

```
struct SYS_CMD_DEVICE_NODE {
    char* cmdPnt;
    char* cmdEnd;
    char cmdBuff[SYS_CMD_BUFFER_DMA_READY[SYS_CMD_MAX_LENGTH+1]];
    const SYS_CMD_API* pCmdApi;
    const void* cmdIoParam;
    SYS_CMD_CONSOLE_IO_PARAM cmdIoType;
    struct SYS_CMD_DEVICE_NODE* next;
    SYS_CMD_STATE cmdState;
};
```

Members

Members	Description
const SYS_CMD_API* pCmdApi;	Cmd IO APIs
const void* cmdIoParam;	channel specific parameter

Description

SYS CMD Command Instance Node Structure
This data structure stores all the data relevant to a uniquely entered command instance. It is a node for a linked list structure to support the Command Processor System Service's command history feature

Remarks

None.

SYS_CMD_INIT Structure

Identifies the system command initialize structure.

File

[sys_command.h](#)

C

```
typedef struct {
    SYS_MODULE_INIT moduleInit;
    uint8_t consoleCmdIoParam;
    SYS_CMD_CONSOLE_IO_PARAM cmdIoType;
} SYS_CMD_INIT;
```

Members

Members	Description
SYS_MODULE_INIT moduleInit;	System module initialization

Description

SYS CMD INIT structure

This structure identifies the system command initialize structure.

Remarks

None.

SYS_CMD_API Structure

Identifies the Command API structure.

File

[sys_command.h](#)

C

```
typedef struct {
    SYS_CMD_MSG_FNC msg;
    SYS_CMD_PRINT_FNC print;
    SYS_CMD_PUTC_FNC putc;
    SYS_CMD_DATA_RDY_FNC isRdy;
    SYS_CMD_GETC_FNC getc;
    SYS_CMD_READC_FNC readc;
} SYS_CMD_API;
```

Members

Members	Description
SYS_CMD_MSG_FNC msg;	Message function API
SYS_CMD_PRINT_FNC print;	Print function API
SYS_CMD_PUTC_FNC putc;	Put single char function API
SYS_CMD_DATA_RDY_FNC isRdy;	Data available API
SYS_CMD_GETC_FNC getc;	Get single data API
SYS_CMD_READC_FNC readc;	Read single data API

Description

SYS CMD API structure

This structure identifies the Command API structure.

Remarks

None.

SYS_CMD_DATA_RDY_FNC Type

Identifies a data available function API.

File

[sys_command.h](#)

C

```
typedef bool (* SYS_CMD_DATA_RDY_FNC)(const void* cmdIoParam);
```

Description

Ready Status Check function API. This handle identifies the interface structure of the data available function API within the Command IO encapsulation.

Remarks

None.

SYS_CMD_DESCRIPTOR Structure

File

[sys_command.h](#)

C

```
typedef struct {
    const char* cmdStr;
    SYS_CMD_FNC cmdFnc;
    const char* cmdDescr;
} SYS_CMD_DESCRIPTOR;
```

Members

Members	Description
const char* cmdStr;	string identifying the command
SYS_CMD_FNC cmdFnc;	function to execute for this command
const char* cmdDescr;	simple command description

Description

a simple command descriptor

SYS_CMD_DESCRIPTOR_TABLE Structure

File

[sys_command.h](#)

C

```
typedef struct {
    int nCmds;
    const SYS_CMD_DESCRIPTOR* pCmd;
    const char* cmdGroupName;
    const char* cmdMenuStr;
} SYS_CMD_DESCRIPTOR_TABLE;
```

Members

Members	Description
int nCmds;	number of commands available in the table
const SYS_CMD_DESCRIPTOR* pCmd;	pointer to an array of command descriptors
const char* cmdGroupName;	name identifying the commands
const char* cmdMenuStr;	help string

Description

table containing the supported commands

SYS_CMD_DEVICE_LIST Structure

Defines the list structure to store a list of command instances.

File

[sys_command.h](#)

C

```
typedef struct {  
    int num;  
    SYS_CMD_DEVICE_NODE* head;  
    SYS_CMD_DEVICE_NODE* tail;  
} SYS_CMD_DEVICE_LIST;
```

Description

SYS CMD Command List Structure

This data structure defines the linked list structure to support the Command Processor System Service's command history feature

Remarks

None.

SYS_CMD_FNC Type

Identifies the command process function API.

File

[sys_command.h](#)

C

```
typedef int (* SYS_CMD_FNC)(SYS_CMD_DEVICE_NODE* pCmdIO, int argc, char **argv);
```

Description

SYS CMD Command Function

Command Process Function API. This handle identifies the interface structure of the command process function API.

Remarks

None.

SYS_CMD_GETC_FNC Type

Identifies a get single character function API.

File

[sys_command.h](#)

C

```
typedef char (* SYS_CMD_GETC_FNC)(const void* cmdIoParam);
```

Description

Get Single Character function API. This handle identifies the interface structure of the get single character function API within the Command I/O encapsulation.

Remarks

None.

SYS_CMD_HANDLE Type

Identifies a particular Command I/O instance.

File

[sys_command.h](#)

C

```
typedef const void* SYS_CMD_HANDLE;
```

Description

Command I/O Handle. This event handle identifies a registered instance of a Command IO object. Every time the application that tries to access the parameters with respect to a particular event, this event handle is used to refer to that event.

Remarks

None.

SYS_CMD_INIT_DATA Structure

Defines the data structure to store each command.

File

[sys_command.h](#)

C

```
typedef struct {
    size_t bytesRead;
    int seqBytesRead;
    char seqBuff[ESC_SEQ_SIZE + 1];
    SYS_MODULE_INDEX moduleIndices[SYS_CMD_DEVICE_MAX_INSTANCES];
    int moduleInFd;
    int moduleOutFd;
} SYS_CMD_INIT_DATA;
```

Description

SYS CMD Command App Init Structure

This data structure stores all the data relevant to a uniquely entered command. It is a node for a linked list structure to support the command history functionality

Remarks

None.

SYS_CMD_MSG_FNC Type

Identifies a message function API.

File

[sys_command.h](#)

C

```
typedef void (* SYS_CMD_MSG_FNC)(const void* cmdIoParam, const char* str);
```

Description

Message function API. This handle identifies the interface structure of the message function API within the Command I/O encapsulation.

Remarks

None.

SYS_CMD_PRINT_FNC Type

Identifies a print function API.

File

[sys_command.h](#)

C

```
typedef void (* SYS_CMD_PRINT_FNC)(const void* cmdIoParam, const char* format, ...);
```

Description

Print function API. This handle identifies the interface structure of the print function API within the Command I/O encapsulation.

Remarks

None.

SYS_CMD_PUTC_FNC Type

Identifies a single character print function API.

File

[sys_command.h](#)

C

```
typedef void (* SYS_CMD_PUTC_FNC)(const void* cmdIoParam, char c);
```

Description

Single Character Print function API. This handle identifies the interface structure of single character print function API within the Command I/O encapsulation.

Remarks

None.

SYS_CMD_READC_FNC Type

Identifies a read single character function API.

File

[sys_command.h](#)

C

```
typedef size_t (* SYS_CMD_READC_FNC)(const void* cmdIoParam);
```

Description

Read Single Character function API

This handle identifies the interface structure of read single character function API within the Command I/O encapsulation.

Remarks

None.

SYS_CMD_STATE Enumeration

Defines the various states that can be achieved by a Command instance.

File

[sys_command.h](#)

C

```
typedef enum {  
    SYS_CMD_STATE_DISABLE,  
    SYS_CMD_STATE_SETUP_READ,  
    SYS_CMD_STATE_WAIT_FOR_READ_DONE,  
    SYS_CMD_STATE_PROCESS_FULL_READ  
} SYS_CMD_STATE;
```

Description

SYS CMD State Machine States

This enumeration defines the various states that can be achieved by the command operation.

Remarks

None.

SYS_CMD_CONSOLE_IO_PARAM Enumeration

Defines whether the command module is single character or full command read

File

[sys_command.h](#)

C

```
typedef enum {  
    SYS_CMD_SINGLE_CHARACTER_READ_CONSOLE_IO_PARAM = 0,  
    SYS_CMD_FULL_COMMAND_READ_CONSOLE_IO_PARAM = 1,  
    SYS_CMD_TELNET_COMMAND_READ_CONSOLE_IO_PARAM = 2  
} SYS_CMD_CONSOLE_IO_PARAM;
```

Description

SYS CMD Console I/O Parameter

This enumeration defines whether the command module is single character or full command read.

Remarks

None.

SYS_CMD_BUFFER_DMA_READY Macro

File

[sys_command.h](#)

C

```
#define SYS_CMD_BUFFER_DMA_READY
```

Description

Define this for MX #define SYS_CMD_BUFFER_DMA_READY __attribute__((coherent)) __attribute__((aligned(4))) //Define this for MZ

SYS_CONSOLE_MESSAGE Macro

File

[sys_console.h](#)

C

```
#define SYS_CONSOLE_MESSAGE(message)
```

Description

```
ifdef SYS_CONSOLE_REMOVE_APIS
```

SYS_CONSOLE_PRINT Macro

File

[sys_console.h](#)

C

```
#define SYS_CONSOLE_PRINT(fmt, ...)
```

Description

This is macro SYS_CONSOLE_PRINT.

SYS_CMD_DEVICE_MAX_INSTANCES Macro

File

[sys_command.h](#)

C

```
#define SYS_CMD_DEVICE_MAX_INSTANCES 1
```

Description

This is macro SYS_CMD_DEVICE_MAX_INSTANCES.

SYS_CMD_MESSAGE Macro

File

[sys_command.h](#)

C

```
#define SYS_CMD_MESSAGE(message)
```

Description

This is macro SYS_CMD_MESSAGE.

SYS_CMD_PRINT Macro

File

[sys_command.h](#)

C

```
#define SYS_CMD_PRINT(fmt, ...)
```

Description

This is macro SYS_CMD_PRINT.

SYS_CMD_CallbackFunction Type

Command Callback Function Handle.

File

[sys_command.h](#)

C

```
typedef void (* SYS_CMD_CallbackFunction)(void *handle);
```

Description

Command Callback Function Handle.

Remarks

None.

SYS_CMD_EVENT Enumeration

Identifies the Command Event Types

File

[sys_command.h](#)

C

```
typedef enum {  
    SYS_CMD_EVENT_WRITE_COMPLETE,  
    SYS_CMD_EVENT_READ_COMPLETE  
} SYS_CMD_EVENT;
```

Description

SYS CMD EVENT structure

This structure identifies the Command Event Types.

Remarks

None.

Files

Files

Name	Description
sys_command.h	Command Processor System Service interface definition.

Description

This section lists the source and header files used by the library.












sys_command.h

Command Processor System Service interface definition.

Enumerations

	Name	Description
	SYS_CMD_CONSOLE_IO_PARAM	Defines whether the command module is single character or full command read
	SYS_CMD_EVENT	Identifies the Command Event Types
	SYS_CMD_STATE	Defines the various states that can be achieved by a Command instance.

Functions


	Name	Description
	SYS_CMD_ADDGRP	Allows clients to add command process to the Command Processor System Service.
	SYS_CMD_DELETE	Removes CMDIO parameters from the Command Processor System Service console.
	SYS_CMD_Initialize	Initializes data for the instance of the Command Processor module.
	SYS_CMD_MESSAGE	Outputs a message to the Command Processor System Service console.
	SYS_CMD_PRINT	Outputs a printout to the Command Processor System Service console.
	SYS_CMD_READY_TO_READ	Allows upper layer application to confirm that the command module is ready to accept command input
	SYS_CMD_READY_TO_WRITE	Allows upper layer application to confirm that the command module is ready to write output to the Console System Service.
	SYS_CMD_RegisterCallback	Registers a callback function with the command service that will be executed when the lower layer read or write queue is empty.
	SYS_CMD_Tasks	Maintains the Command Processor System Service's internal state machine.
	SYS_CMDIO_ADD	Adds CMDIO parameters to the Command Processor System Service console.
	SYS_CMDIO_GET_HANDLE	Gets the CMDIO Handle base via index.

Macros

Name	Description
_promptStr	prompt string
COMMAND_HISTORY_DEPTH	Command Processor System Service Maximum Depth of Command History.
ESC_SEQ_SIZE	standard VT100 escape sequences
LINE_TERM	line terminator
MAX_CMD_ARGS	Command Processor System Service Maximum Number of Argument definitions.
MAX_CMD_GROUP	Command Processor System Service Maximum Number of Command Group definitions.
SYS_CMD_BUFFER_DMA_READY	Define this for MX #define SYS_CMD_BUFFER_DMA_READY __attribute__((coherent)) __attribute__((aligned(4))) //Define this for MZ
SYS_CMD_DEVICE_MAX_INSTANCES	This is macro SYS_CMD_DEVICE_MAX_INSTANCES.
SYS_CMD_MAX_LENGTH	Command Processor System Service Command Buffer Maximum Length definition.
SYS_CMD_MESSAGE	This is macro SYS_CMD_MESSAGE.
SYS_CMD_PRINT	This is macro SYS_CMD_PRINT.
SYS_CMD_READ_BUFFER_SIZE	Command Processor System Service Read Buffer Size definition.
SYS_DEBUG	This is macro SYS_DEBUG.
SYS_ERROR	This is macro SYS_ERROR.

	SYS_ERROR_PRINT	This is macro SYS_ERROR_PRINT.
--	---------------------------------	--------------------------------

Structures

	Name	Description
	SYS_CMD_DEVICE_NODE	Defines the data structure to store each command instance.
	SYS_CMD_API	Identifies the Command API structure.
	SYS_CMD_DESCRIPTOR	a simple command descriptor
	SYS_CMD_DESCRIPTOR_TABLE	table containing the supported commands
	SYS_CMD_DEVICE_LIST	Defines the list structure to store a list of command instances.
	SYS_CMD_INIT	Identifies the system command initialize structure.
	SYS_CMD_INIT_DATA	Defines the data structure to store each command.

Types

	Name	Description
	SYS_CMD_CallbackFunction	Command Callback Function Handle.
	SYS_CMD_DATA_RDY_FNC	Identifies a data available function API.
	SYS_CMD_FNC	Identifies the command process function API.
	SYS_CMD_GETC_FNC	Identifies a get single character function API.
	SYS_CMD_HANDLE	Identifies a particular Command I/O instance.
	SYS_CMD_MSG_FNC	Identifies a message function API.
	SYS_CMD_PRINT_FNC	Identifies a print function API.
	SYS_CMD_PUTC_FNC	Identifies a single character print function API.
	SYS_CMD_READC_FNC	Identifies a read single character function API.

Description

Command Processor System Service Interface Definition

This file contains the interface definition for the Command Processor System Service. It provides a way to interact with the Command Processor subsystem to manage the ASCII command requests from the user supported by the system.

File Name

sys_command.h

Company

Microchip Technology Inc.

Console System Service Library

This section describes the Console System Service Library.

Introduction

The Console System Service routes data or message traffic between a console device and a middleware layer or application. The most common use of the Console Service is to route debug or error messages from a PIC32-based device to a terminal program on a host development system. When fully implemented, the Console Service will be capable of routing data from any supported console device to a variety of middleware layers.

Description

The Console System Service consists of a core layer and a console device layer. The core layer handles module initialization and system functions. It provides a common API for use by applications and middleware libraries to send and receive data to and from one or more console devices. The Console System core layer maintains the status and device descriptor information for each console instance that has been initialized. The maximum number of simultaneous console instances supported by the service is defined by a configuration option. The default is four.

The console device layer contains the functional implementation of the core layer APIs. Each console device may have a different implementation, but the behavior of the API should be uniform across different devices. The console device layer interacts with the peripheral device drivers to send and receive data to and from the hardware peripheral. The details of the data transaction are abstracted by the console device layer and the peripheral device driver.

Both blocking and non-blocking implementations are provided. For a blocking implementation, the API to transfer data to or from the console device will return after the data has been successfully transferred, or an error has occurred. Non-blocking implementations return immediately and the data transfer is controlled by a state machine implemented in the console device layer. The calling program is notified of the data transfer completion by a callback mechanism, or can optionally poll the status of the console device layer to see if it is busy or ready for more data.

The console device layer provides the ability to queue data buffers written to the console. The size of the queue is defined by a configuration option. Read data is queued internally in the console device layer depending on the console device. The size of this queue is also defined by a configuration option. Each console device has configuration options specific to that device. They are explained in their respective sections in this document.

Using the Library

This topic describes the basic architecture of the Console System Service Library and provides information and examples on its use.

Description

Interface Header File: [sys_console.h](#)

The interface to the Console System Service library is defined in the [sys_console.h](#) header file. This file is included by the `sys.h` file. Any C language source (.c) file that uses the Console System Service library should include `sys.h`.

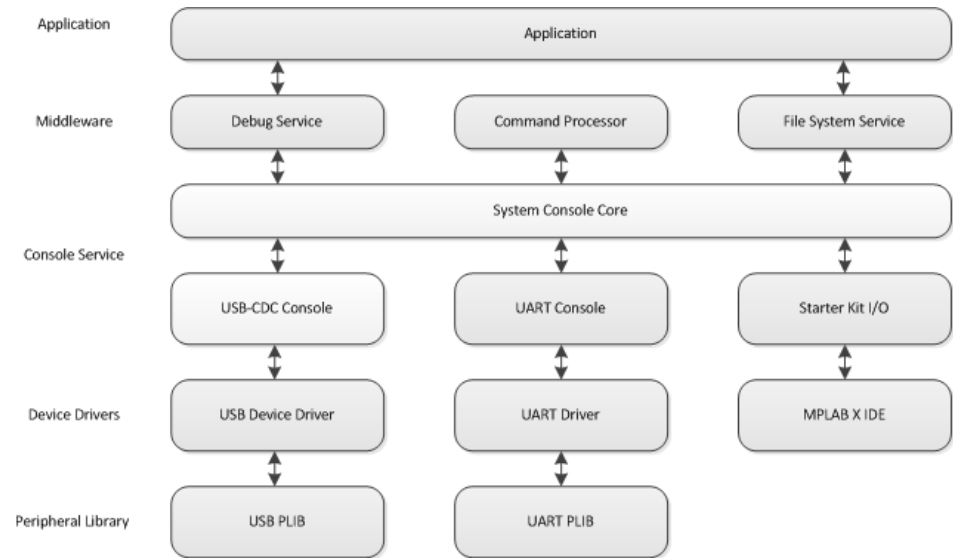
Please refer to the [What is MPLAB Harmony?](#) section for how the System Service interacts with the framework.

Abstraction Model

This library provides a set of functions that send and receive data to and from a console I/O device.

Description

Console Service Software Abstraction Block Diagram



The Console System Service is a middleware library that is part of a layered software architecture. The purpose of this software layering is to allow each module to provide a consistent interface to its client, thus making it easy for the developer to add and remove modules as needed. The console core layer provides a POSIX-like read/write API to applications and middleware libraries to send and receive data to and from the console device. Data and message buffering, along with the details of the data transfer are contained within the console device and device driver layers. Additional APIs are provided to handle system functions, register a callback function for non-blocking implementations, and flush (reset) the internal queues in the event of an error condition.


Library Overview

Refer to the section [System Services Introduction](#) for how the system services operates in a system. The library interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the System Console module.

Library Interface Section	Description
System Functions	Provides system module interfaces, device initialization, tasks and status functions.
Core Functions	Reads or writes data to or from the console I/O device.

How the Library Works

The Console System Service is used to transfer data between an application or middleware layer and a console device.

 **Note:** Not all modes are available on all devices. Please refer to the specific device data sheet to determine the supported modes.

System Functions

This topic describes Console System Service Library system functions.

Description

The Console System Service must be initialized before use. Initialization opens a specific console module instance and initializes the internal data structures for that instance. The initialization routine is provided a pointer to a [SYS_CONSOLE_INIT](#) data structure, which contains a pointer to a device descriptor for a particular console device.

USB CDC Example

```
SYS_CONSOLE_INIT consInit =
{
    .moduleInit = {0},
    .consDevDesc = &consUsbCdcDevDesc,
};
```

UART Example

```
SYS_CONSOLE_INIT consInit =
{
    .moduleInit = {0},
```

```
.consDevDesc = &consUsartDevDesc,  
};
```

The console device descriptor contains information specific to the device associated with that console instance, including pointers to device-specific implementations of the Console System service API. The device descriptor for each console device is declared in the console device implementation. A pointer to the descriptor must be provided to the initialization routine.

Status

The [SYS_CONSOLE_Status](#) function returns the status of the console device module. The internal status of the console device module is updated as the module cycles through its internal state machine. For example, the console device module may need to wait for its corresponding device driver to initialize before moving to an initialized state. Module status is converted from an internal data type to the generic Harmony [SYS_STATUS](#) data type.

Tasks

The [SYS_CONSOLE_Tasks](#) function advances the state machine of the console device module. A pointer to the console device module's `Tasks()` routine is provided as part of the console device descriptor during initialization.

Core Functions

This topic describes core Console System Service functions.

Description

Read

The [SYS_CONSOLE_Read](#) function reads data from a console device into a read buffer. In a blocking implementation, the API will return the number of successfully read bytes when the read operation completes. In a non-blocking implementation, the API will return immediately with number of successfully queued bytes. Completion of the read triggers a callback mechanism in which a user-defined function is executed. A pointer to the read buffer for that transaction is provided as an argument to the callback function to allow console users to distinguish between read events.

For single read events, the user may also poll the status of the console device module.

Write

The [SYS_CONSOLE_Write](#) function writes data from a write buffer or string to a console device. In a blocking implementation, the API will return the number of successfully written bytes when the write operation completes. In a non-blocking implementation, the API will return immediately with the number of successfully queued bytes. Completion of the write triggers a callback mechanism in which a user-defined function is executed. A pointer to the write buffer for that transaction is provided as an argument to the callback function to allow console users to distinguish between write events.

For single write events, the user may also poll the status of the console device module.

Register Callback

Users wishing to take advantage of the callback mechanism must register a callback function with the console service. A separate function must be registered for each event capable of triggering a callback. If no function is registered, the operation will complete normally without triggering any callback function.

Flush

Error conditions may be triggered by a queue overflow or data transfer errors. An error condition may present itself as an unexpected return value from a read or write call, or an error status returned by the [SYS_CONSOLE_Status](#) function. The [SYS_CONSOLE_Flush](#) function is provided to allow the user to terminate any pending transfers and reset the read/write queues internal to the console device.

Console Device Modules

Each supported console device has its own implementation of the Console System service APIs. Function pointers to the console device implementations are provided in the console device descriptor registered to each console instance. This allows the user to utilize the same API for all console devices, making application code more portable across different devices.

Description

Each supported console device has its own implementation of the Console System service APIs. Function pointers to the console device implementations are provided in the console device descriptor registered to each console instance. This allows the user to utilize the same API for all console devices, making application code more portable across different devices.

Application I/O (AppIO)

The AppIO provides a convenient user I/O interface to applications running on a PIC32 using MPLAB X IDE.

Description

Read

The read function accepts user input through the Input Format Field in the PIC AppIO window of MPLAB X IDE. It is the responsibility of the application programmer to properly allocate the read buffer.

Write

The write function sends output to the Output Format Window in MPLAB X IDE, a line at a time. It is the responsibility of the application programmer to properly allocate the write buffer.

USB-CDC Console Device

The USB-CDC Console Device provides a convenient user I/O interface to applications running on a PIC32 with USB support.

Description

In this implementation, the PIC32 acts as a USB device, communicating with a host PC running a terminal emulation program such as Tera Term. This provides a convenient interface for error or debug messages, or a means of communicating with a user interactively. The application does not use the USB-CDC APIs directly, instead all interaction is through the Console System service APIs.

The USB-CDC Console Device supports only non-blocking operation.

Read

While the user may request multiple bytes of data per read, a typical terminal emulator program sends only one byte per USB transaction. When multiple bytes are requested, the USB-CDC Console Device layer will queue individual byte reads until all bytes are read. The size of the read queue is a configuration option. The bytes are sent to the user-supplied read buffer as they are read, and the read callback is triggered when all bytes have been read. Only one read operation can be active at a time.

If the user requests more data bytes than the read queue will hold, the read function will queue and return the size of the read queue. In this scenario, the user can either do multiple reads until all the data is read, or configure the service to use a larger read queue.

The read function accepts a pointer to a read buffer as one of the arguments. It is the responsibility of the application programmer to properly allocate the read buffer.

Write

The USB-CDC console device is capable of writing multiple data bytes per USB transaction, and queuing multiple write transactions. The size of the write queue is a configuration option. If the user tries to queue more write transactions than the write queue can accommodate, the write function will return -1 and set the module status to `SYS_STATUS_ERROR`.

The write function accepts a pointer to a write buffer as one of the arguments. It is the responsibility of the application programmer to properly allocate the write buffer.

State Machine

The USB-CDC Console Device layer implements a state machine that is advanced by each call to the tasks routine. The Console System tasks routine is called periodically by the [SYS_Tasks](#) routine, which is called in a loop by `main()`. The state machine manages initialization of the Console Device, schedules reads and writes, and dispatches callback functions.

Register Callback

The callback registration mechanism associates a user function with a read or write completion event. The callback function includes an argument containing a pointer to the read or write buffer associated with the transfer that initiated the callback. Separate callback functions must be registered for read and write events. If there is no callback function registered, the event will complete as normal, with no callback executed.

Flush

The flush routine in the USB-CDC implementation clears any error conditions, terminates any read or write transactions, and resets the read and write queues. A flush operation is typically triggered in response to an error condition.

UART Console Device

The UART Console Device provides a convenient user I/O interface to applications running on a PIC32 with UART support.

Description

Read

While the user may request multiple bytes of data per read, a typical terminal emulator program sends only one byte per transaction. When multiple bytes are requested, the UART Console Device layer will queue individual byte reads until all bytes are read. The size of the read queue is a configuration option. The bytes are sent to the user-supplied read buffer as they are read, and the read callback is triggered when all bytes have been read. Only one read operation can be active at a time.

If the user requests more data bytes than the read queue will hold, the read function will queue and return the size of the read queue. In this scenario, the user can either do multiple reads until all the data is read, or configure the service to use a larger read queue.

The read function accepts a pointer to a read buffer as one of the arguments. It is the responsibility of the application programmer to properly allocate the read buffer.

Write

The UART console device is capable of writing multiple data bytes per transaction, and queuing multiple write transactions. The size of the write queue is a configuration option. If the user tries to queue more write transactions than the write queue can accommodate, the write function will return -1 and set the module status to SYS_STATUS_ERROR.

The write function accepts a pointer to a write buffer as one of the arguments. It is the responsibility of the application programmer to properly allocate the write buffer.

State Machine

The UART Console Device layer implements a state machine that is advanced by each call to the tasks routine. The Console System tasks routine is called periodically by the [SYS_Tasks](#) routine, which is called in a loop by main(). The state machine manages initialization of the Console Device, schedules reads and writes, and dispatches callback functions.

Register Callback

The callback registration mechanism associates a user function with a read or write completion event. The callback function includes an argument containing a pointer to the read or write buffer associated with the transfer that initiated the callback. Separate callback functions must be registered for read and write events. If there is no callback function registered, the event will complete as normal, with no callback executed.

Flush

The flush routine in the UART implementation clears any error conditions, terminates any read or write transactions, and resets the read and write queues. A flush operation is typically triggered in response to an error condition.

Configuring the Library

The configuration of the Console System Service is based on the file `system_config.h`.

This header file contains the configuration selection for the Console System Service. Based on the selections made, the Console System Service may support the selected features. These configuration settings will apply to all instances of the Console System Service.

This header can be placed anywhere; however, the path of this header needs to be present in the include search path for a successful build. Refer to the Applications Help section for more details.

The Console System Service supports these configuration options:

```
/* Maximum number of simultaneous console instances */
#define CONSOLE_DEVICE_MAX_INSTANCES 1
```

Additional configuration options may be required for the selected Console Device.

Command AppIO Console Device Configuration Options

This topic provided configuration option examples for the Command AppIO Console Device.

Description

```
/* maximum number of console instances */
#define SYS_CONSOLE_DEVICE_MAX_INSTANCES 1

/* number of messages that may be queued up for read */
#define SYS_CONSOLE_APPIO_RD_QUEUE_DEPTH 8

/* number of messages that may be queued up for write */
#define SYS_CONSOLE_APPIO_WR_QUEUE_DEPTH 128
```

USB-CDC Console Device Configuration Options

This topic provided configuration option examples for the USB-CDC Console Device.

Description

USB-CDC Queues

```
/* Number of characters that may be queued for a single read */
#define CONSOLE_USB_CDC_RD_QUEUE_DEPTH 64

/* Number of messages that may be queued for write */
#define CONSOLE_USB_CDC_WR_QUEUE_DEPTH 64
```

USB Controller Driver

```
/* Enables Device Support */
#define DRV_USB_DEVICE_SUPPORT true

/* Disables host support */
#define DRV_USB_HOST_SUPPORT false
```

```

/* Provides 3 endpoints*/
#define DRV_USB_ENDPOINTS_NUMBER      3

/* Only one instance of the USB Peripheral*/
#define DRV_USB_INSTANCES_NUMBER      1

/* Enables interrupt mode */
#define DRV_USB_INTERRUPT_MODE        true

```

USB Device Layer

```

/* Maximum device layer instances */
#define USB_DEVICE_MAX_INSTANCES      1

/* Maximum clients for Device Layer */
#define USB_DEVICE_MAX_CLIENTS        1

/* Maximum function drivers allowed per instance of the USB device layer */
#define USB_DEVICE_FUNCTION_DRIVERS_NUMBER  1

/* EP0 size in bytes */
#define USB_DEVICE_EP0_BUFFER_SIZE     64

```

USB-CDC Function Driver

```

/* Maximum instances of CDC function driver */
#define USB_DEVICE_CDC_INSTANCES_NUMBER  1

/* CDC Transfer Queue Size for both read and
 * write. Applicable to all instances of the
 * function driver */
#define USB_DEVICE_CDC_WRITE_QUEUE_DEPTH  4
#define USB_DEVICE_CDC_READ_QUEUE_DEPTH   1
#define USB_DEVICE_CDC_SSN_QUEUE_DEPTH    1
#define USB_DEVICE_CDC_QUEUE_DEPTH_COMBINED
      (USB_DEVICE_CDC_WRITE_QUEUE_DEPTH +
       USB_DEVICE_CDC_READ_QUEUE_DEPTH +
       USB_DEVICE_CDC_SSN_QUEUE_DEPTH)

```

UART Console Device Configuration Options

This topic provided configuration option examples for the UART Console Device.

Description

```

/* Maximum number of simultaneous console instances */
#define CONSOLE_DEVICE_MAX_INSTANCES    1

/* Number of characters that may be queued for a single read */
#define CONSOLE_UART_RD_QUEUE_DEPTH     1

/* Number of messages that may be queued for write */
#define CONSOLE_UART_WR_QUEUE_DEPTH     64

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

#define DRV_USART_INTERRUPT_MODE         true

#define DRV_USART_INSTANCES_NUMBER      1

#define DRV_USART_CLIENTS_NUMBER        1

#define DRV_USART_BUFFER_QUEUE_SUPPORT   true

#define DRV_USART_BYTE_MODEL_SUPPORT     false

```

```
#define DRV_USART_READ_WRITE_MODEL_SUPPORT    false

#define DRV_USART_WRITE_QUEUE_DEPTH          CONSOLE_UART_RD_QUEUE_DEPTH

#define DRV_USART_READ_QUEUE_DEPTH           CONSOLE_UART_WR_QUEUE_DEPTH

#define DRV_USART_SSN_QUEUE_DEPTH            1

#define DRV_USART_QUEUE_DEPTH_COMBINED       (DRV_USART_WRITE_QUEUE_DEPTH +
                                              DRV_USART_READ_QUEUE_DEPTH +
                                              DRV_USART_SSN_QUEUE_DEPTH)
```

Building the Library

This section lists the files that are available in the Console System Service Library.

Description

The following three tables list and describe the header (.h) and source (.c) files that implement this library. The parent folder for these files is <install-dir>/framework/system/console.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using #include) by any code that uses this library.

Source File Name	Description
sys_console.h	Console System Service Library API header file.

Required File(s)



All of the required files listed in the following table are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

Source File Name	Description
/src/sys_console.c	Console System Service Library implementation.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
/src/sys_console_usb_cdc.c	Required for USB-CDC console device.
/src/sys_console_uart.c	Required for UART console device.
/src/sys_console_local.h	Data structures and definitions used by the Console System Service, but not exposed to clients.
/src/sys_console_usb_cdc_local.h	Data structures and definitions used by the USB-CDC console device, but not exposed to clients.






Module Dependencies

The Console System Service is dependent upon the following modules:





- [Device Control System Service Library](#)
- USART Driver Library (if UART)
- USB Driver Library (if USB-CDC)

Library Interface

a) System Functions

	Name	Description
	SYS_CONSOLE_Initialize	Initializes data for the instance of the Console module and opens the specific module instance. Implementation: Static/Dynamic
	SYS_CONSOLE_Reinitialize	Reinitializes and refreshes the data structure for the instance of the Console module. Implementation: Static/Dynamic
	SYS_CONSOLE_Deinitialize	Deinitializes the specific module instance of the Console module. Implementation: Static/Dynamic
	SYS_CONSOLE_Tasks	Maintains the console's state machine. Implementation: Dynamic
	SYS_CONSOLE_Status	Returns status of the specific instance of the Console module. Implementation: Dynamic

b) Core Functions

	Name	Description
	SYS_CONSOLE_Read	Reads data from the console device. Implementation: Static/Dynamic
	SYS_CONSOLE_Write	Writes data to the console device. Implementation: Static/Dynamic
	SYS_CONSOLE_RegisterCallback	Registers a callback function with the console service that will be executed when the read or write queue is emptied. Implementation: Dynamic
	SYS_CONSOLE_Flush	Flushes the read and write queues and resets an overflow error for the console. Implementation: Static/Dynamic

c) Data Types and Constants

	Name	Description
	SYS_CONSOLE_OBJECT_INSTANCE	System Console object instance structure.
	SYS_CONSOLE_DEVICE	Lists the available console devices.
	SYS_CONSOLE_DEV_DESC	This is type SYS_CONSOLE_DEV_DESC.
	SYS_CONSOLE_INIT	Identifies the system console initialize structure.
	STDERR_FILENO	This is macro STDERR_FILENO.
	STDIN_FILENO	These are in unistd.h
	STDOUT_FILENO	This is macro STDOUT_FILENO.
	SYS_CONSOLE_INDEX_0	Console System Service index definitions.
	SYS_CONSOLE_INDEX_1	This is macro SYS_CONSOLE_INDEX_1.
	SYS_CONSOLE_INDEX_2	This is macro SYS_CONSOLE_INDEX_2.
	SYS_CONSOLE_INDEX_3	This is macro SYS_CONSOLE_INDEX_3.
	CONSOLE_DEVICE_HANDLE_INVALID	This is macro CONSOLE_DEVICE_HANDLE_INVALID.
	SYS_CONSOLE_BUFFER_DMA_READY	This is added here to prevent built errors, just in-case MHC was never run

Description

This section describes the APIs of the Console System Service Library.
Refer to each section for a detailed description.

a) System Functions

SYS_CONSOLE_Initialize Function

Initializes data for the instance of the Console module and opens the specific module instance.
Implementation: Static/Dynamic

File

[sys_console.h](#)

C

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

Returns

If successful, returns a valid handle to an object. Otherwise, it returns [SYS_MODULE_OBJ_INVALID](#). The returned object must be passed as argument to [SYS_CONSOLE_Reinitialize](#), [SYS_CONSOLE_Deinitialize](#), [SYS_CONSOLE_Tasks](#) and [SYS_CONSOLE_Status](#) routines.

Description

This function initializes the Console module, and selects the I/O device to be used. It also initializes any internal data structures.

Remarks

This routine should only be called once during system initialization unless [SYS_Console_Deinitialize](#) is first called to deinitialize the device instance before reinitializing it. If the system was already initialized it safely returns without causing any disturbance.

Preconditions

None.

Example

```
SYS_MODULE_OBJ  objectHandle;

// Populate the console initialization structure
SYS_CONSOLE_INIT consInit =
{
    .moduleInit = {0},
    .consDevDesc = &consUsbCdcDevDesc,
};

objectHandle = SYS_Console_Initialize (SYS_CONSOLE_INDEX_0, (SYS_MODULE_INIT*)&consInit);
if (SYS_MODULE_OBJ_INVALID == objectHandle)
{
    // Handle error
}
```

Parameters

Parameters	Description
index	Index for the instance to be initialized
init	Pointer to a data structure containing any data necessary to initialize the Console System service. This pointer may be null if no data is required because static overrides have been provided.

Function

```
SYS_MODULE_OBJ SYS_CONSOLE_Initialize( const SYS_MODULE_INDEX index,
const SYS_MODULE_INIT * const init )
```

SYS_CONSOLE_Reinitialize Function

Reinitializes and refreshes the data structure for the instance of the Console module.

Implementation: Static/Dynamic

File

[sys_console.h](#)

C

```
bool SYS_CONSOLE_Reinitialize(SYS_MODULE_OBJ object, const SYS_MODULE_INIT * const init);
```

Returns

- true - If successful
- false - If unsuccessful

Description

This function reinitializes and refreshes the data structure for the instance of the Console module using the supplied data.

Remarks

This operation uses the same initialization data structure as the [SYS_CONSOLE_Initialize](#) operation. This function can be called multiple times to reinitialize the module.

Preconditions

The [SYS_CONSOLE_Initialize](#) function should have been called before calling this function.

Example

```
SYS_MODULE_OBJ  objectHandle;

// Populate the console initialization structure
SYS_CONSOLE_INIT consInit =
{
    .moduleInit = {0},
    .consDevDesc = &consUsbCdcDevDesc,
};

SYS_CONSOLE_Reinitialize (objectHandle, (SYS_MODULE_INIT*)&consInit);
```

Parameters

Parameters	Description
object	Identifies the SYS_CONSOLE Object returned by the Initialize interface
init	Pointer to the data structure containing any data necessary to initialize the hardware

Function

```
void SYS_CONSOLE_Reinitialize(  SYS_MODULE_OBJ object,
const                          SYS_MODULE_INIT * const init )
```

SYS_CONSOLE_Deinitialize Function

Deinitializes the specific module instance of the Console module.

Implementation: Static/Dynamic

File

[sys_console.h](#)

C

```
bool SYS_CONSOLE_Deinitialize(SYS_MODULE_OBJ object);
```

Returns

- true - If successful
- false - If unsuccessful

Description

This function deinitializes the specific module instance disabling its operation (and any hardware for driver modules). Resets all of the internal data structures and fields for the specified instance to the default settings.

Remarks

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

Preconditions

The [SYS_CONSOLE_Initialize](#) function should have been called before calling this function.

Example

```
SYS_MODULE_OBJ  object;      // Returned from SYS_CONSOLE_Initialize

SYS_CONSOLE_Deinitialize (object);
```

Parameters

Parameters	Description
object	SYS_CONSOLE object handle, returned from SYS_CONSOLE_Initialize

Function

void SYS_CONSOLE_Deinitialize([SYS_MODULE_OBJ](#) object)

SYS_CONSOLE_Tasks Function

Maintains the console's state machine.

Implementation: Dynamic

File

[sys_console.h](#)

C

```
void SYS_CONSOLE_Tasks(SYS_MODULE_OBJ object);
```

Returns

None

Description

This function is used to maintain the Console System Service internal state machine and implement its ISR for interrupt-driven implementations.

Remarks

This function is normally not called directly by an application. It is called by the system's Tasks routine ([SYS_Tasks](#)) or by the appropriate raw ISR.

Preconditions

The [SYS_CONSOLE_Initialize](#) function must have been called for the specified CONSOLE driver instance.

Example

```
SYS_MODULE_OBJ    object;    // Returned from SYS_CONSOLE_Initialize

while (true)
{
    SYS_CONSOLE_Tasks (object);

    // Do other tasks
}
```

Parameters

Parameters	Description
object	SYS_CONSOLE object handle, returned from SYS_CONSOLE_Initialize

Function

void SYS_CONSOLE_Tasks([SYS_MODULE_OBJ](#) object)

SYS_CONSOLE_Status Function

Returns status of the specific instance of the Console module.

Implementation: Dynamic

File

[sys_console.h](#)

C

```
SYS_STATUS SYS_CONSOLE_Status(SYS_MODULE_OBJ object);
```

Returns

- SYS_STATUS_READY - Indicates that the driver is busy with a previous system level operation and cannot start another. Any value greater than SYS_STATUS_READY is also a normal running state in which the driver is ready to accept new operations.

- `SYS_STATUS_BUSY` - Indicates that the driver is busy with a previous system level operation and cannot start another.
- `SYS_STATUS_ERROR` - Indicates that the driver is in an error state. Any value less than `SYS_STATUS_ERROR` is also an error state.
- `SYS_MODULE_DEINITIALIZED` - Indicates that the driver has been deinitialized. This value is less than `SYS_STATUS_ERROR`.

Description

This function returns the status of the specific module instance disabling its operation (and any hardware for driver modules).

Remarks

None.

Preconditions

The `SYS_CONSOLE_Initialize` function should have been called before calling this function.

Example

```
SYS_MODULE_OBJ    object;    // Returned from SYS_CONSOLE_Initialize
SYS_STATUS        consStatus;

consStatus = SYS_CONSOLE_Status (object);
if (SYS_STATUS_ERROR >= consStatus)
{
    // Handle error
}
```

Parameters

Parameters	Description
object	SYS_CONSOLE object handle, returned from <code>SYS_CONSOLE_Initialize</code>

Function

`SYS_STATUS` `SYS_CONSOLE_Status`(`SYS_MODULE_OBJ` object)

b) Core Functions

SYS_CONSOLE_Read Function

Reads data from the console device.

Implementation: Static/Dynamic

File

`sys_console.h`

C

```
ssize_t SYS_CONSOLE_Read(const SYS_MODULE_INDEX index, int fd, void * buf, size_t count);
```

Returns

Number of bytes actually read.

Description

This function reads the data from the console device.

Remarks

None.

Preconditions

None.

Example

```
ssize_t nr;
char myBuffer[MY_BUFFER_SIZE];
nr = SYS_CONSOLE_Read( SYS_CONSOLE_INDEX_0, STDIN_FILENO, myBuffer, MY_BUFFER_SIZE );
if (nr != MY_BUFFER_SIZE)
{
    // handle error
}
```

```
}
```

Parameters

Parameters	Description
index	Console instance index
fd	I/O stream handle. Maintained for backward compatibility. NULL value can be passed as a parameter.
buf	Buffer to hold the read data.
count	Number of bytes to read.

Function

```
int SYS_CONSOLE_Read( int handle, char *buffer, int len )
```

SYS_CONSOLE_Write Function

Writes data to the console device.

Implementation: Static/Dynamic

File

[sys_console.h](#)

C

```
ssize_t SYS_CONSOLE_Write(const SYS_MODULE_INDEX index, int fd, const char * buf, size_t count);
```

Returns

Number of bytes written or -1 if an error occurred.

Description

This function writes data to the console device.

Remarks

None.

Preconditions

None.

Example

```
ssize_t nr;
char myBuffer[] = "message";
nr = SYS_CONSOLE_Write( SYS_CONSOLE_INDEX_0, STDOUT_FILENO, myBuffer, strlen(myBuffer) );
if (nr != strlen(myBuffer))
{
    // Handle error
}
```

Parameters

Parameters	Description
index	Console instance index
fd	I/O stream handle. Maintained for backward compatibility. NULL value can be passed as a parameter.
buf	Buffer holding the data to be written.
count	Number of bytes to write.

Function

```
ssize_t SYS_CONSOLE_Write(const SYS_MODULE_INDEX index, int fd, const char *buf, size_t count )
```

SYS_CONSOLE_RegisterCallback Function

Registers a callback function with the console service that will be executed when the read or write queue is emptied.

Implementation: Dynamic

File

[sys_console.h](#)

C

```
void SYS_CONSOLE_RegisterCallback(const SYS_MODULE_INDEX index, consoleCallbackFunction cbFunc,
SYS_CONSOLE_EVENT event);
```

Returns

None.

Description

This function is used by an application to register a callback function with the console service. The callback function is called in response to an event. Separate callback functions are required for each event.

Remarks

None.

Preconditions

None.

Example

```
SYS_CONSOLE_RegisterCallback(SYS_CONSOLE_INDEX_0, APP_ReadComplete,
SYS_CONSOLE_EVENT_READ_COMPLETE);
SYS_CONSOLE_RegisterCallback(SYS_CONSOLE_INDEX_0, APP_WriteComplete,
SYS_CONSOLE_EVENT_WRITE_COMPLETE);
```

Parameters

Parameters	Description
index	Console instance index
consCallbackFunc	The name of the callback function
event	Enumerated list of events that can trigger a callback

Function

```
void SYS_CONSOLE_RegisterCallback(const SYS_MODULE_INDEX index,
consoleCallbackFunction cbFunc, SYS_CONSOLE_EVENT event)
```

SYS_CONSOLE_Flush Function

Flushes the read and write queues and resets an overflow error for the console.

Implementation: Static/Dynamic

File

[sys_console.h](#)

C

```
void SYS_CONSOLE_Flush(const SYS_MODULE_INDEX index);
```

Returns

None.

Description

This function flushes the read and write queues and resets an overflow error for the console.

Remarks

None.

Preconditions

None.

Example

```
SYS_CONSOLE_Flush(SYS_CONSOLE_INDEX_0);
```

Parameters

Parameters	Description
index	Console instance index

Function

void SYS_CONSOLE_Flush(const [SYS_MODULE_INDEX](#) index)

c) Data Types and Constants

SYS_CONSOLE_OBJECT_INSTANCE Structure

System Console object instance structure.

File

[sys_console.h](#)

C

```
typedef struct {  
    SYS_STATUS consoleDeviceInstanceStatus;  
    SYS_CONSOLE_DEV_DESC * consoleInstanceDevDesc;  
    CONSOLE_DEVICE_HANDLE consoleDevHandle;  
} SYS_CONSOLE_OBJECT_INSTANCE;
```

Members

Members	Description
SYS_STATUS consoleDeviceInstanceStatus;	State of this instance

Description

SYS_CONSOLE_OBJECT_INSTANCE structure

This data type defines the System Console object instance.

Remarks

None.

SYS_CONSOLE_DEVICE Enumeration

Lists the available console devices.

File

[sys_console.h](#)

C

```
typedef enum {  
    SYS_CONSOLE_DEV_USART,  
    SYS_CONSOLE_DEV_USB_CDC,  
    SYS_CONSOLE_DEV_APPIO,  
    SYS_CONSOLE_NR_DEVS  
} SYS_CONSOLE_DEVICE;
```

Description

Console device enumeration

This enumeration lists all of the available console devices. A console device is a physical peripheral used by the console service to send and receive data.

Remarks

None.

SYS_CONSOLE_DEV_DESC Structure

File

[sys_console.h](#)

C

```
typedef struct {
    SYS_CONSOLE_DEVICE consoleDevice;
    DRV_IO_INTENT intent;
    char (* sysConsoleReadC)(int fd);
    ssize_t (* sysConsoleRead)(int fd, void *buf, size_t count);
    ssize_t (* sysConsoleWrite)(int fd, const void *buf, size_t count);
    void (* sysConsoleRegisterCallback)(consoleCallbackFunction cbFunc, SYS_CONSOLE_EVENT event);
    void (* sysConsoleTasks)(SYS_MODULE_OBJ object);
    SYS_CONSOLE_STATUS (* sysConsoleStatus)(void);
    void (* sysConsoleFlush)(void);
} SYS_CONSOLE_DEV_DESC;
```

Description

This is type SYS_CONSOLE_DEV_DESC.

SYS_CONSOLE_INIT Structure

Identifies the system console initialize structure.

File

[sys_console.h](#)

C

```
typedef struct {
    SYS_MODULE_INIT moduleInit;
    SYS_CONSOLE_DEV_DESC * consDevDesc;
} SYS_CONSOLE_INIT;
```

Members

Members	Description
SYS_MODULE_INIT moduleInit;	System module initialization

Description

SYS Console Initialize structure

This structure identifies the system console initialize structure.

Remarks

None.

STDERR_FILENO Macro

File

[sys_console.h](#)

C

```
#define STDERR_FILENO 2
```

Description

This is macro STDERR_FILENO.

STDIN_FILENO Macro

File

[sys_console.h](#)

C

```
#define STDIN_FILENO 0
```

Description

These are in unistd.h

STDOUT_FILENO Macro**File**

[sys_console.h](#)

C

```
#define STDOUT_FILENO 1
```

Description

This is macro STDOUT_FILENO.

SYS_CONSOLE_INDEX_0 Macro

Console System Service index definitions.

File

[sys_console.h](#)

C

```
#define SYS_CONSOLE_INDEX_0 0
```

Description

SYS Console Module Index Numbers

These constants provide Console System Service index definitions.

Remarks

These constants should be used in place of hard-coded numeric literals.

SYS_CONSOLE_INDEX_1 Macro**File**

[sys_console.h](#)

C

```
#define SYS_CONSOLE_INDEX_1 1
```

Description

This is macro SYS_CONSOLE_INDEX_1.

SYS_CONSOLE_INDEX_2 Macro**File**

[sys_console.h](#)

C

```
#define SYS_CONSOLE_INDEX_2 2
```

Description

This is macro SYS_CONSOLE_INDEX_2.

SYS_CONSOLE_INDEX_3 Macro

File

[sys_console.h](#)

C

```
#define SYS_CONSOLE_INDEX_3 3
```

Description

This is macro SYS_CONSOLE_INDEX_3.

CONSOLE_DEVICE_HANDLE_INVALID Macro

File

[sys_console.h](#)

C

```
#define CONSOLE_DEVICE_HANDLE_INVALID ((CONSOLE_DEVICE_HANDLE)(-1))
```

Description

This is macro CONSOLE_DEVICE_HANDLE_INVALID.

SYS_CONSOLE_BUFFER_DMA_READY Macro

File

[sys_console.h](#)

C

```
#define SYS_CONSOLE_BUFFER_DMA_READY
```

Description

This is added here to prevent built errors, just in-case MHC was never run

Files

Files

Name	Description
sys_console.h	Console System Service interface definitions.

Description

This section lists the source and header files used by the library.



sys_console.h








Console System Service interface definitions.

Enumerations

	Name	Description
	SYS_CONSOLE_DEVICE	Lists the available console devices.

Functions

	Name	Description
	SYS_CONSOLE_Deinitialize	Deinitializes the specific module instance of the Console module. Implementation: Static/Dynamic
	SYS_CONSOLE_Flush	Flushes the read and write queues and resets an overflow error for the console. Implementation: Static/Dynamic

	SYS_CONSOLE_Initialize	Initializes data for the instance of the Console module and opens the specific module instance. Implementation: Static/Dynamic
	SYS_CONSOLE_Read	Reads data from the console device. Implementation: Static/Dynamic
	SYS_CONSOLE_RegisterCallback	Registers a callback function with the console service that will be executed when the read or write queue is emptied. Implementation: Dynamic
	SYS_CONSOLE_Reinitialize	Reinitializes and refreshes the data structure for the instance of the Console module. Implementation: Static/Dynamic
	SYS_CONSOLE_Status	Returns status of the specific instance of the Console module. Implementation: Dynamic
	SYS_CONSOLE_Tasks	Maintains the console's state machine. Implementation: Dynamic
	SYS_CONSOLE_Write	Writes data to the console device. Implementation: Static/Dynamic

Macros

	Name	Description
	CONSOLE_DEVICE_HANDLE_INVALID	This is macro CONSOLE_DEVICE_HANDLE_INVALID.
	STDERR_FILENO	This is macro STDERR_FILENO.
	STDIN_FILENO	These are in unistd.h
	STDOUT_FILENO	This is macro STDOUT_FILENO.
	SYS_CONSOLE_BUFFER_DMA_READY	This is added here to prevent built errors, just in-case MHC was never run
	SYS_CONSOLE_INDEX_0	Console System Service index definitions.
	SYS_CONSOLE_INDEX_1	This is macro SYS_CONSOLE_INDEX_1.
	SYS_CONSOLE_INDEX_2	This is macro SYS_CONSOLE_INDEX_2.
	SYS_CONSOLE_INDEX_3	This is macro SYS_CONSOLE_INDEX_3.
	SYS_CONSOLE_MESSAGE	ifdef SYS_CONSOLE_REMOVE_APIS
	SYS_CONSOLE_PRINT	This is macro SYS_CONSOLE_PRINT.

Structures

	Name	Description
	SYS_CONSOLE_DEV_DESC	This is type SYS_CONSOLE_DEV_DESC.
	SYS_CONSOLE_INIT	Identifies the system console initialize structure.
	SYS_CONSOLE_OBJECT_INSTANCE	System Console object instance structure.

Description

Console System Service Interface Definition

This file contains the interface definition for the Console system service. It provides a way to interact with the Console subsystem to manage the timing requests supported by the system.

File Name

sys_console.h

Company

Microchip Technology Inc.

Debug System Service Library

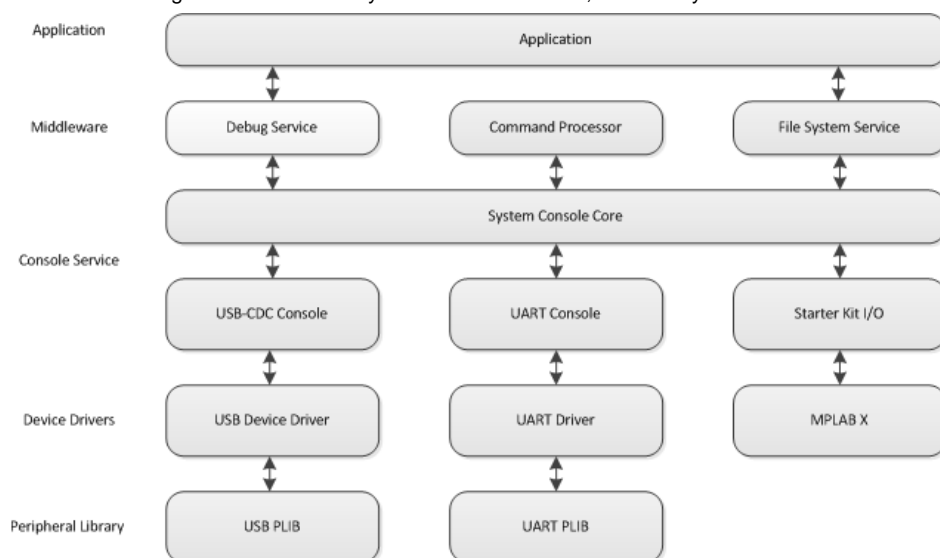
This section describes the Debug System Service Library.

Introduction

This library implements the Debug System Service. The Debug System Service provides a convenient mechanism for the application developer to send formatted or unformatted messages to a system console. The console may direct the messages to a terminal program, file, or any other supported console device. The Debug System Service maintains a global error level, which may be set during initialization and changed dynamically at run-time. Both formatted and unformatted messages can be output or not, depending on the current global error level.

Description

The Debug System Service provides a set of functions that allow the developer to output debug and/or error messages based on a global error level. The messages are sent to the System Console Service, where they are routed to a console device.



Using the Library

This topic describes the basic architecture of the Debug System Service Library and provides information and examples on its use.

Description

Interface Header File: [sys_debug.h](#)

The interface to the Debug System Service library is defined in the [sys_debug.h](#) header file. This file is included by the `sys.h` file. Any C language source (.c) file that uses the Debug System Service Library should include `sys.h`.

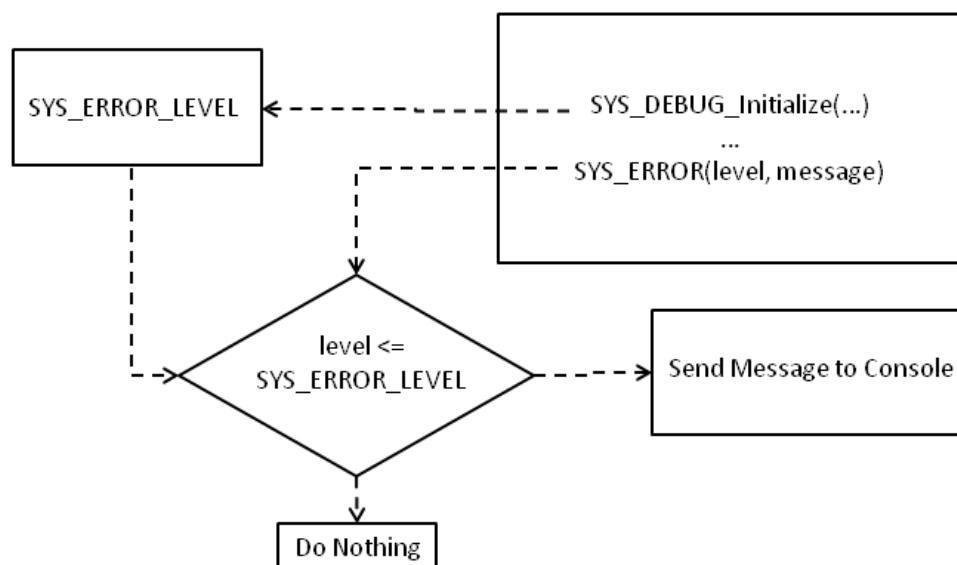
Please refer to the [What is MPLAB Harmony?](#) section for how the Debug System Service interacts with the framework.

Abstraction Model

This library provides a set of functions that allow the developer to quickly and easily provide debug and error messaging during and after program development.

Description

Debug Service Software Abstraction Block Diagram



When the Debug System Service is initialized, it sets the global system error level to the specified level. This level determines the threshold at which debug and error messages are sent to the console. This allows different debug and error reporting verbosity depending on the needs of the developer. The Debug System Service also provides APIs to dynamically set the error level during program execution. This allows the developer to increase or decrease debug verbosity to specific areas of interest within the program.

Library Overview

Refer to the section [System Services Introduction](#) for how the system services operates in a system.

The library interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the Debug System Service module.

Library Interface Section	Description
System Functions	Initializes the Debug System Service and sets the global system error level.
Changing System Error Level Functions	Reads or sets the global system debug level.

How the Library Works

The Debug System Service library can be used by a device driver, middleware layer, or application to report error conditions and output debug messages to a console device during program operation.

Debug System Initialization

The Debug System Service library must be initialized by calling the [SYS_DEBUG_Initialize](#) function. Initialization opens a specific debug module instance and initializes the internal data structures for that instance.

Example: Debug System Service Initialization

```

SYS_DEBUG_INIT debugInit =
{
    .moduleInit = {0},
    .errorLevel = SYS_ERROR_DEBUG
};
  
```

Debug Messages and Error Reporting

The following macros are available to output debug and error messages:

- [SYS_MESSAGE](#)(message) prints a simple message string to the output device
- [SYS_DEBUG](#)(level, message) prints a debug message to the console device if the global error level is equal to or lower than that specified by the "level" argument. The default implementation of [SYS_DEBUG](#)(level, message) resolves to nothing by the preprocessor. This is to allow the developer to leave debug messaging in released code without impacting code size or performance. Typically, the developer would define [SYS_DEBUG](#) to map to `_SYS_DEBUG` in a configuration file for debug builds.

```
#define SYS_DEBUG(level, message) _SYS_DEBUG(level, message)
```

- **SYS_PRINT**(fmt, ...) prints formatted messages to the console. The message formatting is the same as printf.
- **SYS_ERROR**(level, fmt, ...) prints formatted messages to the console if the global error level is equal to or lower than that specified by the "level" argument. The message formatting is the same as printf.

Changing the System Error Level

Two functions are provided to manipulate the global system error level at runtime. This may be useful when you want to increase the debug verbosity for a particular section of code, but not for the entire program.

- **SYS_ERROR_LevelGet** returns the current global system error level
- **SYS_ERROR_LevelSet**(level) sets the current global system error level

Example: Changing the system error level

```
SYS_ERROR_LEVEL lvl;
/* Get current level */
lvl = SYS_ERROR_LevelGet();
/* Set new level */
SYS_ERROR_LevelSet(SYS_ERROR_DEBUG);
// Do something
SYS_DEBUG("Okay until here\n");
/* Restore previous level */
SYS_ERROR_LevelSet(lvl);
```

Configuring the Library

The configuration of the Debug System Service is based on the file `system_config.h`.

This header file contains the configuration selection for the Debug System Service. Based on the selections made, the Debug System Service may support the selected features. These configuration settings will apply to all instances of the Debug System Service.

This header can be placed anywhere; however, the path of this header needs to be present in the include search path for a successful build. Refer to the Applications Help section for more details.

The Debug System Service supports these configuration options:

```
/* Enable debug messaging */
#define SYS_DEBUG(level,message) _SYS_DEBUG(level,message)

/* Circular buffer for formatted output */
#define DEBUG_PRINT_BUFFER_SIZE 512
```

Building the Library

This section lists the files that are available in the Debug System Service Library.

Description

The following three tables list and describe the header (.h) and source (.c) files that implement this library. The parent folder for these files is `<install-dir>/framework/system/debug`.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using `#include`) by any code that uses this library.

Source File Name	Description
sys_debug.h	Debug System Service Library API header file.

Required File(s)



All of the required files listed in the following table are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.

This table lists and describes the source and header files that must *a*lways be included in the MPLAB X IDE project to build this library.

Source File Name	Description
<code>/src/sys_debug.c</code>	Debug System Service Library implementation.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
/src/sys_debug_local.h	Data structures used by the Debug System Service Library, but not exposed to clients.








Module Dependencies

The Debug System Service is dependent upon the following modules:



- [Device Control System Service Library](#)
- [Console System Service Library](#)

Library Interface

a) System Functions

	Name	Description
	SYS_DEBUG_Deinitialize	Deinitializes the specific module instance of the Debug module.
	SYS_DEBUG_Initialize	Initializes the global error level and specific module instance.
	SYS_DEBUG_Reinitialize	Reinitializes and refreshes the data structure for the instance of the Debug module.
	SYS_DEBUG_Status	Returns status of the specific instance of the debug service module.
	SYS_DEBUG_Tasks	Maintains the debug module's state machine.
	SYS_DEBUG_Message	Prints a message to the console regardless of the system error level.
	SYS_DEBUG_Print	Formats and prints a message with a variable number of arguments to the console regardless of the system error level.

b) Changing System Error Level Functions

	Name	Description
	SYS_DEBUG_ErrorLevelGet	Returns the global system Error reporting level.
	SYS_DEBUG_ErrorLevelSet	Sets the global system error reporting level.

c) Data Types and Constants

	Name	Description
	SYS_DEBUG_INIT	Defines the data required to initialize the debug system service.
	SYS_ERROR_LEVEL	System error message priority levels.
	SYS_DEBUG	This is macro SYS_DEBUG.
	SYS_DEBUG_INDEX_0	Debug System Service index.
	SYS_MESSAGE	Prints a message to the console regardless of the system error level.
	SYS_ERROR	This is macro SYS_ERROR.
	SYS_PRINT	Formats and prints an error message with a variable number of arguments regardless of the system error level.
	SYS_DEBUG_MESSAGE	Prints a debug message if the system error level is defined defined at or lower than the level specified.
	SYS_DEBUG_PRINT	Formats and prints an error message if the system error level is defined at or lower than the level specified.
	SYS_DEBUG_BreakPoint	Inserts a software breakpoint instruction when building in Debug mode.
	SYS_ERROR_PRINT	This is macro SYS_ERROR_PRINT.
	_SYS_DEBUG_MESSAGE	Prints a debug message if the specified level is at or below the global system error level.
	_SYS_DEBUG_PRINT	Formats and prints a debug message if the specified level is at or below the global system error level.

Description

This section describes the APIs of the Debug System Service Library.

Refer to each section for a detailed description.

a) System Functions

SYS_DEBUG_Deinitialize Function

Deinitializes the specific module instance of the Debug module.

File

[sys_debug.h](#)

C

```
void SYS_DEBUG_Deinitialize(SYS_MODULE_OBJ object);
```

Returns

None.

Description

This function deinitializes the specific module instance disabling its operation (and any hardware for driver modules). Resets all of the internal data structures and fields for the specified instance to the default settings.

Remarks

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

Preconditions

The [SYS_DEBUG_Initialize](#) function should have been called before calling this function.

Example

```
SYS_MODULE_OBJ    object;    // Returned from SYS_DEBUG_Initialize
SYS_DEBUG_Deinitialize (object);
```

Parameters

Parameters	Description
object	SYS DEBUG object handle, returned from SYS_DEBUG_Initialize

Function

```
void SYS_DEBUG_Deinitialize( SYS\_MODULE\_OBJ object )
```

SYS_DEBUG_Initialize Function

Initializes the global error level and specific module instance.

File

[sys_debug.h](#)

C

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

Returns

If successful, returns a valid handle to an object. Otherwise, it returns [SYS_MODULE_OBJ_INVALID](#).

Description

This function initializes the global error level. It also initializes any internal system debug module data structures.

Remarks

This routine should only be called once during system initialization. If the system was already initialized it safely returns without causing any disturbance.

Preconditions

None.

Example

```
SYS_MODULE_OBJ objectHandle;
SYS_DEBUG_INIT debugInit =
{
    .moduleInit = {0},
    .errorLevel = SYS_ERROR_DEBUG,
};

objectHandle = SYS_Debug_Initialize (SYS_DEBUG_CONSOLE, (SYS_MODULE_INIT*)&debugInit);
```

```
if (SYS_MODULE_OBJ_INVALID == objectHandle)
{
    // Handle error
}
```

Parameters

Parameters	Description
index	Index for the instance to be initialized
init	Pointer to a data structure containing any data necessary to initialize the debug service. This pointer may be null if no data is required because static overrides have been provided.

Function

```
SYS_MODULE_OBJ SYS_DEBUG_Initialize( const SYS_MODULE_INDEX index,
const SYS_MODULE_INIT * const init )
```

SYS_DEBUG_Reinitialize Function

Reinitializes and refreshes the data structure for the instance of the Debug module.

File

sys_debug.h

C

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

Returns

None

Description

This function reinitializes and refreshes the data structure for the instance of the Debug module using the supplied data.

Remarks

This operation uses the same initialization data structure as the [SYS_DEBUG_Initialize](#) operation. This function can be called multiple times to reinitialize the module.

Preconditions

The [SYS_DEBUG_Initialize](#) function should have been called before calling this function.

Example

```
SYS_MODULE_OBJ objectHandle;

// Populate the console initialization structure
SYS_DEBUG_INIT dbgInit =
{
    .moduleInit = {0},
    .errorLevel = SYS_ERROR_DEBUG,
};

SYS_DEBUG_Reinitialize (objectHandle, (SYS_MODULE_INIT*)&dbgInit);
```

Parameters

Parameters	Description
object	Identifies the SYS DEBUG Object returned by the Initialize interface
init	Pointer to the data structure containing any data necessary to initialize the hardware

Function

```
void SYS_DEBUG_Reinitialize( SYS_MODULE_OBJ object,
const SYS_MODULE_INIT * const init )
```

SYS_DEBUG_Status Function

Returns status of the specific instance of the debug service module.

File

[sys_debug.h](#)

C

```
SYS_STATUS SYS_DEBUG_Status(SYS_MODULE_OBJ object);
```

Returns

- SYS_STATUS_READY - Indicates that the module is running and ready to service requests. Any value greater than SYS_STATUS_READY is also a normal running state in which the module is ready to accept new operations.
- SYS_STATUS_BUSY - Indicates that the module is busy with a previous system level operation and cannot start another.
- SYS_STATUS_ERROR - Indicates that the module is in an error state. Any value less than SYS_STATUS_ERROR is also an error state.
- SYS_STATUS_UNINITIALIZED - Indicates that the module has not been initialized or has been deinitialized.

Description

This function returns the status of the specific debug service module instance.

Remarks

None.

Preconditions

The [SYS_CONSOLE_Initialize](#) function should have been called before calling this function.

Example

```
SYS_MODULE_OBJ    object;    // Returned from SYS_CONSOLE_Initialize
SYS_STATUS        debugStatus;

debugStatus = SYS_DEBUG_Status (object);
if (SYS_STATUS_ERROR >= debugStatus)
{
    // Handle error
}
```

Parameters

Parameters	Description
object	Debug module object handle, returned from SYS_DEBUG_Initialize

Function

```
SYS_STATUS SYS_DEBUG_Status( SYS_MODULE_OBJ object )
```

SYS_DEBUG_Tasks Function

Maintains the debug module's state machine.

File

[sys_debug.h](#)

C

```
void SYS_DEBUG_Tasks(SYS_MODULE_OBJ object);
```

Returns

None.

Description

This function is used to maintain the debug module's internal state machine and implement its ISR for interrupt-driven implementations.

Remarks

This function is normally not called directly by an application. It is called by the system's Tasks routine ([SYS_Tasks](#)) or by the appropriate raw ISR.

Preconditions

The [SYS_DEBUG_Initialize](#) function must have been called for the specified CONSOLE driver instance.

Example

```
SYS_MODULE_OBJ    object;    // Returned from SYS_DEBUG_Initialize

while (true)
{
    SYS_DEBUG_Tasks (object);

    // Do other tasks
}
```

Parameters

Parameters	Description
object	SYS DEBUG object handle, returned from SYS_DEBUG_Initialize

Function

```
void SYS_DEBUG_Tasks( SYS\_MODULE\_OBJ object )
```

SYS_DEBUG_Message Function

Prints a message to the console regardless of the system error level.

File

[sys_debug.h](#)

C

```
void SYS_DEBUG_Message(const char * message);
```

Returns

None.

Description

This function prints a message to the console regardless of the system error level. It can be used as an implementation of the [SYS_MESSAGE](#) and [SYS_DEBUG_MESSAGE](#) macros.

Remarks

Do not call this function directly. Call the [SYS_MESSAGE](#) or [SYS_DEBUG_MESSAGE](#) macros instead.

The default [SYS_MESSAGE](#) and [SYS_DEBUG_MESSAGE](#) macro definitions remove the messages and message function calls from the source code. To access and utilize the messages, define the [SYS_DEBUG_USE_CONSOLE](#) macro or override the definitions of the individual macros.

Preconditions

[SYS_DEBUG_Initialize](#) must have returned a valid object handle and the [SYS_DEBUG_Tasks](#) function must be called by the system to complete the message request.

Example

```
// In system_config.h:
#define SYS_MESSAGE(message)  SYS_DEBUG_Message(message)

// In source (.c) files:
SYS_MESSAGE("My Message\r");
```

Parameters

Parameters	Description
message	Pointer to a message string to be displayed.

Function

```
SYS_DEBUG_Message( const char* message );
```

SYS_DEBUG_Print Function

Formats and prints a message with a variable number of arguments to the console regardless of the system error level.

File

[sys_debug.h](#)

C

```
void SYS_DEBUG_Print(const char * format, ...);
```

Returns

None.

Description

This function formats and prints a message with a variable number of arguments to the console regardless of the system error level. It can be used to implement the [SYS_PRINT](#) and [SYS_DEBUG_PRINT](#) macros.

Remarks

The format string and arguments follow the printf convention.

Do not call this function directly. Call the [SYS_PRINT](#) or [SYS_DEBUG_PRINT](#) macros instead.

The default [SYS_PRINT](#) and [SYS_DEBUG_PRINT](#) macro definitions remove the messages and message function calls. To access and utilize the messages, define the `SYS_DEBUG_USE_CONSOLE` macro or override the definitions of the individual macros.

Preconditions

[SYS_DEBUG_Initialize](#) must have returned a valid object handle and the [SYS_DEBUG_Tasks](#) function must be called by the system to complete the message request.

Example

```
// In system_config.h
#define SYS_PRINT(format, ...) SYS_DEBUG_Print(format, ##__VA_ARGS__)

// In source code
int result;

result = SomeOperation();
if (result > MAX_VALUE)
{
    SYS_PRINT("Result of %d exceeds max value", result);
}
```

Parameters

Parameters	Description
format	Pointer to a buffer containing the format string for the message to be displayed.
...	Zero or more optional parameters to be formatted as defined by the format string.

Function

```
SYS_DEBUG_Print( const char* format, ... )
```

b) Changing System Error Level Functions

SYS_DEBUG_ErrorLevelGet Function

Returns the global system Error reporting level.

File

[sys_debug.h](#)

C

```
SYS_ERROR_LEVEL SYS_DEBUG_ErrorLevelGet();
```

Returns

The global System Error Level.

Description

This function returns the global System Error reporting level.

Remarks

None.

Preconditions

[SYS_DEBUG_Initialize](#) must have returned a valid object handle.

Example

```
SYS_ERROR_LEVEL level;  
  
level = SYS_DEBUG_ErrorLevelGet();
```

Function

[SYS_ERROR_LEVEL](#) SYS_DEBUG_ErrorLevelGet(void);

SYS_DEBUG_ErrorLevelSet Function

Sets the global system error reporting level.

File

[sys_debug.h](#)

C

```
void SYS_DEBUG_ErrorLevelSet(SYS_ERROR_LEVEL level);
```

Returns

None.

Description

This function sets the global system error reporting level.

Remarks

None.

Preconditions

[SYS_DEBUG_Initialize](#) must have returned a valid object handle.

Example

```
SYS_DEBUG_ErrorLevelSet(SYS_ERROR_WARNING);
```

Parameters

Parameters	Description
level	The desired system error level.

Function

```
void SYS_DEBUG_ErrorLevelSet( SYS\_ERROR\_LEVEL level);
```

c) Data Types and Constants

SYS_DEBUG_INIT Structure

Defines the data required to initialize the debug system service.

File

[sys_debug.h](#)

C

```
typedef struct {
```

```
SYS_MODULE_INIT moduleInit;  
SYS_ERROR_LEVEL errorLevel;  
SYS_MODULE_INDEX consoleIndex;  
} SYS_DEBUG_INIT;
```

Members

Members	Description
SYS_MODULE_INIT moduleInit;	System module initialization
SYS_ERROR_LEVEL errorLevel;	Initial system error level setting.
SYS_MODULE_INDEX consoleIndex;	Console index to receive debug messages

Description

SYS Debug Initialize structure

This structure defines the data required to initialize the debug system service.

Remarks

None.

SYS_ERROR_LEVEL Enumeration

System error message priority levels.

File

[sys_debug.h](#)

C

```
typedef enum {  
    SYS_ERROR_FATAL = 0,  
    SYS_ERROR_ERROR = 1,  
    SYS_ERROR_WARNING = 2,  
    SYS_ERROR_INFO = 3,  
    SYS_ERROR_DEBUG = 4  
} SYS_ERROR_LEVEL;
```

Members

Members	Description
SYS_ERROR_FATAL = 0	Errors that have the potential to cause a system crash.
SYS_ERROR_ERROR = 1	Errors that have the potential to cause incorrect behavior.
SYS_ERROR_WARNING = 2	Warnings about potentially unexpected behavior or side effects.
SYS_ERROR_INFO = 3	Information helpful to understanding potential errors and warnings.
SYS_ERROR_DEBUG = 4	Verbose information helpful during debugging and testing.

Description

SYS_ERROR_LEVEL enumeration

This enumeration defines the supported system error message priority values.

Remarks

Used by debug message macros to compare individual message priority against a global system-wide error message priority level to determine if an individual message should be displayed.

SYS_DEBUG Macro

File

[sys_command.h](#)

C

```
#define SYS_DEBUG(level,message) SYS_DEBUG_MESSAGE(level,message)
```

Description

This is macro SYS_DEBUG.

SYS_DEBUG_INDEX_0 Macro

Debug System Service index.

File

[sys_debug.h](#)

C

```
#define SYS_DEBUG_INDEX_0 0
```

Description

SYS Debug Module Index Number

This constant defines a symbolic name for the debug system service index.

Remarks

There can only be a single debug system service instance in the system.

SYS_MESSAGE Macro

Prints a message to the console regardless of the system error level.

File

[sys_debug.h](#)

C

```
#define SYS_MESSAGE(message)
```

Returns

None.

Description

Macro: SYS_MESSAGE(const char* message)

This macro is used to print a message to the console regardless of the system error level. It can be mapped to any desired implementation.

Remarks

By default, this macro is defined as nothing, effectively removing all code generated by calls to it. To process SYS_MESSAGE calls, this macro must be defined in a way that maps calls to it to the desired implementation (see example, above).

This macro can be mapped to the system console service (along with other system debug macros) by defining SYS_DEBUG_USE_CONSOLE in the system configuration (system_config.h) instead of defining it individually.

Preconditions

If mapped to the _SYS_MESSAGE function, then the system debug service must be initialized and running.

Example

```
// In system_config.h:
#define SYS_MESSAGE(message) _SYS_MESSAGE(message)

// In source (.c) files:
SYS_MESSAGE("My Message");
```

Parameters

Parameters	Description
message	Pointer to a buffer containing the message string to be displayed.

SYS_ERROR Macro

File

[sys_command.h](#)

C

```
#define SYS_ERROR(level,fmt, ...) SYS_ERROR_PRINT(level, fmt, ##__VA_ARGS__)
```

Description

This is macro SYS_ERROR.

SYS_PRINT Macro

Formats and prints an error message with a variable number of arguments regardless of the system error level.

File

[sys_debug.h](#)

C

```
#define SYS_PRINT(fmt, ...)
```

Returns

None.

Description

This function formats and prints an error message with a variable number of if the system error level is defined at or lower than the level specified.

Remarks

The format string and arguments follow the printf convention. This function is called by the macros SYS_PRINT and [SYS_ERROR_PRINT](#)

By default, this macro is defined as nothing, effectively removing all code generated by calls to it. To process SYS_PRINT calls, this macro must be defined in a way that maps calls to it to the desired implementation (see example, above).

This macro can be mapped to the system console service (along with other system debug macros) by defining SYS_DEBUG_USE_CONSOLE in the system configuration (system_config.h) instead of defining it individually.

Preconditions

SYSTEM_CURRENT_ERROR_LEVEL must be defined.

Example

```
// In system_config.h
#define SYS_PRINT(level, format, ...) _SYS_DEBUG_PRINT(level, format, ##__VA_ARGS__)

// In source code
int result;

result = SomeOperation();
if (result > MAX_VALUE)
{
    SYS_PRINT("Result of %d exceeds max value", result);
    // Take appropriate action
}
```

Parameters

Parameters	Description
format	Pointer to a buffer containing the format string for the message to be displayed.
...	Zero or more optional parameters to be formatted as defined by the format string.

Function

```
SYS_PRINT(const char* format, ...);
```

SYS_DEBUG_MESSAGE Macro

Prints a debug message if the system error level is defined defined at or lower than the level specified.

File

[sys_debug.h](#)

C

```
#define SYS_DEBUG_MESSAGE(level,message)
```

Returns

None.

Description

This function prints a debug message if the system error level is defined at or lower than the level specified.

Remarks

By default, this macro is defined as nothing, effectively removing all code generated by calls to it. To process SYS_DEBUG_MESSAGE calls, this macro must be defined in a way that maps calls to it to the desired implementation (see example, above).

This macro can be mapped to the system console service (along with other system debug macros) by defining SYS_DEBUG_USE_CONSOLE in the system configuration (system_config.h) instead of defining it individually.

Preconditions

SYSTEM_CURRENT_ERROR_LEVEL must be defined as SYS_ERROR_DEBUG.

Example

```
#define SYS_DEBUG_MESSAGE _SYS_DEBUG_MESSAGE
SYS_ERROR_LevelSet(SYS_ERROR_DEBUG);
SYS_DEBUG_MESSAGE("System Debug Message nr");
```

Parameters

Parameters	Description
level	The current error level threshold for displaying the message.
message	Pointer to a buffer containing the message to be displayed.

Function

```
SYS_DEBUG_MESSAGE( const char* message )
```

SYS_DEBUG_PRINT Macro

Formats and prints an error message if the system error level is defined at or lower than the level specified.

File

[sys_debug.h](#)

C

```
#define SYS_DEBUG_PRINT(level, fmt, ...)
```

Returns

None.

Description

Macro: SYS_DEBUG_PRINT([SYS_ERROR_LEVEL](#) level, const char* format, ...)

This macro formats and prints an error message if the system error level is defined at or lower than the level specified.

Remarks

The format string and arguments follow the printf convention. This function is called by the macros [SYS_PRINT](#) and SYS_DEBUG_PRINT.

By default, this macro is defined as nothing, effectively removing all code generated by calls to it. To process [SYS_MESSAGE](#) calls, this macro must be defined in a way that maps calls to it to the desired implementation (see example, above).

This macro can be mapped to the system console service (along with other system debug macros) by defining SYS_DEBUG_USE_CONSOLE in the system configuration (system_config.h) instead of defining it individually.

Preconditions

[SYS_DEBUG_Initialize](#) must have returned a valid object handle.

Example

```
// In system_config.h
#define SYS_DEBUG_PRINT(level, format, ...) _SYS_DEBUG_PRINT(level, format, ##__VA_ARGS__)
```

```
// In source code
int result;

result = SomeOperation();
if (result > MAX_VALUE)
{
    SYS_DEBUG_PRINT(SYS_ERROR_WARNING, "Result of %d exceeds max valuern", result);
    // Take appropriate action
}
```

Parameters

Parameters	Description
level	The current error level threshold for displaying the message.
format	Pointer to a buffer containing the format string for the message to be displayed.
...	Zero or more optional parameters to be formatted as defined by the format string.

SYS_DEBUG_BreakPoint Macro

Inserts a software breakpoint instruction when building in Debug mode.

File

[sys_debug.h](#)

C

```
#define SYS_DEBUG_BreakPoint
```

Returns

None.

Description

Macro: SYS_DEBUG_BreakPoint(void)

This macro inserts a software breakpoint instruction when building in Debug mode.

Remarks

Compiles out if not built for debugging.

Preconditions

None.

Example

```
if (myDebugTestFailed)
{
    SYS_DEBUG_BreakPoint();
}
```

SYS_ERROR_PRINT Macro

File

[sys_command.h](#)

C

```
#define SYS_ERROR_PRINT(level, fmt, ...) do { if((level) <= gblErrLvl) SYS_CMD_PRINT(fmt, ##__VA_ARGS__); }
while (0)
```

Description

This is macro SYS_ERROR_PRINT.

SYS_DEBUG_MESSAGE Macro

Prints a debug message if the specified level is at or below the global system error level.

File

[sys_debug.h](#)

C

```
#define _SYS_DEBUG_MESSAGE(level, message) do { if((level) <= SYS_DEBUG_ErrorLevelGet())  
SYS_DEBUG_Message(message); }while(0)
```

Returns

None.

Description

Macro: `_SYS_DEBUG_MESSAGE(SYS_ERROR_LEVEL level, const char* message)`

This macro prints a debug message if the specified level is at or below the global error level. It can be used to implement the [SYS_DEBUG_MESSAGE](#) macro.

Remarks

Do not call this macro directly. Call the [SYS_DEBUG_MESSAGE](#) macro instead.

The default [SYS_DEBUG_MESSAGE](#) macro definition removes the message and function call from the source code. To access and utilize the message, define the `SYS_DEBUG_USE_CONSOLE` macro or override the definition of the [SYS_DEBUG_MESSAGE](#) macro.

Preconditions

[SYS_DEBUG_Initialize](#) must have returned a valid object handle.

Example

```
// In system_config.h  
#define SYS_DEBUG_MESSAGE(level,message) _SYS_DEBUG_MESSAGE(level,message)  
  
// In library source (.c) code.  
SYS_DEBUG_MESSAGE(SYS_ERROR_WARNING, "My debug warning messagenr");
```

Parameters

Parameters	Description
level	The current error level threshold for displaying the message.
message	Pointer to a buffer containing the message to be displayed.

SYS_DEBUG_PRINT Macro

Formats and prints a debug message if the specified level is at or below the global system error level.

File

[sys_debug.h](#)

C

```
#define _SYS_DEBUG_PRINT(level, format, ...) do { if((level) <= SYS_DEBUG_ErrorLevelGet())  
SYS_DEBUG_Print(format, ##__VA_ARGS__); } while (0)
```

Returns

None.

Description

Macro: `_SYS_DEBUG_PRINT(SYS_ERROR_LEVEL level, const char* format, ...)`

This function formats and prints a debug message if the specified level is at or below the global system error level. It can be used to implement the [SYS_DEBUG_PRINT](#) macro.

Remarks

Do not call this macro directly. Call the [SYS_DEBUG_PRINT](#) macro instead.

The default [SYS_DEBUG_PRINT](#) macro definition removes the message and function call from the source code. To access and utilize the message, define the `SYS_DEBUG_USE_CONSOLE` macro or override the definition of the [SYS_DEBUG_PRINT](#) macro.

Preconditions

[SYS_DEBUG_Initialize](#) must have returned a valid object handle.

Example

```
// In system_config.h
#define SYS_DEBUG_PRINT(level, format, ...) _SYS_DEBUG_PRINT(level, format, ##__VA_ARGS__)

// In source code
int result;

result = SomeOperation();
if (result > MAX_VALUE)
{
    SYS_DEBUG_PRINT(SYS_ERROR_WARNING, "Result of %d exceeds max value", result);
    // Take appropriate action
}
```

Parameters

Parameters	Description
level	The current error level threshold for displaying the message.
format	Pointer to a buffer containing the format string for the message to be displayed.
...	Zero or more optional parameters to be formatted as defined by the format string.

Files

Files

Name	Description
sys_debug.h	Defines the common debug definitions and interfaces used by MPLAB Harmony libraries to report errors and debug information to the user.

Description

This section lists the source and header files used by the library.










sys_debug.h

Defines the common debug definitions and interfaces used by MPLAB Harmony libraries to report errors and debug information to the user.

Enumerations

	Name	Description
	SYS_ERROR_LEVEL	System error message priority levels.

Functions

	Name	Description
	SYS_DEBUG_Deinitialize	Deinitializes the specific module instance of the Debug module.
	SYS_DEBUG_ErrorLevelGet	Returns the global system Error reporting level.
	SYS_DEBUG_ErrorLevelSet	Sets the global system error reporting level.
	SYS_DEBUG_Initialize	Initializes the global error level and specific module instance.
	SYS_DEBUG_Message	Prints a message to the console regardless of the system error level.
	SYS_DEBUG_Print	Formats and prints a message with a variable number of arguments to the console regardless of the system error level.
	SYS_DEBUG_Reinitialize	Reinitializes and refreshes the data structure for the instance of the Debug module.
	SYS_DEBUG_Status	Returns status of the specific instance of the debug service module.
	SYS_DEBUG_Tasks	Maintains the debug module's state machine.

Macros

	Name	Description
	_SYS_DEBUG_MESSAGE	Prints a debug message if the specified level is at or below the global system error level.
	_SYS_DEBUG_PRINT	Formats and prints a debug message if the specified level is at or below the global system error level.
	SYS_DEBUG_BreakPoint	Inserts a software breakpoint instruction when building in Debug mode.
	SYS_DEBUG_INDEX_0	Debug System Service index.

	SYS_DEBUG_MESSAGE	Prints a debug message if the system error level is defined defined at or lower than the level specified.
	SYS_DEBUG_PRINT	Formats and prints an error message if the system error level is defined at or lower than the level specified.
	SYS_MESSAGE	Prints a message to the console regardless of the system error level.
	SYS_PRINT	Formats and prints an error message with a variable number of arguments regardless of the system error level.

Structures

	Name	Description
	SYS_DEBUG_INIT	Defines the data required to initialize the debug system service.

Description

Debug System Services Library Header

This header file defines the common debug definitions and interface macros (summary below) and prototypes used by MPLAB Harmony libraries to report errors and debug information to the user.

File Name

sys_debug.h

Company

Microchip Technology Inc.

Device Control System Service Library

This section describes the Device Control System Service Library.

Introduction

This library provides a low-level abstraction of the Device Control System Service Library that is available on the Microchip family of PIC32 microcontrollers with a convenient C language interface. It can be used to simplify low-level access to the module without the necessity of interacting directly with the module's registers, there by hiding differences from one microcontroller variant to another.

Description

The Device Control System Service provides the developer with simple APIs to configure and control device-specific functions. These functions may involve one or more peripherals, or no peripherals at all.

Using the Library

This topic describes the basic architecture of the Device Control System Service Library and provides information and examples on its use.

Description

Interface Header File: [sys_devcon.h](#)

The interface to the Device Control System Service library is defined in the [sys_devcon.h](#) header file.

Please refer to the What is MPLAB Harmony? section for how the Device Control System Service interacts with the framework.

Abstraction Model

This topic provides a description of the software abstraction for the Device Control System Service.

Description

The Device Control System Service is a collection of device-specific operations that may not be tied to any specific peripheral, multiple peripherals, or may not involve any peripherals at all. This service provides functions to optimize system performance, perform cache management, and do other system-level tasks that may be needed by multiple modules.

Library Overview

Refer to the section [System Services Introduction](#) for how the system services operates in a system.

The library interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the Device Control module.

Library Interface Section	Description
System Interaction Functions	Provides system module APIs. Device initialization, deinitialization, reinitialization and status functions.
Core Functionality Functions	Device-specific configuration and control functions.
Cache Functions	Configuration and control functions specific to L1 Cache.
Data Types and Constants	Device-specific enumerations, structures, and constants.

How the Library Works

System Interaction

Initialization and Reinitialization

Initialization of the Device Control System Service initializes the status of the module and sets the state of the internal state machine.

The Device Control Status and Tasks routines are provided as stubs for future enhancement. They are not currently required for this system service.

Core Functionality

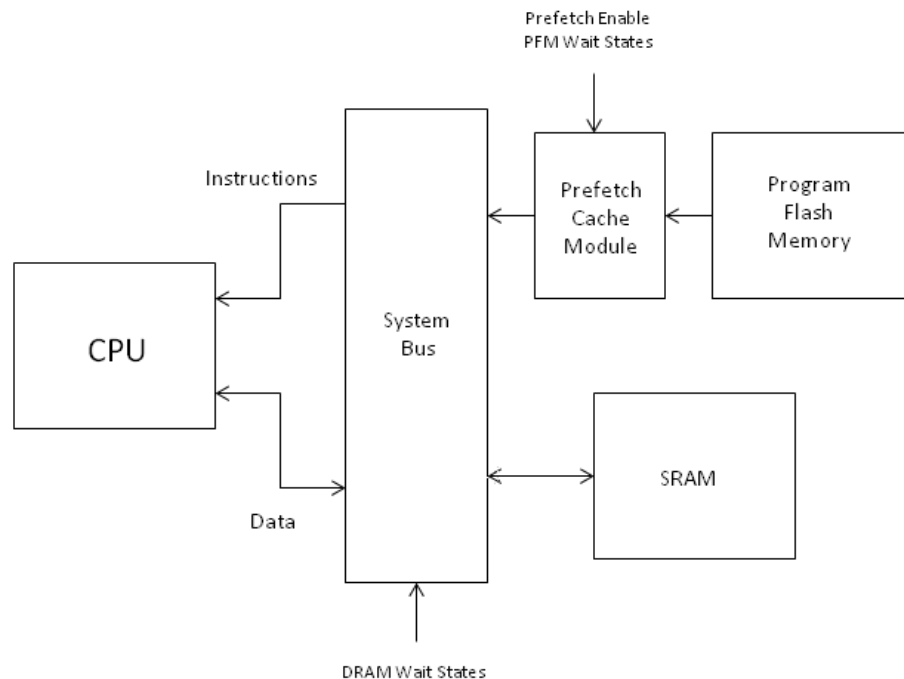
System Performance Tuning

When any PIC32 device is powered up, the Program Flash Memory and Data RAM memory wait states are not set to optimum values. By setting these wait states to the minimum required for the programmed clock speed, enabling the prefetch cache (if available), performance can be optimized. This service provides a single API that performs this optimization across all PIC32 devices.

The `SYS_DEVCON_PerformanceConfig` function takes the current system clock frequency as an argument and uses this value to set the program Flash wait states to the optimum value. It enables the Prefetch Cache module on both PIC32MX and PIC32MZ devices, and sets the SRAM Wait state for PIC32MX devices.

```
SYS_DEVCON_PerformanceConfig( unsigned int sysclk )
```

Performance Tuning Block Diagram



System Locking and Unlocking

Many operations within the PIC32 device require the system to be unlocked before the operation is performed. This is a safety feature which prevents errant code from altering critical settings by mistake. The unlock sequence consists of writing a series of "magic numbers" to a specific register in sequence. The critical operation is then performed, and the system is relocked by writing to the same register. This service provides convenient APIs that work across all PIC32 devices to perform this lock/unlock sequence.

```
SYS_DEVCON_SystemUnlock();
// critical operation
SYS_DEVCON_SystemLock();
```



Note: For the operations that require a lock/unlock sequence, refer to the specific device data sheet to determine the requirement for your device.

L1 Cache

This topic provides a description of the management functions for the Level 1 (L1) Cache that is available in PIC32MZ family devices.

Description

The Device Control System Service contains an interface for controlling the L1 Cache in the PIC32MZ family of devices.

The L1 cache is divided into two parts, a Data Cache (D-cache) and an Instruction Cache (I-cache). These blocks of high-speed memory both serve to compensate for the lengthy access time of main memory, by fetching instructions and data for the CPU ahead of time. The CPU can then access the information directly through the cache in a single clock cycle, rather than having to wait multiple clock cycles for accesses to main memory. The L1 cache provides a drastic increase in performance, but the user must be aware of hazards that exist when using the cache.

Cache Coherency

Cache coherency is the discipline of ensuring that the data stored in main memory matches the corresponding data in the cache. The majority of the cache-related APIs deal with cache coherency. These functions allow the user to flush, clean and invalidate entire cache(s), or just a range of addresses within the cache.

Caches most often lose coherency when a bus master other than the CPU modifies memory. This happens frequently with DMA. Two examples are provided in the following section.

Examples

Example 1:

Imagine a situation where you would like to transfer data from a source buffer to a destination buffer using DMA. You would write data to the source buffer, start the DMA transfer, and then expect that the same data now appears in the destination buffer. With the cache in write-back mode (the default mode for the PIC32MZ family), this will not be the case. When transferring data out of memory using DMA, it is possible that the desired data is held in the D-cache, but has never been written back to main memory. Therefore, in this case, you write data to the source buffer and it gets stored in cache. When the DMA transfer executes, it will pull the data from the source buffer out of RAM and then transfer it to the destination buffer in RAM. The problem is that the fresh data was stored in the cache but never written back to RAM, so what has happened is that stale data was copied over rather than the intended data. What is needed is a way to force the cache to write its data back to main memory before the DMA transfer. This is known as a *write-back operation* and would be performed with the use of the function:

[SYS_DEVCON_DataCacheClean](#)(uint32_t addr, size_t len)

The example application, `devcon_cache_clean`, in the Device Control System Service Examples demonstrates this situation and shows how to resolve the issue.

Example 2:

The second situation involves writing data into memory using DMA. Imagine that the cache is holding a chunk of data known as `destination_buffer`. You then execute a DMA transfer to copy some new data from a source buffer into `destination_buffer`. The issue here is that main memory now contains the correct data, but the cache holds a copy of stale data for `destination_buffer`. The CPU cannot see this problem and it will keep pulling the data out of the cache, not even realizing that it's stale. What is needed is a way to tell the cache to pull the fresh data out of main memory, to replace the stale data that the cache contains. This is known as an *invalidate operation*. It is performed with the use of the function:

[SYS_DEVCON_DataCacheInvalidate](#)(uint32_t addr, size_t len)

The example application, `devcon_cache_invalidate`, in the Device Control System Service Examples demonstrates this situation and shows how to resolve the issue.

Configuring the Library

No configuration is required for the Device Control System Service.

Building the Library

This section lists the files that are available in the Device Control System Service Library.

Description

The following three tables list and describe the header (.h) and source (.c) files that implement this library. The parent folder for these files is `<install-dir>/framework/system/devcon`.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using `#include`) by any code that uses this library.

Source File Name	Description
sys_devcon.h	Device Control System Service Library API header file.

Required File(s)



All of the required files listed in the following table are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

Source File Name	Description
<code>/src/sys_devcon.c</code>	Device Control System Service Library implementation.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
src/sys_devcon_pic32mx.c	PIC32MX-specific implementation of SYS_DEVCON_PerformanceConfig .
/src/sys_devcon_pic32mz.c	PIC32MZ-specific implementation of SYS_DEVCON_PerformanceConfig .
/src/sys_devcon_local.h	Data structures and definitions used by the Device Control System Service, but not exposed to clients.
/src/sys_devcon_cache_pic3mz.S	PIC32MZ-specific implementation to support L1 Cache.

Module Dependencies

The Device Control System Service is not dependent upon any other modules.

Library Interface

a) System Interaction Functions


	Name	Description
⇒	SYS_DEVCON_Deinitialize	Deinitializes the specific module instance of the DEVCON module
⇒	SYS_DEVCON_Initialize	Initializes data for the instance of the Device Control module and opens the specific module instance.
⇒	SYS_DEVCON_Reinitialize	Reinitializes and refreshes the hardware for the instance of the Device Control module.
⇒	SYS_DEVCON_Status	Returns status of the specific instance of the Device Control module.
⇒	SYS_DEVCON_Tasks	Maintains the system Device Control state machine.

b) Core Functionality Functions

	Name	Description
⇒	SYS_DEVCON_SystemLock	Performs a system lock sequence by writing to the SYSKEY register.
⇒	SYS_DEVCON_SystemUnlock	Performs a system unlock sequence by writing to the SYSKEY register.
⇒	SYS_DEVCON_JTAGDisable	Disables the JTAG port on the device.
⇒	SYS_DEVCON_JTAGEnable	Enables the JTAG port on the device.
⇒	SYS_DEVCON_TraceDisable	Disables the trace output port on the device.
⇒	SYS_DEVCON_TraceEnable	Enables the trace output port on the device.
⇒	SYS_DEVCON_PerformanceConfig	Configures the PFM wait states and Prefetch Cache module for maximum performance.
⇒	SYS_DEVCON_PowerModeEnter	Puts the device in a low-power state.

c) Cache Functions

	Name	Description
⇒	SYS_DEVCON_CacheClean	Writes back and invalidates an address range in either cache.
⇒	SYS_DEVCON_CacheCoherencyGet	Returns the current cache coherency attribute for kseg0.
⇒	SYS_DEVCON_CacheCoherencySet	Sets the cache coherency attribute for kseg0.
⇒	SYS_DEVCON_CacheFlush	Flushes the L1 cache.
⇒	SYS_DEVCON_CacheInit	Initializes the L1 cache.
⇒	SYS_DEVCON_CacheSync	Synchronizes the instruction and data caches.
⇒	SYS_DEVCON_DataCacheClean	Writes back and invalidates an address range in the data cache.
⇒	SYS_DEVCON_DataCacheFlush	Flushes the L1 data cache.
⇒	SYS_DEVCON_DataCacheInvalidate	Invalidates an address range in the data cache.
⇒	SYS_DEVCON_DataCacheLock	Fetches and locks a block of data in the data cache.
⇒	SYS_DEVCON_DataCacheAssociativityGet	Returns the number of ways in the data cache.
⇒	SYS_DEVCON_DataCacheLineSizeGet	Returns the data cache line size.
⇒	SYS_DEVCON_DataCacheLinesPerWayGet	Returns the number of lines per way in the data cache.
⇒	SYS_DEVCON_DataCacheSizeGet	Returns the total number of bytes in the data cache.
⇒	SYS_DEVCON_InstructionCacheFlush	Flushes (invalidates) the L1 instruction cache.
⇒	SYS_DEVCON_InstructionCacheInvalidate	Invalidates an address range in the instruction cache.
⇒	SYS_DEVCON_InstructionCacheLock	Fetches and locks a block of instructions in the instruction cache.
⇒	SYS_DEVCON_InstructionCacheAssociativityGet	Returns the number of ways in the instruction cache.
⇒	SYS_DEVCON_InstructionCacheLineSizeGet	Returns the instruction cache line size.
⇒	SYS_DEVCON_InstructionCacheLinesPerWayGet	Returns the number of lines per way in the instruction cache.

	SYS_DEVCON_InstructionCacheSizeGet	Returns the total number of bytes in the instruction cache.
---	--	---

d) Data Types and Constants

	Name	Description
	SYS_DEVCON_HANDLE	Identifies a particular registered event instance.
	SYS_DEVCON_INIT	Identifies the system device control initialize structure.
	SYS_DEVCON_INDEX_0	Device Control System Service index definitions.
	SYS_CACHE_COHERENCY	System L1 cache coherency settings.
	SYS_POWER_MODE	System power mode settings.

Description

This section describes the APIs of the Device Control System Service Library.

Refer to each section for a detailed description.

a) System Interaction Functions

SYS_DEVCON_Deinitialize Function

Deinitializes the specific module instance of the DEVCON module

File

[sys_devcon.h](#)

C

```
void SYS_DEVCON_Deinitialize(SYS_MODULE_OBJ object);
```

Returns

None.

Description

This function deinitializes the specific module instance disabling its operation (and any hardware for driver modules). Resets all of the internal data structures and fields for the specified instance to the default settings.

Remarks

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

Preconditions

The [SYS_DEVCON_Initialize](#) function should have been called before calling this function.

Example

```
SYS_MODULE_OBJ    object;    // Returned from SYS_DEVCON_Initialize
SYS_STATUS        status;

SYS_DEVCON_Deinitialize (object);

status = SYS_DEVCON_Status (object);
if (SYS_MODULE_DEINITIALIZED == status)
{
    // Check again later if you need to know
    // when the SYS DEVCON is deinitialized.
}
```

Parameters

Parameters	Description
object	SYS DEVCON object handle, returned from SYS_DEVCON_Initialize

Function

```
void SYS_DEVCON_Deinitialize( SYS_MODULE_OBJ object )
```

SYS_DEVCON_Initialize Function

Initializes data for the instance of the Device Control module and opens the specific module instance.

File

[sys_devcon.h](#)

C

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

Returns

If successful, returns a valid handle to an object. Otherwise, it returns [SYS_MODULE_OBJ_INVALID](#). The returned object must be passed as argument to [SYS_DEVCON_Reinitialize](#), [SYS_DEVCON_Deinitialize](#), [SYS_DEVCON_Tasks](#) and [SYS_DEVCON_Status](#) routines.

Description

This function initializes the instance of the Device Control module, using the specified initialization data. It also initializes any internal data structures.

Remarks

This routine should only be called once during system initialization unless [SYS_DEVCON_Deinitialize](#) is first called to deinitialize the device instance before reinitializing it. If the system was already initialized it safely returns without causing any disturbance.

Preconditions

None.

Example

```
SYS_MODULE_OBJ  objectHandle;
SYS_DEVCON_INIT  initConfig;

// Populate the DEVCON initialization structure
initConfig.moduleInit.value = SYS_MODULE_POWER_RUN_FULL;

objectHandle = SYS_DEVCON_Initialize (SYS_DEVCON_INDEX_0, (SYS_MODULE_INIT*)&initConfig);
if (SYS_MODULE_OBJ_INVALID == objectHandle)
{
    // Handle error
}
```

Parameters

Parameters	Description
index	Index for the instance to be initialized
init	Pointer to a data structure containing any data necessary to initialize the Device Control module. This pointer may be null if no data is required because static overrides have been provided.

Function

```
SYS_MODULE_OBJ SYS_DEVCON_Initialize( const SYS_MODULE_INDEX index,
const          SYS_MODULE_INIT * const init )
```

SYS_DEVCON_Reinitialize Function

Reinitializes and refreshes the hardware for the instance of the Device Control module.

File

[sys_devcon.h](#)

C

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

Returns

None

Description

This function reinitializes the instance of the Device Control module using the supplied data. It modifies the internal data structure.

Remarks

This operation uses the same initialization data structure as the [SYS_DEVCON_Initialize](#) operation. This operation can be used to change the power state of a DEVCON module. This function can be called multiple times to reinitialize the module. This operation uses the same initialization data structure as the Initialize operation. This operation can also be used to refresh the hardware registers as defined by the initialization data.

Preconditions

The [SYS_DEVCON_Initialize](#) function should have been called before calling this function.

Example

```
SYS_MODULE_OBJ  objectHandle;
SYS_DEVCON_INIT  initConfig;
SYS_STATUS      devconStatus;

// Populate the device control initialization structure
initConfig.moduleInit.value = SYS_MODULE_POWER_RUN_FULL;

SYS_DEVCON_Reinitialize (objectHandle, (SYS_MODULE_INIT*)&initConfig);

devconStatus = SYS_DEVCON_Status (object);
if (SYS_STATUS_ERROR >= devconStatus)
{
    // Handle error
}
```

Parameters

Parameters	Description
object	Identifies the SYS DEVCON Object returned by the Initialize interface
init	Pointer to the data structure containing any data necessary to initialize the hardware

Function

```
void SYS_DEVCON_Reinitialize(  SYS\_MODULE\_OBJ object,
const      SYS\_MODULE\_INIT * const init )
```

SYS_DEVCON_Status Function

Returns status of the specific instance of the Device Control module.

File

[sys_devcon.h](#)

C

```
SYS_STATUS SYS\_DEVCON\_Status(SYS\_MODULE\_OBJ object);
```

Returns

- [SYS_STATUS_READY](#) - Indicates that any previous operations have succeeded and the module is ready for additional operations. Any value greater than [SYS_STATUS_READY](#) is also a normal running state in which the driver is ready to accept new operations.
- [SYS_STATUS_BUSY](#) - Indicates that the driver is busy with a previous system level operation and cannot start another.
- [SYS_STATUS_ERROR](#) - Indicates that the driver is in an error state. Any value less than [SYS_STATUS_ERROR](#) is also an error state.
- [SYS_MODULE_DEINITIALIZED](#) - Indicates that the driver has been deinitialized. This value is less than [SYS_STATUS_ERROR](#). Once the Initialize operation has been called, the Deinitialize operation must be called before the Initialize operation can be called again.

Description

This function returns the status of the specific module instance disabling its operation (and any hardware for driver modules).

Remarks

None.

Preconditions

The [SYS_DEVCON_Initialize](#) function should have been called before calling this function.

Example

```
SYS_MODULE_OBJ    object;    // Returned from SYS_DEVCON_Initialize
SYS_STATUS        tmrStatus;

devconStatus = SYS_DEVCON_Status (object);
else if (SYS_STATUS_ERROR >= devconStatus)
{
    // Handle error
}
```

Parameters

Parameters	Description
object	SYS DEVCON object handle, returned from SYS_DEVCON_Initialize

Function

[SYS_STATUS](#) SYS_DEVCON_Status([SYS_MODULE_OBJ](#) object)

SYS_DEVCON_Tasks Function

Maintains the system Device Control state machine.

File

[sys_devcon.h](#)

C

```
void SYS_DEVCON_Tasks( SYS_MODULE_OBJ object );
```

Returns

None.

Description

This function is used to maintain the system Device Control internal state machine.

Remarks

This function is normally not called directly by an application. It is called by the system's Tasks routine ([SYS_Tasks](#)) or by the appropriate raw ISR.

Preconditions

The [SYS_DEVCON_Initialize](#) function must have been called for the specified DEVCON driver instance.

Example

```
SYS_MODULE_OBJ    object;    // Returned from SYS_DEVCON_Initialize

while (true)
{
    SYS_DEVCON_Tasks (object);

    // Do other tasks
}
```

Parameters

Parameters	Description
object	SYS DEVCON object handle, returned from SYS_DEVCON_Initialize

Function

void SYS_DEVCON_Tasks([SYS_MODULE_OBJ](#) object)

b) Core Functionality Functions

SYS_DEVCON_SystemLock Function

Performs a system lock sequence by writing to the SYSKEY register.

File

[sys_devcon.h](#)

C

```
void SYS_DEVCON_SystemLock( );
```

Returns

None.

Description

Performs a system lock sequence by writing to the SYSKEY register. A system lock sequence is required after performing the action that required a system lock sequence.

Remarks

None.

Preconditions

None.

Example

```
SYS_DEVCON_SystemUnlock( );  
// Perform some action  
SYS_DEVCON_SystemLock( );
```

Function

```
void SYS_DEVCON_SystemLock( void )
```

SYS_DEVCON_SystemUnlock Function

Performs a system unlock sequence by writing to the SYSKEY register.

File

[sys_devcon.h](#)

C

```
void SYS_DEVCON_SystemUnlock( );
```

Returns

None.

Description

Performs a system unlock sequence by writing to the SYSKEY register. A system unlock sequence is required before performing certain actions such as changing a clock frequency or I/O unlocking.

Remarks

None.

Preconditions

None.

Example

```
SYS_DEVCON_SystemUnlock( );  
// Perform some action  
SYS_DEVCON_SystemLock( );
```

Function

```
void SYS_DEVCON_SystemUnlock( void )
```

SYS_DEVCON_JTAGDisable Function

Disables the JTAG port on the device.

File

[sys_devcon.h](#)

C

```
void SYS_DEVCON_JTAGDisable();
```

Returns

None.

Description

This function disables the JTAG port on the device.

Remarks

None.

Preconditions

None.

Example

```
SYS_DEVCON_JTAGDisable();
```

Function

```
void SYS_DEVCON_JTAGDisable( void )
```

SYS_DEVCON_JTAGEnable Function

Enables the JTAG port on the device.

File

[sys_devcon.h](#)

C

```
void SYS_DEVCON_JTAGEnable();
```

Returns

None.

Description

This function enables the JTAG port on the device.

Remarks

None.

Preconditions

None.

Example

```
SYS_DEVCON_JTAGEnable();
```

Function

```
void SYS_DEVCON_JTAGEnable( void )
```

SYS_DEVCON_TraceDisable Function

Disables the trace output port on the device.

File

[sys_devcon.h](#)

C

```
void SYS_DEVCON_TraceDisable();
```

Returns

None.

Description

This function disables the trace output port on the device.

Remarks

None.

Preconditions

None.

Example

```
SYS_DEVCON_TraceDisable();
```

Function

```
void SYS_DEVCON_TraceDisable( void )
```

SYS_DEVCON_TraceEnable Function

Enables the trace output port on the device.

File

[sys_devcon.h](#)

C

```
void SYS_DEVCON_TraceEnable();
```

Returns

None.

Description

This function enables the trace output port on the device.

Remarks

None.

Preconditions

None.

Example

```
SYS_DEVCON_TraceEnable();
```

Function

```
void SYS_DEVCON_TraceEnable( void )
```

SYS_DEVCON_PerformanceConfig Function

Configures the PFM wait states and Prefetch Cache module for maximum performance.

File

[sys_devcon.h](#)

C

```
void SYS_DEVCON_PerformanceConfig(unsigned int sysclk);
```

Returns

None.

Description

This function configures the PFM wait states and Prefetch Cache module for maximum performance.

Remarks

None.

Preconditions

The [SYS_DEVCON_Initialize](#) function should have been called before calling this function.

Example

```
SYS_DEVCON_PerformanceConfig(SYS_FREQUENCY);
```

Function

```
void SYS_DEVCON_PerformanceConfig( unsigned int sysclk )
```

SYS_DEVCON_PowerModeEnter Function

Puts the device in a low-power state.

File

[sys_devcon.h](#)

C

```
void SYS_DEVCON_PowerModeEnter( SYS_POWER_MODE pwrMode );
```

Returns

None.

Description

This function places the device in a low-power state.

Remarks

None.

Preconditions

None.

Example

```
SYS_DEVCON_PowerModeEnter( SYS_POWER_MODE_SLEEP );
```

Parameters

Parameters	Description
pwrMode	The low power state to enter.

Function

```
void SYS_DEVCON_PowerModeEnter( SYS_POWER_MODE pwrMode)
```

c) Cache Functions

SYS_DEVCON_CacheClean Function

Writes back and invalidates an address range in either cache.

File

[sys_devcon.h](#)

C

```
void SYS_DEVCON_CacheClean(uint32_t addr, size_t len);
```

Returns

None.

Description

This function writes back (data) and invalidates (data and address) an address range in either cache.

Remarks

None.

Preconditions

None.

Example

```
SYS_DEVCON_CacheClean(&myData, sizeof(myData));
```

Parameters

Parameters	Description
addr	Starting address of data block.
len	Size of data block.

Function

```
void SYS_DEVCON_CacheClean( uint32_t addr, size_t len )
```

SYS_DEVCON_CacheCoherencyGet Function

Returns the current cache coherency attribute for kseg0.

File

[sys_devcon.h](#)

C

```
SYS_CACHE_COHERENCY SYS_DEVCON_CacheCoherencyGet();
```

Returns

The current cache coherency attribute for kseg0.

Description

This function returns the current cache coherency attribute for kseg0.

Remarks

None.

Preconditions

None.

Example

```
SYS_CACHE_COHERENCY cacheCoherency;  
cacheCoherency = SYS_DEVCON_CacheCoherencyGet();
```

Function

```
SYS_CACHE_COHERENCY SYS_DEVCON_CacheCoherencyGet( void )
```

SYS_DEVCON_CacheCoherencySet Function

Sets the cache coherency attribute for kseg0.

File

[sys_devcon.h](#)

C

```
void SYS_DEVCON_CacheCoherencySet(SYS_CACHE_COHERENCY cacheCoherency);
```

Returns

None.

Description

This function sets the cache coherency attribute for kseg0.

Remarks

Use with caution. May cause unpredictable behavior in a running system.

Preconditions

None.

Example

```
SYS_DEVCON_CacheCoherencySet(SYS_CACHE_WRITEBACK_WRITEALLOCATE);
```

Parameters

Parameters	Description
cacheCoherency	The desired coherency attribute.

Function

```
void SYS_DEVCON_CacheCoherencySet(SYS_CACHE_COHERENCY cacheCoherency)
```

SYS_DEVCON_CacheFlush Function

Flushes the L1 cache.

File

[sys_devcon.h](#)

C

```
void SYS_DEVCON_CacheFlush();
```

Returns

None.

Description

This function flushes both instruction and data caches. Invalidate entire instruction cache; writes back and invalidates the entire data cache.

Remarks

Simplest way to synchronize caches with memory, but not necessarily the most efficient.

Preconditions

None.

Example

```
SYS_DEVCON_CacheFlush();
```

Function

```
void SYS_DEVCON_CacheFlush(void)
```

SYS_DEVCON_CacheInit Function

Initializes the L1 cache.

File

[sys_devcon.h](#)

C

```
void SYS_DEVCON_CacheInit(SYS_CACHE_COHERENCY cacheCoherency);
```

Returns

None.

Description

This function initializes both instruction and data caches. Invalidates all entries and zeros all tags. Sets coherency attribute for kseg0.

Remarks

Use with caution. Invalidates all cache lines without writing data back to memory. Any dirty data in the cache will be lost.

Preconditions

None.

Example

```
SYS_DEVCON_CacheInit(SYS_CACHE_WRITEBACK_WRITEALLOCATE);
```

Parameters

Parameters	Description
cacheCoherency	The desired kseg0 coherency attribute.

Function

```
void SYS_DEVCON_CacheInit( SYS_CACHE_COHERENCY cacheCoherency )
```

SYS_DEVCON_CacheSync Function

Synchronizes the instruction and data caches.

File

[sys_devcon.h](#)

C

```
void SYS_DEVCON_CacheSync(uint32_t addr, size_t len);
```

Returns

None.

Description

This function synchronizes the instruction and data caches. Used when modifying the instruction stream (breakpoints, self-modifying code, relocating executable code to RAM). Flushes an address range from the data cache and invalidates that same range from the instruction cache.

Remarks

None.

Preconditions

None.

Example

```
SYS_DEVCON_CacheSync(&myCode, myCodeSize);
```

Parameters

Parameters	Description
addr	Starting address of code block.
len	Size of code block.

Function

```
void SYS_DEVCON_CacheSync( uint32_t addr, size_t len )
```

SYS_DEVCON_DataCacheClean Function

Writes back and invalidates an address range in the data cache.

File

[sys_devcon.h](#)

C

```
void SYS_DEVCON_DataCacheClean(uint32_t addr, size_t len);
```

Returns

None.

Description

This function writes back and invalidates an address range in the data cache.

Remarks

None.

Preconditions

None.

Example

```
SYS_DEVCON_DataCacheClean(&myData, sizeof(myData));
```

Parameters

Parameters	Description
addr	Starting address of data block.
len	Size of data block.

Function

```
void SYS_DEVCON_DataCacheClean( uint32_t addr, size_t len )
```

SYS_DEVCON_DataCacheFlush Function

Flushes the L1 data cache.

File

[sys_devcon.h](#)

C

```
void SYS_DEVCON_DataCacheFlush();
```

Returns

None.

Description

This function writes back and invalidates the entire data cache.

Remarks

None.

Preconditions

None.

Example

```
SYS_DEVCON_DataCacheFlush();
```

Function

```
void SYS_DEVCON_DataCacheFlush( void )
```

SYS_DEVCON_DataCacheInvalidate Function

Invalidates an address range in the data cache.

File

[sys_devcon.h](#)

C

```
void SYS_DEVCON_DataCacheInvalidate(uint32_t addr, size_t len);
```

Returns

None.

Description

This function invalidates an address range in the data cache.

Remarks

Use caution. Any dirty data in the cache will be lost.

Preconditions

None.

Example

```
SYS_DEVCON_DataCacheInvalidate(&myData, sizeof(myData));
```

Parameters

Parameters	Description
addr	Starting address of data block.
len	Size of data block.

Function

```
void SYS_DEVCON_DataCacheInvalidate( uint32_t addr, size_t len )
```

SYS_DEVCON_DataCacheLock Function

Fetches and locks a block of data in the data cache.

File

[sys_devcon.h](#)

C

```
void SYS_DEVCON_DataCacheLock(uint32_t addr, size_t len);
```

Returns

None.

Description

This function fetches and locks a block of data in the data cache.

Remarks

Global flush functions will invalidate and unlock any cache lines locked with this function.

Preconditions

None.

Example

```
SYS_DEVCON_DataCacheLock(&myData, sizeof(myData));
```

Parameters

Parameters	Description
addr	Starting address of data block.
len	Size of data block.

Function

```
void SYS_DEVCON_DataCacheLock( uint32_t addr, size_t len )
```

SYS_DEVCON_DataCacheAssociativityGet Function

Returns the number of ways in the data cache.

File

[sys_devcon.h](#)

C

```
size_t SYS_DEVCON_DataCacheAssociativityGet();
```

Returns

The number of ways in the data cache.

Description

This function returns the number of ways in the data cache.

Remarks

None.

Preconditions

None.

Example

```
unsigned int size;  
size = SYS_DEVCON_DataCacheAssociativityGet();
```

Function

```
size_t SYS_DEVCON_DataCacheAssociativityGet(void)
```

SYS_DEVCON_DataCacheLineSizeGet Function

Returns the data cache line size.

File

[sys_devcon.h](#)

C

```
size_t SYS_DEVCON_DataCacheLineSizeGet();
```

Returns

The number of bytes per line in the data cache.

Description

This function returns the data cache line size.

Remarks

None.

Preconditions

None.

Example

```
unsigned int size;
```

```
size = SYS_DEVCON_DataCacheLineSizeGet();
```

Function

```
size_t SYS_DEVCON_DataCacheLineSizeGet(void)
```

SYS_DEVCON_DataCacheLinesPerWayGet Function

Returns the number of lines per way in the data cache.

File

[sys_devcon.h](#)

C

```
size_t SYS_DEVCON_DataCacheLinesPerWayGet();
```

Returns

The number of lines per way in the data cache.

Description

This function returns the number of lines per way in the data cache.

Remarks

None.

Preconditions

None.

Example

```
unsigned int size;  
size = SYS_DEVCON_DataCacheLinesPerWayGet();
```

Function

```
size_t SYS_DEVCON_DataCacheLinesPerWayGet(void)
```

SYS_DEVCON_DataCacheSizeGet Function

Returns the total number of bytes in the data cache.

File

[sys_devcon.h](#)

C

```
size_t SYS_DEVCON_DataCacheSizeGet();
```

Returns

The total number of bytes in the data cache.

Description

This function returns the total number of bytes in the data cache.

Remarks

None.

Preconditions

None.

Example

```
unsigned int size;  
size = SYS_DEVCON_DataCacheSizeGet();
```

Function

```
size_t SYS_DEVCON_DataCacheSizeGet(void)
```

SYS_DEVCON_InstructionCacheFlush Function

Flushes (invalidates) the L1 instruction cache.

File

[sys_devcon.h](#)

C

```
void SYS_DEVCON_InstructionCacheFlush();
```

Returns

None.

Description

This function invalidates the entire instruction cache.

Remarks

None.

Preconditions

None.

Example

```
SYS_DEVCON_InstructionCacheFlush();
```

Function

```
void SYS_DEVCON_InstructionCacheFlush( void )
```

SYS_DEVCON_InstructionCacheInvalidate Function

Invalidates an address range in the instruction cache.

File

[sys_devcon.h](#)

C

```
void SYS_DEVCON_InstructionCacheInvalidate(uint32_t addr, size_t len);
```

Returns

None.

Description

This function invalidates an address range in the instruction cache.

Remarks

None.

Preconditions

None.

Example

```
SYS_DEVCON_InstructionCacheInvalidate(&myCode, myCodeSize);
```

Parameters

Parameters	Description
addr	Starting address of code block.
len	Size of code block.

Function

```
void SYS_DEVCON_InstructionCacheInvalidate( uint32_t addr, size_t len )
```

SYS_DEVCON_InstructionCacheLock Function

Fetches and locks a block of instructions in the instruction cache.

File

[sys_devcon.h](#)

C

```
void SYS_DEVCON_InstructionCacheLock(uint32_t addr, size_t len);
```

Returns

None.

Description

This function fetches and locks a block of instructions in the instruction cache.

Remarks

Global flush functions will invalidate and unlock any cache lines locked with this function.

Preconditions

None.

Example

```
SYS_DEVCON_InstructionCacheLock(&myCode, myCodeSize);
```

Parameters

Parameters	Description
addr	Starting address of code block.
len	Size of code block.

Function

```
void SYS_DEVCON_InstructionCacheLock( uint32_t addr, size_t len )
```

SYS_DEVCON_InstructionCacheAssociativityGet Function

Returns the number of ways in the instruction cache.

File

[sys_devcon.h](#)

C

```
size_t SYS_DEVCON_InstructionCacheAssociativityGet();
```

Returns

The number of ways in the instruction cache.

Description

This function returns the number of ways in the instruction cache.

Remarks

None.

Preconditions

None.

Example

```
unsigned int size;  
size = SYS_DEVCON_InstructionCacheAssociativityGet();
```

Function

```
size_t SYS_DEVCON_InstructionCacheAssociativityGet(void)
```

SYS_DEVCON_InstructionCacheLineSizeGet Function

Returns the instruction cache line size.

File

[sys_devcon.h](#)

C

```
size_t SYS_DEVCON_InstructionCacheLineSizeGet();
```

Returns

The number of bytes per line in the instruction cache.

Description

This function returns the instruction cache line size.

Remarks

None.

Preconditions

None.

Example

```
unsigned int size;  
size = SYS_DEVCON_InstructionCacheLineSizeGet();
```

Function

```
size_t SYS_DEVCON_InstructionCacheLineSizeGet(void)
```

SYS_DEVCON_InstructionCacheLinesPerWayGet Function

Returns the number of lines per way in the instruction cache.

File

[sys_devcon.h](#)

C

```
size_t SYS_DEVCON_InstructionCacheLinesPerWayGet();
```

Returns

The number of lines per way in the instruction cache.

Description

This function returns the number of lines per way in the instruction cache.

Remarks

None.

Preconditions

None.

Example

```
unsigned int size;  
size = SYS_DEVCON_InstructionCacheLinesPerWayGet();
```

Function

```
size_t SYS_DEVCON_InstructionCacheLinesPerWayGet(void)
```

SYS_DEVCON_InstructionCacheSizeGet Function

Returns the total number of bytes in the instruction cache.

File

[sys_devcon.h](#)

C

```
size_t SYS_DEVCON_InstructionCacheSizeGet();
```

Returns

The total number of bytes in the instruction cache.

Description

This function returns the total number of bytes in the instruction cache.

Remarks

None.

Preconditions

None.

Example

```
unsigned int size;
size = SYS_DEVCON_InstructionCacheSizeGet();
```

Function

```
size_t SYS_DEVCON_InstructionCacheSizeGet(void)
```

d) Data Types and Constants**SYS_DEVCON_HANDLE Type**

Identifies a particular registered event instance.

File

[sys_devcon.h](#)

C

```
typedef int8_t SYS_DEVCON_HANDLE;
```

Description

SYS DEVCON Handle. This event handle identifies a registered instance of an event. Every time the application that tries to access the parameters with respect to a particular event, shall used this event handle to refer to that event.

Remarks

None.

SYS_DEVCON_INIT Structure

Identifies the system device control initialize structure.

File

[sys_devcon.h](#)

C

```
typedef struct {
    SYS_MODULE_INIT moduleInit;
} SYS_DEVCON_INIT;
```

Members

Members	Description
SYS_MODULE_INIT moduleInit;	System module initialization

Description

SYS_DEVCON Initialize structure

This structure identifies the system device control initialize structure.

Remarks

None.

SYS_DEVCON_INDEX_0 Macro

Device Control System Service index definitions.

File

[sys_devcon.h](#)

C

```
#define SYS_DEVCON_INDEX_0 0
```

Description

SYS Device Control Module Index Numbers

These constants provide Device Control System Service index definitions.

Remarks

These constants should be used in place of hard-coded numeric literals.

SYS_CACHE_COHERENCY Enumeration

System L1 cache coherency settings.

File

[sys_devcon.h](#)

C

```
typedef enum {  
    SYS_CACHE_WRITETHROUGH_NO_WRITEALLOCATE = 0,  
    SYS_CACHE_WRITETHROUGH_WRITEALLOCATE = 1,  
    SYS_CACHE_DISABLE = 2,  
    SYS_CACHE_WRITEBACK_WRITEALLOCATE = 3  
} SYS_CACHE_COHERENCY;
```

Description

SYS_CACHE_COHERENCY enumeration

This enumeration defines the supported system L1 cache coherency settings.

Remarks

Used to read or write cache coherency policy for kseg0.

SYS_POWER_MODE Enumeration

System power mode settings.

File

[sys_devcon.h](#)

C

```
typedef enum {  
    SYS_POWER_MODE_IDLE,  
    SYS_POWER_MODE_SLEEP  
} SYS_POWER_MODE;
```

Description

SYS_POWER_MODE enumeration

This enumeration defines the supported system power mode settings.

Remarks

None.

Files

Files

Name	Description
sys_devcon.h	Device Control (DEVCON) System Service interface definition.

Description

This section lists the source and header files used by the library.































sys_devcon.h





Device Control (DEVCON) System Service interface definition.

Enumerations

	Name	Description
	SYS_CACHE_COHERENCY	System L1 cache coherency settings.
	SYS_POWER_MODE	System power mode settings.

Functions

	Name	Description
	SYS_DEVCON_CacheClean	Writes back and invalidates an address range in either cache.
	SYS_DEVCON_CacheCoherencyGet	Returns the current cache coherency attribute for kseg0.
	SYS_DEVCON_CacheCoherencySet	Sets the cache coherency attribute for kseg0.
	SYS_DEVCON_CacheFlush	Flushes the L1 cache.
	SYS_DEVCON_CacheInit	Initializes the L1 cache.
	SYS_DEVCON_CacheSync	Synchronizes the instruction and data caches.
	SYS_DEVCON_DataCacheAssociativityGet	Returns the number of ways in the data cache.
	SYS_DEVCON_DataCacheClean	Writes back and invalidates an address range in the data cache.
	SYS_DEVCON_DataCacheFlush	Flushes the L1 data cache.
	SYS_DEVCON_DataCacheInvalidate	Invalidates an address range in the data cache.
	SYS_DEVCON_DataCacheLineSizeGet	Returns the data cache line size.
	SYS_DEVCON_DataCacheLinesPerWayGet	Returns the number of lines per way in the data cache.
	SYS_DEVCON_DataCacheLock	Fetches and locks a block of data in the data cache.
	SYS_DEVCON_DataCacheSizeGet	Returns the total number of bytes in the data cache.
	SYS_DEVCON_Deinitialize	Deinitializes the specific module instance of the DEVCON module
	SYS_DEVCON_Initialize	Initializes data for the instance of the Device Control module and opens the specific module instance.
	SYS_DEVCON_InstructionCacheAssociativityGet	Returns the number of ways in the instruction cache.
	SYS_DEVCON_InstructionCacheFlush	Flushes (invalidates) the L1 instruction cache.
	SYS_DEVCON_InstructionCacheInvalidate	Invalidates an address range in the instruction cache.
	SYS_DEVCON_InstructionCacheLineSizeGet	Returns the instruction cache line size.
	SYS_DEVCON_InstructionCacheLinesPerWayGet	Returns the number of lines per way in the instruction cache.
	SYS_DEVCON_InstructionCacheLock	Fetches and locks a block of instructions in the instruction cache.
	SYS_DEVCON_InstructionCacheSizeGet	Returns the total number of bytes in the instruction cache.
	SYS_DEVCON_JTAGDisable	Disables the JTAG port on the device.
	SYS_DEVCON_JTAGEnable	Enables the JTAG port on the device.
	SYS_DEVCON_PerformanceConfig	Configures the PFM wait states and Prefetch Cache module for maximum performance.
	SYS_DEVCON_PowerModeEnter	Puts the device in a low-power state.
	SYS_DEVCON_Reinitialize	Reinitializes and refreshes the hardware for the instance of the Device Control module.
	SYS_DEVCON_Status	Returns status of the specific instance of the Device Control module.
	SYS_DEVCON_SystemLock	Performs a system lock sequence by writing to the SYSKEY register.

	SYS_DEVCON_SystemUnlock	Performs a system unlock sequence by writing to the SYSKEY register.
	SYS_DEVCON_Tasks	Maintains the system Device Control state machine.
	SYS_DEVCON_TraceDisable	Disables the trace output port on the device.
	SYS_DEVCON_TraceEnable	Enables the trace output port on the device.

Macros

	Name	Description
	SYS_DEVCON_INDEX_0	Device Control System Service index definitions.

Structures

	Name	Description
	SYS_DEVCON_INIT	Identifies the system device control initialize structure.

Types

	Name	Description
	SYS_DEVCON_HANDLE	Identifies a particular registered event instance.

Description

Device Control System Service Interface Definition

This file contains the interface definition for the Device Control System Service. It provides a way to interact with the Device Control subsystem to manage the device control requests supported by the system.

File Name

sys_devcon.h

Company

Microchip Technology Inc.

Direct Memory Access (DMA) System Service Library

This section describes the Direct Memory Access (DMA) System Service Library.

Introduction

This library provides an interface to interact with the DMA subsystem to control and manage the data transfer between different peripherals and/or memory without intervention from the CPU.

Description

The Direct Memory Access (DMA) controller is a bus master module that is useful for data transfers between different peripherals without intervention from the CPU. The source and destination of a DMA transfer can be any of the memory-mapped modules. For example, memory, or one of the Peripheral Bus (PBUS) devices such as the SPI, UART, and so on.

Using the Library

This topic describes the basic architecture of the DMA System Service Library and provides information and examples on its use.

Description

Interface Header File: [sys_dma.h](#)

The interface to the DMA System Service library is defined in the [sys_dma.h](#) header file. This file is included by the [system.h](#) file. Any C language source (.c) file that uses the DMA System Service Library should include [system.h](#).

Please refer to the What is MPLAB Harmony? section for how the DMA System Service interacts with the framework.

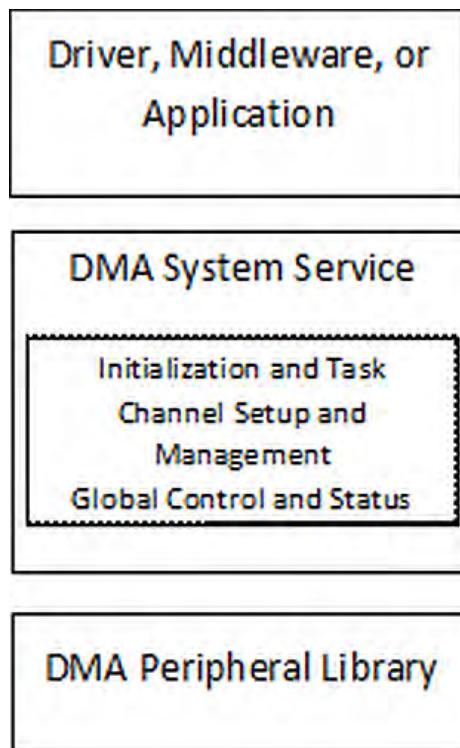
Abstraction Model

This model explains how the system interfaces with the DMA System Service and the application.

Description

This library provides an abstraction of the DMA subsystem that is used by device drivers, middleware libraries and applications to transmit and receive data.

DMA Software Abstraction Block Diagram



DMA System Service

The DMA system services provide support for initializing the DMA controller, managing the transfer state machine, setup/management of

channels, and global module control and status management.

Initialization and Tasks

Each software module (device driver, middleware, or application) that needs to use the DMA for data transfer must enable the DMA controller. This is normally done by calling the initialization routine of the DMA subsystem in the module's initialization routine, which is called by the [SYS_Initialize](#) service. The initialization routine returns a DMA module object, which should be used as a parameter in the call to Task routines.

The Task routines implement the data transfer state machine for synchronous and asynchronous data transfer operations. If Asynchronous (interrupt) mode of operation is desired, the Task routine([SYS_DMA_TasksISR](#)) should be called from the respective channel ISR. If Synchronous mode of operation if desired, the Task routine([SYS_DMA_Tasks](#)) should be called from the [SYS_Tasks](#) function.

Channel Setup and Management

Any module that needs to use the DMA system service must request for channel allocation. An allocated channel is used to setup the channel parameters like the mode of operation(Basic, CRC, chaining etc). Setup the transfer trigger types(Synchronous/Asynchronous). Add a transfer by Setting up the source, destination address and transfer sizes. The DMA transfer starts either forcefully or based on events according to the setting. The channel status events are used to manage the data transfer.

Global Control and Status Management

Provides for control and status of the DMA module. The user can suspend a DMA operation or alternatively resume an already suspended operation. The status of last DMA operation can also be retrieved.

Library Overview

Please refer to the [System Service Introduction](#) for a detailed description of MPLAB Harmony system services.

The library interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the DMA module.

Library Interface Section	Description
Initialization and Task Functions	These functions initialize and enable the DMA subsystem and manage the task State machine.
Channel Setup and Management Functions	These DMA Channel and Setup Management functions enable: <ul style="list-style-type: none"> • Channel Allocation and Release • Setting up of a channel for basic and advanced modes of operations • Channel Enable/Disable • Adding a data transfer • Synchronous data transfer start/abort • Asynchronous data transfer start/abort • Channel Busy status • Channel Computed CRC • Setting the transfer event trigger callback
Global Control and Status Functions	These DMA functions enable global control and status.

How the Library Works

Initialization and Tasks

Describes the functions that can be used for DMA initialization, and provides examples of their usage.

Description

Initialization Function Usage

The DMA subsystem is initialized by calling the initialization routine. The routine also enables the DMA module.

Function Name: [SYS_DMA_Initialize](#)

Example

//To Enable the Stop in Idle mode feature

```
SYS_MODULE_OBJ objectHandle;
SYS_DMA_INIT initParam;

initParam.sidl = SYS_DMA_SIDL_ENABLE;

objectHandle = SYS_DMA_Initialize((SYS_MODULE_INIT*)&initParam);
```

```

if(SYS_MODULE_OBJ_INVALID == objectHandle)
{
    // Handle error
}

```

Task Function Usage

The task routine implements the data transfer state machine and returns a callback on the completion, abortion or error in a data transfer.

Function Name: [SYS_DMA_Tasks](#)

Example

```

// 'object' Returned from SYS_DMA_Initialize
while (true)
{
    SYS_DMA_Tasks (object);
    // Do other tasks
}

```



Note: The [SYS_DMA_Tasks](#) function should be used in Synchronous (polling) mode only. This function should not be called from an ISR.

Function Name: [SYS_DMA_TasksError](#)

Example

```

// 'object' Returned from SYS_DMA_Initialize
while (true)
{
    SYS_DMA_TasksError (object);
    // Do other tasks
}

```



Note: The [SYS_DMA_TasksError](#) function should be used in Synchronous (polling) mode only. This function should not be called from an ISR.

Function Name: [SYS_DMA_TasksISR](#)

Example

```

// 'object' Returned from SYS_DMA_Initialize
// Channel 3 is setup for receiving data by USART peripheral in interrupt mode

// Following is the DMA Channel 3 ISR

void __ISR(_DMA3_VECTOR,ipl5) _InterruptHandler_BT_USART_RX_DMA_CHANNEL(void)
{
    SYS_DMA_TasksISR(object, DMA_CHANNEL_3);
}

```



Note: The [SYS_DMA_TasksISR](#) function should be used in Asynchronous (interrupt) mode only. This function should not be called from the [SYS_Tasks](#) function.

Function Name: [SYS_DMA_TasksErrorISR](#)

Example

```

// 'object' Returned from SYS_DMA_Initialize
// Channel 3 is setup for receiving data by USART peripheral in interrupt mode

// Following is the DMA Channel 3 ISR

void __ISR(_DMA3_VECTOR,ipl5) _InterruptHandler_BT_USART_RX_DMA_CHANNEL(void)
{
    SYS_DMA_TasksErrorISR(object, DMA_CHANNEL_3);
}

```



Note: The [SYS_DMA_TasksErrorISR](#) function should be used in Asynchronous (interrupt) mode only. This function should not be called from the [SYS_Tasks](#) function.

Channel Setup and Management

Describes the functions that can be used for DMA channel setup and management, and provides examples of their usage.

Description

Channel Allocation and Release Functions

Channel Allocate and Release functions allocate/release a particular channel from the available channels on the particular device.

Channel Allocation Function

Channel Allocate function takes a parameter specifying the requested channel number. If the requested channel is available the function allocates the channel and returns a channel handle. If the user is not particular about any specific channel, the user can specify `DMA_CHANNEL_ANY` enumerator values. When `DMA_CHANNEL_ANY` values are specified an available channel is allocated and a channel handle is returned. The function returns an invalid channel handle `SYS_DMA_CHANNEL_HANDLE_INVALID` when the requested channel is not available for allocation.

The valid channel handle returned by this function must be used in all subsequent DMA channel function calls.

Function Name: `SYS_DMA_ChannelAllocate`

Example 1

```
/* The following examples requests for allocation of a channel handle with Channel number 2 */
SYS_DMA_CHANNEL_HANDLE  handle
DMA_CHANNEL             channel;
channel = DMA_CHANNEL_2;
handle  = SYS_DMA_ChannelAllocate(channel);
```

Example 2

```
/* The following example requests for allocation of a channel handle with no choice of channel number */
SYS_DMA_CHANNEL_HANDLE  handle
DMA_CHANNEL             channel;
channel = DMA_CHANNEL_ANY;
handle  = SYS_DMA_ChannelAllocate(channel);
```

Precondition: DMA should be initialized by calling `SYS_DMA_Initialize`.

Channel Release Function

Channel Release function Deallocates and frees the DMA channel specified by the channel handle.

Function Name: `SYS_DMA_ChannelRelease`

Example

```
/* The following example requests for release of a channel handle with Channel number 2 */
DMA_CHANNEL             channel;
SYS_DMA_CHANNEL_HANDLE  handle;
channel = DMA_CHANNEL_2;
handle  = SYS_DMA_ChannelAllocate(channel);
SYS_DMA_ChannelRelease(handle);
```

Precondition: DMA should be initialized by calling `SYS_DMA_Initialize`.

Channel Setup Functions

The DMA subsystem supports the following modes of operations. The setting up of these modes of operation can be done by the following functions.

- Basic Transfer Mode
- Pattern Match Abort Mode
- Channel Chaining Mode
- Channel Auto Enable Mode
- CRC Mode



Note: The operation modes are not mutually exclusive.

General Channel Setup Function

This function sets up the channel for the supported operating modes. The function does the following:

- Sets up the channel priority
- Enables the specified mode
- Sets up the DMA asynchronous transfer mode

If the DMA channel transfer is intended to be synchronous, the parameter 'eventSrc' (asynchronous trigger source) can be specified as 'DMA_TRIGGER_SOURCE_NONE'. When the channel trigger source is specified as 'DMA_TRIGGER_SOURCE_NONE', The DMA channel transfer needs to be forcefully started by calling the respective function.

**Notes:**

1. Enabling of the available operation mode is not mutually exclusive. More than one operation mode can be enabled by bitwise ORing the operating mode enable parameter.
2. To setup the specific features of the supported operation modes the corresponding function needs to be called after calling this function.

Function Name: [SYS_DMA_ChannelSetup](#)

Example

```
/* Configure channel number, priority and enables basic and CRC mode */
SYS_DMA_CHANNEL_HANDLE  handle;
DMA_CHANNEL             channel;
SYS_DMA_CHANNEL_OP_MODE modeEnable;
DMA_TRIGGER_SOURCE      eventSrc;
channel                 = DMA_CHANNEL_2;
modeEnable              = (SYS_DMA_CHANNEL_OP_MODE_BASIC | SYS_DMA_CHANNEL_OP_MODE_CRC);
eventSrc                = DMA_TRIGGER_USART_1_TRANSMIT;
handle                  = SYS_DMA_ChannelAllocate(channel);
SYS_DMA_ChannelSetup(handle, modeEnable, eventSrc);
```

Precondition: DMA should be initialized by calling [SYS_DMA_Initialize](#).

Pattern Match Abort Mode Function

This function sets up the termination of DMA operation when the specified pattern is matched. Additionally on supported devices, the function also sets up the ignoring of part of a pattern (8-bit) from match abort pattern (16-bit).

Before calling this function the pattern match termination mode must have been enabled by calling the general channel setup function

Function Name: [SYS_DMA_ChannelSetupMatchAbortMode](#)

Example 1

```
/* The following code is for a device with 8-bit pattern value and no support for pattern match ignore feature */
SYS_DMA_CHANNEL_HANDLE  handle;
DMA_CHANNEL             channel;
uint16_t                pattern;
DMA_PATTERN_LENGTH      length;
SYS_DMA_CHANNEL_IGNORE_MATCH ignore;
uint8_t                 ignorePattern;
channel                 = DMA_CHANNEL_2;
priority                = DMA_CHANNEL_PRIORITY_1;
pattern                 = 0x00; //Stop transfer on detection of a NULL character
length                  = DMA_PATTERN_LENGTH_NONE;
ignore                  = SYS_DMA_CHANNEL_IGNORE_MATCH_DISABLE;
ignorePattern           = 0;
handle                  = SYS_DMA_ChannelAllocate(channel);
SYS_DMA_ChannelSetupMatchAbortMode(handle, pattern, length, ignoreEnable, ignorePattern);
```

Example 2

```
/* The following code is for a device with 16-bit pattern value and support for pattern match ignore feature */
SYS_DMA_CHANNEL_HANDLE  handle;
DMA_CHANNEL             channel;
uint16_t                pattern;
DMA_PATTERN_LENGTH      length;
SYS_DMA_CHANNEL_IGNORE_MATCH ignore;
uint8_t                 ignorePattern;
priority                = DMA_CHANNEL_PRIORITY_1;
channel                 = DMA_CHANNEL_2;
pattern                 = 0x0D0A; //Stop transfer on detection of '\r\n'
length                  = DMA_PATTERN_MATCH_LENGTH_2BYTES;
ignore                  = SYS_DMA_CHANNEL_IGNORE_MATCH_ENABLE;
ignorePattern           = 0x00; \\ Any null character between the termination pattern '\r' and '\n' is ignored.
handle                  = SYS_DMA_ChannelAllocate(channel);
SYS_DMA_ChannelSetupMatchAbortMode(handle, pattern, length, ignore, ignorePattern);
```

Preconditions:

1. DMA should be initialized by calling [SYS_DMA_Initialize](#).
2. Channel should be setup and pattern match mode enabled by calling [SYS_DMA_ChannelSetup](#).

CRC Operation Mode Setup Function

This function sets up the CRC computation features of the channel. Before calling this function the CRC mode must have been enabled by calling the general channel setup function

Function Name: `SYS_DMA_ChannelSetupCRCMode`

Example

```
/* DMA calculation using the CRC background mode */
SYS_DMA_CHANNEL_HANDLE handle;
DMA_CHANNEL channel;
SYS_DMA_CHANNEL_OPERATION_MODE_CRC crc;
channel = DMA_CHANNEL_2;
crc.type = DMA_CRC_LFSR;
crc.mode = SYS_DMA_CHANNEL_CRC_MODE_BACKGROUND;
crc.polyLength = 16;
crc.bitOrder = DMA_CRC_BIT_ORDER_LSB;
crc.byteOrder = DMA_CRC_BYTEORDER_NO_SWAPPING;
crc.writeOrder = SYS_DMA_CRC_WRITE_ORDER_MAINTAIN;
crc.data = 0xFFFF;
crc.xorBitMask = 0x1021;
handle = SYS_DMA_ChannelAllocate(channel);
SYS_DMA_ChannelCRCSet(handle, crc);
```

Preconditions:

1. DMA should be initialized by calling [SYS_DMA_Initialize](#).
2. Channel should be setup and CRC mode enabled by calling [SYS_DMA_ChannelSetup](#).

Channel Enable/Disable Functions

The Enable/Disable functions allow to enable/disable a channel on the run.



Note: When a data transfer is added by calling [SYS_DMA_ChannelTransferAdd](#), the channel is automatically enabled.

Function Name: [SYS_DMA_ChannelEnable](#)

Example

```
SYS_DMA_CHANNEL_HANDLE handle;
DMA_CHANNEL channel;
channel = DMA_CHANNEL_2;
handle = SYS_DMA_ChannelAllocate(channel);
SYS_DMA_ChannelEnable(handle);
```

Precondition: DMA should be initialized by calling [SYS_DMA_Initialize](#).

Function Name: [SYS_DMA_ChannelDisable](#)

Example

```
SYS_DMA_CHANNEL_HANDLE handle;
DMA_CHANNEL channel;
channel = DMA_CHANNEL_2;
handle = SYS_DMA_ChannelAllocate(channel);
SYS_DMA_ChannelDisable(handle);
```

Precondition: DMA should be initialized by calling [SYS_DMA_Initialize](#).

Channel Data Transfer Function

This function adds a data transfer to a DMA channel and Enables the channel to start data transfer. The source and the destination addresses, source and destination lengths, The number of bytes transferred per cell event are set and the channel is enabled to start the data transfer.

Function Name: [SYS_DMA_ChannelTransferAdd](#)

Example

```
/* Add 10 bytes of data transfer to UART */
SYS_DMA_CHANNEL_HANDLE handle;
uint8_t buf[10];
void *srcAddr;
void *destAddr;
size_t srcSize;
size_t destSize;
size_t cellSize;
DMA_CHANNEL channel;
channel = DMA_CHANNEL_2;
srcAddr = (uint8_t *) buf;
srcSize = 10;
destAddr = (uint8_t*) &U2TXREG; //Uart 2 TX register is the DMA destination
destSize = 1;
cellSize = 1;
handle = SYS_DMA_ChannelAllocate(channel);
```

```
SYS_DMA_ChannelTransferAdd(handle,srcAddr,srcSize,destAddr,destSize,cellSize);
```

Preconditions:

1. DMA should be initialized by calling [SYS_DMA_Initialize](#).
2. Channel should be setup and enabled by calling [SYS_DMA_ChannelSetup](#).

Synchronous Data Transfer Start/Abort Functions

These functions allows to force start/abort data transfer on the selected channel.



Note: The DMA should have been set up, transfer added before calling these functions.

Function Name: [SYS_DMA_ChannelForceStart](#)

Example

```
SYS_DMA_CHANNEL_HANDLE  handle;
DMA_CHANNEL             channel;
channel = DMA_CHANNEL_2;
handle  = SYS_DMA_ChannelAllocate(channel);
SYS_DMA_ChannelForceStart(handle);
```

Function Name: [SYS_DMA_ChannelForceAbort](#)

Example

```
SYS_DMA_CHANNEL_HANDLE  handle;
DMA_CHANNEL             channel;
channel = DMA_CHANNEL_2;
handle  = SYS_DMA_ChannelAllocate(channel);
SYS_DMA_ChannelForceAbort(handle);
```

Asynchronous Data Transfer Abort Function

This function sets an event source and enables cell transfer abort event for the same for the selected channel.



Note: Asynchronous data transfer is started in general channel setup function. A channel is setup for asynchronous data transfer by default when the appropriate trigger source is specified in general channel setup function call.

Function Name: [SYS_DMA_ChannelAbortEventSet](#)

Example

```
SYS_DMA_CHANNEL_HANDLE  handle;
DMA_CHANNEL             channel;
DMA_TRIGGER_SOURCE      eventSrc;
channel = DMA_CHANNEL_2;
eventSrc = DMA_TRIGGER_CTMU;
handle  = SYS_DMA_ChannelAllocate(channel);
SYS_DMA_ChannelAbortEventSet(handle, eventSrc);
```

Preconditions:

1. DMA should be initialized by calling [SYS_DMA_Initialize](#).
2. Channel should be setup and enabled by calling [SYS_DMA_ChannelSetup](#).

Channel Busy Status Function

This function gets the busy status of the selected channel.

Function Name: [SYS_DMA_ChannelIsBusy](#)

Example

```
SYS_DMA_CHANNEL_HANDLE  handle;
DMA_CHANNEL             channel;
bool                    busyStat;
channel = DMA_CHANNEL_2;
handle  = SYS_DMA_ChannelAllocate(channel);
busyStat = SYS_DMA_ChannelIsBusy(handle);
```

Precondition: DMA should be initialized by calling [SYS_DMA_Initialize](#).

Channel Computed CRC Function

This function gets the computed CRC.



Note: The CRC generator must have been previously set up by calling the CRC mode setup function.

To get the computed CRC value this function must be called after the block transfer completion event.

Function Name: `SYS_DMA_ChannelGetCRC`

Example

```
SYS_DMA_CHANNEL_HANDLE handle;
DMA_CHANNEL channel;
uint32_t computedCRC;
channel = DMA_CHANNEL_2;
handle = SYS_DMA_ChannelAllocate(channel);
computedCRC = SYS_DMA_ChannelCRCSet();
```

Precondition: DMA should be initialized by calling [SYS_DMA_Initialize](#).

Channel Transfer Event Handler Set Function

This function allows to set an event handler for the transfer complete, abort or error events

Function Name: [SYS_DMA_ChannelTransferEventHandlerSet](#)

Example

```
SYS_DMA_CHANNEL_HANDLE handle;
DMA_CHANNEL channel;
MY_APP_OBJ myAppObj; //Application specific object
channel = DMA_CHANNEL_2;
handle = SYS_DMA_ChannelAllocate(channel);
SYS_DMA_ChannelTransferEventHandlerSet(handle, APP_DMASYSSTransferEventHandler, (uintptr_t)&myAppObj);
```

// Event Processing Technique. Event is received when the transfer is processed.

```
void APP_DMASYSSTransferEventHandler(SYS_DMA_TRANSFER_EVENT event, SYS_DMA_CHANNEL_HANDLE handle,
uintptr_t contextHandle)
{
    switch(event)
    {
        case SYS_DMA_TRANSFER_EVENT_COMPLETE:
            // This means the data was transferred.
            break;
        case SYS_DMA_TRANSFER_EVENT_ERROR:
            // Error handling here.
            break;
        default:
            break;
    }
}
```

Precondition: DMA should be initialized by calling [SYS_DMA_Initialize](#).

Channel Error Get Function

This function returns the error associated with the channel access.

Function Name: [SYS_DMA_ChannelErrorGet](#)

Example

```
SYS_DMA_CHANNEL_HANDLE handle;
DMA_CHANNEL channel;
channel = DMA_CHANNEL_2;
handle = SYS_DMA_ChannelAllocate(channel);
// Do Channel setup and Transfer Add.
// In the Even Handler Check if there was an Error
if(SYS_DMA_ERROR_ADDRESS_ERROR == SYS_DMA_ChannelErrorGet(handle))
{
    // There was an address error.
    // Do error handling here.
}
```

Precondition: DMA should be initialized by calling [SYS_DMA_Initialize](#).

Global Control and Status

Global Control and Status Function Usage:

Suspend/Resume Functions

These functions allows an enabled DMA module to suspend DMA operations. Operation of the suspended module can also be resumed using these functions.

Function Name: [SYS_DMA_Suspend](#)

Example 1

```
SYS_DMA_Suspend();
```

Precondition: DMA should be initialized by calling [SYS_DMA_Initialize](#).

Function Name: [SYS_DMA_Resume](#)

Example 2

```
SYS_DMA_Resume();
```

Precondition: DMA should be initialized by calling [SYS_DMA_Initialize](#).

Busy Status Function

This function gets the busy status of DMA module.

Function Name: [SYS_DMA_IsBusy](#)

Example

```
bool busyStat;
```

```
busyStat = SYS_DMA_IsBusy();
```

Precondition: DMA should be initialized by calling [SYS_DMA_Initialize](#).

Memory to Memory Transfer

Provides a code example for a memory to memory data transfer.

Description

The following code is a typical usage example of using the DMA System Service for a memory to memory data transfer.

```
/* Destination Memory Address pointer */
uint8_t          *pDmaSrc;
/* Source Memory Address pointer */
uint8_t          *pDmaDst;
/* Transfer Size */
size_t           txferSize;

/* Maximum Transfer Size */
#define MAX_TRANSFER_SIZE      100
/* The data pattern to be transferred*/
#define SOURCE_TRANSFER_PATTERN 'W'

/*****
The below routine sets up a memory to memory data transfer. It also
registers an event handler which is called when the data transfer
completes */
void SYS_DMA_TEST_Memory2Memory(void)
{
    uint8_t          *pSrcTemp;
    uint32_t          index;
    SYS_DMA_INIT      dmaInit;
    SYS_DMA_CHANNEL_HANDLE channelHandle;

    /* Initializing source and destination variables */
    pDmaSrc           = (uint8_t *) NULL;
    pDmaDst           = (uint8_t *) NULL;
    pSrcTemp          = (uint8_t *) NULL;
    txferSize         = MAX_TRANSFER_SIZE;
    pDmaSrc            = (uint8_t*) malloc(txferSize);
    pDmaDst            = (uint8_t*) malloc(txferSize);
    pSrcTemp           = pDmaSrc;

    if(((uint8_t*) NULL != pDmaSrc) && ((uint8_t*) NULL != pDmaDst))
    {
        /* Initialize the source memory block with the pattern */
        for(index=0; index < txferSize; index++)
        {
            *pSrcTemp++ = SOURCE_TRANSFER_PATTERN;
        }

        /* Initialize the DMA system service */
        dmaInit.sidl = SYS_DMA_SIDL_DISABLE;
```

```

sysObj = SYS_DMA_Initialize((SYS_MODULE_INIT*)&dmaInit);

/* Allocate a DMA channel */
channelHandle = SYS_DMA_ChannelAllocate(DMA_CHANNEL_1);
if(SYS_DMA_CHANNEL_HANDLE_INVALID != channelHandle)
{
    /* Register an event handler for the channel */
    SYS_DMA_ChannelTransferEventHandlerSet(channelHandle,
        Sys_DMA_Mem2Mem_Event_Handler, (uintptr_t)&sysContext);

    /* Setup the channel */
    SYS_DMA_ChannelSetup(channelHandle,
        SYS_DMA_CHANNEL_OP_MODE_BASIC,
        DMA_TRIGGER_SOURCE_NONE);

    /* Add the memory block transfer request */
    SYS_DMA_ChannelTransferAdd(channelHandle,pDmaSrc,txferSize,
        pDmaDst,txferSize,txferSize);

    /* Start the DMA transfer */
    SYS_DMA_ChannelForceStart(channelHandle);
}
else
{
    /* Channel Handle not available */
    ;
}
}

/* Below is an Event handler for the registered memory to memory DMA channel */
static void Sys_DMA_Mem2Mem_Event_Handler(SYS_DMA_TRANSFER_EVENT event,
    SYS_DMA_CHANNEL_HANDLE handle, uintptr_t contextHandle)
{
    int32_t fail=0;
    uint32_t index;

    /* Success event */
    if(SYS_DMA_TRANSFER_EVENT_COMPLETE == event)
    {
        /* Verify the contents of destination block matches the
         * memory contents of the source block */
        for(index=0; index < txferSize; index++)
        {
            if(*pDmaSrc++!=*pDmaDst++) // compare the buffers
            {
                fail = -1;
            }
        }
        if(0==fail)
        {
            /* Transfer Success */
        }
        else
        {
            /* Transfer Failed */
        }
    }
    /* Failure Event */
    else if(SYS_DMA_TRANSFER_EVENT_ABORT == event)
    {
    }
}

```

Configuring the Library

The configuration of the DMA System Service is based on the file `system_config.h`.

This header file contains the configuration selection for the DMA system service. Based on the selections made, the DMA System Service may support the selected features. These configuration settings will apply to all instances of the DMA System Service.

This header can be placed anywhere; however, the path of this header needs to be present in the include search path for a successful build. Refer to the Applications Help section for more details.

Building the Library

This section lists the files that are available in the Direct Memory Access (DMA) System Service Library.

Description

The following three tables list and describe the header (.h) and source (.c) files that implement this library. The parent folder for these files is `<install-dir>/framework/system/dma`.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using `#include`) by any code that uses this library.

Source File Name	Description
<code>sys_dma.h</code>	DMA System Service Library API header file.

Required File(s)



All of the required files listed in the following table are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

Source File Name	Description
<code>/src/sys_dma.c</code>	DMA System Service Library implementation.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
N/A	There are no optional files for the DMA System Service Library.

Module Dependencies

The DMA System Service Library does not depend on any other modules.

















Library Interface

a) Initialization and Task Functions




	Name	Description
⇒	<code>SYS_DMA_Tasks</code>	Maintains the system service's state machine.
⇒	<code>SYS_DMA_TasksError</code>	Maintains the system service's error state machine.
⇒	<code>SYS_DMA_Initialize</code>	Initializes and Enables the DMA Controller.
⇒	<code>SYS_DMA_TasksErrorISR</code>	Maintains the system service's state machine and implements its ISR.
⇒	<code>SYS_DMA_TasksISR</code>	Maintains the system service's state machine and implements its ISR.

b) Channel Setup and Management Functions

	Name	Description
⇒	<code>SYS_DMA_ChannelAbortEventSet</code>	Sets an event source and enables cell transfer abort event for the same for the selected channel.
⇒	<code>SYS_DMA_ChannelAllocate</code>	Allocates the specified DMA channel and returns a handle to it.

	SYS_DMA_ChannelDisable	Disables a channel.
	SYS_DMA_ChannelEnable	Enables a channel.
	SYS_DMA_ChannelForceAbort	Force abort of transfer on the selected channel.
	SYS_DMA_ChannelForceStart	Force start of transfer on the selected channel.
	SYS_DMA_ChannelsIsBusy	Returns the busy status of the specified DMA channel.
	SYS_DMA_ChannelRelease	Deallocates and frees the channel specified by the handle.
	SYS_DMA_ChannelSetupMatchAbortMode	Setup the pattern match abort mode.
	SYS_DMA_ChannelTransferAdd	Adds a data transfer to a DMA channel and Enables the channel to start data transfer.
	SYS_DMA_ChannelErrorGet	This function returns the error(if any) associated with the last client request.
	SYS_DMA_ChannelTransferEventHandlerSet	This function allows a DMA system service client to set an event handler.
	SYS_DMA_ChannelCRCGet	Returns the computed CRC.
	SYS_DMA_ChannelCRCSet	Sets up the CRC operation mode.
	SYS_DMA_ChannelSetup	Setup the DMA channel parameters.
	SYS_DMA_ChannelDestinationTransferredSizeGet	Returns the number of bytes transferred to destination.
	SYS_DMA_ChannelSourceTransferredSizeGet	Returns the number of bytes transferred from source.
	SYS_DMA_ChannelTransferSet	Sets up a data transfer to a DMA channel.

c) Global Control and Status Functions

	Name	Description
	SYS_DMA_IsBusy	Returns the busy status of the DMA module.
	SYS_DMA_Resume	Resume DMA operations.
	SYS_DMA_Suspend	Suspend DMA transfers.

d) Data Types and Constants

	Name	Description
	SYS_DMA_CHANNEL_CHAIN_PRIO	Identifies the priority of chaining channel.
	SYS_DMA_CHANNEL_CRC_MODE	Identifies the operation mode of the CRC module.
	SYS_DMA_CHANNEL_OP_MODE	Identifies the available DMA operating modes.
	SYS_DMA_CHANNEL_OPERATION_MODE_CRC	Defines the attributes for CRC operation mode.
	SYS_DMA_CRC_WRITE_ORDER	Identifies altering/maintaining write order post CRC computation.
	SYS_DMA_TRANSFER_EVENT	Enumeration of possible DMA System service events.
	SYS_DMA_CHANNEL_HANDLE	Assigns the handle for requested channel.
	SYS_DMA_CHANNEL_IGNORE_MATCH	Enables/Disables the feature to ignore a byte between a 2-byte match abort pattern.
	SYS_DMA_INIT	Indicates the error information for the DMA System service.
	SYS_DMA_SIDL	Enable/Disable DMA operations in Idle mode.
	SYS_DMA_CHANNEL_COUNT	Identifies the maximum number of DMA channel handles to be defined.
	SYS_DMA_CHANNEL_HANDLE_INVALID	Defines an invalid channel handle.

Description

This section describes the APIs of the DMA System Service Library.

Refer to each section for a detailed description.

a) Initialization and Task Functions

SYS_DMA_Tasks Function

Maintains the system service's state machine.

File

[sys_dma.h](#)

C

```
void SYS_DMA_Tasks(SYS_MODULE_OBJ object);
```


Returns

None.

Description

This function is used to maintain the DMA system service's internal state machine. This function is specifically designed for non DMA interrupt trigger implementations(polling mode), and should be used only in polling mode. this function should be called from the [SYS_Tasks\(\)](#) function.

Remarks

This function is normally not called directly by an application. It is called by the system's Tasks routine ([SYS_Tasks](#)).

This function is deprecated. Any calls made to this functions will have no effect.

Preconditions

DMA should have been initialized by calling [SYS_DMA_Initialize](#).

Example

```
// 'object' Returned from SYS_DMA_Initialize

while (true)
{
    SYS_DMA_Tasks ((object) );

    // Do other tasks
}
```

Parameters

Parameters	Description
object	Object handle for the DMA module (returned from SYS_DMA_Initialize)

Function

```
void SYS_DMA_Tasks( SYS\_MODULE\_OBJ object)
```

SYS_DMA_TasksError Function

Maintains the system service's error state machine.

File

[sys_dma.h](#)

C

```
void SYS_DMA_TasksError(SYS\_MODULE\_OBJ object);
```

Returns

None.

Description

This function is used to maintain the DMA system service's error state machine. This function is specifically designed for non DMA interrupt trigger implementations(polling mode), and should be used only in polling mode. this function should be called from the [SYS_Tasks\(\)](#) function.

Remarks

This function is normally not called directly by an application. It is called by the system's Tasks routine ([SYS_Tasks](#)).

Preconditions

DMA should have been initialized by calling [SYS_DMA_Initialize](#).

Example

```
// 'object' Returned from SYS_DMA_Initialize

while (true)
{
    SYS_DMA_TasksError (object);

    // Do other tasks
}
```

Parameters

Parameters	Description
object	Object handle for the DMA module (returned from SYS_DMA_Initialize)

Function

```
void SYS_DMA_TasksError( SYS\_MODULE\_OBJ object)
```

SYS_DMA_Initialize Function

Initializes and Enables the DMA Controller.

File

[sys_dma.h](#)

C

```
SYS\_MODULE\_OBJ SYS_DMA_Initialize(const SYS\_MODULE\_INIT * const init);
```

Returns

If successful, returns a valid handle to the DMA module object. Otherwise, it returns [SYS_MODULE_OBJ_INVALID](#).

Description

This function Enables the DMA module. Enable/Disable stop in idle mode feature based on the passed parameter value.

This function initializes the DMA module making it ready for clients to open and use it. The initialization data is specified by the init parameter.

Remarks

This function must be called before any other DMA systems service routines are called.

Not all features are available on all micro-controllers.

Preconditions

None.

Example

```
SYS\_MODULE\_OBJ objectHandle;
SYS\_DMA\_INIT initParam;

initParam.sidl = SYS\_DMA\_SIDL\_ENABLE;
objectHandle = SYS\_DMA\_Initialize(DRV\_I2S\_INDEX\_1,
                                   (SYS\_MODULE\_INIT*)initParam);
if (SYS\_MODULE\_OBJ\_INVALID == objectHandle)
{
    // Handle error
}
```

Parameters

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

Function

```
SYS\_MODULE\_OBJ SYS\_DMA\_Initialize(const SYS\_MODULE\_INIT * const init)
```

SYS_DMA_TasksErrorISR Function

Maintains the system service's state machine and implements its ISR.

File

[sys_dma.h](#)

C

```
void SYS\_DMA\_TasksErrorISR(SYS\_MODULE\_OBJ object, DMA\_CHANNEL activeChannel);
```

Returns

None.

Description

This function is used to maintain the DMA system service's internal error state machine and implement its ISR for DMA interrupt trigger implementations(interrupt mode). This function is specifically designed for DMA interrupt trigger implementations (interrupt mode). In interrupt mode, this function should be called in the interrupt service routine of the DMA channel that is associated with this transfer.

Remarks

It is called by appropriate raw ISR.

This function may execute in an ISR context and will never block or access any resources that may cause it to block.

Preconditions

DMA should have been initialized by calling [SYS_DMA_Initialize](#).

Example

```
// 'object' Returned from SYS_DMA_Initialize

void __ISR(_DMA3_VECTOR,ipl5) _InterruptHandler_BT_USART_RX_DMA_CHANNEL(void)
{
    // ....
    SYS_DMA_TasksErrorISR(object, DMA_CHANNEL_3);
}
```

Parameters

Parameters	Description
object	Object handle for the DMA module (returned from SYS_DMA_Initialize)
activeChannel	DMA channel number of the ISR being serviced.

Function

void SYS_DMA_TasksErrorISR([SYS_MODULE_OBJ](#) object, DMA_CHANNEL activeChannel)

SYS_DMA_TasksISR Function

Maintains the system service's state machine and implements its ISR.

File

[sys_dma.h](#)

C

```
void SYS_DMA_TasksISR(SYS_MODULE_OBJ object, DMA_CHANNEL activeChannel);
```

Returns

None.

Description

This function is used to maintain the DMA system service's internal state machine and implement its ISR for DMA interrupt trigger implementations(interrupt mode). This function is specifically designed for DMA interrupt trigger implementations (interrupt mode). In interrupt mode, this function should be called in the interrupt service routine of the DMA channel that is associated with this transfer.

Remarks

It is called by appropriate raw ISR.

This function may execute in an ISR context and will never block or access any resources that may cause it to block.

Preconditions

DMA should have been initialized by calling [SYS_DMA_Initialize](#).

Example

```
// 'object' Returned from SYS_DMA_Initialize

void __ISR(_DMA3_VECTOR,ipl5) _InterruptHandler_BT_USART_RX_DMA_CHANNEL(void)
```

```
{
    SYS_DMA_TasksISR(object, DMA_CHANNEL_3);
}
```

Parameters

Parameters	Description
object	Object handle for the DMA module (returned from SYS_DMA_Initialize)
activeChannel	DMA channel number of the ISR being serviced.

Function

```
void SYS_DMA_TasksISR( SYS\_MODULE\_OBJ object, DMA_CHANNEL activeChannel)
```

b) Channel Setup and Management Functions

SYS_DMA_ChannelAbortEventSet Function

Sets an event source and enables cell transfer abort event for the same for the selected channel.

File

[sys_dma.h](#)

C

```
void SYS_DMA_ChannelAbortEventSet(SYS_DMA_CHANNEL_HANDLE handle, DMA_TRIGGER_SOURCE eventSrc);
```

Returns

None.

Description

This functions enables a cell transfer abort event for the selected source event.

Remarks

If the parameter 'eventSrc' is specified as DMA_TRIGGER_SOURCE_NONE the current DMA transfer will be aborted. The behavior is a same as calling [SYS_DMA_ChannelForceAbort](#).

Preconditions

DMA should have been initialized by calling [SYS_DMA_Initialize](#). DMA channel should have been allocated by calling [SYS_DMA_ChannelAllocate](#). The function [SYS_DMA_ChannelSetup](#) must have been called to setup and enable the required features. The function [SYS_DMA_ChannelTransferAdd](#) to add a transfer.

Example

```
// 'handle' is a valid handle returned
// by the SYS_DMA_ChannelAllocate function.

// pDmaSrc, pDmaDst is the source,destination address
// txferSrcSize, txferDesSize is the source,destination transfer size
// cellSize is the cell size
MY_APP_OBJ      myAppObj;
// Client registers an event handler with service. This is done once

SYS_DMA_ChannelTransferEventHandlerSet( handle, APP_DMATransferEventHandler,
                                       (uintptr_t)&myAppObj );

SYS_DMA_ChannelSetup(handle, SYS_DMA_CHANNEL_OP_MODE_BASIC,
                    DMA_TRIGGER_SOURCE_NONE);

SYS_DMA_ChannelTransferAdd(handle,pDmaSrc,txferSrcSize,
                          pDmaDst,txferDesSize,cellSize);

SYS_DMA_ChannelAbortEventSet(handle, DMA_TRIGGER_CTMU);

SYS_DMA_ChannelForceStart(handle);
```

```
if(SYS_DMA_CHANNEL_HANDLE_INVALID == handle)
{
    // Error handling here
}

// Event Processing Technique. Event is received when
// the transfer is processed.

void APP_DMATransferEventHandle( SYS_DMA_TRANSFER_EVENT event,
                                SYS_DMA_CHANNEL_HANDLE handle, uintptr_t contextHandle )
{
    switch(event)
    {
        case SYS_DMA_TRANSFER_EVENT_ABORT:

            // This means the data was transferred.
            break;

        case SYS_DMA_TRANSFER_EVENT_ERROR:

            // Error handling here.

            if(SYS_DMA_ERROR_ADDRESS_ERROR == DRV_I2S_ErrorGet(myI2SHandle))
            {
                // There was an address error.
                // Do error handling here.
            }

            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	Handle of the DMA channel as returned by the SYS_DMA_ChannelAllocate function.
eventSrc	The event causing the cell transfer abort

Function

```
void SYS_DMA_ChannelAbortEventSet
(
    SYS_DMA_CHANNEL_HANDLE handle,
    DMA_TRIGGER_SOURCE eventSrc
)
```

SYS_DMA_ChannelAllocate Function

Allocates the specified DMA channel and returns a handle to it.

File

[sys_dma.h](#)

C

```
SYS_DMA_CHANNEL_HANDLE SYS_DMA_ChannelAllocate(DMA_CHANNEL channel);
```

Returns

The channel handle for the requested channel number.

If an error occurs, the return value is [SYS_DMA_CHANNEL_HANDLE_INVALID](#). Error can occur.

- if the requested channel number is invalid.
- if the requested channel number is not free.

Description

This function opens the specified DMA channel and provides a handle that must be provided to all other client-level operations to identify the caller and the DMA channel.

Remarks

The handle returned is valid until the [SYS_DMA_ChannelRelease](#) routine is called. This function must be called before any other DMA channel Setup and management routines are called

Preconditions

Function [SYS_DMA_Initialize](#) must have been called before calling this function.

Example

```
DMA_CHANNEL channel;
SYS_DMA_CHANNEL_HANDLE handle

channel = DMA_CHANNEL_2;
handle = SYS_DMA_ChannelAllocate(channel);
if (SYS_DMA_CHANNEL_HANDLE_INVALID == handle)
{
    // Failed to allocate the channel
}
else
{
    // Proceed with setting up the channel and adding the transfer
}
```

Parameters

Parameters	Description
channel	Channel number requested for allocation. When channel number specified is DMA_CHANNEL_ANY a random channel is allocated for DMA transfers.

Function

[SYS_DMA_CHANNEL_HANDLE](#) SYS_DMA_ChannelAllocate (DMA_CHANNEL channel)

SYS_DMA_ChannelDisable Function

Disables a channel.

File

[sys_dma.h](#)

C

```
void SYS_DMA_ChannelDisable(SYS_DMA_CHANNEL_HANDLE handle);
```

Returns

None.

Description

This function disables a channel.

Remarks

None.

Preconditions

DMA should have been initialized by calling [SYS_DMA_Initialize](#). DMA channel should have been allocated by calling [SYS_DMA_ChannelAllocate](#). The function [SYS_DMA_ChannelSetup](#) must have been called to setup and enable the required features. A DMA channel should have been enabled either by calling '[SYS_DMA_ChannelTransferAdd](#)' or '[SYS_DMA_ChannelEnable](#)'

Example

```
// 'handle' is a valid handle returned
// by the SYS_DMA_ChannelAllocate function.

SYS_DMA_ChannelDisable(handle);
```

Parameters

Parameters	Description
handle	Handle of the DMA channel as returned by the SYS_DMA_ChannelAllocate function.

Function

```
void SYS_DMA_ChannelDisable( SYS\_DMA\_CHANNEL\_HANDLE handle)
```

SYS_DMA_ChannelEnable Function

Enables a channel.

File

[sys_dma.h](#)

C

```
void SYS_DMA_ChannelEnable( SYS\_DMA\_CHANNEL\_HANDLE handle );
```

Returns

None.

Description

This function enables a channel.

Remarks

This function may not required to be called when starting DMA setup (by [SYS_DMA_ChannelSetup](#)) and transfer Add (by [SYS_DMA_ChannelTransferAdd](#)). But may be needed to be called in the registered callback to enable the channel and continue the data transfer with the existing transfer parameters previously set with '[SYS_DMA_ChannelTransferAdd](#)'. The DMA channel is by default disabled on the completion of block transfer(callback generated)

Preconditions

DMA should have been initialized by calling [SYS_DMA_Initialize](#). DMA channel should have been allocated by calling [SYS_DMA_ChannelAllocate](#). The function [SYS_DMA_ChannelSetup](#) must have been called to setup and enable the required features.

Example

```
// 'handle' is a valid handle returned  
// by the SYS_DMA_ChannelAllocate function.
```

```
SYS_DMA_ChannelEnable(handle);
```

Parameters

Parameters	Description
handle	Handle of the DMA channel as returned by the SYS_DMA_ChannelAllocate function.

Function

```
void SYS_DMA_ChannelEnable( SYS\_DMA\_CHANNEL\_HANDLE handle)
```

SYS_DMA_ChannelForceAbort Function

Force abort of transfer on the selected channel.

File

[sys_dma.h](#)

C

```
void SYS_DMA_ChannelForceAbort( SYS\_DMA\_CHANNEL\_HANDLE handle );
```

Returns

None.

Description

The function aborts a DMA transfer to occur for the selected channel.

Remarks

This function must be used to abort the DMA transfer when the channel has been setup (by calling [SYS_DMA_ChannelSetup](#)) with the eventSrc as DMA_TRIGGER_SOURCE_NONE. and [SYS_DMA_ChannelAbortEventSet](#) has not been called.

Preconditions

DMA should have been initialized by calling [SYS_DMA_Initialize](#). DMA channel should have been allocated by calling [SYS_DMA_ChannelAllocate](#). The function [SYS_DMA_ChannelSetup](#) must have been called to setup and enable the required features. The function '[SYS_DMA_ChannelTransferAdd](#)' to add a transfer.

Example

```
// 'handle' is a valid handle returned
// by the SYS_DMA_ChannelAllocate function.

// pDmaSrc, pDmaDst is the source, destination address
// txferSrcSize, txferDesSize is the source, destination transfer size
// cellSize is the cell size

MY_APP_OBJ      myAppObj;
// Client registers an event handler with service. This is done once

SYS_DMA_ChannelTransferEventHandlerSet( handle, APP_DMATransferEventHandler,
                                       (uintptr_t)&myAppObj );

SYS_DMA_ChannelSetup(handle, SYS_DMA_CHANNEL_OP_MODE_BASIC,
                    DMA_TRIGGER_SOURCE_NONE);

SYS_DMA_ChannelTransferAdd(handle, pDmaSrc, txferSrcSize,
                          pDmaDst, txferDesSize, cellSize);

SYS_DMA_ChannelForceStart(handle);

if(SYS_DMA_CHANNEL_HANDLE_INVALID == handle)
{
    // Error handling here
}

....
....
// Client may need to abort a transfer
SYS_DMA_ChannelForceAbort(handle);

// Event Processing Technique. Event is received when
// the transfer is processed.

void APP_DMATransferEventHandler( SYS_DMA_TRANSFER_EVENT event,
                                SYS_DMA_CHANNEL_HANDLE handle, uintptr_t contextHandle )
{
    switch(event)
    {
        case SYS_DMA_TRANSFER_EVENT_ABORT:

            // This means the data was transferred.
            break;

        case SYS_DMA_TRANSFER_EVENT_ERROR:

            // Error handling here.

            if(SYS_DMA_ERROR_ADDRESS_ERROR == DRV_I2S_ErrorGet(myI2SHandle))
            {
                // There was an address error.
                // Do error handling here.
            }

            break;
    }
}
```



```

        default:
            break;
    }
}

```

Parameters

Parameters	Description
handle	Handle of the DMA channel as returned by the SYS_DMA_ChannelAllocate function.

Function

```
void SYS_DMA_ChannelForceAbort( SYS\_DMA\_CHANNEL\_HANDLE handle)
```

SYS_DMA_ChannelForceStart Function

Force start of transfer on the selected channel.

File

[sys_dma.h](#)

C

```
void SYS_DMA_ChannelForceStart( SYS\_DMA\_CHANNEL\_HANDLE handle );
```

Returns

None.

Description

The function force start a DMA transfer to occur for the selected channel.

Remarks

This function must be used to start the DMA transfer when the channel has been setup (by calling [SYS_DMA_ChannelSetup](#)) with the eventSrc as DMA_TRIGGER_SOURCE_NONE.

Preconditions

DMA should have been initialized by calling [SYS_DMA_Initialize](#). DMA channel should have been allocated by calling [SYS_DMA_ChannelAllocate](#). The function [SYS_DMA_ChannelSetup](#) must have been called to setup and enable the required features. The function '[SYS_DMA_ChannelTransferAdd](#)' to add a transfer.

Example

```

// 'handle' is a valid handle returned
// by the SYS_DMA_ChannelAllocate function.

// pDmaSrc, pDmaDst is the source, destination address
// txferSrcSize, txferDesSize is the source, destination transfer size
// cellSize is the cell size
MY_APP_OBJ      myAppObj;
// Client registers an event handler with service. This is done once

SYS_DMA_ChannelTransferEventHandlerSet( handle, APP_DMATransferEventHandle,
                                       (uintptr_t)&myAppObj );

SYS_DMA_ChannelSetup(handle, SYS_DMA_CHANNEL_OP_MODE_BASIC,
                    DMA_TRIGGER_SOURCE_NONE);

SYS_DMA_ChannelTransferAdd(handle, pDmaSrc, txferSrcSize,
                          pDmaDst, txferDesSize, cellSize);

SYS_DMA_ChannelForceStart(handle);

if(SYS_DMA_CHANNEL_HANDLE_INVALID == handle)
{
    // Error handling here
}

// Event Processing Technique. Event is received when
// the transfer is processed.

```

```
void APP_DMATransferEventHandle( SYS_DMA_TRANSFER_EVENT event,
                                SYS_DMA_CHANNEL_HANDLE handle, uintptr_t contextHandle )
{
    switch(event)
    {
        case SYS_DMA_TRANSFER_EVENT_COMPLETE:

            // This means the data was transferred.
            break;

        case SYS_DMA_TRANSFER_EVENT_ERROR:

            // Error handling here.

            if(SYS_DMA_ERROR_ADDRESS_ERROR == DRV_I2S_ErrorGet(myI2SHandle))
            {
                // There was an address error.
                // Do error handling here.
            }

            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	Handle of the DMA channel as returned by the SYS_DMA_ChannelAllocate function.

Function

```
void SYS_DMA_ChannelForceStart( SYS\_DMA\_CHANNEL\_HANDLE handle)
```

SYS_DMA_ChannelsBusy Function

Returns the busy status of the specified DMA channel.

File

```
sys\_dma.h
```

C

```
bool SYS_DMA_ChannelIsBusy( SYS\_DMA\_CHANNEL\_HANDLE handle );
```

Returns

- bool - true, if the selected DMA channel is active or enabled
- false, if the selected DMA channel is inactive or disabled

Description

This function returns the busy status of the selected DMA channel

Remarks

This feature may not be available on all devices. Refer to the specific device data sheet to determine availability.

Preconditions

DMA should have been initialized by calling [SYS_DMA_Initialize](#). DMA channel should have been allocated by calling [SYS_DMA_ChannelAllocate](#).

Example

```
bool busyStat;

busyStat = SYS_DMA_ChannelGetBusy(handle);
```

Parameters

Parameters	Description
handle	Handle of the DMA channel as returned by the SYS_DMA_ChannelAllocate function.

Function

bool SYS_DMA_ChannelsIsBusy([SYS_DMA_CHANNEL_HANDLE](#) handle)

SYS_DMA_ChannelRelease Function

Deallocates and frees the channel specified by the handle.

File

[sys_dma.h](#)

C

```
void SYS_DMA_ChannelRelease(SYS_DMA_CHANNEL_HANDLE handle);
```

Returns

None.

Description

This function deallocates an allocated-channel of the DMA module, invalidating the handle.

Remarks

After calling this routine, the handle passed in "handle" must not be used with any of the remaining service's routines. A new handle must be obtained by calling [SYS_DMA_ChannelAllocate](#) before the caller may use the service again

Preconditions

DMA should have been initialized by calling [SYS_DMA_Initialize](#). DMA channel should have been allocated by calling [SYS_DMA_ChannelAllocate](#)

Example

```
DMA_CHANNEL channel;
SYS_DMA_CHANNEL_HANDLE handle;

channel = DMA_CHANNEL_2;
handle = SYS_DMA_ChannelAllocate(channel);
SYS_DMA_ChannelRelease(handle);
```

Parameters

Parameters	Description
handle	A valid allocated-channel handle, returned from the service's Allocate routine

Function

```
void SYS_DMA_ChannelRelease( SYS\_DMA\_CHANNEL\_HANDLE handle)
```

SYS_DMA_ChannelSetupMatchAbortMode Function

Setup the pattern match abort mode.

File

[sys_dma.h](#)

C

```
void SYS_DMA_ChannelSetupMatchAbortMode(SYS_DMA_CHANNEL_HANDLE handle, uint16_t pattern, DMA_PATTERN_LENGTH length, SYS_DMA_CHANNEL_IGNORE_MATCH ignore, uint8_t ignorePattern);
```

Returns

None.

Description

This function sets up the termination of DMA operation when the specified pattern is matched. Additionally on supported parts the function also

sets up the ignoring of part of a pattern(8-bit) from match abort pattern(16-bit).

Remarks

The parameter 'pattern' (8-bit or 16-bit) is device-specific. Not all features are available on all devices. Refer to the specific device data sheet for details.

Preconditions

DMA should have been initialized by calling [SYS_DMA_Initialize](#). DMA channel should have been allocated by calling [SYS_DMA_ChannelAllocate](#). The function [SYS_DMA_ChannelSetup](#) must be called to enable `SYS_DMA_CHANNEL_OP_MODE_MATCH_ABORT` before setting up pattern match mode features.

Example

```
// Example 1
// The following code is for a device with an 8-bit pattern value and no
// support for pattern match ignore feature

// 'handle' is a valid handle returned
// by the SYS_DMA_ChannelAllocate function.
uint16_t          pattern;
DMA_PATTERN_LENGTH length;
SYS_DMA_CHANNEL_IGNORE_MATCH ignore;
uint8_t           ignorePattern;

pattern          = 0x00; //Stop transfer on detection of a NULL character
length           = DMA_PATTERN_LENGTH_NONE;
ignore           = SYS_DMA_CHANNEL_IGNORE_MATCH_DISABLE;
ignorePattern    = 0;
SYS_DMA_ChannelSetupMatchAbortMode(handle, pattern, length,
                                   ignoreEnable, ignorePattern);

// Example 2
// The following code is for a device with a 16-bit pattern value and
// support for pattern match ignore feature

// 'handle' is a valid handle returned
// by the SYS_DMA_ChannelAllocate function.
uint16_t          pattern;
DMA_PATTERN_LENGTH length;
SYS_DMA_CHANNEL_IGNORE_MATCH ignore;
uint8_t           ignorePattern;

pattern          = 0x0D0A; //Stop transfer on detection of 'rn'
length           = DMA_PATTERN_MATCH_LENGTH_2BYTES;
ignore           = SYS_DMA_CHANNEL_IGNORE_MATCH_ENABLE;
ignorePattern    = 0x00; \ Any null character between the termination pattern
                        \ 'r' and 'n' is ignored.
SYS_DMA_ChannelSetupMatchAbortMode(handle, pattern, length,
                                   ignore, ignorePattern);
```

Parameters

Parameters	Description
handle	Handle of the DMA channel as returned by the SYS_DMA_ChannelAllocate function.
pattern	The pattern that needs to be matched to abort a DMA transfer.
length	Match pattern length can be 1-byte or 2-byte.
ignore	Enable/Disable a byte between a 2-byte pattern match.
ignorePattern	The part of the pattern(8-bit) that needs to be ignored from the match abort pattern(16-bit)

Function

```
void SYS_DMA_ChannelSetupMatchAbortMode
(
    SYS\_DMA\_CHANNEL\_HANDLE handle,
    uint16_t pattern,
    DMA_PATTERN_LENGTH length,
    SYS\_DMA\_CHANNEL\_IGNORE\_MATCH ignore,
```

```
uint8_t ignorePattern
)
```

SYS_DMA_ChannelTransferAdd Function

Adds a data transfer to a DMA channel and Enables the channel to start data transfer.

File

[sys_dma.h](#)

C

```
void SYS_DMA_ChannelTransferAdd(SYS_DMA_CHANNEL_HANDLE handle, const void * srcAddr, size_t srcSize, const
void * destAddr, size_t destSize, size_t cellSize);
```

Returns

None.

Description

This function adds a data transfer characteristics for a DMA channel. The The source and the destination addresses, source and destination lengths, The number of bytes transferred per cell event are set. It also enables the channel to start data transfer.

If the requesting client registered an event callback with the service, the service will issue a SYS_DMA_TRANSFER_EVENT_COMPLETE or SYS_DMA_TRANSFER_EVENT_ABORT event if the transfered was processed successfully of SYS_DMA_TRANSFER_EVENT_ERROR event if the transfer was not processed successfully.

Remarks

For PIC32MZ series of devices, if the source/destination addresses are RAM memory addresses, the the source/destination buffers should be made coherent to avoid the cache coherency issues. For example:

```
uint8_t buffer[1024];
// The buffer can be made coherent by adding the 'coherent' attribute, as follows:
uint8_t __attribute__((coherent)) __attribute__((aligned(16))) buffer[1024];
```

Preconditions

DMA should have been initialized by calling [SYS_DMA_Initialize](#). DMA channel should have been allocated by calling [SYS_DMA_ChannelAllocate](#). The function [SYS_DMA_ChannelSetup](#) must have been called to setup and enable the required features.

Example

```
// Add 10 bytes of data transfer to UART TX

// 'handle' is a valid handle returned
// by the SYS_DMA_ChannelAllocate function.
MY_APP_OBJ      myAppObj;
uint8_t         buf[10];
void            *srcAddr;
void            *destAddr;
size_t          srcSize;
size_t          destSize;
size_t          cellSize;

srcAddr         = (uint8_t *) buf;
srcSize         = 10;
destAddr        = (uint8_t*) &U2TXREG;
destSize        = 1;
cellSize        = 1;

// User registers an event handler with system service. This is done once

SYS_DMA_ChannelTransferEventHandlerSet(handle, APP_DMASysTransferEventHandler,
                                       (uintptr_t)&myAppObj);

SYS_DMA_ChannelTransferAdd(handle,srcAddr,srcSize,destAddr,destSize,cellSize);

if(SYS_DMA_CHANNEL_HANDLE_INVALID == handle)
{
    // Error handling here
}
```

```
// Event Processing Technique. Event is received when
// the transfer is processed.

void APP_DMASYSSTransferEventHandler(SYS_DMA_TRANSFER_EVENT event,
    SYS_DMA_CHANNEL_HANDLE handle, uintptr_t contextHandle)
{
    switch(event)
    {
        case SYS_DMA_TRANSFER_EVENT_COMPLETE:

            // This means the data was transferred.
            break;

        case SYS_DMA_TRANSFER_EVENT_ERROR:

            // Error handling here.
            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	Handle of the DMA channel as returned by the SYS_DMA_ChannelAllocate function.
srcAddr	Source of the DMA transfer
srcSize	Size of the source
destAddr	Destination of the DMA transfer
destSize	Size of the destination
cellSize	Size of the cell

Function

```
void SYS_DMA_ChannelTransferAdd
(
    SYS_DMA_CHANNEL_HANDLE handle,
    const void *srcAddr, size_t srcSize
    const void *destAddr, size_t destSize,
    size_t cellSize
)
```

SYS_DMA_ChannelErrorGet Function

This function returns the error(if any) associated with the last client request.

File

[sys_dma.h](#)

C

```
SYS_DMA_ERROR SYS_DMA_ChannelErrorGet(SYS_DMA_CHANNEL_HANDLE handle);
```

Returns

A SYS_DMA_ERROR type indicating last known error status.

Description

This function returns the error(if any) associated with the last client request. If the service send a SYS_DMA_TRANSFER_EVENT_ERROR to the client, the client can call this function to know the error cause. The error status will be updated on every operation and should be read frequently (ideally immediately after the service operation has completed) to know the relevant error status.

Remarks

It is the client's responsibility to make sure that the error status is obtained frequently. The service will update the error status regardless of whether this has been examined by the client.

Preconditions

The [SYS_DMA_Initialize](#) routine must have been called for the DMA sub system. [SYS_DMA_ChannelAllocate](#) must have been called to obtain a valid opened channel handle.

Example

```
// 'handle' is a valid handle returned
// by the SYS_DMA_ChannelAllocate function.

// pDmaSrc, pDmaDst is the source,destination address
// txferSrcSize, txferDesSize is the source,destination transfer size
// cellSize is the cell size
MY_APP_OBJ          myAppObj;
// Client registers an event handler with service. This is done once

SYS_DMA_ChannelTransferEventHandlerSet( handle, APP_DMATransferEventHandler,
                                       (uintptr_t)&myAppObj );

SYS_DMA_ChannelSetup(handle, SYS_DMA_CHANNEL_OP_MODE_BASIC,
                     DMA_TRIGGER_SOURCE_NONE);

SYS_DMA_ChannelTransferAdd(handle,pDmaSrc,txferSrcSize,
                           pDmaDst,txferDesSize,cellSize);

SYS_DMA_ChannelForceStart(handle);

if(SYS_DMA_CHANNEL_HANDLE_INVALID == handle)
{
    // Error handling here
}

// Event Processing Technique. Event is received when
// the transfer is processed.

void APP_DMATransferEventHandler( SYS_DMA_TRANSFER_EVENT event,
                                SYS_DMA_CHANNEL_HANDLE handle, uintptr_t contextHandle )
{
    switch(event)
    {
        case SYS_DMA_TRANSFER_EVENT_COMPLETE:

            // This means the data was transferred.
            break;

        case SYS_DMA_TRANSFER_EVENT_ERROR:

            // Error handling here.

            if(SYS_DMA_ERROR_ADDRESS_ERROR == SYS_DMA_ChannelErrorGet(handle))
            {
                // There was an address error.
                // Do error handling here.
            }

            break;

        default:
            break;
    }
}
```

Parameters

Parameters	Description
handle	Handle of the DMA channel as returned by the SYS_DMA_ChannelAllocate function.

Function

`SYS_DMA_ERROR SYS_DMA_ChannelErrorGet(SYS_DMA_CHANNEL_HANDLE handle)`

SYS_DMA_ChannelTransferEventHandlerSet Function

This function allows a DMA system service client to set an event handler.

File

[sys_dma.h](#)

C

```
void SYS_DMA_ChannelTransferEventHandlerSet(SYS\_DMA\_CHANNEL\_HANDLE handle, const
SYS\_DMA\_CHANNEL\_TRANSFER\_EVENT\_HANDLER eventHandler, const uintptr\_t contextHandle);
```

Returns

None.

Description

This function allows a client to set an event handler. The client may want to receive transfer related events in cases when it uses non-blocking read and write functions. The event handler should be set before the client intends to perform operations that could generate events.

This function accepts a contextHandle parameter. This parameter could be set by the client to contain (or point to) any client specific data object that should be associated with this DMA channel.

Remarks

None.

Preconditions

DMA should have been initialized by calling [SYS_DMA_Initialize](#). DMA channel should have been allocated by calling [SYS_DMA_ChannelAllocate](#).

Example

```
// 'handle' is a valid handle returned
// by the SYS_DMA_ChannelAllocate function.
MY_APP_OBJ          myAppObj;
uint8_t             buf[10];
void                *srcAddr;
void                *destAddr;
size_t              srcSize;
size_t              destSize;
size_t              cellSize;

srcAddr      = (uint8_t *) buf;
srcSize      = 10;
destAddr     = (uint8_t*) &U2TXREG;
destSize     = 1;
cellSize     = 1;
channelHandle = SYS_DMA_ChannelAllocate(channel);
// User registers an event handler with system service. This is done once

SYS_DMA_ChannelTransferEventHandlerSet(handle, APP_DMASYSSTransferEventHandler,
                                       (uintptr_t)&myAppObj);

SYS_DMA_ChannelTransferAdd(handle,srcAddr,srcSize,destAddr,destSize,cellSize);

if(SYS_DMA_CHANNEL_HANDLE_INVALID == handle)
{
    // Error handling here
}

// Event Processing Technique. Event is received when
// the transfer is processed.

void APP_DMASYSSTransferEventHandler(SYS_DMA_TRANSFER_EVENT event,
                                     SYS\_DMA\_CHANNEL\_HANDLE handle, uintptr\_t contextHandle)
{
    switch(event)
```



```
{
    case SYS_DMA_TRANSFER_EVENT_COMPLETE:

        // This means the data was transferred.
        break;

    case SYS_DMA_TRANSFER_EVENT_ERROR:

        // Error handling here.
        break;

    default:
        break;
}
```

Parameters

Parameters	Description
handle	A valid channel handle, returned from the system service's Allocate routine
eventHandler	Pointer to the event handler function.
contextHandle	Value identifying the context of the application/driver/middleware that registered the event handling function.

Function

```
void SYS_DMA_ChannelTransferEventHandlerSet
(
    SYS_DMA_CHANNEL_HANDLE handle,
    const SYS_DMA_CHANNEL_TRANSFER_EVENT_HANDLER eventHandler,
    const uintptr_t contextHandle
)
```

SYS_DMA_ChannelCRCGet Function

Returns the computed CRC.

File

```
sys_dma.h
```

C

```
uint32_t SYS_DMA_ChannelCRCGet();
```

Returns

uint32_t - The Computed CRC.

Description

This function returns the computed CRC

Remarks

To get the computed CRC value this function must be called after the block transfer completion event (i.e., after getting and processing the callback registered with [SYS_DMA_ChannelTransferEventHandlerSet](#)).

This feature may not be available on all devices. Refer to the specific device data sheet to determine availability.

Preconditions

DMA should have been initialized by calling [SYS_DMA_Initialize](#). DMA channel should have been allocated by calling [SYS_DMA_ChannelAllocate](#). The function [SYS_DMA_ChannelSetup](#) must be called to enable SYS_DMA_CHANNEL_OP_MODE_CRC before setting up the CRC mode. The CRC generator must have been previously setup using [SYS_DMA_ChannelCRCSet](#).

Example

```
uint32_t          computedCRC;

computedCRC      = SYS_DMA_ChannelCRCGet();
```

Function

```
uint32_t SYS_DMA_ChannelCRCGet(void)
```

SYS_DMA_ChannelCRCSet Function

Sets up the CRC operation mode.

File

[sys_dma.h](#)

C

```
void SYS_DMA_ChannelCRCSet(SYS_DMA_CHANNEL_HANDLE handle, SYS_DMA_CHANNEL_OPERATION_MODE_CRC crc);
```

Returns

None.

Description

This function sets up the CRC computation features.

Remarks

This feature may not be available on all devices. Refer to the specific device data sheet to determine availability.

Preconditions

DMA should have been initialized by calling [SYS_DMA_Initialize](#). DMA channel should have been allocated by calling [SYS_DMA_ChannelAllocate](#). The function [SYS_DMA_ChannelSetup](#) must be called to enable SYS_DMA_CHANNEL_OP_MODE_CRC before setting up the CRC mode.

Example

```
//Example 1
// DMA calculation using the CRC background mode

// 'handle' is a valid handle returned
// by the SYS_DMA_ChannelAllocate function.
SYS_DMA_CHANNEL_OPERATION_MODE_CRC  crc;

crc.type           = DMA_CRC_LFSR;
crc.mode           = SYS_DMA_CHANNEL_CRC_MODE_BACKGROUND;
crc.polyLength     = 16;
crc.bitOrder       = DMA_CRC_BIT_ORDER_LSB;
crc.byteOrder      = DMA_CRC_BYTEORDER_NO_SWAPPING;
crc.writeOrder     = SYS_DMA_CRC_WRITE_ORDER_MAINTAIN;
crc.data           = 0xFFFF;
crc.xorBitMask     = 0x1021;
SYS_DMA_ChannelCRCSet(handle, crc);
```

Parameters

Parameters	Description
handle	Handle of the DMA channel as returned by the SYS_DMA_ChannelAllocate function.
crc.type	CRC will calculate an IP header checksum or an LFSR CRC.
crc.mode	Compute the CRC in Background/Append mode.
crc.polyLength	Denotes the length of the polynomial.
crc.bitOrder	CRC is calculated LSb/MSb first.
crc.byteOrder	Byte selection order input pre-CRC Generator
crc.writeOrder	Write byte order selection post-CRC computation
crc.data	Computed/Seed CRC
crc.xorBitMask	Enable/Disable XOR bit mask on the corresponding bits when mode is LFSR

Function

```
void SYS_DMA_ChannelCRCSet
(
    SYS\_DMA\_CHANNEL\_HANDLE handle,
```

[SYS_DMA_CHANNEL_OPERATION_MODE_CRC](#) crc

)

SYS_DMA_ChannelSetup Function

Setup the DMA channel parameters.

File

[sys_dma.h](#)

C

```
void SYS_DMA_ChannelSetup(SYS_DMA_CHANNEL_HANDLE handle, SYS_DMA_CHANNEL_OP_MODE modeEnable,
DMA_TRIGGER_SOURCE eventSrc);
```

Returns

None.

Description

This function sets up the DMA channel parameters. It sets the channel priority and enables the mode of operations for the current system design.

Remarks

If [SYS_DMA_CHANNEL_OP_MODE_MATCH_ABORT](#), [SYS_DMA_CHANNEL_OP_MODE_CHAIN](#) or [SYS_DMA_CHANNEL_OP_MODE_CRC](#) mode of operation is enabled, then corresponding mode setup API's needs to be called to set the related parameters.

If the parameter 'eventSrc' is specified as [DMA_TRIGGER_SOURCE_NONE](#) then [SYS_DMA_ChannelForceStart](#) must be called to start the DMA channel transfer.

Not all features are available on all microcontrollers.

Preconditions

DMA should have been initialized by calling [SYS_DMA_Initialize](#). DMA channel should have been allocated by calling [SYS_DMA_ChannelAllocate](#).

Example

```
// 'handle' is a valid handle returned
// by the SYS_DMA_ChannelAllocate function.

SYS_DMA_CHANNEL_OP_MODE    modeEnable;
DMA_TRIGGER_SOURCE         eventSrc;

channel                    =    DMA_CHANNEL_2;
modeEnable                =    (SYS_DMA_CHANNEL_OP_MODE_BASIC | SYS_DMA_CHANNEL_OP_MODE_CRC);
eventSrc                  =    DMA_TRIGGER_USART_1_TRANSMIT;
// Setup channel number, and enables basic and CRC mode
SYS_DMA_ChannelSetup(handle, modeEnable, eventSrc);
```

Parameters

Parameters	Description
handle	Handle of the DMA channel as returned by the SYS_DMA_ChannelAllocate function.
priority	The priority to be associated to the channel.
modeEnable	The supported operating modes to be enabled. This parameter can be logically ORed to specify multiple options.
eventSrc	The event causing the cell transfer start.

Function

```
void SYS_DMA_ChannelSetup
(
    SYS\_DMA\_CHANNEL\_HANDLE handle,
    SYS\_DMA\_CHANNEL\_OP\_MODE modeEnable
    DMA_TRIGGER_SOURCE eventSrc
)
```

SYS_DMA_ChannelDestinationTransferredSizeGet Function

Returns the number of bytes transferred to destination.

File

[sys_dma.h](#)

C

```
size_t SYS_DMA_ChannelDestinationTransferredSizeGet(SYS_DMA_CHANNEL_HANDLE handle);
```

Returns

size_t - Returns the number of bytes received from the submitted size.

Description

When a data transfer request is submitted. At any time while the transmission is in progress the size of the amount of data transferred to destination can be known by calling this function.

Remarks

None.

Preconditions

DMA should have been initialized by calling [SYS_DMA_Initialize](#). DMA channel should have been allocated by calling [SYS_DMA_ChannelAllocate](#). Data transfer should have been initiated by calling [SYS_DMA_ChannelTransferAdd](#) or [SYS_DMA_ChannelTransferSet](#).

Example

```
// 'handle' is a valid handle returned
// by the SYS_DMA_ChannelAllocate function.
MY_APP_OBJ      myAppObj;
uint8_t         buf[10];
void            *srcAddr;
void            *destAddr;
size_t          srcSize;
size_t          destSize;
size_t          cellSize;
size_t          transferredSize;

srcAddr      = (uint8_t*) &U2RXREG;
srcSize      = 1;
destAddr     = (uint8_t *) buf ;
destSize     = 10;
cellSize     = 1;
channelHandle = SYS_DMA_ChannelAllocate(channel);
// User registers an event handler with system service. This is done once

SYS_DMA_ChannelTransferEventHandlerSet(handle, APP_DMASYSSTransferEventHandler,
                                       (uintptr_t)&myAppObj);

SYS_DMA_ChannelTransferAdd(handle,srcAddr,srcSize,destAddr,destSize,cellSize);

if(SYS_DMA_CHANNEL_HANDLE_INVALID == handle)
{
    // Error handling here
}

// The data is being transferred after adding the transfer to the DMA channel.
// The user can get to know dynamically the amount of data
// transmitted to destination by calling SYS_DMA_ChannelDestinationTransferredSizeGet
transferredSize = SYS_DMA_ChannelDestinationTransferredSizeGet(channelHandle);
```

Parameters

Parameters	Description
handle	A valid channel handle, returned from the system service's Allocate routine

Function

```
size_t SYS_DMA_ChannelDestinationTransferredSizeGet( SYS\_DMA\_CHANNEL\_HANDLE handle)
```

SYS_DMA_ChannelSourceTransferredSizeGet Function

Returns the number of bytes transferred from source.

File

[sys_dma.h](#)

C

```
size_t SYS_DMA_ChannelSourceTransferredSizeGet(SYS_DMA_CHANNEL_HANDLE handle);
```

Returns

size_t - Returns the number of bytes transferred from the submitted size.

Description

When a data transfer request is submitted. At any time while the transmission is in progress the size of the amount of data transferred from source can be known by calling this function.

Remarks

None.

Preconditions

DMA should have been initialized by calling [SYS_DMA_Initialize](#). DMA channel should have been allocated by calling [SYS_DMA_ChannelAllocate](#). Data transfer should have been initiated by calling [SYS_DMA_ChannelTransferAdd](#) or [SYS_DMA_ChannelTransferSet](#).

Example

```
// 'handle' is a valid handle returned
// by the SYS_DMA_ChannelAllocate function.
MY_APP_OBJ      myAppObj;
uint8_t          buf[10];
void             *srcAddr;
void             *destAddr;
size_t           srcSize;
size_t           destSize;
size_t           cellSize;
size_t           transferredSize;

srcAddr      = (uint8_t *) buf;
srcSize      = 10;
destAddr     = (uint8_t*) &U2TXREG;
destSize     = 1;
cellSize     = 1;
channelHandle = SYS_DMA_ChannelAllocate(channel);
// User registers an event handler with system service. This is done once

SYS_DMA_ChannelTransferEventHandlerSet(handle, APP_DMASYSSTransferEventHandler,
                                       (uintptr_t)&myAppObj);

SYS_DMA_ChannelTransferAdd(handle,srcAddr,srcSize,destAddr,destSize,cellSize);

if(SYS_DMA_CHANNEL_HANDLE_INVALID == handle)
{
    // Error handling here
}

// The data is being transferred after adding the transfer to the DMA channel.
// The user can get to know dynamically the amount of data
// transmitted from source by calling SYS_DMA_ChannelSourceTransferredSizeGet
transferredSize = SYS_DMA_ChannelSourceTransferredSizeGet(channelHandle);
```

Parameters

Parameters	Description
handle	A valid channel handle, returned from the system service's Allocate routine

Function

size_t SYS_DMA_ChannelSourceTransferredSizeGet([SYS_DMA_CHANNEL_HANDLE](#) handle)

SYS_DMA_ChannelTransferSet Function

Sets up a data transfer to a DMA channel.

File

[sys_dma.h](#)

C

```
void SYS_DMA_ChannelTransferSet(SYS_DMA_CHANNEL_HANDLE handle, const void * srcAddr, size_t srcSize, const void * destAddr, size_t destSize, size_t cellSize);
```

Returns

None.

Description

This function sets up data transfer characteristics for a DMA channel. The source and the destination addresses, source and destination lengths, The number of bytes transferred per cell event are set. This function does not enable the DMA channel. The channel has to be explicitly enabled to start the data transfer. The above functions could be used in situations where the user intends to setup transfer parameters but do not intend to enable the channel immediately. For example to chain to DMA channels in a cyclic order where the channels remains disabled. The channel is enabled automatically by the DMA when the transfer trigger condition occurs.

Remarks

For PIC32MZ series of devices, if the source/destination addresses are RAM memory addresses, the source/destination buffers should be made coherent to avoid the cache coherency issues. For example:

```
uint8_t buffer[1024];
// The buffer can be made coherent by adding the 'coherent' attribute, as follows:
uint8_t __attribute__((coherent)) __attribute__((aligned(16))) buffer[1024];
```

Preconditions

DMA should have been initialized by calling [SYS_DMA_Initialize](#). DMA channel should have been allocated by calling [SYS_DMA_ChannelAllocate](#). The function [SYS_DMA_ChannelSetup](#) must have been called to setup and enable the required features.

Example

```
// Set up 10 bytes of data transfer to UART TX

// 'handle' is a valid handle returned
// by the SYS_DMA_ChannelAllocate function.
MY_APP_OBJ      myAppObj;
uint8_t         buf[10];
void            *srcAddr;
void            *destAddr;
size_t          srcSize;
size_t          destSize;
size_t          cellSize;

srcAddr         = (uint8_t *) buf;
srcSize         = 10;
destAddr        = (uint8_t *) &U2TXREG;
destSize        = 1;
cellSize        = 1;

// User registers an event handler with system service. This is done once

SYS_DMA_ChannelTransferEventHandlerSet(handle, APP_DMASYSSTransferEventHandler,
                                       (uintptr_t)&myAppObj);

SYS_DMA_ChannelTransferSet(handle, srcAddr, srcSize, destAddr, destSize, cellSize);
SYS_DMA_ChannelEnable(handle);

if(SYS_DMA_CHANNEL_HANDLE_INVALID == handle)
{
    // Error handling here
```

```

}

// Event Processing Technique. Event is received when
// the transfer is processed.

void APP_DMASysTransferEventHandler(SYS_DMA_TRANSFER_EVENT event,
    SYS_DMA_CHANNEL_HANDLE handle, uintptr_t contextHandle)
{
    switch(event)
    {
        case SYS_DMA_TRANSFER_EVENT_COMPLETE:

            // This means the data was transferred.
            break;

        case SYS_DMA_TRANSFER_EVENT_ERROR:

            // Error handling here.
            break;

        default:
            break;
    }
}

```

Parameters

Parameters	Description
handle	Handle of the DMA channel as returned by the SYS_DMA_ChannelAllocate function.
srcAddr	Source of the DMA transfer
srcSize	Size of the source
destAddr	Destination of the DMA transfer
destSize	Size of the destination
cellSize	Size of the cell

Function

```

void SYS_DMA_ChannelTransferSet
(
    SYS_DMA_CHANNEL_HANDLE handle,
    const void *srcAddr, size_t srcSize
    const void *destAddr, size_t destSize,
    size_t cellSize
)

```

c) Global Control and Status Functions

SYS_DMA_IsBusy Function

Returns the busy status of the DMA module.

File

[sys_dma.h](#)

C

```
bool SYS_DMA_IsBusy();
```

Returns

Boolean

- true - The DMA module is active
- false - The DMA module is inactive and disabled

```
bool busyStat;
busyStat = SYS_DMA_IsBusy();
```

Description

This function returns the busy status of the DMA module

Remarks

This feature may not be available on all devices. Refer to the specific device data sheet to determine availability.

Preconditions

DMA should have been initialized by calling [SYS_DMA_Initialize](#).

Function

```
bool SYS_DMA_IsBusy(void)
```

SYS_DMA_Resume Function

Resume DMA operations.

File

[sys_dma.h](#)

C

```
void SYS_DMA_Resume( );
```

Returns

None.

Description

This function disables DMA suspend. It resumes the DMA operation suspended by calling [SYS_DMA_Suspend](#). The DMA operates normally.

Remarks

None

Preconditions

DMA should have been initialized by calling [SYS_DMA_Initialize](#).

Example

```
SYS_DMA_Resume( );
```

Function

```
void SYS_DMA_Resume(void)
```

SYS_DMA_Suspend Function

Suspend DMA transfers.

File

[sys_dma.h](#)

C

```
void SYS_DMA_Suspend( );
```

Returns

None.

Description

This function suspends DMA transfers to allow CPU uninterrupted access to data bus

Remarks

None

Preconditions

DMA should have been initialized by calling [SYS_DMA_Initialize](#).

Example

```
SYS_DMA_Suspend( ) ;
```

Function

```
void SYS_DMA_Suspend(void)
```

d) Data Types and Constants

SYS_DMA_CHANNEL_CHAIN_PRIO Enumeration

Identifies the priority of chaining channel.

File

[sys_dma.h](#)

C

```
typedef enum {  
    SYS_DMA_CHANNEL_CHAIN_PRIO_HIGH,  
    SYS_DMA_CHANNEL_CHAIN_PRIO_LOW  
} SYS_DMA_CHANNEL_CHAIN_PRIO;
```

Members

Members	Description
SYS_DMA_CHANNEL_CHAIN_PRIO_HIGH	Chain to channel higher in natural priority
SYS_DMA_CHANNEL_CHAIN_PRIO_LOW	Chain to channel lower in natural priority

Description

DMA Channel chaining priority

This data type identifies the priority of chaining channel.

Channel chaining priority identifies the channel that will be enabled on completion of block data transfer on the master channel.

Remarks

None.

SYS_DMA_CHANNEL_CRC_MODE Enumeration

Identifies the operation mode of the CRC module.

File

[sys_dma.h](#)

C

```
typedef enum {  
    SYS_DMA_CHANNEL_CRC_MODE_BACKGROUND,  
    SYS_DMA_CHANNEL_CRC_MODE_APPEND  
} SYS_DMA_CHANNEL_CRC_MODE;
```

Members

Members	Description
SYS_DMA_CHANNEL_CRC_MODE_BACKGROUND	DMA reads the data from the source, passes it through the CRC module and writes it to the destination. the calculated CRC is left in the DCRCDATA register at the end of the block transfer.
SYS_DMA_CHANNEL_CRC_MODE_APPEND	DMA only feeds source data to the CRC module. it does not write source data to the destination address. When a block transfer complete or pattern abort event occur, The DMA writes the CRC value to the destination address

Description

DMA channel CRC mode

This data type identifies CRC operating mode. The CRC computation can be in background mode or append mode.

Remarks

None.

SYS_DMA_CHANNEL_OP_MODE Enumeration

Identifies the available DMA operating modes.

File

[sys_dma.h](#)

C

```
typedef enum {
    SYS_DMA_CHANNEL_OP_MODE_BASIC,
    SYS_DMA_CHANNEL_OP_MODE_MATCH_ABORT,
    SYS_DMA_CHANNEL_OP_MODE_CHAIN_HIGH,
    SYS_DMA_CHANNEL_OP_MODE_CHAIN_LOW,
    SYS_DMA_CHANNEL_OP_MODE_AUTO,
    SYS_DMA_CHANNEL_OP_MODE_CRC
} SYS_DMA_CHANNEL_OP_MODE;
```

Members

Members	Description
SYS_DMA_CHANNEL_OP_MODE_BASIC	The normal DMA operation mode. The DMA channel will transfer data from a source to a destination without CPU intervention
SYS_DMA_CHANNEL_OP_MODE_MATCH_ABORT	Pattern Match abort mode allows the user to end a transfer if data written during a transaction matches a specific pattern, as defined by the DCHxDAT register
SYS_DMA_CHANNEL_OP_MODE_CHAIN_HIGH	Channel chaining operating mode enhances the normal DMA channel operations. A channel(slave) can be chained to an adjacent channel(master). The slave channel will be enabled when a block transfer of the master channel completes. only one of the below two chaining priorities must be provided. Chaining priority High.
SYS_DMA_CHANNEL_OP_MODE_CHAIN_LOW	Chaining priority Low
SYS_DMA_CHANNEL_OP_MODE_AUTO	Auto enable operating mode allows a channel to be kept active, even if a block transfer completes or pattern match occurs. This prevents the user from having to re-enable the channel each time a block transfer completes.
SYS_DMA_CHANNEL_OP_MODE_CRC	CRC operating mode allows computation of 16, 32-bit CRC.

Description

DMA channel operation modes

This data type Identifies the available DMA operating modes.

The supported DMA modes are not mutually exclusive, but can be simultaneously operational.

Remarks

These supported DMA modes can be logically OR'd together. They are passed into the [SYS_DMA_ChannelSetup](#) function through the modeEnable parameter.

SYS_DMA_CHANNEL_OPERATION_MODE_CRC Structure

Defines the attributes for CRC operation mode.

File

[sys_dma.h](#)

C

```
typedef struct {
    DMA_CRC_TYPE type;
    SYS_DMA_CHANNEL_CRC_MODE mode;
    uint8_t polyLength;
    DMA_CRC_BIT_ORDER bitOrder;
    DMA_CRC_BYTE_ORDER byteOrder;
    SYS_DMA_CRC_WRITE_ORDER writeOrder;
    uint32_t data;
    uint32_t xorBitMask;
} SYS_DMA_CHANNEL_OPERATION_MODE_CRC;
```

Members

Members	Description
DMA_CRC_TYPE type;	CRC type (PLIB-level). Type Selection identifies whether the CRC module will calculate an IP header checksum or an LFSR CRC
SYS_DMA_CHANNEL_CRC_MODE mode;	CRC mode, Background or Append
uint8_t polyLength;	Polynomial length, This value is redundant when the selected CRC type is IP header checksum. When the CRC type is LFSR, Identifies the length of the polynomial.
DMA_CRC_BIT_ORDER bitOrder;	Bit order (PLIB-level). Identifies whether the CRC is computed LSb or MSb first
DMA_CRC_BYTE_ORDER byteOrder;	Byte order (PLIB-level). Identifies the byte selection order input pre-CRC Generator.
SYS_DMA_CRC_WRITE_ORDER writeOrder;	Post CRC Write order
uint32_t data;	CRC data feed: While enabling the CRC mode, this field gives the seed for the CRC computation. On block transfer complete or pattern match abort the field have the computed CRC.
uint32_t xorBitMask;	XOR bit mask, This value is redundant when the selected CRC type is IP header checksum. When the CRC type is LFSR, Identifies the XOR bit mask input to the shift register during CRC computation.

Description

DMA channel CRC mode

This data type defines the attributes for CRC operation mode.

Remarks

This feature may not be available on all devices. Refer to the specific device data sheet to determine availability.

SYS_DMA_CRC_WRITE_ORDER Enumeration

Identifies altering/maintaining write order post CRC computation.

File

[sys_dma.h](#)

C

```
typedef enum {
    SYS_DMA_CRC_WRITE_ORDER_MAINTAIN,
    SYS_DMA_CRC_WRITE_ORDER_CHANGE
} SYS_DMA_CRC_WRITE_ORDER;
```

Members

Members	Description
SYS_DMA_CRC_WRITE_ORDER_MAINTAIN	Source data is written to the destination unchanged regardless of the selected pre-CRC byte order.
SYS_DMA_CRC_WRITE_ORDER_CHANGE	Source data is written to the destination reordered according to the selected pre-CRC byte order.

Description

DMA channel CRC write order

This data type identifies write byte order selection post CRC computation.

Remarks

None.

SYS_DMA_TRANSFER_EVENT Enumeration

Enumeration of possible DMA System service events.

File

[sys_dma.h](#)

C

```
typedef enum {
    SYS_DMA_TRANSFER_EVENT_COMPLETE,
```

```
SYS_DMA_TRANSFER_EVENT_ERROR,  
SYS_DMA_TRANSFER_EVENT_ABORT  
} SYS_DMA_TRANSFER_EVENT;
```

Members

Members	Description
SYS_DMA_TRANSFER_EVENT_COMPLETE	Data was transferred successfully.
SYS_DMA_TRANSFER_EVENT_ERROR	Error while processing the request
SYS_DMA_TRANSFER_EVENT_ABORT	Data transfer was aborted.

Description

DMA System service Events

This data type provides an enumeration of all possible DMA System service events.

Remarks

None.

SYS_DMA_CHANNEL_HANDLE Type

Assigns the handle for requested channel.

File

[sys_dma.h](#)

C

```
typedef uintptr_t SYS_DMA_CHANNEL_HANDLE;
```

Description

DMA System Service Channel Handle

A channel handle value is returned by a call to the [SYS_DMA_ChannelAllocate](#) function. This handle is associated with the channel number that is used for data transfer and it allows the application to track the data transfer.

The channel handle once assigned to a client expires when the client calls [SYS_DMA_ChannelRelease](#). exits.

Remarks

None.

SYS_DMA_CHANNEL_IGNORE_MATCH Enumeration

Enables/Disables the feature to ignore a byte between a 2-byte match abort pattern.

File

[sys_dma.h](#)

C

```
typedef enum {  
    SYS_DMA_CHANNEL_IGNORE_MATCH_DISABLE,  
    SYS_DMA_CHANNEL_IGNORE_MATCH_ENABLE  
} SYS_DMA_CHANNEL_IGNORE_MATCH;
```

Members

Members	Description
SYS_DMA_CHANNEL_IGNORE_MATCH_DISABLE	Disabling the feature to ignore a byte between a 2-byte match abort pattern.
SYS_DMA_CHANNEL_IGNORE_MATCH_ENABLE	Enable the feature to ignore a byte between a 2-byte match abort pattern.

Description

DMA Channel ignore pattern match

This data type allows enabling/disabling the feature to ignore a byte between a 2-byte match abort pattern.

Remarks

This feature may not be available on all devices. Refer to the specific device data sheet to determine availability.

SYS_DMA_INIT Enumeration

Indicates the error information for the DMA System service.

File

[sys_dma.h](#)

C

```
typedef enum {  
    } SYS_DMA_INIT;
```

Description

DMA System service Error

This enumeration indicates the error information for the DMA System service.

Remarks

None.

SYS_DMA_SIDL Enumeration

Enable/Disable DMA operations in Idle mode.

File

[sys_dma.h](#)

C

```
typedef enum {  
    SYS_DMA_SIDL_DISABLE,  
    SYS_DMA_SIDL_ENABLE  
} SYS_DMA_SIDL;
```

Members

Members	Description
SYS_DMA_SIDL_DISABLE	When the CPU enters idle mode, the DMA module continue operations
SYS_DMA_SIDL_ENABLE	When the CPU enters idle mode, the DMA module will discontinue operations

Description

DMA stop in idle mode

This data type allows enabling/disabling of DMA operations in idle mode.

Remarks

The field 'sidl' (in init data structure 'SYS_DMA_INIT') is enable/disable while initializing the DMA module by calling [SYS_DMA_Initialize](#).

This feature may not be available on all devices. Refer to the specific device data sheet to determine availability.

SYS_DMA_CHANNEL_COUNT Macro

Identifies the maximum number of DMA channel handles to be defined.

File

[sys_dma.h](#)

C

```
#define SYS_DMA_CHANNEL_COUNT DMA_NUMBER_OF_CHANNELS
```

Description

DMA System Service Channel Count

This constant identifies the maximum number of DMA channel handles that should be defined.

Remarks

This value is derived from device-specific header files defined as part of the peripheral libraries.

SYS_DMA_CHANNEL_HANDLE_INVALID Macro

Defines an invalid channel handle.

File

[sys_dma.h](#)

C

```
#define SYS_DMA_CHANNEL_HANDLE_INVALID ((SYS_DMA_CHANNEL_HANDLE) (-1))
```

Description

DMA System Service Invalid Channel Handle

This is the definition of an invalid channel handle. An invalid buffer handle is returned by [SYS_DMA_ChannelAllocate](#) function if the channel number request was not successful.

Remarks

None.

Files

Files

Name	Description
sys_dma.h	DMA System Service.

Description

This section lists the source and header files used by the library.













sys_dma.h















DMA System Service.

Enumerations

	Name	Description
	SYS_DMA_CHANNEL_CHAIN_PRIO	Identifies the priority of chaining channel.
	SYS_DMA_CHANNEL_CRC_MODE	Identifies the operation mode of the CRC module.
	SYS_DMA_CHANNEL_IGNORE_MATCH	Enables/Disables the feature to ignore a byte between a 2-byte match abort pattern.
	SYS_DMA_CHANNEL_OP_MODE	Identifies the available DMA operating modes.
	SYS_DMA_CRC_WRITE_ORDER	Identifies altering/maintaining write order post CRC computation.
	SYS_DMA_INIT	Indicates the error information for the DMA System service.
	SYS_DMA_SIDL	Enable/Disable DMA operations in Idle mode.
	SYS_DMA_TRANSFER_EVENT	Enumeration of possible DMA System service events.

Functions

	Name	Description
	SYS_DMA_ChannelAbortEventSet	Sets an event source and enables cell transfer abort event for the same for the selected channel.
	SYS_DMA_ChannelAllocate	Allocates the specified DMA channel and returns a handle to it.
	SYS_DMA_ChannelCRCGet	Returns the computed CRC.
	SYS_DMA_ChannelCRCSet	Sets up the CRC operation mode.
	SYS_DMA_ChannelDestinationTransferredSizeGet	Returns the number of bytes transferred to destination.
	SYS_DMA_ChannelDisable	Disables a channel.
	SYS_DMA_ChannelEnable	Enables a channel.
	SYS_DMA_ChannelErrorGet	This function returns the error(if any) associated with the last client request.
	SYS_DMA_ChannelForceAbort	Force abort of transfer on the selected channel.
	SYS_DMA_ChannelForceStart	Force start of transfer on the selected channel.
	SYS_DMA_ChannelsBusy	Returns the busy status of the specified DMA channel.
	SYS_DMA_ChannelRelease	Deallocates and frees the channel specified by the handle.

	SYS_DMA_ChannelSetup	Setup the DMA channel parameters.
	SYS_DMA_ChannelSetupMatchAbortMode	Setup the pattern match abort mode.
	SYS_DMA_ChannelSourceTransferredSizeGet	Returns the number of bytes transferred from source.
	SYS_DMA_ChannelTransferAdd	Adds a data transfer to a DMA channel and Enables the channel to start data transfer.
	SYS_DMA_ChannelTransferEventHandlerSet	This function allows a DMA system service client to set an event handler.
	SYS_DMA_ChannelTransferSet	Sets up a data transfer to a DMA channel.
	SYS_DMA_Initialize	Initializes and Enables the DMA Controller.
	SYS_DMA_IsBusy	Returns the busy status of the DMA module.
	SYS_DMA_Resume	Resume DMA operations.
	SYS_DMA_Suspend	Suspend DMA transfers.
	SYS_DMA_Tasks	Maintains the system service's state machine.
	SYS_DMA_TasksError	Maintains the system service's error state machine.
	SYS_DMA_TasksErrorISR	Maintains the system service's state machine and implements its ISR.
	SYS_DMA_TasksISR	Maintains the system service's state machine and implements its ISR.

Macros

	Name	Description
	SYS_DMA_CHANNEL_COUNT	Identifies the maximum number of DMA channel handles to be defined.
	SYS_DMA_CHANNEL_HANDLE_INVALID	Defines an invalid channel handle.

Structures

	Name	Description
	SYS_DMA_CHANNEL_OPERATION_MODE_CRC	Defines the attributes for CRC operation mode.

Types

	Name	Description
	SYS_DMA_CHANNEL_HANDLE	Assigns the handle for requested channel.

Description

DMA System Service Library Interface Definition

This file contains the interface definition for the DMA System Service. It provides a way to interact with the DMA subsystem to manage the data transfer between different peripherals and/or memory without intervention from the CPU.

File Name

sys_dma.h

Company

Microchip Technology Inc.

File System Service Library

This section describes the File System Service Library.

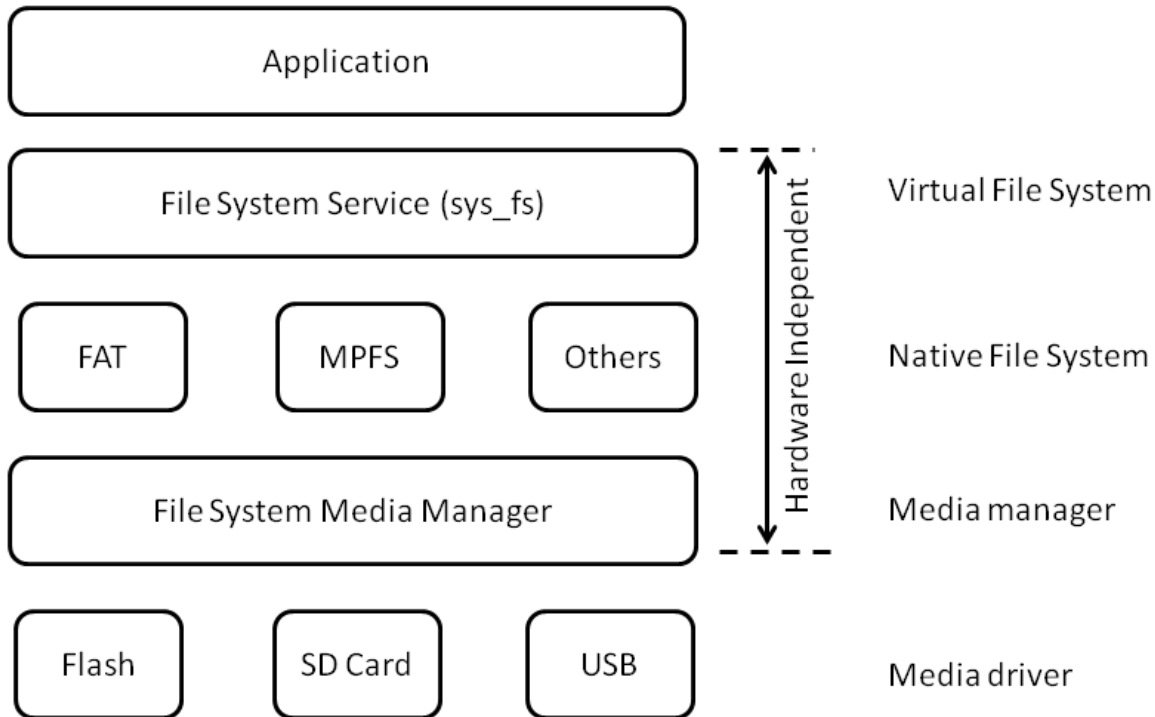
Introduction

Introduction to the MPLAB Harmony File System (FS).

Description

The MPLAB Harmony File System (FS) provides file system services to MPLAB Harmony based applications. The architecture of the File System Service is shown in the following figure.

File System Architecture




The File System Service provides an application programming interface (API) through which a utility or user program requests services of a file system. Some file system APIs may also include interfaces for maintenance operations, such as creating or initializing a file system and verifying the file system for integrity.

The File System service is really a framework designed to support multiple file systems (native file system) and multiple media in the same application. Examples of native file systems are FAT12, FAT16, FAT32, and the Microchip File System (MPFS) among others. MPFS is a read-only file system, which is used to store the files that do not change for example Web pages, images, etc. Each of these native file systems have a common set of APIs that can be used to access the files of that particular native file system.

The File System Service abstracts the native file system calls and provides a common interface to the user/application layer. For example, while the application layer requests for a file read or write from a disk, due to the presence of this abstraction, the application need not be bothered about the native file system implemented on that disk. Instead, the application can call the read/write API of the File System, which in turn translates to the read/write command of the native file system used on the required disk.

This simplifies the implementation of the higher application layer and also provides a mechanism to add more native file system to the File System framework in the future.

 **Note:** "File System Service" and "sys_fs" are synonymous.

File System Service Library Porting Guide

This section provides information for porting an existing File System (FS) application that uses legacy MLA code to MPLAB Harmony.

Description

Differences between the legacy MLA code and MPLAB Harmony are described and examples are provided in the following topics:

- [Initialization](#)
- [System Configuration](#)
- [Mounting a Volume](#)
- [Opening a File](#)
- [Reading a File](#)
- [Writing a File](#)
- [Closing a File](#)
- [File EOF](#)
- [File Tell](#)
- [File Seek](#)
- [SYS_FS_Tasks](#)

Comparison of API Names

This topic lists the API names for MLA versus the API name in MPLAB Harmony.

Description

API Name in MLA	API Name in MPLAB Harmony
N/A	SYS_FS_Mount
N/A	SYS_FS_Unmount
N/A	SYS_FS_FileError
FSInit	SYS_FS_Initialize
N/A	SYS_FS_Tasks
Fsfclose	SYS_FS_FileClose
FsfEOF	SYS_FS_FileEOF
FSfread	SYS_FS_FileRead
FSftell	SYS_FS_FileTell
FSfwrite	SYS_FS_FileWrite
N/A	SYS_FS_FileSize
wFSfopen	SYS_FS_FileOpen
N/A	SYS_FS_FileStat
FSfseek	SYS_FS_FileSeek

Initialization

This topic describes the differences in initialization.

Description

MLA Initialization (Legacy code):

In legacy MLA code, the initialization involves modification of the header file `HardwareProfile.h`. The header file has definitions for the SPI modules used (SPI1 or SPI2), configuration for SPI module, clock frequency, pin mapping for SD card, and pin remapping for SPI (SDI, SDO and SCLK pins). The following two images illustrate these modifications.

```

HardwareProfile.h
523
524     #elif defined (__PIC32MX__)
525         // Registers for the SPI module you want to use
526         #define MDD_USE_SPI_1
527         // #define MDD_USE_SPI_2
528
529         // SPI Configuration
530         #define SPI_START_CFG_1 (PRI_PRESCAL_64_1 | SEC_PRESCAL_8_1 | MASTER_ENABLE_ON | SPI_CKE_ON | SPI_SMP_ON)
531         #define SPI_START_CFG_2 (SPI_ENABLE)
532
533         // Define the SPI frequency
534         #define SPI_FREQUENCY (20000000)
535
536         #if defined MDD_USE_SPI_1
537             // Description: SD-SPI Chip Select Output bit
538             #define SD_CS LATBbits.LATB1
539             // Description: SD-SPI Chip Select TRIS bit
540             #define SD_CS_TRIS TRISBbits.TRISB1
541
542             // Description: SD-SPI Card Detect Input bit
543             #define SD_CD PORTFbits.RF0
544             // Description: SD-SPI Card Detect TRIS bit
545             #define SD_CD_TRIS TRISFbits.TRISF0
546
547             // Description: SD-SPI Write Protect Check Input bit
548             #define SD_WE PORTFbits.RF1
549             // Description: SD-SPI Write Protect Check TRIS bit
550             #define SD_WE_TRIS TRISFbits.TRISF1
551
552             // Description: The main SPI control register
553
554
555
556         // Description: The definition for the SPI baud rate generator register (PIC32)
557         #define SPIBRG SPI1BRG
558
559         // Tris pins for SCK/SDI/SDO lines
560
561         // Tris pins for SCK/SDI/SDO lines
562         #if defined (__32MX460F512L__)
563             // Description: The TRIS bit for the SCK pin
564             #define SPICLOCK TRISDbits.TRISD10
565             // Description: The TRIS bit for the SDI pin
566             #define SPIIN TRISCbits.TRISC4
567             // Description: The TRIS bit for the SDO pin
568             #define SPIOUT TRISDbits.TRISD0
569         #else // example: PIC32MX360F512L
570             // Description: The TRIS bit for the SCK pin
571             #define SPICLOCK TRISFbits.TRISF6
572             // Description: The TRIS bit for the SDI pin
573             #define SPIIN TRISFbits.TRISF7
574             // Description: The TRIS bit for the SDO pin
575             #define SPIOUT TRISFbits.TRISF8
576         #endif
577
578         // SPI library functions
579         #define putcSPI putcSPI1
580         #define getcSPI getcSPI1
581         #define OpenSPI(config1, config2) OpenSPI1(config1, config2)
582
583         #elif defined MDD_USE_SPI_2
584             // Description: SD-SPI Chip Select Output bit
585             #define SD_CS LATBbits.LATB9
586
587
588
589
590
591
592
593
594
595

```

Definitions for SPI modules to be used

Configuration of SPI module and SPI baud rate

Pin mapping for SD card

Device specific SPI pin remapping based on "#define"

SPI module number specific functions based on "#define"

Inside the main function, the initialization of interrupts and the clock occurs in a series of function calls. Then, the media detect function is called for the media used. Later, the FS init function is called, which internally performs the disk mount, and if the mount process and FS initialization is successful, the function returns as '1'.

```

177
178 int main (void)
179 {
180     FSFILE * pointer;
181     #if defined(SUPPORT_LFN)
182     char count = 80;
183     #endif
184     char * pointer2;
185     SearchRec rec;
186     unsigned char attributes;
187     unsigned char size = 0, i;
188
189     // Turn on the interrupts
190     INTEnableSystemMultiVectoredInt();
191     SYSTEMConfigPerformance(GetSystemClock());
192     mOSCSetPBDIV(OSC_PB_DIV_2);
193     //Initialize the RTCC
194     RtcInit();
195     while(RtcGetClkStat() != RTCC_CLK_ON); // wait for the SOSC to be actually running and RTCC to have its clock source
196     // could wait here at most 32ms
197     RtcOpen(0x10073000, 0x07011602, 0);
198
199     while (!MDD_MediaDetect());
200
201     // Initialize the library
202     while (!FSInit());
203
204     #ifndef ALLOW_WRITES
205     // Create a file
206     pointer = wfsopen ((const unsigned short int *) &str1[0], "w");

```

main() function

Clock and interrupt initialization

Media detection and FS initialization

MPLAB Harmony Initialization:

In the case of MPLAB Harmony, the initialization is done by the [SYS_Initialize](#) function, which is called from the main function.

Inside the [SYS_Initialize](#) function, the BSP, interrupt, clock, port remapping, driver initialization, FS initialization and application-specific initialization is done. All initialization is done by calling specific system function calls, as shown in the following image. For more information, refer to the demonstration code provided as a part of the MPLAB Harmony release package.

```

196
197 void SYS_Initialize ( void* data )
198 {
199     /* Initialize the BSP */
200     BSP_Initialize();
201
202     /* Initialize the interrupt system */
203     SYS_INT_Initialize();
204
205     /* Remap the SPI pins */
206     SYS_PORTS_RemapOutput(PORTS_ID_0, OUTPUT_FUNC_SDO2, OUTPUT_PIN_RPG8);
207     SYS_PORTS_RemapInput(PORTS_ID_0, INPUT_FUNC_SDO2, INPUT_PIN_RPG8);
208
209     /* set priority for SPI interrupt source */
210     SYS_INT_VectorPrioritySet(INT_VECTOR_SPI);
211     SYS_INT_VectorPrioritySet(INT_VECTOR_SPI);
212
213     /* set sub-priority for SPI interrupt */
214     SYS_INT_VectorSubprioritySet(INT_VECTOR_SPI);
215     SYS_INT_VectorSubprioritySet(INT_VECTOR_SPI);
216
217     /* Initialize the global interrupt */
218     SYS_INT_Enable();
219
220     /* Initialize the clock system service. This is used
221      * by the SPI Driver. */
222     clkObject.peripheralClock = 100000000;
223
224     /* Initialize the SPI driver */
225     appDrvObjects.drivSPIObject = DRV_SPI_Initialize(DRV_SPI_INDEX_0,
226     (SYS_MODULE_INIT *) &drvSPIInit);
227
228     /* Initialize the SDCARD driver */
229     appDrvObjects.drivSDCARDObject = DRV_SDCARD_Initialize(DRV_SDCARD_INDEX_0,
230     (SYS_MODULE_INIT *) &drvSDCARDInit);
231
232     /* Initialize the SYS_FS Layer */
233     SYS_FS_Initialize( (const void *) sysFSInit );
234
235     /* Initialize the Application */
236     APP_Initialize();
237
238
239 }

```

Initialization of BSP for specific demo board

Initialization of

- Ports
- Interrupt
- Clock

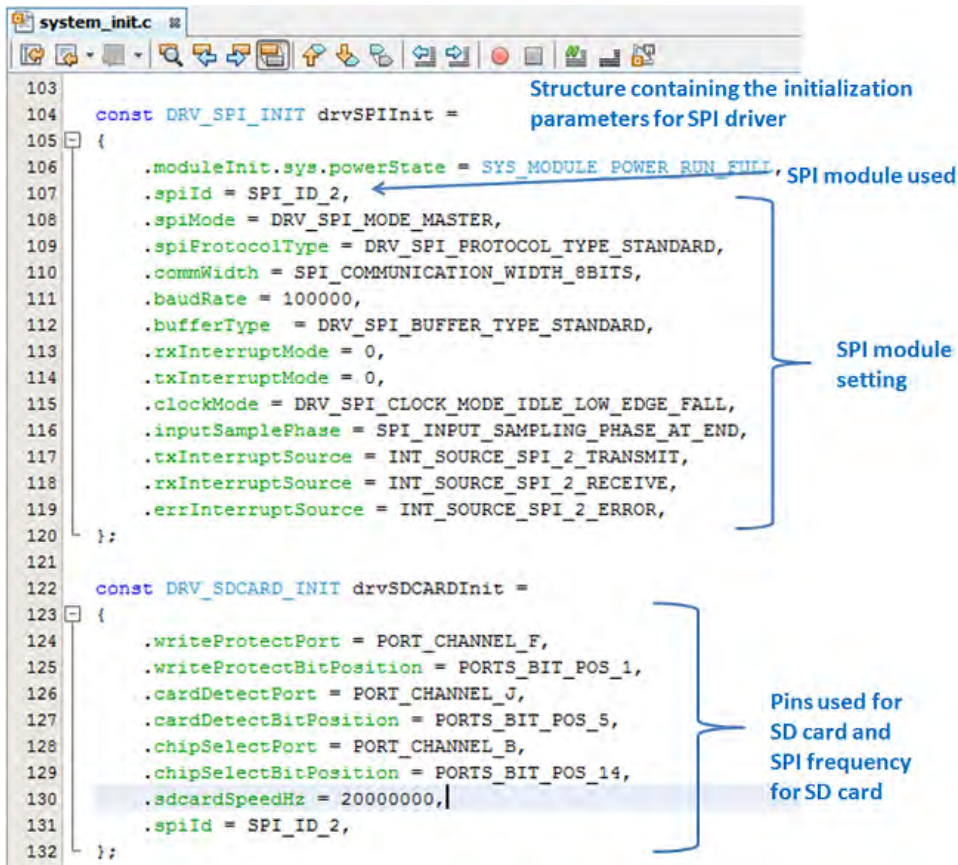
All initialization done by system calls. No "#defines"

Driver Initialization

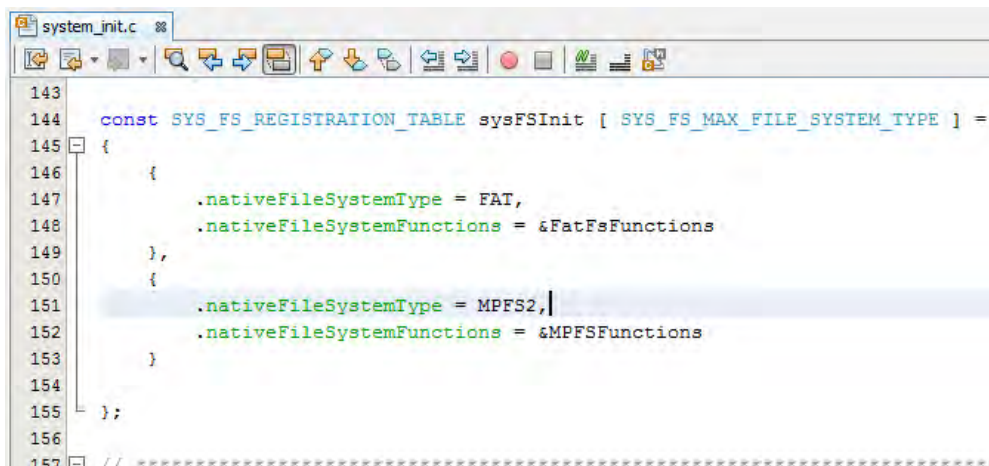
FS Initialization

Any application specific initialization

The SPI module selection and configuration of the SPI module is done through the SPI driver initialization structure. Similarly, the selection of pins related to SD card functionality and the selection of the SPI clock frequency is done through the SD card driver initialization structure. The initialization structures are passed as input parameters during driver initialization function calls. There are no #define used, as was done in the case of legacy MLA code.



Since MPLAB Harmony supports multiple file systems, there is a structure which needs to be defined and then passed as an input parameter to the SYS_FS_Init function.



System Configuration

This topic describes differences in system configuration.

Description

system_config.h file in MPLAB Harmony:

The file `system_config.h` contains the various system configurations required to run an application. Examples of system configurations related to, but not limited to drivers are:

- SPI driver configuration: such as `DRV_SPI_INSTANCES_NUMBER`, `DRV_SPI_CLIENTS_NUMBER`, `DRV_SPI_INTERRUPT_MODE`, `DRV_SPI_BUFFER_SIZE`, etc.
- SD Card driver configuration: such as `DRV_SDCARD_INSTANCES_NUMBER`, `DRV_SDCARD_CLIENTS_NUMBER`, `DRV_SDCARD_QUEUE_POOL_SIZE`, etc.

These configurations are required to configure the drivers used for the FS. Since the driver concept was not present in the legacy MLA code, this is a new addition to MPLAB Harmony. To get detailed information about each of the configuration parameter, please refer to the specific driver

documentation.

Media manager configuration:

- **SYS_FS_MEDIA_NUMBER** - Number of media that will be used in the application. For example, if the application uses SD card and Mass storage device the **SYS_FS_MEDIA_NUMBER** should be defined as 2.
- **SYS_FS_VOLUME_NUMBER** - Number of volumes that will be used in the application. MPLAB Harmony supports multi-partitioned media. If the application uses a SD card that has three partitions and a Mass storage device, which has one partition, the **SYS_FS_VOLUME_NUMBER** should be defined as 4.
- Clock related configurations are present in the file, which are used to set the clock for the device

File system related configuration:

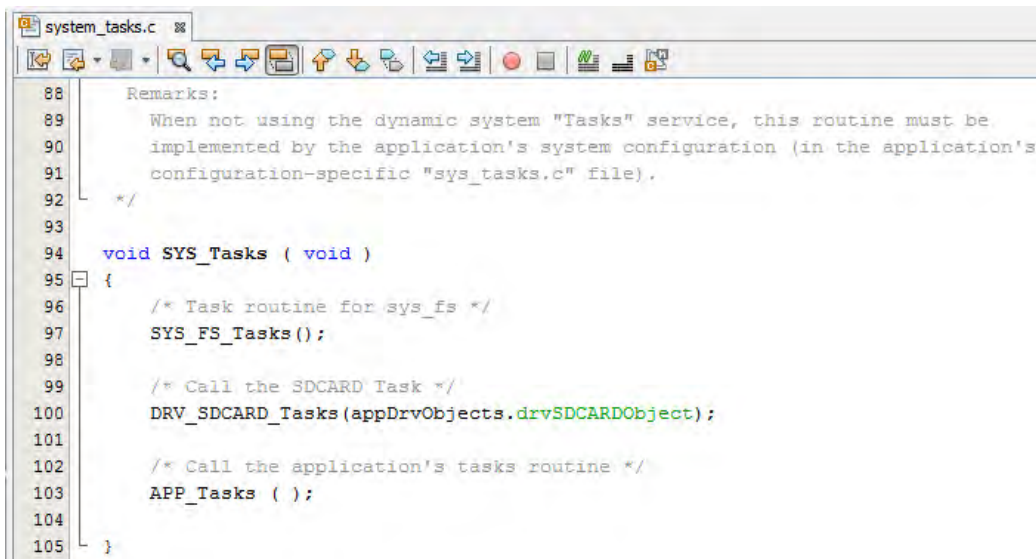
- **SYS_FS_MAX_FILE_SYSTEM_TYPE** - Number of native file system that will be used in the application. MPLAB Harmony supports multiple native file systems. If the application uses two file systems, FAT FS and MPFS2, **SYS_FS_MAX_FILE_SYSTEM_TYPE** should be defined as 2.
- **SYS_FS_MAX_FILES** - Maximum number of files that will be opened by the application at a time
- Other application-specific configuration settings

SYS_Tasks in MPLAB Harmony:

In the case of MPLAB Harmony, certain tasks need to be executed periodically. These tasks are executed from a common function called **SYS_Tasks**, which is in turn called from the main loop (while(1)) loop. While the concept of running certain functions from the main loop is not new, the legacy MLA demonstration code did not have any functions that had to run from the main loop. However, in case of MPLAB Harmony, it is mandatory that certain tasks should be running periodically from the main loop.

In the case of the FS application, tasks that have to run from **SYS_Tasks** function are:

- **SYS_FS_Tasks** - This task maintains the working of the SYS_FS layer and other file system related layers. It is extremely essential that this function runs periodically from the **SYS_Tasks** function.
- Driver task - Consider a case where the FS application uses a SD card; there is no hardware module that controls the SD card. In such a case, the task routine for the SD card must run from **SYS_Tasks**. This is required so that the internal driver mechanism for the SD card continues working.
- Any other application related task



```

88      Remarks:
89      When not using the dynamic system "Tasks" service, this routine must be
90      implemented by the application's system configuration (in the application's
91      configuration-specific "sys_tasks.c" file).
92      */
93
94      void SYS_Tasks ( void )
95      {
96          /* Task routine for sys_fs */
97          SYS_FS_Tasks();
98
99          /* Call the SDCARD Task */
100         DRV_SDCARD_Tasks(appDrvObjects.drvcSDCARDObject);
101
102         /* Call the application's tasks routine */
103         APP_Tasks ( );
104
105     }

```

Mounting a Volume

This topic describes differences when mounting a volume.

Description

Mounting a Volume in MLA versus MPLAB Harmony:

In MLA, the complete mounting of a volume was achieved by calling the two functions, MDD_MediaDetect and FSInit, until the function returns success. The features of mounting a volume in MLA were:

- There are two functions that completely achieve the mounting process
- Both functions were of input type as "void" (no input argument)
- Both functions were called in blocking mode

```

193 //Initialize the RTCC
194 RtccInit();
195 while(RtccGetClkStat()!=RTCC_CLK_ON); // wait for the SOSC to be actually runni
196 // could wait here at most 32ms
197 RtccOpen(0x10073000, 0x07011602, 0);
198
199 while (!MDD_MediaDetect());
200
201 // Initialize the library
202 while (!FSInit());
203
204 #ifdef ALLOW_WRITES
205 // Create a file
206 pointer = wFSfopen ((const unsigned short int *)&str1[0], "w");
207 if (pointer == NULL)
208     while(1);
209
210

```

In MPLAB Harmony, the mounting of a volume is achieved by calling the [SYS_FS_Mount](#) function until the function returns success. The features of mounting a volume in MPLAB Harmony are:

- [SYS_FS_Mount](#) function includes both media detection and mounting the volume. Please note that, MPLAB Harmony FS still needs the function [SYS_FS_Initialize](#) to be called during system initialization. Though the name [SYS_FS_Initialize](#) seems similar to the MLA function [FSInit](#), the task achieved by calling these functions are very different.
 - [SYS_FS_Initialize](#) (MPLAB Harmony function) - Just does the initialization of SYS_FS layer. It does not do any mounting of volumes.
 - [FSInit](#) (MLA function) - Did the initialization of FS and also did mount the disk.
- [SYS_FS_Mount](#) function accepts input parameter such as devName, mntName, fileType, etc. To know each of the parameter in detail, please refer to the documentation of the [SYS_FS_Mount](#) function.
- [SYS_FS_Mount](#) has to be running in a non-blocking mode. The following image shows an application where [SYS_FS_Mount](#) is called from the state machine implementation. This is required as the implementation of the state machine allows the [SYS_Tasks](#) function to be run periodically. This is quite different from the earlier implementations of MLA.

```

140
141 switch(appData.state)
142 {
143     case APP_MOUNT_DISK:
144         if(SYS_FS_Mount("/dev/mmcblk0", "/mnt/myDevice", FAT, 0, NULL) != 0)
145         {
146             /* The disk could not be mounted, Try
147              * mounting again until success. */
148
149             appData.state = APP_MOUNT_DISK;
150         }
151         else
152         {
153             /* Mount was successful. Unmount the disk, for testing */
154
155             appData.state = APP_UNMOUNT_DISK;
156         }
157         break;
158

```

Opening a File

This topic describes differences when opening a file.

Description

Opening a file in MLA versus MPLAB Harmony:

Opening a file remains similar for both MLA and MPLAB Harmony. However, a critical difference in MPLAB Harmony is that, while opening the file, the complete path of the file must be specified. In addition, the path must be preceded by the string `/mnt/`. The complete path is required because the MPLAB Harmony FS implements a multi-partition media support. Therefore, the file path should include the name of volume (assigned media partition), and from where the file is to be opened. For more details, please refer to the documentation of the [SYS_FS_FileOpen](#) function. The following two figures illustrate opening a file in MLA as opposed to MPLAB Harmony.

In MLA, the error in opening a file was indicated by the file open function returning a NULL, while, in MPLAB harmony, the file open returns [SYS_FS_HANDLE_INVALID](#).

```

244 #endif
245
246 // Open file 1 in read mode
247 pointer = wFSfopen ((const unsigned short int *)&str1[0], "r");
248 if (pointer == NULL)
249     while(1);
250
251 if (wFSrename ((const unsigned short int *)&str3[0], pointer))
252     while(1);
253
254 // Read one four-byte object
255 if (FSfread (receiveBuffer, 4, 1, pointer) != 1)
256     while(1);

```

Figure 1: Opening a File in MLA

```

173
174     case APP_OPEN_FILE:
175
176         appData.fileHandle = SYS_FS_FileOpen("/mnt/myDrive/FILE_TOO_LONG_NAME_EXAMPLE_123.JPG",
177             (SYS_FS_FILE_OPEN_READ));
178         if(appData.fileHandle == SYS_FS_HANDLE_INVALID)
179         {
180             /* Could not open the file. Error out*/
181             appData.state = APP_ERROR;
182         }
183         else
184         {
185             appData.fileHandle = SYS_FS_FileOpen("/mnt/myDrive/FILE_TOO_LONG_NAME_EXAMPLE_123.JPG",

```

Figure 2: Opening a File in MPLAB Harmony

Reading a File

This topic describes the differences between legacy MLA code for FS and MPLAB Harmony FS, while reading a file.

Description

Reading a file in MLA versus MPLAB Harmony:

Reading a file remains similar for both MLA and MPLAB Harmony, as depicted in the following figures. The smaller differences are that, the name of the function to read the file is different, the order of parameters passed are different, and MPLAB Harmony only takes three parameters. The MPLAB Harmony function only enables byte based access.

Although both the MLA and MPLAB Harmony file read functions return the number of bytes read, the MPLAB Harmony functions also return '-1' if there were any errors while reading the file.

Demonstration3.c

```

253
254     // Read one four-byte object
255     if (FSfread (receiveBuffer, 4, 1, pointer) != 1)
256         while(1);
257
258     // Check if this is the end of the file- it shouldn't be
259     if (FSfeof (pointer))

```

Figure 1: Reading a File in MLA

```

201
202     if(SYS_FS_FileRead((void *)appData.data, 512, appData.fileHandle) == -1)
203     {
204         /* There was an error while reading the file.
205         * Close the file and error out. */
206
207         SYS_FS_FileClose(appData.fileHandle);
208         appData.state = APP_ERROR;
209     }

```

Figure 2: Reading a File in MPLAB Harmony

Writing a File

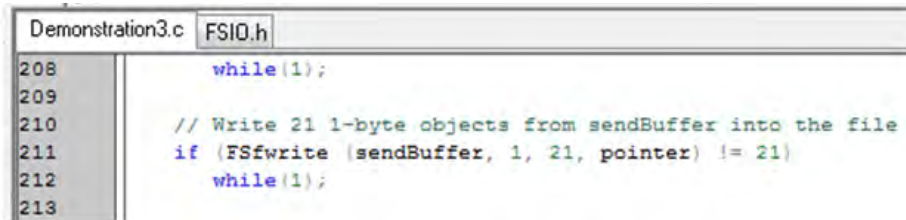
This topic describes the differences between legacy MLA code for FS and MPLAB Harmony FS, while writing to a file.

Description

Writing to a file in MLA versus MPLAB Harmony:

Writing to a file remains similar for both MLA and MPLAB Harmony, as depicted in the following figures. The smaller differences are that the name of the function to write to the file is different, the order of parameters passed are different, and MPLAB Harmony only takes three parameters. The MPLAB Harmony function only enables byte based access.

Although both the MLA and MPLAB Harmony file read functions return the number of bytes written, the MPLAB Harmony functions also return '-1' if there were any errors while writing to the file.

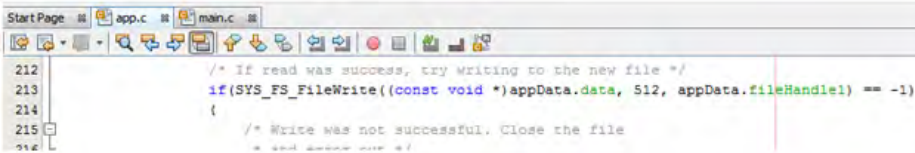


```

208     while(1);
209
210     // Write 21 1-byte objects from sendBuffer into the file
211     if (FSfwrite (sendBuffer, 1, 21, pointer) != 21)
212         while(1);
213

```

Figure 1: Writing to a File in MLA



```

212     /* If read was success, try writing to the new file */
213     if(SYS_FS_FileWrite((const void *)appData.data, 512, appData.fileHandle1) == -1)
214     {
215         /* Write was not successful. Close the file
216         * and return error if

```

Figure 2: Writing to a File in MPLAB Harmony

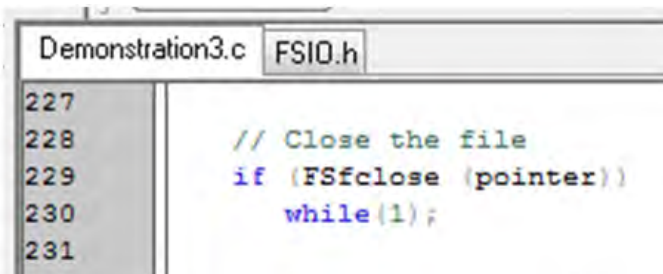
Closing a File

This topic describes the differences between legacy MLA code for FS and MPLAB Harmony FS, while closing a file.

Description

Closing a file in MLA versus MPLAB Harmony:

Closing a file remains similar for both MLA and MPLAB Harmony, as depicted in the following figures. The smaller difference is that the name of the function is different.

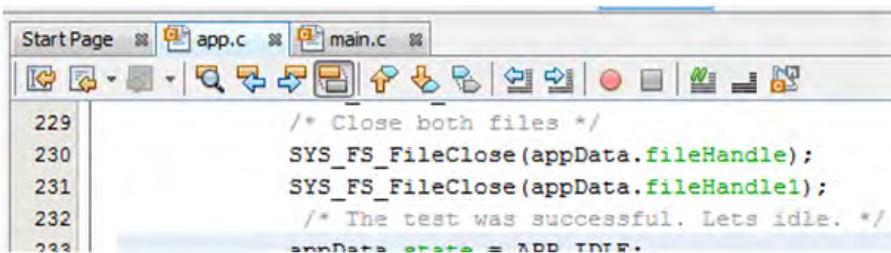


```

227
228     // Close the file
229     if (FSfclose (pointer))
230         while(1);
231

```

Figure 1: Closing a File in MLA



```

229     /* Close both files */
230     SYS_FS_FileClose (appData.fileHandle);
231     SYS_FS_FileClose (appData.fileHandle1);
232     /* The test was successful. Lets idle. */
233     appData.state = APP_IDLE;

```

Figure 2: Closing a File in MPLAB Harmony

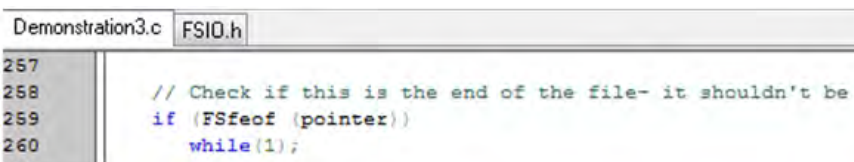
File EOF

This topic describes the differences between legacy MLA code for FS and MPLAB Harmony FS, while checking for end of file.

Description

Checking for EOF in MLA versus MPLAB Harmony:

The EOF function remains similar for both MLA and MPLAB Harmony, as depicted in the following figures. The smaller difference is that the name of the function is different.

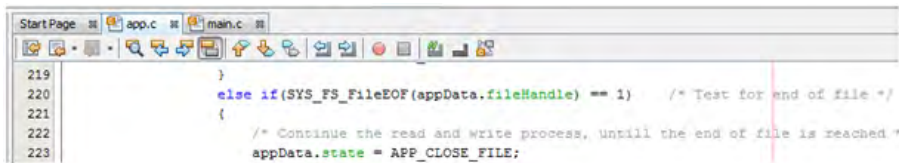


```

257
258     // Check if this is the end of the file- it shouldn't be
259     if (FSfeof (pointer))
260         while(1);
261

```


Figure 1: EOF in MLA



```

219     }
220     else if(SYS_FS_FileEOF(appData.fileHandle) == 1) /* Test for end of file */
221     {
222         /* Continue the read and write process, untill the end of file is reached */
223         appData.state = APP_CLOSE_FILE;

```

Figure 2: EOF in MPLAB Harmony

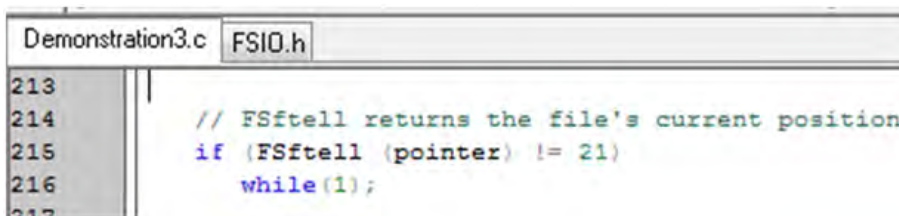
File Tell

This topic describes the differences between legacy MLA code for FS and MPLAB Harmony FS, while checking for file pointer (tell).

Description

Checking for file pointer (tell) in MLA versus MPLAB Harmony:

The function to check for file pointer (tell) remains similar for both MLA and MPLAB Harmony, as depicted in the following figures. The smaller difference is that the name of the function is different.

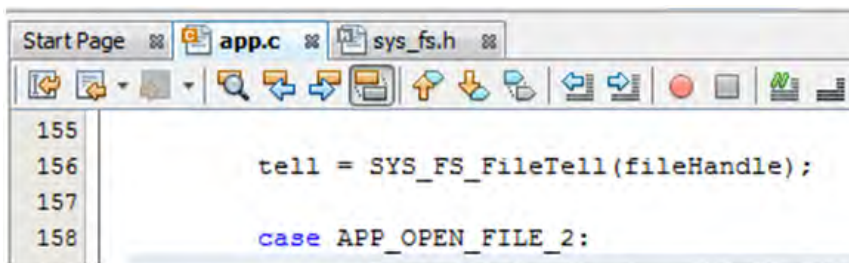


```

213
214     // FSftell returns the file's current position
215     if (FSftell(pointer) != 21)
216         while(1);
217

```

Figure 1: Tell Function in MLA



```

155
156     tell = SYS_FS_FileTell(fileHandle);
157
158     case APP_OPEN_FILE_2:

```

Figure 2: Tell Function in MPLAB Harmony

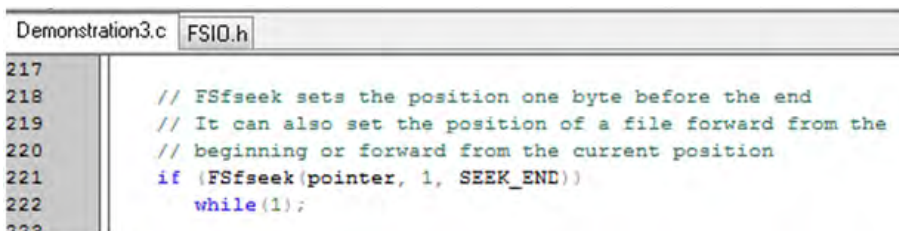
File Seek

This topic describes the differences between legacy MLA code for FS and MPLAB Harmony FS, while performing a file seek.

Description

File seek in MLA versus MPLAB Harmony:

The function to perform file seek remains similar for both MLA and MPLAB Harmony, as depicted in the following figures. The smaller difference is that the name of the function is different.

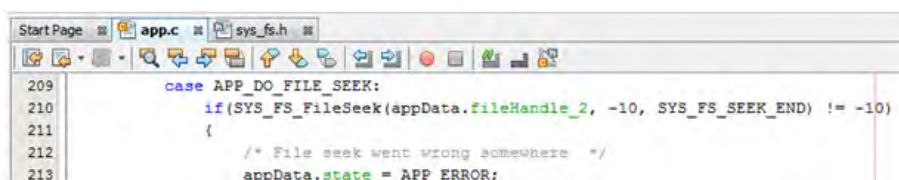


```

217
218     // FSfseek sets the position one byte before the end
219     // It can also set the position of a file forward from the
220     // beginning or forward from the current position
221     if (FSfseek(pointer, 1, SEEK_END))
222         while(1);
223

```

Figure 1: File Seek in MLA



```

209     case APP_DO_FILE_SEEK:
210         if(SYS_FS_FileSeek(appData.fileHandle_2, -10, SYS_FS_SEEK_END) != -10)
211         {
212             /* File seek went wrong somewhere */
213             appData.state = APP_ERROR;

```

Figure 2: File Seek in MPLAB Harmony

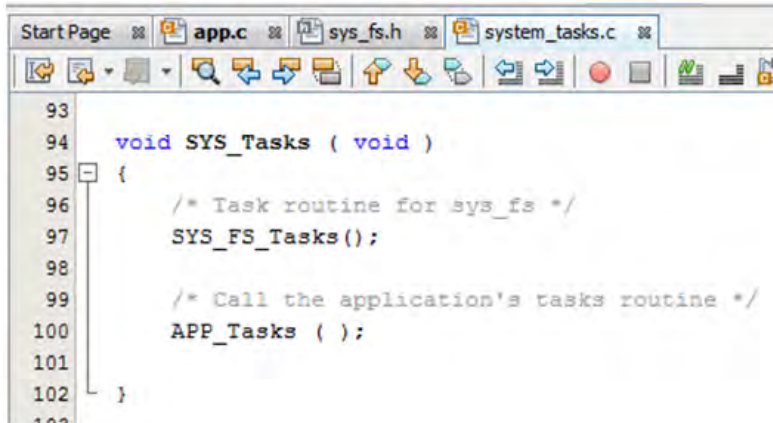
SYS_FS_Tasks

This topic describes the differences between legacy MLA code for FS and MPLAB Harmony FS with respect to the [SYS_FS_Tasks](#) function.

Description

SYS_FS_Tasks in MLA Versus MPLAB Harmony:

MPLAB Harmony needs the [SYS_FS_Tasks](#) function to be running periodically from the [SYS_Tasks](#) function. The MLA code did not have any such function that had to be running periodically.

A screenshot of the MPLAB Harmony IDE showing the `system_tasks.c` file. The code defines the `SYS_Tasks` function, which calls `SYS_FS_Tasks` and `APP_Tasks`.

```
93
94 void SYS_Tasks ( void )
95 {
96     /* Task routine for sys_fs */
97     SYS_FS_Tasks();
98
99     /* Call the application's tasks routine */
100    APP_Tasks ( );
101
102 }
```

Figure 1: Running [SYS_FS_Tasks](#) Function from [SYS_Tasks](#)

Using the Library

This topic describes the basic architecture of the File System Service Library and provides information and examples on its use.

Description

Interface Header File: [sys_fs.h](#)

The interface to the File System Service library is defined in the [sys_fs.h](#) header file. This file is included by the [system.h](#) file. Any C language source (.c) file that uses the File System Service Library should include [system.h](#).

Please refer to the What is MPLAB Harmony? section for how the File System Service interacts with the framework.

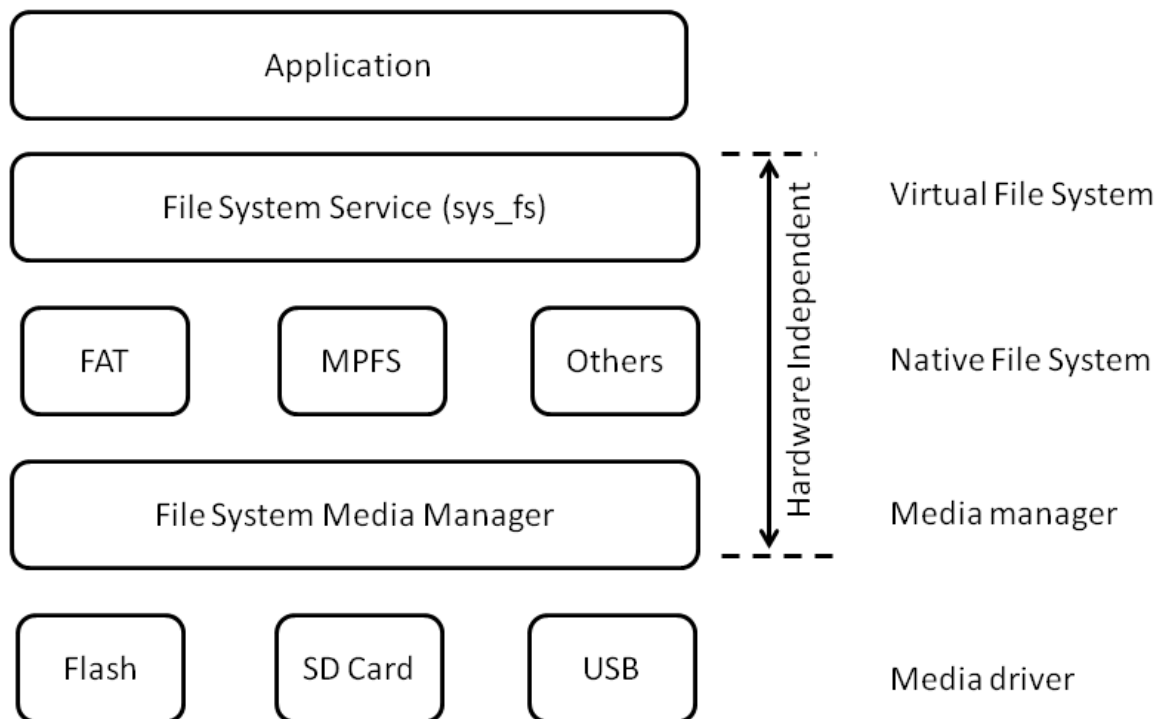
Abstraction Model

This topic describes the abstraction model of the MPLAB Harmony File System.

Description

The FS framework features a modular and layered architecture, as shown in the following figure.

FS Framework Architecture



As seen in the figure, the FS Framework consists of the following major blocks:

- The **Driver** for the physical media has to be included as a part of the FS Framework. This layer provides a low-level interface to access the physical media. This layer also enables multiple instances of media. Examples of drivers are:
 - NVM driver – To access files using NVM (Flash memory)
 - SPI driver – To access files from SD card, which interfaces using the SPI peripheral
- The **Media driver** provides a mechanism to access the media as "sectors". Sectors are the smallest storage element accessed by a file system and are contiguous memory locations. Typically, each sector has 512 bytes. Depending on the requirement, in some cases, the driver and media driver could be combined as one layer.
- The **Media manager** implements a disk and sector based media access mechanism. It also performs disk allocated/deallocated on media attach/detach. Due to the implementation of this layer, the FS Framework can support multiple disks. The media manager detects and analyzes a media based on its Master Boot Record (MBR). Therefore, it is mandatory for the media to have a MBR for it to work with the FS.
- The **Native file system** implements support for the media file system format. Examples of native file systems are: FAT12, FAT32, and MPFS, among other. At present, only the FAT and MPFS file systems are supported by the FS framework; however, more native file systems can be included.
- The **Virtual file system (or SYS_FS)** layer provides a file system independent file system operation interface. This layer translates virtual file systems calls to native file system calls. Due to this layer, applications can now support multiple file systems. Interfaces provided by this layer, but not limited to, include:
 - SYS_FS_mount
 - SYS_FS_open
 - SYS_FS_read
 - SYS_FS_write
 - SYS_FS_close

Library Overview

Please refer to the [System Service Introduction](#) for a detailed description of MPLAB Harmony system services.

The library interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the File System module.

Library Interface Section	Description
File and Directory Operation Functions	Provides file and directory operation interface routines.
General Operation Functions	Provides general operation interface routines.

How the Library Works

This topic provides information on how the MPLAB Harmony File System works.

Description

The MPLAB Harmony File System (FS) provides embedded application developers with a file system framework for retrieving and storing data from various media.

The MPLAB Harmony file system is designed to support multiple file systems (native file systems) and multiple media at the same time. Examples of native file systems are FAT12, FAT32, MPFS, and JFS, among others. Each of these native file systems has a common set of APIs that can be used to access the files of that particular native file system. The FS is a part of the MPLAB Harmony installation and is accompanied by demonstration applications that highlight usage. These demonstrations can also be modified or updated to build custom applications.

FS features include the following:

- Support for multiple file system (FAT, MPFS)
- Supports multiple physical media (NVM, SD card)
- More physical media can be interfaced with the FS, once the driver is available for the media
- Modular and Layered architecture

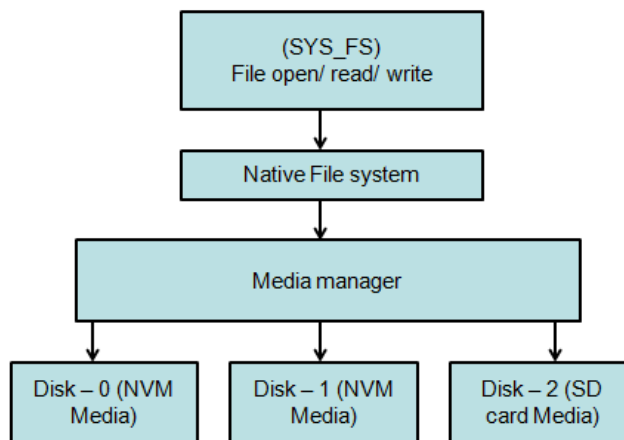
Application Interaction

This topic describes how an application must interact with the File System.

Description

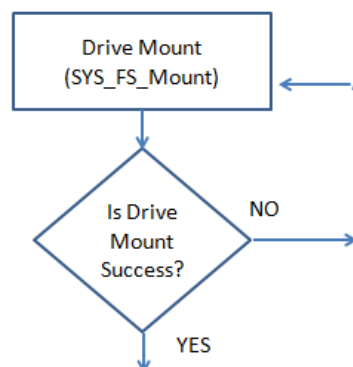
The interaction of various layers is shown in the following figure.

Application Interaction with FS Framework



In the process of using the FS Framework, the application must first **mount** the media drive for the FS Framework to access the media. Unless the mounting process returns successfully, the application should continue trying to mount the drive. If the drive is not attached, the mounting process will fail. In such a situation, the application should not proceed further unless the mounting is success.

Application Mounts a Drive

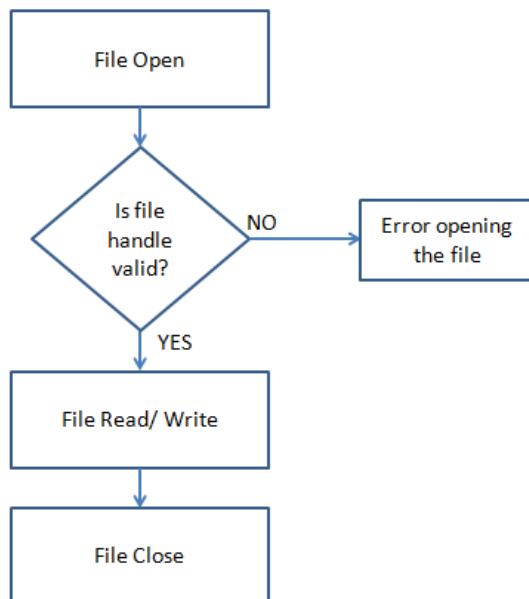


Once the drive is mounted, the application code can then **open the file** from the drive with different attributes (such as read-only or write). If the file open returns a valid handle, the application can proceed further. Otherwise, the application will enter an error state. The reason for an invalid

handle could be that the application was trying to read a file from the drive that does not exist. Another reason for an invalid handle is when the application tries to write to a drive that is write-protected.

Once the file is opened, the valid file handle is further used to **read/write** data to the file. Once the required operation is performed on the file, the file can then be **closed** by the application by passing the file handle. The following figure illustrates the process.

Further File System Operations



Using the File System

This topic describes how to use the File System.

Description

Use the Available Library Demonstration Applications

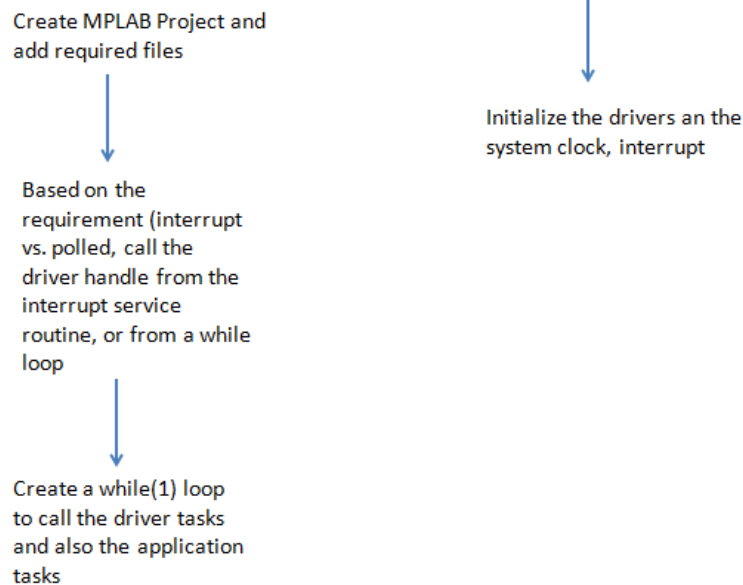
The FS framework release package contains a set of demonstration applications that are representative of common scenario (single/multi-media and single/multi-native file systems). These demonstrations can be easily modified to include application-specific initialization and application logic. The application logic must be non-blocking and could be implemented as a state machine.

- The application specific initialization can be called in the [SYS_Initialize](#) function (in the `system_init.c` file). The [SYS_Initialize](#) function is called when the device comes out of Power-on Reset (POR).
- The application logic can be called in the [SYS_Tasks](#) function (in the `system_tasks.c` file). The application logic can interact with the FS layer by using relevant API calls, as provided in the `APP_Tasks` (in the `app.c` file)

Building a FS Application from Scratch

In a case where the available demonstration applications do not meet the end application requirements, an application to use the FS framework can be created from scratch. The following figure shows a flowchart for the steps that need to be performed.

Steps to Create a FS Application

**Step 1:**

Create a MPLAB Project and add the required FS framework files to the project. The following files are needed to build the a FS project

- `system_config.h` - This file should contain the compile time configuration macros for the Driver, Media Layer, and FS layer. The file also contains the clock speed setting, which is set for the microcontroller.
- `system_init.c` - This file should contain the initial settings for each driver. It should also call the functions required to initialize different drivers to be used by the FS.
- `ff.c`, `diskio.c`, `mpfs.c`, `sys_fs.c`, `sys_fs_media_manager.c` - These files are part of the FS, which must be included in the project
- `sys_int_pic32.c` and `plib_int_pic32.c` - These files implement the system interrupt service that is required by the FS
- `sys_ports.c` - If the FS is using a SD card as media, this file needs to be included (for Chip Select)
- Driver - The driver for media to be used by FS should also be included
- Application specific files - These file will implement the application logic

Step 2:

Since the MPLAB Harmony drivers included with the File System operate in interrupt mode, a driver Handler should be defined as follows:

```

/* Use this for PIC32MX */
void __ISR ( _SPI1_VECTOR, ipl4 ) _InterruptHandler_SPI_stub ( void )
{
    DRV_SPI_Tasks( (SYS_MODULE_OBJ)appDrvObjects.drvSPIObject);
}

/* Use this for PIC32MZ */
void __ISR ( _SPI2_RX_VECTOR, ipl4 ) _InterruptHandler_SPI_RX_stub ( void )
{
    DRV_SPI_Tasks( (SYS_MODULE_OBJ)appDrvObjects.drvSPIObject);
}

void __ISR ( _SPI2_TX_VECTOR, ipl4 ) _InterruptHandler_SPI_TX_stub ( void )
{
    DRV_SPI_Tasks( (SYS_MODULE_OBJ)appDrvObjects.drvSPIObject);
}
  
```

Step 3:

The application should create a `while(1)` loop that continuously updates the driver layer State Machine and the application state machine. This requires the application state machine to be non-blocking.

/ This while(1) loop will continuously update the driver layer state machine and the application state machine */*

```

while(1)
{
    /* Task routine for sys_fs */
    SYS_FS_Tasks();

    /* Call the SDCARD Task */
    DRV_SDCARD_Tasks(appDrvObjects.drvSDCARDObject);
}
  
```

```

    /* Call the application's tasks routine */
    APP_Tasks ( );

}

```

Step 4:

If interrupt-based operation is needed, the interrupts need to be enabled first. The application should then initialize the driver layer. Refer to the driver specific documentation regarding usage.

```

/* Initialize the interrupt system */

SYS_INT_Initialize();

/* Initialize the global interrupts */
SYS_INT_Enable();

/* set priority for SPI interrupt source */
SYS_INT_VectorPrioritySet(INT_VECTOR_SPI1, INT_PRIORITY_LEVEL3);

/* set sub-priority for SPI interrupt source */
SYS_INT_VectorSubprioritySet(INT_VECTOR_SPI1, INT_SUBPRIORITY_LEVEL3);

/* Initialize the global interrupts */
SYS_INT_Enable();

/* Initialize the SPI driver */
appDrvObjects.drvSPIObject = DRV_SPI_Initialize(DRV_SPI_INDEX_0,
    (SYS_MODULE_INIT *)&drvSPIInit);

/* Initialize the SDCARD driver*/
appDrvObjects.drvSDCARDObject = DRV_SDCARD_Initialize(DRV_SDCARD_INDEX_0,
    (SYS_MODULE_INIT *)&drvSDCARDInit);

```

Step 5:

The application code can be implemented as a non-blocking state machine inside the APP_TASKS function, as shown in the following example for an application to read a file and write the content into another newly created file.

The input file TEST.JPG is not provided with the release package. It could be any arbitrary JPEG file chosen by the user, and then suitably renamed to TEST1.JPG. The reason for choosing a JPEG file for test purposes is that the duplicate file TEST1.JPG created by the FS demonstration could be easily verified for correctness by inserting the SD card in the computer and opening the TEST1.JPG file.

```

void APP_Tasks ( void )
{
    /* The application task state machine */

    switch(appData.state)
    {
        case APP_MOUNT_DISK:
            if(SYS_FS_Mount("/dev/mmcblk1", "/mnt/myDrive", FAT, 0, NULL) != 0)
            {
                /* The disk could not be mounted. Try
                 * mounting again until success. */

                appData.state = APP_MOUNT_DISK;
            }
            else
            {
                /* Mount was successful. Open a file.
                 * Let the switch case fall through. */
                appData.state = APP_OPEN_FILE;
            }
            break;

        case APP_OPEN_FILE:

            appData.fileHandle = SYS_FS_FileOpen("/mnt/myDrive/TEST.JPG",
                (FA_READ));
            if(appData.fileHandle == SYS_FS_HANDLE_INVALID)
            {
                /* Could not open the file. Error out*/
                appData.state = APP_ERROR;
            }
    }
}

```

```

else
{
    appData.fileHandle1 = SYS_FS_FileOpen("/mnt/myDrive/TEST1.JPG",
        (FA_WRITE|FA_CREATE_ALWAYS));
    if(appData.fileHandle == SYS_FS_HANDLE_INVALID)
    {
        /* Could not open the file. Error out*/
        appData.state = APP_ERROR;
    }
    else
    {
        /* Check the file to be read */
        appData.state = APP_CHECK_FILE;
    }
}
break;

case APP_CHECK_FILE:
    /* check the size of file */
    fileSize = SYS_FS_FileSize(appData.fileHandle);
    /* since, we will read 512 bytes at a time, find the number of times, the read has to be
    performed */
    sectorCounter = integralSector = (fileSize/512);
    /* find the remaining bytes */
    balanceSector = (fileSize%512);

    appData.state = APP_READ_WRITE_TO_FILE;

    break;

case APP_READ_WRITE_TO_FILE:

    if(SYS_FS_FileRead((void *)appData.data, 512, appData.fileHandle) == -1)
    {
        /* There was an error while reading the file.
        * Close the file and error out. */

        SYS_FS_FileClose(appData.fileHandle);
        appData.state = APP_ERROR;
    }
    else if(SYS_FS_FileWrite((const void *)appData.data, 512, appData.fileHandle1) == -1)
    {
        /* Write was not successful. Close the file
        * and error out.*/
        SYS_FS_FileClose(appData.fileHandle1);
        appData.state = APP_ERROR;
    }
    else
    {
        sectorCounter--;
        /* if entire integral sectors are written, write the balance sector*/
        if(sectorCounter == 0)
        {
            if(SYS_FS_FileRead((void *)appData.data, balanceSector, appData.fileHandle) == -1)
            {
                /* There was an error while reading the file.
                * Close the file and error out. */

                SYS_FS_FileClose(appData.fileHandle);
                appData.state = APP_ERROR;
            }
            else if(SYS_FS_FileWrite((const void *)appData.data, balanceSector,
appData.fileHandle1) == -1)
            {
                /* Write was not successful. Close the file
                * and error out.*/
                SYS_FS_FileClose(appData.fileHandle1);
                appData.state = APP_ERROR;
            }
        }
    }
}

```



```
        else
        {
            appData.state = APP_CLOSE_FILE;
        }
    }
}
break;

case APP_CLOSE_FILE:
    /* Close both files */
    SYS_FS_FileClose(appData.fileHandle);
    SYS_FS_FileClose(appData.fileHandle1);
    /* The test was successful. Lets idle. */
    appData.state = APP_IDLE;
    break;

case APP_IDLE:
    /* The application comes here when the demonstration
       has completed successfully. Switch on green LED. */
    BSP_SwitchONLED(LED_2);
    break;
case APP_ERROR:
    /* The application comes here when the demonstration
       has failed. Switch on the red LED.*/
    BSP_SwitchONLED(LED_1);
    break;
default:
    break;
}

} //End of APP_Tasks
```

Configuring the Library

Lists and describes the library configuration macros.

Macros

	Name	Description
	SYS_FS_AUTOMOUNT_ENABLE	Enable/Disable Auto Mount Feature of File system
	SYS_FS_MAX_FILE_SYSTEM_TYPE	Number of file systems used in the application
	SYS_FS_MAX_FILES	Number of simultaneous files access
	SYS_FS_MEDIA_MAX_BLOCK_SIZE	Media Sector Size information
	SYS_FS_MEDIA_NUMBER	Number of media used in the application
	SYS_FS_VOLUME_NUMBER	This macro defines number of volumes used in the application

Description

The configuration of the File System Service is based on the file `system_config.h`. This header file contains the configuration selection for the File System service. Based on the selections made, the File System Service may support the selected features. These configuration settings will apply to all instances of the File System Service. This header can be placed anywhere; however, the path of this header needs to be present in the include search path for a successful build. Refer to the Applications Help section for more details.

SYS_FS_AUTOMOUNT_ENABLE Macro

Enable/Disable Auto Mount Feature of File system

File

[sys_fs_config_template.h](#)

C

```
#define SYS_FS_AUTOMOUNT_ENABLE 1
```

Description

Enable/Disable Auto Mount Feature of File system
Enable/Disable Auto Mount Feature of File system

Remarks

None.

SYS_FS_MAX_FILE_SYSTEM_TYPE Macro

Number of file systems used in the application

File

[sys_fs_config_template.h](#)

C

```
#define SYS_FS_MAX_FILE_SYSTEM_TYPE 1
```

Description

Number of File system types
Number of fil systems used in the application

Remarks

None.

SYS_FS_MAX_FILES Macro

Number of simultaneous files access

File

[sys_fs_config_template.h](#)

C

```
#define SYS_FS_MAX_FILES 1
```

Description

Number of simultaneous files access
Number of simultaneous files access

Remarks

None.

SYS_FS_MEDIA_MAX_BLOCK_SIZE Macro

Media Sector Size information

File

[sys_fs_config_template.h](#)

C

```
#define SYS_FS_MEDIA_MAX_BLOCK_SIZE 512
```

Description

Media Sector Size information
Media Sector Size information

Remarks

None.

SYS_FS_MEDIA_NUMBER Macro

Number of media used in the application

File

[sys_fs_config_template.h](#)

C

```
#define SYS_FS_MEDIA_NUMBER 1
```

Description

Number of media used in the application

Number of media used in the application

Remarks

None.

SYS_FS_VOLUME_NUMBER Macro

This macro defines number of volumes used in the application

File

[sys_fs_config_template.h](#)

C

```
#define SYS_FS_VOLUME_NUMBER 1
```

Description

Number of Volumes

This macro defines the number of volumes used in the application

Remarks

None.

Building the Library

This section lists the files that are available in the File System Service Library.

Description

The following three tables list and describe the header (.h) and source (.c) files that implement this library. The parent folder for these files is <install-dir>/framework/system/fs.

Interface File(s)



All of the required files listed in the following table are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.

This table lists and describes the header files that must be included (i.e., using #include) by any code that uses this library.

Source File Name	Description
/sys_fs.h	This file contains function and type declarations required to Interact with the MPLAB Harmony File System Framework.
/fat_fs/src/file_system/ff.h	FAT File System module include file. This file should be included when using the FAT File System.
/mpfs/mpfs.h	This file contains the interface definition for handling the Microchip File System (MPFS). This file should be included when using MPFS.

Required File(s)

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

Source File Name	Description
/src/dynamic/sys_fs.c	This file contains implementation of File System interfaces.
/src/dynamic/sys_fs_media_manager.c	This file contains implementation File System Media Manager functions.
/fat_fs/src/file_system/ff.c	This file implements the FAT File system functions. This file should be included when using FAT File System.
/fat_fs/src/hardware_access/diskio.c	Low-level disk I/O module for FAT File System. This file should be included when using FAT File System.
/mpfs/src/mpfs.c	This file implements the MPFS functions. This file should be included when using MPFS.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
N/A	There are no optional files for this library.


























Module Dependencies







The File System Service Library depends on the following modules:

- NVM Driver Library (if media is NVM)
- Secure Digital (SD) Card Driver Library (if media is a SD Card)
- SPI Driver Library (if media is a SD Card)
- [Device Control System Service Library](#)
- [Interrupt System Service Library](#)
- [Ports System Service Library](#)
- [Clock System Service Library](#)













Library Interface

a) File and Directory Operation Functions

	Name	Description
	SYS_FS_FileClose	Closes a file.
	SYS_FS_FileEOF	Checks for end of file.
	SYS_FS_FileOpen	Opens a file.
	SYS_FS_FileSeek	Moves the file pointer by the requested offset.
	SYS_FS_FileSize	Returns the size of the file in bytes.
	SYS_FS_FileTell	Obtains the file pointer position.
	SYS_FS_FileNameGet	Reads the file name.
	SYS_FS_FilePrintf	Writes a formatted string into a file.
	SYS_FS_FileTestError	Checks for errors in the file.
	SYS_FS_FileDirectoryModeSet	Sets the mode for the file or directory.
	SYS_FS_FileDirectoryRemove	Removes a file or directory.
	SYS_FS_FileDirectoryRenameMove	Renames or moves a file or directory.
	SYS_FS_FileDirectoryTimeSet	Sets or changes the time for a file or directory.
	SYS_FS_FileTruncate	Truncates a file
	SYS_FS_FileCharacterPut	Writes a character to a file.
	SYS_FS_FileStringGet	Reads a string from the file into a buffer.
	SYS_FS_FileStringPut	Writes a string to a file.
	SYS_FS_Error	Returns the last error.
	SYS_FS_FileError	Returns the file specific error.
	SYS_FS_FileRead	Read data from the file.
	SYS_FS_FileWrite	Writes data to the file.
	SYS_FS_CurrentWorkingDirectoryGet	Gets the current working directory
	SYS_FS_DirectoryChange	Changes to a the directory specified.
	SYS_FS_DirectoryMake	Makes a directory.
	SYS_FS_DirClose	Closes an opened directory.

	SYS_FS_DirOpen	Open a directory
	SYS_FS_DirRead	Reads the specified directory.
	SYS_FS_DirSearch	Searches for a file or directory.
	SYS_FS_FileSync	Flushes the cached information when writing to a file.
	SYS_FS_DirRewind	Rewinds to the beginning of the directory.
	SYS_FS_FileStat	Gets file status.

b) General Operation Functions

	Name	Description
	SYS_FS_Initialize	Initializes the file system abstraction layer (sys_fs layer).
	SYS_FS_Mount	Mounts the file system.
	SYS_FS_Tasks	Maintains the File System tasks and functionalities.
	SYS_FS_Unmount	Unmounts the file system.
	SYS_FS_CurrentDriveGet	Gets the current drive
	SYS_FS_CurrentDriveSet	Sets the drive.
	SYS_FS_DriveLabelGet	Gets the drive label.
	SYS_FS_DriveLabelSet	Sets the drive label
	SYS_FS_DriveFormat	Formats a drive.
	SYS_FS_DrivePartition	Partitions a physical drive (media).
	SYS_FS_DriveSectorGet	Obtains total number of sectors and number of free sectors for the specified drive.
	SYS_FS_EventHandlerSet	Allows a client to identify an event handling function for the file system to call back when mount/unmount operation has completed.

c) Data Types and Constants

	Name	Description
	SYS_FS_ERROR	Lists the various error cases.
	SYS_FS_FILE_OPEN_ATTRIBUTES	Lists the various attributes (modes) in which a file can be opened.
	SYS_FS_FILE_SEEK_CONTROL	Lists the various modes of file seek.
	SYS_FS_FILE_SYSTEM_TYPE	Enumerated data type identifying native file systems supported.
	SYS_FS_FSTAT	File System status
	SYS_FS_FUNCTIONS	SYS FS Function signature structure for native file systems.
	SYS_FS_HANDLE	This type defines the file handle.
	SYS_FS_REGISTRATION_TABLE	The sys_fs layer has to be initialized by passing this structure with suitably initialized members.
	SYS_FS_RESULT	Lists the various results of a file operation.
	FAT_FS_MAX_LFN	Maximum length of the Long File Name.
	FAT_FS_MAX_SS	Lists the definitions for FAT file system sector size.
	FAT_FS_USE_LFN	Lists the definitions for FAT file system LFN selection.
	SYS_FS_HANDLE_INVALID	Invalid file handle
	SYS_FS_FILE_DIR_ATTR	Enumerated data type identifying the various attributes for file/directory.
	SYS_FS_TIME	The structure to specify the time for a file or directory.
	SYS_FS_FORMAT	Specifies the partitioning rule.
	SYS_FS_EVENT	Identifies the possible file system events.
	SYS_FS_EVENT_HANDLER	Pointer to the File system Handler function.

Description

This section describes the APIs of the File System Service Library.

Refer to each section for a detailed description.

a) File and Directory Operation Functions

SYS_FS_FileClose Function

Closes a file.

File

[sys_fs.h](#)

C

```
SYS_FS_RESULT SYS_FS_FileClose(SYS_FS_HANDLE handle);
```

Returns

SYS_FS_RES_SUCCESS - File close operation was successful. SYS_FS_RES_FAILURE - File close operation failed. The reason for the failure can be retrieved with [SYS_FS_Error](#) or [SYS_FS_FileError](#).

Description

This function closes an opened file.

Remarks

None.

Preconditions

A valid file handle must be obtained before closing a file.

Example

```
SYS_FS_HANDLE fileHandle;
fileHandle = SYS_FS_FileOpen("/mnt/myDrive/FILE.JPG",
    (SYS_FS_FILE_OPEN_READ));

if(fileHandle != SYS_FS_HANDLE_INVALID)
{
    // File open is successful
}

SYS_FS_FileClose(fileHandle);
```

Parameters

Parameters	Description
handle	A valid handle, which was obtained while opening the file.

Function

```
SYS_FS_RESULT SYS_FS_FileClose
(
    SYS_FS_HANDLE handle
);
```

SYS_FS_FileEOF Function

Checks for end of file.

File

[sys_fs.h](#)

C

```
bool SYS_FS_FileEOF(SYS_FS_HANDLE handle);
```

Returns

On success returns true indicating that the file pointer has reached the end of the file. On failure returns false. This could be due to file pointer having not reached the end of the file. Or due to an invalid file handle. The reason for the failure can be retrieved with [SYS_FS_Error](#) or [SYS_FS_FileError](#).

Description

Checks whether or not the file position indicator is at the end of the file.

Remarks

None.

Preconditions

A valid file handle must be obtained before knowing a EOF.

Example

```
SYS_FS_HANDLE fileHandle;
bool eof;

fileHandle = SYS_FS_FileOpen("/mnt/myDrive/FILE.JPG",
    (SYS_FS_FILE_OPEN_READ));

if(fileHandle != SYS_FS_HANDLE_INVALID)
{
    // File open is successful
}
...
...

eof = SYS_FS_FileEOF(fileHandle);

if(eof == false)
{
    // Check the error state using SYS_FS_FileError
}
```

Parameters

Parameters	Description
handle	file handle obtained during file Open.

Function

```
bool SYS_FS_FileEOF
(
    SYS_FS_HANDLE handle
);
```

SYS_FS_FileOpen Function

Opens a file.

File

[sys_fs.h](#)

C

```
SYS_FS_HANDLE SYS_FS_FileOpen(const char* fname, SYS_FS_FILE_OPEN_ATTRIBUTES attributes);
```

Returns

On success - A valid file handle will be returned On failure - [SYS_FS_HANDLE_INVALID](#). The reason for the failure can be retrieved with [SYS_FS_Error](#).

Description

This function opens a file with the requested attributes.

Remarks

None.

Preconditions

Prior to opening a file, the name of the volume on which the file resides should be known and the volume should be mounted.

Example

```
SYS_FS_HANDLE fileHandle;

fileHandle = SYS_FS_FileOpen("/mnt/myDrive/FILE.JPG",
    (SYS_FS_FILE_OPEN_READ));
```

```
if(fileHandle != SYS_FS_HANDLE_INVALID)
{
    // File open succeeded.
}
else
{
    // File open failed.
}

// Using SYS_FS_CurrentDriveSet () function.

SYS_FS_HANDLE fileHandle;

SYS_FS_CurrentDriveSet("/mnt/myDrive");

fileHandle = SYS_FS_FileOpen("FILE.JPG", (SYS_FS_FILE_OPEN_READ));
if(fileHandle != SYS_FS_HANDLE_INVALID)
{
    // File open succeeded.
}
else
{
    // File open failed.
}
```

Parameters

Parameters	Description
fname	The name of the file to be opened along with the path. The fname
format is as follows	"/mnt/volumeName/dirName/fileName". volumeName is the name of the volume/drive. dirName is the name of the directory under which the file is located. fileName is the name of the file to be opened. The "/mnt/volumeName" portion from the fName can be omitted if the SYS_FS_CurrentDriveSet () has been invoked to set the current drive/volume.
attributes	Access mode of the file, of type SYS_FS_FILE_OPEN_ATTRIBUTES

Function

```
SYS_FS_HANDLE SYS_FS_FileOpen
(
    const char* fname,
    SYS_FS_FILE_OPEN_ATTRIBUTES attributes
);
```

SYS_FS_FileSeek Function

Moves the file pointer by the requested offset.

File

[sys_fs.h](#)

C

```
int32_t SYS_FS_FileSeek(SYS_FS_HANDLE fildes, int32_t offset, SYS_FS_FILE_SEEK_CONTROL whence);
```

Returns

On success - The number of bytes by which file pointer is moved (0 or positive number) On Failure - (-1) If the chosen offset value was (-1), the success or failure can be determined with [SYS_FS_Error](#).

Description

This function sets the file pointer for a open file associated with the file handle, as follows: whence = SYS_FS_SEEK_SET - File offset is set to offset bytes from the beginning. whence = SYS_FS_SEEK_CUR - File offset is set to its current location plus offset. whence = SYS_FS_SEEK_END - File offset is set to the size of the file plus offset. The offset specified for this option should be negative for the file pointer to be valid.

Trying to move the file pointer using SYS_FS_FileSeek, beyond the range of file will only cause the pointer to be moved to the last location of the file.

Remarks

None.

Preconditions

A valid file handle must be obtained before seeking a file.

Example

```
SYS_FS_HANDLE fileHandle;
int status;

fileHandle = SYS_FS_FileOpen("/mnt/myDrive/FILE.JPG",
    (SYS_FS_FILE_OPEN_READ));

if(fileHandle != SYS_FS_HANDLE_INVALID)
{
    // File open is successful
}
...
...

status = SYS_FS_FileSeek(fileHandle, 5, SYS_FS_SEEK_CUR);

if((status != -1) && (status == 5))
{
    // Success
}
```

Parameters

Parameters	Description
handle	A valid file handle obtained during file open.
offset	The number of bytes which act as file offset. This value could be a positive or negative value.
whence	Type of File Seek operation as specified in SYS_FS_FILE_SEEK_CONTROL .

Function

```
int32_t SYS_FS_FileSeek
(
    SYS_FS_HANDLE handle,
    int32_t offset,
    SYS_FS_FILE_SEEK_CONTROL whence
);
```

SYS_FS_FileSize Function

Returns the size of the file in bytes.

File

[sys_fs.h](#)

C

```
int32_t SYS_FS_FileSize(SYS_FS_HANDLE handle);
```

Returns

On success returns the size of the file in bytes. On failure returns -1. The reason for the failure can be retrieved with [SYS_FS_Error](#) or [SYS_FS_FileError](#).

Description

This function returns the size of the file as pointed by the handle.

Remarks

None.

Preconditions

A valid file handle must be obtained before knowing a file size.

Example

```
SYS_FS_HANDLE fileHandle;
long fileSize;

fileHandle = SYS_FS_FileOpen("/mnt/myDrive/FILE.JPG",
    (SYS_FS_FILE_OPEN_READ));

if(fileHandle != SYS_FS_HANDLE_INVALID)
{
    // File open is successful
}
...
...

fileSize = SYS_FS_FileSize(fileHandle);

if(fileSize != -1)
{
    // Success
}
```

Parameters

Parameters	Description
handle	File handle obtained during file Open.

Function

```
int32_t SYS_FS_FileSize
(
    SYS_FS_HANDLE handle
);
```

SYS_FS_FileTell Function

Obtains the file pointer position.

File

[sys_fs.h](#)

C

```
int32_t SYS_FS_FileTell(SYS_FS_HANDLE handle);
```

Returns

On success returns the current file position. On failure returns -1. The reason for the failure can be retrieved with [SYS_FS_Error](#) or [SYS_FS_FileError](#).

Description

Obtains the current value of the file position indicator for the file pointed to by handle.

Remarks

None.

Preconditions

A valid file handle must be obtained before performing a file tell.

Example

```
SYS_FS_HANDLE fileHandle;
int32_t tell;

fileHandle = SYS_FS_FileOpen("/mnt/myDrive/FILE.JPG",
    (SYS_FS_FILE_OPEN_READ));
```

```
if(fileHandle != SYS_FS_HANDLE_INVALID)
{
    // File open is successful
}
...
...

tell = SYS_FS_FileTell(fileHandle);

if(tell != -1)
{
    // Success
}
```

Parameters

Parameters	Description
handle	File handle obtained during file Open.

Function

```
int32_t SYS_FS_FileTell
(
    SYS_FS_HANDLE handle
);
```

SYS_FS_FileNameGet Function

Reads the file name.

File

```
sys_fs.h
```

C

```
bool SYS_FS_FileNameGet(SYS_FS_HANDLE handle, uint8_t* cName, uint16_t wLen);
```

Returns

Returns true if the file name was read successfully. Returns false if the file name was not read successfully. The reason for the failure can be retrieved with [SYS_FS_Error](#).

Description

This function reads the file name of a file that is already open.

Remarks

None.

Preconditions

The file handle referenced by handle is already open.

Example

```
SYS_FS_HANDLE fileHandle;
bool stat;
uint8_t fileName[255];

fileHandle = SYS_FS_FileOpen("/mnt/myDrive/FILE.JPG",
    (SYS_FS_FILE_OPEN_READ));

if(fileHandle != SYS_FS_HANDLE_INVALID)
{
    // File open is successful
}
...
...

stat = SYS_FS_FileNameGet(fileHandle, fileName, 8 );
```

```
if(stat == false)
{
    // file not located based on handle passed
    // Check the error state using SYS_FS_FileError
}
```

Parameters

Parameters	Description
handle	File handle obtained during file Open.
cName	Where to store the name of the file.
wLen	The maximum length of data to store in cName.

Function

```
bool SYS_FS_FileNameGet
(
    SYS_FS_HANDLE handle,
    uint8_t* cName,
    uint16_t wLen
);
```

SYS_FS_FilePrintf Function

Writes a formatted string into a file.

File

[sys_fs.h](#)

C

```
SYS_FS_RESULT SYS_FS_FilePrintf(SYS_FS_HANDLE handle, const char * string, ...);
```

Returns

SYS_FS_RES_SUCCESS - Formatted string write operation was successful. SYS_FS_RES_FAILURE - Formatted string write operation was unsuccessful. The reason for the failure can be retrieved with [SYS_FS_Error](#) or [SYS_FS_FileError](#).

Description

This function writes a formatted string into a file.

Remarks

None.

Preconditions

The file into which a string has to be written, must exist and should be open.

Example

```
SYS_FS_RESULT res;
SYS_FS_HANDLE fileHandle;

fileHandle = SYS_FS_FileOpen("/mnt/myDrive/FILE.txt", (SYS_FS_FILE_OPEN_WRITE_PLUS));

if(fileHandle != SYS_FS_HANDLE_INVALID)
{
    // File open is successful
}

// Write a string
res = SYS_FS_FilePrintf(fileHandle, "%d", 1234);
if( res != SYS_FS_RES_SUCCESS)
{
    // write operation failed.
}
```

Parameters

Parameters	Description
handle	File handle to which formatted string is to be written.
string	Pointer to formatted string which has to be written into file.

Function

```
SYS_FS_RESULT SYS_FS_FilePrintf  
(  
    SYS_FS_HANDLE handle,  
    const char *string,  
    ...  
);
```

SYS_FS_FileTestError Function

Checks for errors in the file.

File

[sys_fs.h](#)

C

```
bool SYS_FS_FileTestError(SYS_FS_HANDLE handle);
```

Returns

On success returns false indicating that the file has no errors. On failure returns true. The reason for the failure can be retrieved with [SYS_FS_Error](#) or [SYS_FS_FileError](#).

Description

This function checks whether or not file has any errors.

Remarks

None.

Preconditions

A valid file handle must be obtained before passing to the function

Example

```
SYS_FS_HANDLE fileHandle;  
bool err;  
  
fileHandle = SYS_FS_FileOpen("/mnt/myDrive/FILE.JPG", (SYS_FS_FILE_OPEN_READ));  
  
if(fileHandle != SYS_FS_HANDLE_INVALID)  
{  
    // File open is successful  
}  
...  
...  
  
err = SYS_FS_FileTestError(fileHandle);  
if(err == true)  
{  
    // either file has error, or there  
    // was an error in working with the "SYS_FS_FileTestError" function  
}
```

Parameters

Parameters	Description
handle	file handle obtained during file Open.

Function

```
bool SYS_FS_FileTestError
(
    SYS_FS_HANDLE handle
);
```

SYS_FS_FileDirectoryModeSet Function

Sets the mode for the file or directory.

File

[sys_fs.h](#)

C

```
SYS_FS_RESULT SYS_FS_FileDirectoryModeSet(const char* path, SYS_FS_FILE_DIR_ATTR attr, SYS_FS_FILE_DIR_ATTR
mask);
```

Returns

SYS_FS_RES_SUCCESS - Mode set operation was successful. SYS_FS_RES_FAILURE - Mode set operation was unsuccessful. The reason for the failure can be retrieved with [SYS_FS_Error](#).

Description

This function sets the mode for a file or directory from the specified list of attributes.

Remarks

None.

Preconditions

The file or directory for which the mode is to be set must exist.

Example

```
// Set read-only flag, clear archive flag and others are retained.
SYS_FS_FileDirectoryModeSet("file.txt", SYS_FS_ATTR_RDO, SYS_FS_ATTR_RDO | SYS_FS_ATTR_ARC);
```

Parameters

Parameters	Description
path	Path for the file/directory, for which the mode is to be set.
attr	Attribute flags to be set in one or more combination of the type SYS_FS_FILE_DIR_ATTR . The specified flags are set and others are cleared.
mask	Attribute mask of type SYS_FS_FILE_DIR_ATTR that specifies which attribute is changed. The specified attributes are set or cleared.

Function

```
SYS_FS_RESULT SYS_FS_FileDirectoryModeSet
(
    const char* path,
    SYS_FS_FILE_DIR_ATTR attr,
    SYS_FS_FILE_DIR_ATTR mask
);
```

SYS_FS_FileDirectoryRemove Function

Removes a file or directory.

File

[sys_fs.h](#)

C

```
SYS_FS_RESULT SYS_FS_FileDirectoryRemove(const char* path);
```

Returns

SYS_FS_RES_SUCCESS - Indicates that the file or directory remove operation was successful. SYS_FS_RES_FAILURE - Indicates that the file or directory remove operation was unsuccessful. The reason for the failure can be retrieved with [SYS_FS_Error](#).

Description

This function removes a file or directory as specified by the path.

Remarks

None.

Preconditions

- The disk has to be mounted before a directory could be removed.
- The file or directory to be removed has to present.
- The file/sub-directory must not have read-only attribute (AM_RDO), or the function will be rejected with FR_DENIED.
- The sub-directory must be empty and must not be current directory, or the function will be rejected with FR_DENIED.
- The file/sub-directory must not be opened.

Example

```
SYS_FS_RESULT res;

res = SYS_FS_FileDirectoryRemove("Dir1");

if(res == SYS_FS_RES_FAILURE)
{
    // Directory remove operation failed
}
//...
//...
```

Parameters

Parameters	Description
path	Path of the File or directory to be removed.

Function

```
SYS\_FS\_RESULT SYS_FS_FileDirectoryRemove
(
    const char* path
);
```

SYS_FS_FileDirectoryRenameMove Function

Renames or moves a file or directory.

File

[sys_fs.h](#)

C

```
SYS_FS_RESULT SYS_FS_FileDirectoryRenameMove(const char * oldPath, const char * newPath);
```

Returns

SYS_FS_RES_SUCCESS - Rename/move operation was successful. SYS_FS_RES_FAILURE - Rename/move operation was unsuccessful. The reason for the failure can be retrieved with [SYS_FS_Error](#).

Description

This function renames or moves a file or directory.

Remarks

This function cannot move files/ directory from one drive to another. Do not rename/ move files which are open.

Preconditions

The file or directory to be renamed or moved must exist. This function cannot move files or directories from one drive to another. Do not rename or

move files that are open.

Example

```
SYS_FS_RESULT res;

// rename "file.txt" to "renamed_file.txt"
res = SYS_FS_FileDirectoryRenameMove("file.txt", "renamed_file.txt");
if( res != SYS_FS_RES_SUCCESS)
{
    // Rename operation failed.
}

// Now, move "renamed_file.txt" inside directory "Dir1"
res = SYS_FS_FileDirectoryRenameMove("renamed_file.txt", "Dir1/renamed_file.txt");
if( res != SYS_FS_RES_SUCCESS)
{
    // File move operation failed.
}
```

Parameters

Parameters	Description
oldPath	Path for the file/directory, which has to be renamed/moved.
newPath	New Path for the file/directory.

Function

```
SYS_FS_RESULT SYS_FS_FileDirectoryRenameMove
(
    const char *oldPath,
    const char *newPath
);
```

SYS_FS_FileDirectoryTimeSet Function

Sets or changes the time for a file or directory.

File

[sys_fs.h](#)

C

```
SYS_FS_RESULT SYS_FS_FileDirectoryTimeSet(const char* path, SYS_FS_TIME * time);
```

Returns

SYS_FS_RES_SUCCESS - Set time operation was successful. SYS_FS_RES_FAILURE - Set time operation was unsuccessful. The reason for the failure can be retrieved with [SYS_FS_Error](#).

Description

This function sets or change the time for a file or directory.

Remarks

None.

Preconditions

The file/directory for which time is to be set must exist.

Example

```
void setTime(void)
{
    SYS_FS_RESULT res;
    SYS_FS_TIME time;

    time.packedTime = 0;

    // All FAT FS times are calculated based on 0 = 1980
}
```



```

time.discreteTime.year = (2013 - 1980); // Year is 2013
time.discreteTime.month = 8;           // Month (August)
time.discreteTime.day = 9;             // Day (9)
time.discreteTime.hour = 15;           // 3 PM
time.discreteTime.minute = 06;         // 06 minutes
time.discreteTime.second = 00;         // 00 seconds

res = SYS_FS_FileDirectoryTimeSet("file.txt", &time);
if( res != SYS_FS_RES_SUCCESS)
{
    // time change has gone wrong
}
}

```

Parameters

Parameters	Description
path	A path for the file/directory, for which the time is to be set.
ptr	Pointer to the structure of type SYS_FS_TIME , which contains the time data to be set.

Function

```

SYS_FS_RESULT SYS_FS_FileDirectoryTimeSet
(
    const char* path,
    SYS_FS_TIME *time
);

```

SYS_FS_FileTruncate Function

Truncates a file

File

[sys_fs.h](#)

C

```
SYS_FS_RESULT SYS_FS_FileTruncate(SYS_FS_HANDLE handle);
```

Returns

[SYS_FS_RES_SUCCESS](#) - File truncate operation was successful. [SYS_FS_RES_FAILURE](#) - File truncate operation was unsuccessful. The reason for the failure can be retrieved with [SYS_FS_Error](#) or [SYS_FS_FileError](#).

Description

This function truncates the file size to the current file read/write pointer. This function has no effect if the file read/write pointer is already pointing to end of the file.

Remarks

None.

Preconditions

A valid file handle has to be passed as input to the function. The file has to be opened in a mode where writes to file is possible (such as read plus or write mode).

Example

```

SYS_FS_HANDLE fileHandle;
size_t nbytes;
size_t bytes_read;
SYS_FS_RESULT res;

fileHandle = SYS_FS_FileOpen("/mnt/myDrive/FILE.JPG",
    (SYS_FS_FILE_OPEN_READ));

if(fileHandle != SYS_FS_HANDLE_INVALID)
{
    // File open is successful
}

```

```
// Read the file content
nbytes = sizeof(buf);
bytes_read = SYS_FS_FileRead(buf, nbytes, fileHandle);
// Truncate the file
res = SYS_FS_FileTruncate(fileHandle);
if(res != SYS_FS_RES_SUCCESS)
{
    // Truncation failed.
}

SYS_FS_FileClose(fileHandle);
```

Parameters

Parameters	Description
handle	A valid handle which was obtained while opening the file.

Function

```
SYS_FS_RESULT SYS_FS_FileTruncate
(
    SYS_FS_HANDLE handle
);
```

SYS_FS_FileCharacterPut Function

Writes a character to a file.

File

[sys_fs.h](#)

C

```
SYS_FS_RESULT SYS_FS_FileCharacterPut(SYS_FS_HANDLE handle, char data);
```

Returns

SYS_FS_RES_SUCCESS - Write operation was successful. SYS_FS_RES_FAILURE - Write operation was unsuccessful. The reason for the failure can be retrieved with [SYS_FS_Error](#) or [SYS_FS_FileError](#).

Description

This function writes a character to a file.

Remarks

None.

Preconditions

The file into which a character has to be written, has to be present and should have been opened.

Example

```
SYS_FS_RESULT res;
SYS_FS_HANDLE fileHandle;

fileHandle = SYS_FS_FileOpen("/mnt/myDrive/FILE.JPG", (SYS_FS_FILE_OPEN_WRITE_PLUS));
if(fileHandle != SYS_FS_HANDLE_INVALID)
{
    // File open is successful
}

// Write a character to the file.
res = SYS_FS_FileCharacterPut(fileHandle, 'c');
if(res != SYS_FS_RES_SUCCESS)
{
    // Character write operation failed.
}
```

Parameters

Parameters	Description
handle	file handle to which the character is to be written.
data	character to be written to the file.

Function

```
SYS_FS_RESULT SYS_FS_FileCharacterPut
(
    SYS_FS_HANDLE handle,
    char data
);
```

SYS_FS_FileStringGet Function

Reads a string from the file into a buffer.

File

[sys_fs.h](#)

C

```
SYS_FS_RESULT SYS_FS_FileStringGet(SYS_FS_HANDLE handle, char* buff, uint32_t len);
```

Returns

SYS_FS_RES_SUCCESS - String read operation was successful. SYS_FS_RES_FAILURE - String read operation was unsuccessful. The reason for the failure can be retrieved with [SYS_FS_Error](#) or [SYS_FS_FileError](#).

Description

This function reads a string of specified length from the file into a buffer. The read operation continues until

1. 'n' is stored
2. reached end of the file or
3. the buffer is filled with len - 1 characters.

The read string is terminated with a '0'.

Remarks

None.

Preconditions

The file from which a string has to be read, has to be present and should have been opened.

Example

```
SYS_FS_RESULT res;
SYS_FS_HANDLE fileHandle;
char buffer[100];

fileHandle = SYS_FS_FileOpen("/mnt/myDrive/FILE.JPG", (SYS_FS_FILE_OPEN_WRITE_PLUS));
if(fileHandle != SYS_FS_HANDLE_INVALID)
{
    // File open is successful
}

// Read a string from the file.
res = SYS_FS_FileStringGet(fileHandle, buffer, 50);
if( res != SYS_FS_RES_SUCCESS)
{
    //String read operation failed.
}
```

Parameters

Parameters	Description
handle	Handle of the file from which string is to be read.
buff	Buffer in which the string is to be stored.

len	length of string to be read.
-----	------------------------------

Function

```
SYS_FS_RESULT SYS_FS_FileStringGet
(
    SYS_FS_HANDLE handle,
    char* buff,
    uint32_t len
);
```

SYS_FS_FileStringPut Function

Writes a string to a file.

File

[sys_fs.h](#)

C

```
SYS_FS_RESULT SYS_FS_FileStringPut(SYS_FS_HANDLE handle, const char * string);
```

Returns

SYS_FS_RES_SUCCESS - String write operation was successful. SYS_FS_RES_FAILURE - String write operation was unsuccessful. The reason for the failure can be retrieved with [SYS_FS_Error](#) or [SYS_FS_FileError](#).

Description

This function writes a string into a file. The string to be written should be NULL terminated. The terminator character will not be written.

Remarks

None.

Preconditions

The file into which a string has to be written, has to be present and should have been opened.

Example

```
SYS_FS_RESULT res;
SYS_FS_HANDLE fileHandle;

fileHandle = SYS_FS_FileOpen("/mnt/myDrive/FILE.JPG", SYS_FS_FILE_OPEN_WRITE_PLUS));
if(fileHandle != SYS_FS_HANDLE_INVALID)
{
    // File open is successful
}

// Write a string
res = SYS_FS_FileStringPut(fileHandle, "Hello World");
if(res != SYS_FS_RES_SUCCESS)
{
    // String write operation failed.
}
```

Parameters

Parameters	Description
handle	File handle to which string is to be written.
string	Pointer to the null terminated string which has to be written into file.

Function

```
SYS_FS_RESULT SYS_FS_FileStringPut
(
    SYS_FS_HANDLE handle,
    const char *string
);
```

SYS_FS_Error Function

Returns the last error.

File

[sys_fs.h](#)

C

```
SYS_FS_ERROR SYS_FS_Error();
```

Returns

Error code of type [SYS_FS_ERROR](#).

Description

When a file system operation fails, the application can know the reason of failure by calling the SYS_FS_Error. This function only reports the errors which are not file (or file handle) specific. For example, for functions such as [SYS_FS_Mount](#) and [SYS_FS_FileOpen](#), which do not take handle, any errors happening inside such function calls could be reported using SYS_FS_Error function. Even for functions, which take handle as its input parameters, the SYS_FS_Error function can be used to report the type of error for cases where the passed handle itself is invalid.

Remarks

None.

Preconditions

This function has to be called immediately after a failure is observed while doing a file operation. Any subsequent failure will overwrite the cause of previous failure.

Example

```
SYS_FS_HANDLE fileHandle;
SYS_FS_ERROR err;

fileHandle = SYS_FS_FileOpen("/mnt/myDrive/FILE.JPG",
                             (SYS_FS_FILE_OPEN_READ));

if(fileHandle == SYS_FS_HANDLE_INVALID)
{
    // If failure, now know the specific reason for failure
    err = SYS_FS_Error();
}
```

Function

```
SYS\_FS\_ERROR SYS_FS_Error
(
    void
)
```

SYS_FS_FileError Function

Returns the file specific error.

File

[sys_fs.h](#)

C

```
SYS_FS_ERROR SYS_FS_FileError(SYS_FS_HANDLE handle);
```

Returns

Error code of type [SYS_FS_ERROR](#).

Description

For file system functions which accepts valid handle, any error happening in those functions could be retrieved with SYS_FS_FileError. This function returns errors which are file specific.

Please note that if an invalid handle is passed to a file system function, in such a case, SYS_FS_FileError will not return the correct type of error,

as the handle was invalid. Therefore, it would be prudent to check the errors using the [SYS_FS_Error](#) function.

Remarks

None.

Preconditions

This function has to be called immediately after a failure is observed while doing a file operation. Any subsequent failure will overwrite the cause of previous failure.

Example

```
...
const char *buf = "Hello World";
size_t nbytes;
size_t bytes_written;
SYS_FS_HANDLE fd;
SYS_FS_ERROR err;
...

bytes_written = SYS_FS_FileWrite((const void *)buf, nbytes, fd);

if(bytes_written == -1)
{
    // error while writing file
    // find the type (reason) of error
    err = SYS_FS_FileError(fd);
}
```

Parameters

Parameters	Description
handle	A valid file handle

Function

```
SYS_FS_ERROR SYS_FS_FileError
(
    SYS_FS_HANDLE handle
);
```

SYS_FS_FileRead Function

Read data from the file.

File

[sys_fs.h](#)

C

```
size_t SYS_FS_FileRead(SYS_FS_HANDLE handle, void * buf, size_t nbytes);
```

Returns

On success returns the number of bytes read successfully(0 or positive number). On failure returns -1. The reason for the failure can be retrieved with [SYS_FS_Error](#) or [SYS_FS_FileError](#).

Description

This function attempts to read nbytes bytes of data from the file associated with the file handle into the buffer pointed to by buf.

Remarks

None.

Preconditions

A valid file handle must be obtained before reading a file.

Example

```
...
char buf[20];
```

```
size_t nbytes;  
size_t bytes_read;  
SYS_FS_HANDLE fd;  
...  
nbytes = sizeof(buf);  
bytes_read = SYS_FS_FileRead(fd, buf, nbytes);  
...
```

Parameters

Parameters	Description
handle	File handle obtained during file open.
buf	Pointer to buffer into which data is read.
nbyte	Number of bytes to be read

Function

```
size_t SYS_FS_FileRead  
(  
    SYS_FS_HANDLE handle,  
    void *buf,  
    size_t nbyte  
);
```

SYS_FS_FileWrite Function

Writes data to the file.

File

[sys_fs.h](#)

C

```
size_t SYS_FS_FileWrite(SYS_FS_HANDLE handle, const void * buf, size_t nbyte);
```

Returns

On success returns the number of bytes written successfully(0 or positive number). On failure returns -1. The reason for the failure can be retrieved with [SYS_FS_Error](#) or [SYS_FS_FileError](#).

Description

This function attempts to write nbyte bytes from the buffer pointed to by buf to the file associated with the file handle.

Remarks

None.

Preconditions

A valid file handle must be obtained before writing a file.

Example

```
...  
const char *buf = "Hello World";  
size_t nbytes;  
size_t bytes_written;  
SYS_FS_HANDLE fd;  
...  
  
bytes_written = SYS_FS_FileWrite(fd, (const void *)buf, nbytes);  
...
```

Parameters

Parameters	Description
handle	File handle obtained during file open.
buf	Pointer to buffer from which data is to be written
nbyte	Number of bytes to be written

Function

```
size_t SYS_FS_FileWrite
(
    SYS_FS_HANDLE handle,
    const void *buf,
    size_t nbyte
);
```

SYS_FS_CurrentWorkingDirectoryGet Function

Gets the current working directory

File

[sys_fs.h](#)

C

```
SYS_FS_RESULT SYS_FS_CurrentWorkingDirectoryGet(char * buff, uint32_t len);
```

Returns

SYS_FS_RES_SUCCESS - Get current working directory operation was successful. SYS_FS_RES_FAILURE - Get current working directory operation was unsuccessful. The reason for the failure can be retrieved with [SYS_FS_Error](#).

Description

This function gets the current working directory path along with the working drive.

Remarks

None.

Preconditions

At least one disk must be mounted.

Example

```
SYS_FS_RESULT res;
char buffer[16];

switch(appState)
{
    case TRY_MOUNT:
        if(SYS_FS_Mount("/dev/mmcb1kal", "/mnt/myDrive", FAT, 0, NULL) != SYS_FS_RES_SUCCESS)
        {
            // Failure, try mounting again
        }
        else
        {
            // Mount was successful. Create a directory.
            appState = CREATE_DIR;
        }
        break;

    case CREATE_DIR:
        res = SYS_FS_DirectoryMake("Dir1");
        if(res == SYS_FS_RES_FAILURE)
        {
            // Directory creation failed
            appState = ERROR;
        }
        else
        {
            // Directory creation was successful. Change to the new
            // directory.
            appState = CHANGE_DIR;
        }
        break;
```



```
    case CHANGE_DIR:
        res = SYS_FS_DirectoryChange("Dir1");
        if(res == SYS_FS_RES_FAILURE)
        {
            // Directory change failed
            appState = ERROR;
        }
        else
        {
            // Directory change was successful. Get current working
            // directory
            appState = GET_CWD;
        }
        break;

    case GET_CWD:
        res = SYS_FS_CurrentWorkingDirectoryGet(buffer, 15);
        if(res == SYS_FS_RES_FAILURE)
        {
            // Get current directory operation failed
            appState = ERROR;
        }
        break;
}
```

Parameters

Parameters	Description
buff	Pointer to a buffer which will contain the name of the current working directory and drive, once the function completes.
len	Size of the buffer.

Function

```
SYS_FS_RESULT SYS_FS_CurrentWorkingDirectoryGet
(
    char *buff,
    uint32_t len
);
```

SYS_FS_DirectoryChange Function

Changes to a the directory specified.

File

[sys_fs.h](#)

C

```
SYS_FS_RESULT SYS_FS_DirectoryChange(const char* path);
```

Returns

SYS_FS_RES_SUCCESS - Indicates that the directory change operation was successful. SYS_FS_RES_FAILURE - Indicates that the directory change operation was unsuccessful. The reason for the failure can be retrieved with [SYS_FS_Error](#).

Description

This function changes the present directory to a new directory.

Remarks

None.

Preconditions

The disk has to be mounted and the directory to be changed must exist.

Example

```
SYS_FS_RESULT res;
```

```
res = SYS_FS_DirectoryChange("Dir1");

if(res == SYS_FS_RES_FAILURE)
{
    // Directory change failed
}
```

Parameters

Parameters	Description
path	Path of the directory to be changed to.

Function

```
SYS_FS_RESULT SYS_FS_DirectoryChange
(
    const char* path
);
```

SYS_FS_DirectoryMake Function

Makes a directory.

File

[sys_fs.h](#)

C

```
SYS_FS_RESULT SYS_FS_DirectoryMake(const char* path);
```

Returns

SYS_FS_RES_SUCCESS - Indicates that the creation of the directory was successful. SYS_FS_RES_FAILURE - Indicates that the creation of the directory was unsuccessful. The reason for the failure can be retrieved with [SYS_FS_Error](#).

Description

This function makes a new directory as per the specified path.

Remarks

None.

Preconditions

The disk has to be mounted before a directory could be made.

Example

```
SYS_FS_RESULT res;

res = SYS_FS_DirectoryMake("Dir1");

if(res == SYS_FS_RES_FAILURE)
{
    // Directory make failed
}
```

Parameters

Parameters	Description
path	Path of the new directory

Function

```
SYS_FS_RESULT SYS_FS_DirectoryMake
(
    const char* path
);
```

SYS_FS_DirClose Function

Closes an opened directory.

File

[sys_fs.h](#)

C

```
SYS_FS_RESULT SYS_FS_DirClose(SYS_FS_HANDLE handle);
```

Returns

SYS_FS_RES_SUCCESS - Directory close operation was successful. SYS_FS_RES_FAILURE - Directory close operation was unsuccessful. The reason for the failure can be retrieved with [SYS_FS_Error](#) or [SYS_FS_FileError](#).

Description

This function closes a directory that was opened earlier opened with the [SYS_FS_DirOpen](#) function.

Remarks

None.

Preconditions

A valid directory handle must be obtained before closing the directory.

Example

```
SYS_FS_HANDLE dirHandle;

dirHandle = SYS_FS_DirOpen("/mnt/myDrive/Dir1");

if(dirHandle != SYS_FS_HANDLE_INVALID)
{
    // Directory open is successful
}

// Perform required operation on the directory

// Close the directory
if(SYS_FS_DirClose(dirHandle) == SYS_FS_RES_FAILURE)
{
    // Close operation failed.
}
```

Parameters

Parameters	Description
handle	directory handle obtained during directory open.

Function

```
SYS_FS_RESULT SYS_FS_DirClose
(
    SYS_FS_HANDLE handle
);
```

SYS_FS_DirOpen Function

Open a directory

File

[sys_fs.h](#)

C

```
SYS_FS_HANDLE SYS_FS_DirOpen(const char* path);
```

Returns

On success a valid handle to the directory will be returned. On failure [SYS_FS_HANDLE_INVALID](#) will be returned. The reason for the failure can be retrieved with [SYS_FS_Error](#).

Description

This function opens the requested directory.

Remarks

None

Preconditions

The volume on which the directory is present should be mounted.

Example

```
SYS_FS_HANDLE dirHandle;

dirHandle = SYS_FS_DirOpen("/mnt/myDrive/Dir1");
// For root directory, end with a "/"
// dirHandle = SYS_FS_DirOpen("/mnt/myDrive/");

if(dirHandle != SYS_FS_HANDLE_INVALID)
{
    // Directory open is successful
}
```

Parameters

Parameters	Description
path	Path to the directory along with the volume name. The string of volume and directory name has to be preceded by "/mnt/". Also, the volume name and directory name has to be separated by a slash "/". If the directory specified is only the root directory, the path has to be ended with "/".

Function

```
SYS\_FS\_HANDLE SYS_FS_DirOpen
(
    const char* path
);
```

SYS_FS_DirRead Function

Reads the specified directory.

File

[sys_fs.h](#)

C

```
SYS_FS_RESULT SYS_FS_DirRead(SYS_FS_HANDLE handle, SYS_FS_FSTAT * stat);
```

Returns

SYS_FS_RES_SUCCESS - Indicates that the directory read operation was successful. SYS_FS_RES_FAILURE - Indicates that the directory read operation was unsuccessful. The reason for the failure can be retrieved with [SYS_FS_Error](#) or [SYS_FS_FileError](#).

Description

This function reads the files and directories specified in the open directory.

Remarks

None.

Preconditions

A valid directory handle must be obtained before reading a directory.

Example

```
SYS_FS_HANDLE dirHandle;
SYS_FS_FSTAT stat;
char longFileName[300];
uintptr_t longFileSize;

dirHandle = SYS_FS_DirOpen("/mnt/myDrive/Dir1");

if(dirHandle != SYS_FS_HANDLE_INVALID)
{
    // Directory open is successful
}

// If long file name is used, the following elements of the "stat"
// structure needs to be initialized with address of proper buffer.
stat.lfname = longFileName;
stat.lfsize = 300;

if(SYS_FS_DirRead(dirHandle, &stat) == SYS_FS_RES_FAILURE)
{
    // Directory read failed.
}
```

Parameters

Parameters	Description
handle	Directory handle obtained during directory open.
stat	Empty structure of type SYS_FS_FSTAT , where the properties of the open directory will be populated after the SYS_FS_DirRead function returns successfully. If LFN is used, then the "lfname" member of the SYS_FS_FSTAT structure should be initialized with the address of suitable buffer. Also, the "lfsize" should be initialized with the size of the buffer. Once the function returns, the buffer whose address is held in "lfname" will have the file name(long file name)

Function

```
SYS_FS_RESULT SYS_FS_DirRead
(
    SYS_FS_HANDLE handle,
    SYS_FS_FSTAT *stat
);
```

SYS_FS_DirSearch Function

Searches for a file or directory.

File

[sys_fs.h](#)

C

```
SYS_FS_RESULT SYS_FS_DirSearch(SYS_FS_HANDLE handle, const char * name, SYS_FS_FILE_DIR_ATTR attr,
SYS_FS_FSTAT * stat);
```

Returns

[SYS_FS_RES_SUCCESS](#) - Indicates that the file or directory was found. The stat parameter will contain information about the file or directory.
[SYS_FS_RES_FAILURE](#) - Indicates that the file or directory was not found. The reason for the failure can be retrieved with [SYS_FS_Error](#) or [SYS_FS_FileError](#).

Description

This function searches for the requested file or directory. The file or directory is specified in the attr parameter, which is of type [SYS_FS_FILE_DIR_ATTR](#).

Remarks

None.

Preconditions

A valid directory handle must be obtained before searching the directory.

Example

```
SYS_FS_HANDLE dirHandle;
SYS_FS_FSTAT stat;
char longFileName[300];
uintptr_t longFileSize;

dirHandle = SYS_FS_DirOpen("/mnt/myDrive/Dir1");

if(dirHandle != SYS_FS_HANDLE_INVALID)
{
    // Directory open is successful
}

// If long file name is used, the following elements of the "stat"
// structure needs to be initialized with address of proper buffer.
stat.lfname = longFileName;
stat.lfsize = 300;

if(SYS_FS_DirSearch(dirHandle, "FIL*.*", SYS_FS_ATTR_ARC, &stat) == SYS_FS_RES_FAILURE)
{
    // Specified file not found
}
else
{
    // File found. Read the complete file name from "stat.lfname" and
    // other file parameters from the "stat" structure
}
```

Parameters

Parameters	Description
handle	directory handle obtained during directory open.
name	name of file or directory needed to be searched. The file
name can have wild card entries as follows	<ul style="list-style-type: none">- Indicates the rest of the filename or extension can vary (e.g. FILE.*)
?	Indicates that one character in a filename can vary (e.g. F?LE.T?T)
attr	Attribute of the name of type SYS_FS_FILE_DIR_ATTR . This attribute specifies whether to search a file or a directory. Other attribute types could also be specified.
stat	Empty structure of type SYS_FS_FSTAT , where the properties of the file/directory will be populated. If LFN is used, then the "lfname" member of the SYS_FS_FSTAT structure should be initialized with address of suitable buffer. Also, the "lfsize" should be initialized with the size of buffer. Once the function returns, the buffer whose address is held in "lfname" will have the file name (long file name).

Function

[SYS_FS_RESULT](#) SYS_FS_DirSearch

```
(
    SYS\_FS\_HANDLE handle,
    const char * name,
    SYS\_FS\_FILE\_DIR\_ATTR attr,
    SYS\_FS\_FSTAT *stat
);
```

SYS_FS_FileSync Function

Flushes the cached information when writing to a file.

File

[sys_fs.h](#)

C

```
SYS_FS_RESULT SYS_FS_FileSync(SYS_FS_HANDLE handle);
```

Returns

SYS_FS_RES_SUCCESS - File sync operation was successful. SYS_FS_RES_FAILURE - File sync operation was unsuccessful. The reason for the failure can be retrieved with [SYS_FS_Error](#) or [SYS_FS_FileError](#).

Description

This function flushes the cached information when writing to a file. The SYS_FS_FileSync function performs the same process as [SYS_FS_FileClose](#) function; however, the file is left open and can continue read/write/seek operations to the file.

Remarks

None.

Preconditions

A valid file handle has to be passed as input to the function. The file which has to be flushed, has to be present and should have been opened in write mode.

Example

```
SYS_FS_RESULT res;
SYS_FS_HANDLE fileHandle;
const char *buf = "Hello World";
size_t nbytes;
size_t bytes_written;

fileHandle = SYS_FS_FileOpen("/mnt/myDrive/FILE.JPG", (SYS_FS_FILE_OPEN_WRITE_PLUS));

if(fileHandle != SYS_FS_HANDLE_INVALID)
{
    // File open is successful
}

// Write data to the file
bytes_written = SYS_FS_FileWrite((const void *)buf, nbytes, fileHandle);

// Flush the file
res = SYS_FS_FileSync(fileHandle);
if( res != SYS_FS_RES_SUCCESS)
{
    // renaming has gone wrong
}
```

Parameters

Parameters	Description
handle	valid file handle

Function

```
SYS_FS_RESULT SYS_FS_FileSync
(
    SYS_FS_HANDLE handle
);
```

SYS_FS_DirRewind Function

Rewinds to the beginning of the directory.

File

[sys_fs.h](#)

C

```
SYS_FS_RESULT SYS_FS_DirRewind(SYS_FS_HANDLE handle);
```

Returns

SYS_FS_RES_SUCCESS - Directory rewind operation was successful. SYS_FS_RES_FAILURE - Directory rewind operation was unsuccessful. The reason for the failure can be retrieved with [SYS_FS_Error](#) or [SYS_FS_FileError](#).

Description

This function rewinds the directory to the start. Once a search of directory or directory read is completed, the rewind function is used to begin searching the directory from the start.

Remarks

None.

Preconditions

A valid directory handle must be obtained before reading a directory.

Example

```
SYS_FS_HANDLE dirHandle;
SYS_FS_FSTAT stat;
char longFileName[300];
uintptr_t longFileSize;

dirHandle = SYS_FS_DirOpen("/mnt/myDrive/Dir1");

if(dirHandle != SYS_FS_HANDLE_INVALID)
{
    // Directory open is successful
}

// If long file name is used, the following elements of the "stat"
// structure needs to be initialized with address of proper buffer.
stat.lfname = longFileName;
stat.lfsize = 300;

if(SYS_FS_DirRead(dirHandle, &stat) == SYS_FS_RES_FAILURE)
{
    // Directory read operation failed.
}

// Do more search
// Do some more search

// Now, rewind the directory to begin search from start

if(SYS_FS_DirRewind(dirHandle) == SYS_FS_RES_FAILURE)
{
    // Directory rewind failed.
}
```

Parameters

Parameters	Description
handle	directory handle obtained during directory open.

Function

```
SYS_FS_RESULT SYS_FS_DirRewind
(
    SYS_FS_HANDLE handle
);
```

SYS_FS_FileStat Function

Gets file status.

File

[sys_fs.h](#)

C

```
SYS_FS_RESULT SYS_FS_FileStat(const char * fname, SYS_FS_FSTAT * buf);
```

Returns

SYS_FS_RES_SUCCESS - File stat operation was successful. SYS_FS_RES_FAILURE - File stat operation was unsuccessful. The reason for the failure can be retrieved with [SYS_FS_Error](#).

Description

This function obtains information about a file associated with the file name, and populates the information in the structure pointed to by buf. This function can read the status of file regardless of whether a file is opened or not.

Remarks

None.

Preconditions

Prior to opening a file, the name of the volume on which the file resides should be known and the volume should be mounted.

Example

```
SYS_FS_fStat fileStat;

if(SYS_FS_FileStat("/mnt/myDrive/FILE.TXT", &fileStat) == SYS_FS_RES_SUCCESS)
{
    // Successfully read the status of file "FILE.TXT"
}
```

Parameters

Parameters	Description
fname	Name of the file with the path and the volume name. The string of volume and file name has to be preceded by "/mnt/". Also, the volume name and file name has to be separated by a slash "/".
buf	pointer to SYS_FS_FSTAT structure.

Function

```
SYS_FS_RESULT SYS_FS_FileStat
(
    const char *fname,
    SYS_FS_FSTAT *buf
)
```

b) General Operation Functions

SYS_FS_Initialize Function

Initializes the file system abstraction layer (sys_fs layer).

File

[sys_fs.h](#)

C

```
SYS_FS_RESULT SYS_FS_Initialize(const void* initData);
```

Returns

SYS_FS_RES_SUCCESS - SYS FS Layer was initialized successfully. SYS_FS_RES_FAILURE - SYS FS Layer initialization failed. The reason for the failure can be retrieved with [SYS_FS_Error](#).

Description

This function initializes the abstraction layer (sys_fs layer) and sets up the necessary parameters.

Preconditions

This is the first function to be called during usage of sys_fs. Calling other functions of sys_fs without initializing the sys_fs will cause unpredictable

behavior.

Example

```
// This code shows an example of how the SYS FS is initialized
// Only one file system is used

#define SYS_FS_MAX_FILE_SYSTEM_TYPE          1

// Function pointer table for FAT FS
const SYS_FS_FUNCTIONS FatFsFunctions =
{
    .mount      = f_mount,
    .unmount    = f_unmount,
    .open       = f_open,
    .read       = f_read,
    .write      = f_write,
    .close      = f_close,
    .seek       = f_lseek,
    .tell       = f_tell,
    .eof        = f_eof,
    .size       = f_size,
    .fstat      = f_stat,
};

const SYS_FS_REGISTRATION_TABLE sysFSInit [ SYS_FS_MAX_FILE_SYSTEM_TYPE ] =
{
    {
        .nativeFileSystemType = FAT,
        .nativeFileSystemFunctions = &FatFsFunctions
    }
};

SYS_FS_Initialize((const void *)sysFSInit);
```

Parameters

Parameters	Description
initData	Pointer to an array of type SYS_FS_REGISTRATION_TABLE . The number of elements of array is decided by the definition SYS_FS_MAX_FILE_SYSTEM_TYPE . If the application uses one file system (say only FAT FS), SYS_FS_MAX_FILE_SYSTEM_TYPE is defined to be 1. Otherwise, if the application uses 2 file systems (say FAT FS and MPFS2), SYS_FS_MAX_FILE_SYSTEM_TYPE is defined to be 2.

Function

```
SYS_FS_RESULT SYS_FS_Initialize
(
    const void* initData
);
```

SYS_FS_Mount Function

Mounts the file system.

File

[sys_fs.h](#)

C

```
SYS_FS_RESULT SYS_FS_Mount(const char * devName, const char * mountName, SYS_FS_FILE_SYSTEM_TYPE
filesystemtype, unsigned long mountflags, const void * data);
```

Returns

SYS_FS_RES_SUCCESS - Mount was successful. SYS_FS_RES_FAILURE - Mount was unsuccessful. The reason for the failure can be retrieved with [SYS_FS_Error](#).

Description

The mount command attaches the file system specified to a volume. The call to the mount should be non blocking in nature. The application code

has to allow the [SYS_FS_Tasks](#) to run periodically while calling the SYS_FS_Mount function. If the SYS_FS_Mount is called in a blocking mode, then the [SYS_Tasks\(\)](#) never gets a chance to run and therefore, the media will not be analyzed and finally, the SYS_FS_Mount will never succeed. This will result in a deadlock.

There is no mechanism available for the application to know if the specified volume (devName) is really attached or not. The only available possibility is to keep trying to mount the volume (with the devname), until success is achieved.

It is prudent that the application code implements a time-out mechanism while trying to mount a volume (by calling SYS_FS_Mount). The trial for mount should continue at least 10 times before assuming that the mount will never succeed. This has to be done for every new volume to be mounted.

The standard names for volumes (devName) used in the MPLAB Harmony file system is as follows: NVM - "nvm" "media number" "volume number" SD card - "mmcblk" "media number" "volume number" MSD - "sd" "media number" "volume number"

Where, "media number" a, b, c... depends on the number of the type of connected media, and where, "volume number" 1, 2, 3... depends on the number of partitions in that media.

The convention for assigning names to volumes is further described below with examples:

If a SD card (with four partitions) is attached to the system, and assuming all four partitions are recognized, there will be four devNames:

1. mmcblk1
2. mmcblk2
3. mmcblk3 and
4. mmcblk4

Subsequently, if NVM media is attached that has only one partition, the devname will be: nvma1.

Later, if another SD card is attached to the system that has one partition, the devname will be mmcblk1.

Finally, there will be six volume names (or devNames), which are available for the application to be mounted and used for the file system.

Remarks

None

Preconditions

The "devName" name for the volume has to be known. The file system type with which each of the volumes are formatted has to be known. Trying to mount a volume with a file system which is different from what the volume is actually formatted, will cause mount failure.

Example

```
switch(appState)
{
    case TRY_MOUNT:
        if(SYS_FS_Mount("/dev/mmcblk1", "/mnt/myDrive", FAT, 0, NULL) != SYS_FS_RES_SUCCESS)
        {
            // Failure, try mounting again
        }
        else
        {
            // Mount was successful. Do further file operations
            appState = DO_FURTHER_STUFFS;
        }
        break;
}
```

Parameters

Parameters	Description
devName	The device name (name of volume) which needs to be mounted. The devName has to be preceded by the string "/dev/".
mountName	Mount name for the device to be mounted. This is a name provided by the user. In future, while accessing the mounted volume (say, during SYS_FS_FileOpen operation), the mountName is used to refer the path for file. The mount name has to be preceded by the string "/mnt/".
filesystemtype	Native file system of SYS_FS_FILE_SYSTEM_TYPE type.
mountflags	Mounting control flags. This parameter is reserved for future enhancements. Therefore, always pass zero.
data	The data argument is interpreted by the different file systems. This parameter is reserved for future enhancements. Therefore, always pass NULL.

Function

[SYS_FS_RESULT](#) SYS_FS_Mount
(

```
const char *devName,  
const char *mountName,  
    SYS_FS_FILE_SYSTEM_TYPE filesystemtype,  
unsigned long mountflags,  
const void *data  
);
```

SYS_FS_Tasks Function

Maintains the File System tasks and functionalities.

File

[sys_fs.h](#)

C

```
void SYS_FS_Tasks ( );
```

Returns

None.

Description

This function is used to run the various tasks and functionalities of sys_fs layer.

Remarks

This function is not called directly by an application. It is called by the system's Tasks routine ([SYS_Tasks](#)).

Preconditions

The [SYS_FS_Initialize](#) routine must have been called before running the tasks.

Example

```
void SYS_Tasks ( void )  
{  
    SYS_FS_Tasks ( );  
    // Do other tasks  
}
```

Function

```
void SYS_FS_Tasks  
(  
void  
);
```

SYS_FS_Unmount Function

Unmounts the file system.

File

[sys_fs.h](#)

C

```
SYS_FS_RESULT SYS_FS_Unmount(const char * mountName);
```

Returns

SYS_FS_RES_SUCCESS - Unmount was successful. SYS_FS_RES_FAILURE - Unmount was unsuccessful. The reason for the failure can be retrieved with [SYS_FS_Error](#).

Description

This function removes (unmounts) the attachment of the volume from the file system.

Preconditions

The volume name has to be known in order to pass as input to Unmount. The specified volume name to be unmounted should have been already mounted.

Example

```
if(SYS_FS_Unmount("/mnt/myDrive") != SYS_FS_RES_SUCCESS)
{
    // Failure, try unmounting again
}
else
{
    // Unmount was successful.
}
```

Parameters

Parameters	Description
mountName	Mount name for the volume to be unmounted. The mount name has to be preceded by the string "/mnt/".

Function

```
SYS_FS_RESULT SYS_FS_Unmount
(
    const char *mountName
);
```

SYS_FS_CurrentDriveGet Function

Gets the current drive

File

[sys_fs.h](#)

C

```
SYS_FS_RESULT SYS_FS_CurrentDriveGet(char* buffer);
```

Returns

SYS_FS_RES_SUCCESS - Current drive get operation was successful. SYS_FS_RES_FAILURE - Current drive get operation was unsuccessful. The reason for the failure can be retrieved with [SYS_FS_Error](#).

Description

This function gets the present drive being used. The drive information is populated in the buffer.

Remarks

None.

Preconditions

The disk has to be mounted.

Example

```
SYS_FS_RESULT res;
char buffer[255];

res = SYS_FS_CurrentDriveGet(buffer);
if(res == SYS_FS_RES_FAILURE)
{
    // Operation failed.
}
```

Parameters

Parameters	Description
buffer	Pointer to buffer which will hold the name of present drive being used.

Function

```
SYS_FS_RESULT SYS_FS_CurrentDriveGet
(
```

```
char* buffer
);
```

SYS_FS_CurrentDriveSet Function

Sets the drive.

File

[sys_fs.h](#)

C

```
SYS_FS_RESULT SYS_FS_CurrentDriveSet(const char* path);
```

Returns

SYS_FS_RES_SUCCESS - Current drive set operation was successful. SYS_FS_RES_FAILURE - Current drive set operation was unsuccessful. The reason for the failure can be retrieved with [SYS_FS_Error](#).

Description

This function sets the present drive to the one as specified by the path. By default, the drive mounted last becomes the current drive for the system. This is useful for applications where only one drive (volume) is used. In such an application, there is no need to call the SYS_FS_CurrentDriveSet function. However, in the case of an application where there are multiple volumes, the user can select the current drive for the application by calling this function.

Remarks

None.

Preconditions

The disk has to be mounted.

Example

```
SYS_FS_RESULT res;

res = SYS_FS_CurrentDriveSet("/mnt/myDrive");
if(res == SYS_FS_RES_FAILURE)
{
    // Drive change failed
}
```

Parameters

Parameters	Description
path	Path for the drive to be set.

Function

```
SYS_FS_RESULT SYS_FS_CurrentDriveSet
(
    const char* path
);
```

SYS_FS_DriveLabelGet Function

Gets the drive label.

File

[sys_fs.h](#)

C

```
SYS_FS_RESULT SYS_FS_DriveLabelGet(const char* drive, char * buff, uint32_t * sn);
```

Returns

SYS_FS_RES_SUCCESS - Drive label information retrieval was successful. SYS_FS_RES_FAILURE - Drive label information retrieval was unsuccessful. The reason for the failure can be retrieved with [SYS_FS_Error](#).

Description

This function gets the label for the drive specified. If no drive is specified, the label for the current drive is obtained.

Remarks

None.

Preconditions

At least one disk must be mounted.

Example

```
SYS_FS_RESULT res;
char buffer[255];
uint32_t serialNo;

switch(appState)
{
    case TRY_MOUNT:
        if(SYS_FS_Mount("/dev/mmcb1kal", "/mnt/myDrive", FAT, 0, NULL) != SYS_FS_RES_SUCCESS)
        {
            // Failure, try mounting again
        }
        else
        {
            // Mount was successful. Get label now
            appState = GET_LABEL;
        }
        break;

    case GET_LABEL:
        res = SYS_FS_DriveLabelGet("/mnt/myDrive", buffer, &serialNo);

        if(res == SYS_FS_RES_FAILURE)
        {
            // Fetching drive label information failed
        }
        //...
        //...
        break;
}
```

Parameters

Parameters	Description
drive	Pointer to buffer which will hold the name of drive being for which the label is requested. If this string is NULL, then then label of the current drive is obtained by using this function.
buff	Buffer which will hold the string of label.
sn	Serial number of the drive. If this information is not needed, it can be set as NULL.

Function

```
SYS_FS_RESULT SYS_FS_DriveLabelGet
(
    const char* drive,
    char *buff,
    uint32_t *sn
);
```

SYS_FS_DriveLabelSet Function

Sets the drive label

File

sys_fs.h

C

```
SYS_FS_RESULT SYS_FS_DriveLabelSet(const char * drive, const char * label);
```

Returns

SYS_FS_RES_SUCCESS - Drive label set operation was successful. SYS_FS_RES_FAILURE - Drive label set operation was unsuccessful. The reason for the failure can be retrieved with [SYS_FS_Error](#).

Description

This function sets the label for the drive specified. If no drive is specified, the label for the current drive is set.

Remarks

None.

Preconditions

At least one disk must be mounted.

Example

```
SYS_FS_RESULT res;

switch(appState)
{
    case TRY_MOUNT:
        if(SYS_FS_Mount("/dev/mmcblk1", "/mnt/myDrive", FAT, 0, NULL) != SYS_FS_RES_SUCCESS)
        {
            // Failure, try mounting again
        }
        else
        {
            // Mount was successful. Get label now
            appState = GET_LABEL;
        }
        break;

    case GET_LABEL:
        res = SYS_FS_DriveLabelSet("/mnt/myDrive", "MY_LABEL");
        if(res == SYS_FS_RES_FAILURE)
        {
            // Drive label get failed
        }
        //...
        //...
        break;
}
```

Parameters

Parameters	Description
drive	Pointer to string that holds the name of drive being for which the label is to be set. If this string is NULL, the label of the current drive is set by using this function.
label	Pointer to string which contains the label to be set.

Function

```
SYS_FS_RESULT SYS_FS_DriveLabelSet
(
    const char* drive,
    const char *label
);
```

SYS_FS_DriveFormat Function

Formats a drive.

File

[sys_fs.h](#)

C

```
SYS_FS_RESULT SYS_FS_DriveFormat(const char* drive, SYS_FS_FORMAT fmt, uint32_t clusterSize);
```

Returns

SYS_FS_RES_SUCCESS - Drive format was successful. SYS_FS_RES_FAILURE - Drive format was unsuccessful. The reason for the failure can be retrieved with [SYS_FS_Error](#).

Description

This function formats a logic drive (create a FAT file system on the logical drive), as per the format specified.

If the logical drive that has to be formatted has been bound to any partition (1-4) by multiple partition feature, the FAT volume is created into the specified partition. In this case, the second argument `fmt` is ignored. The physical drive must have been partitioned prior to using this function.

Remarks

None.

Preconditions

At least one disk must be mounted. The physical drive must have already been partitioned.

Example

```
SYS_FS_RESULT res;

switch(appState)
{
    case TRY_MOUNT:
        if(SYS_FS_Mount("/dev/mmcblk1", "/mnt/myDrive", FAT, 0, NULL) != SYS_FS_RES_SUCCESS)
        {
            // Failure, try mounting again
        }
        else
        {
            // Mount was successful. Format now.
            appState = FORMAT_DRIVE;
        }
        break;

    case FORMAT_DRIVE:
        res = SYS_FS_DriveFormat("/mnt/myDrive", SYS_FS_FORMAT_SFD, 0);
        if(res == SYS_FS_RES_FAILURE)
        {
            // Format of the drive failed.
        }
        //...
        break;
}
```

Parameters

Parameters	Description
drive	Pointer to buffer which will hold the name of drive being for which the format is to be done. If this string is NULL, then then current drive will be formatted. It is important to end the drive name with a "/".
fmt	Format type.
clusterSize	Cluster size. The value must be sector (size * n), where n is 1 to 128 and power of 2. When a zero is given, the cluster size depends on the volume size.

Function

```
SYS_FS_RESULT SYS_FS_DriveFormat
(
    const char* drive,
    SYS_FS_FORMAT fmt,
    uint32_t clusterSize
);
```

SYS_FS_DrivePartition Function

Partitions a physical drive (media).

File

[sys_fs.h](#)

C

```
SYS_FS_RESULT SYS_FS_DrivePartition(const char * path, const uint32_t partition[], void * work);
```

Returns

SYS_FS_RES_SUCCESS - Partition was successful. SYS_FS_RES_FAILURE - Partition was unsuccessful. The reason for the failure can be retrieved with [SYS_FS_Error](#).

Description

This function partitions a physical drive (media) into requested partition sizes. This function will alter the MBR of the physical drive and make it into multi partitions. Windows operating systems do not support multi partitioned removable media. Maximum 4 partitions can be created on a media.

Remarks

None

Preconditions

Prior to partitioning the media, the media should have a valid MBR and it should be mounted as a volume with the file system.

Example

```
//=====
// Initially, consider the case of a SD card that has only one partition.
//=====
SYS_FS_RESULT res;

// Following 4 element array specifies the size of 2 partitions as
// 256MB (=524288 sectors). The 3rd and 4th partition are not created
// since, the sizes of those are zero.
uint32_t plist[] = {524288, 524288, 0, 0};

// Work area for function SYS_FS_DrivePartition
char work[FAT_FS_MAX_SS];

switch(appState)
{
    case TRY_MOUNT:
        if(SYS_FS_Mount("/dev/mmcbk1a1", "/mnt/myDrive", FAT, 0, NULL) != SYS_FS_RES_SUCCESS)
        {
            // Failure, try mounting again
        }
        else
        {
            // Mount was successful. Partition now.
            appState = PARTITION_DRIVE;
        }
        break;

    case PARTITION_DRIVE:
        res = SYS_FS_DrivePartition("/mnt/myDrive", plist, work);
        if(res == SYS_FS_RES_FAILURE)
        {
            // Drive partition went wrong
        }
        else
        {
            // Partition was successful. Power cycle the board so that
            // all partitions are recognized. Then try mounting both
            // partitions.
        }
        break;
}
```

```

    default:
        break;
}

//=====
//The following code is after the SD card is partitioned and then
//powered ON.
//=====
SYS_FS_RESULT res;

switch(appState)
{
    case TRY_MOUNT_1ST_PARTITION:
        if(SYS_FS_Mount("/dev/mmcblk1", "/mnt/myDrive1", FAT, 0, NULL) != SYS_FS_RES_SUCCESS)
        {
            // Failure, try mounting again
            appState = TRY_MOUNT_1ST_PARTITION;
        }
        else
        {
            // Mount was successful. Mount second partition.
            appState = TRY_MOUNT_2ND_PARTITION;
        }
        break;

    case TRY_MOUNT_2ND_PARTITION:
        if(SYS_FS_Mount("/dev/mmcblk2", "/mnt/myDrive2", FAT, 0, NULL) != SYS_FS_RES_SUCCESS)
        {
            // Failure, try mounting again
            appState = TRY_MOUNT_2ND_PARTITION;
        }
        else
        {
            // Mount was successful. Try formatting first partition.
            appState = TRY_FORMATING_1ST_PARTITION;
        }
        break;

    case TRY_FORMATING_1ST_PARTITION:
        if(SYS_FS_DriveFormat("/mnt/myDrive1/", SYS_FS_FORMAT_FDISK, 0) == SYS_FS_RES_FAILURE)
        {
            // Failure
        }
        else
        {
            // Try formatting second partitions.
            appState = TRY_FORMATING_2ND_PARTITION;
        }

    case TRY_FORMATING_2ND_PARTITION:
        if(SYS_FS_DriveFormat("/mnt/myDrive2/", SYS_FS_FORMAT_FDISK, 0) == SYS_FS_RES_FAILURE)
        {
            // Failure
        }
        else
        {
            // Use both partitions as 2 separate volumes.
        }

    default:
        break;
}

```

Parameters

Parameters	Description
path	Path to the volume with the volume name. The string of volume name has to be preceded by "/mnt". Also, the volume name and directory name has to be separated by a slash "/".

partition	Array with 4 items, where each item mentions the sizes of each partition in terms of number of sector. 0th element of array specifies the number of sectors for first partition and 3rd element of array specifies the number of sectors for fourth partition.
work	Pointer to the buffer for function work area. The size must be at least FAT_FS_MAX_SS bytes.

Function

```

SYS_FS_RESULT SYS_FS_DrivePartition
(
    const char *path,
    const uint32_t partition[],
    void * work
);

```

SYS_FS_DriveSectorGet Function

Obtains total number of sectors and number of free sectors for the specified drive.

File

[sys_fs.h](#)

C

```

SYS_FS_RESULT SYS_FS_DriveSectorGet(const char * path, uint32_t * totalSectors, uint32_t * freeSectors);

```

Returns

SYS_FS_RES_SUCCESS - Sector information get operation was successful. SYS_FS_RES_FAILURE - Sector information get operation was unsuccessful. The reason for the failure can be retrieved with [SYS_FS_Error](#).

Description

Function to obtain the total number of sectors and number of free sectors in a drive (media).

Remarks

None.

Preconditions

The drive for which the information is to be retrieved should be mounted.

Example

```

uint32_t totalSectors, freeSectors;
SYS_FS_RESULT res;

if(SYS_FS_Mount("/dev/mmcblk1", "/mnt/myDrive", FAT, 0, NULL) != SYS_FS_RES_SUCCESS)
{
    // Failure, try mounting again
}
else
{
    // Mount was successful.
    // Do other FS stuffs.
}
// Perform usual FS tasks.
//....
//....

// Now, determine the total sectors and free sectors
res = SYS_FS_DriveSectorGet("/mnt/myDrive", &totalSectors, &freeSectors);
if(res == SYS_FS_RES_FAILURE)
{
    //Sector information get operation failed.
}

```

Parameters

Parameters	Description
path	Path to the volume with the volume name. The string of volume name must be preceded by "/mnt/". Also, the volume name and directory name must be separated by a slash "/".
totalSectors	Pointer to a variable passed to the function, which will contain the total number of sectors available in the drive (media).
freeSectors	Pointer to a variable passed to the function, which will contain the free number of sectors available in the drive (media).

Function

```
SYS_FS_RESULT SYS_FS_DriveSectorGet
(
    const char* path,
    uint32_t *totalSectors,
    uint32_t *freeSectors
);
```

SYS_FS_EventHandlerSet Function

Allows a client to identify an event handling function for the file system to call back when mount/unmount operation has completed.

File

sys_fs.h

C

```
void SYS_FS_EventHandlerSet(const void * eventHandler, const uintptr_t context);
```

Returns

None.

Description

This function allows a client to identify an event handling function for the File System to call back when mount/unmount operation has completed. The file system will pass mount name back to the client by calling "eventHandler".

Remarks

If the client does not want to be notified when the mount/unmount operation has completed, it does not need to register a callback.

Preconditions

The [SYS_FS_Initialize\(\)](#) routine must have been called.

Example

```
// Client registers an event handler with file system. This is done once.
SYS_FS_EventHandlerSet(APP_SysFSEventHandler, (uintptr_t)NULL);

// Event Processing Technique. Event is received when operation is done.
void APP_SysFSEventHandler
(
    SYS_FS_EVENT event,
    void* eventData,
    uintptr_t context
)
{
    switch(event)
    {
        case SYS_FS_EVENT_MOUNT:
            if(0 == strcmp((const char *)mountName, "/mnt/myDrive1"))
            {
                gSDCardMountFlag = true;
            }
            else if(0 == strcmp((const char *)mountName, "/mnt/myDrive2"))
            {
                gNVMMountFlag = true;
            }
    }
}
```

```
        }
        break;

    case SYS_FS_EVENT_UNMOUNT:
        if(0 == strcmp((const char *)mountName, "/mnt/myDrive1"))
        {
            gSDCardMountFlag = false;
        }
        else if(0 == strcmp((const char *)mountName, "/mnt/myDrive2"))
        {
            gNVMMountFlag = false;
        }
        appData.state = APP_ERROR;
        break;

    case SYS_FS_EVENT_ERROR:
        break;

    }
}
```

Parameters

Parameters	Description
eventHandler	Pointer to the event handler function implemented by the user
context	The value of parameter will be passed back to the client unchanged, when the eventHandler function is called. It can be used to identify any client specific data object that identifies the instance of the client module (for example, it may be a pointer to the client module's state structure).

Function

```
void SYS_FS_EventHandlerSet
(
    const void * eventHandler,
    const uintptr_t context
);
```

c) Data Types and Constants

SYS_FS_ERROR Enumeration

Lists the various error cases.

File

```
sys_fs.h
```

C

```
typedef enum {
    SYS_FS_ERROR_OK = 0,
    SYS_FS_ERROR_DISK_ERR,
    SYS_FS_ERROR_INT_ERR,
    SYS_FS_ERROR_NOT_READY,
    SYS_FS_ERROR_NO_FILE,
    SYS_FS_ERROR_NO_PATH,
    SYS_FS_ERROR_INVALID_NAME,
    SYS_FS_ERROR_DENIED,
    SYS_FS_ERROR_EXIST,
    SYS_FS_ERROR_INVALID_OBJECT,
    SYS_FS_ERROR_WRITE_PROTECTED,
    SYS_FS_ERROR_INVALID_DRIVE,
    SYS_FS_ERROR_NOT_ENABLED,
    SYS_FS_ERROR_NO_FILESYSTEM,
    SYS_FS_ERROR_FORMAT_ABORTED,
    SYS_FS_ERROR_TIMEOUT,
    SYS_FS_ERROR_LOCKED,
    SYS_FS_ERROR_NOT_ENOUGH_CORE,
    SYS_FS_ERROR_TOO_MANY_OPEN_FILES,
```

```

SYS_FS_ERROR_INVALID_PARAMETER,
SYS_FS_ERROR_NOT_ENOUGH_FREE_VOLUME,
SYS_FS_ERROR_FS_NOT_SUPPORTED,
SYS_FS_ERROR_FS_NOT_MATCH_WITH_VOLUME,
SYS_FS_ERROR_NOT_SUPPORTED_IN_NATIVE_FS
} SYS_FS_ERROR;

```

Members

Members	Description
SYS_FS_ERROR_OK = 0	Success
SYS_FS_ERROR_DISK_ERR	(1) A hard error occurred in the low level disk I/O layer
SYS_FS_ERROR_INT_ERR	(2) Assertion failed
SYS_FS_ERROR_NOT_READY	(3) The physical drive cannot work
SYS_FS_ERROR_NO_FILE	(4) Could not find the file
SYS_FS_ERROR_NO_PATH	(5) Could not find the path
SYS_FS_ERROR_INVALID_NAME	(6) The path name format is invalid
SYS_FS_ERROR_DENIED	(7) Access denied due to prohibited access or directory full
SYS_FS_ERROR_EXIST	(8) Access denied due to prohibited access
SYS_FS_ERROR_INVALID_OBJECT	(9) The file/directory object is invalid
SYS_FS_ERROR_WRITE_PROTECTED	(10) The physical drive is write protected
SYS_FS_ERROR_INVALID_DRIVE	(11) The logical drive number is invalid
SYS_FS_ERROR_NOT_ENABLED	(12) The volume has no work area
SYS_FS_ERROR_NO_FILESYSTEM	(13) There is no valid volume
SYS_FS_ERROR_FORMAT_ABORTED	(14) The Format() aborted due to any parameter error
SYS_FS_ERROR_TIMEOUT	(15) Could not get a grant to access the volume within defined period
SYS_FS_ERROR_LOCKED	(16) The operation is rejected according to the file sharing policy
SYS_FS_ERROR_NOT_ENOUGH_CORE	(17) LFN working buffer could not be allocated
SYS_FS_ERROR_TOO_MANY_OPEN_FILES	(18) Number of open files
SYS_FS_ERROR_INVALID_PARAMETER	(19) Given parameter is invalid
SYS_FS_ERROR_NOT_ENOUGH_FREE_VOLUME	(20) Too many mounts requested. Not enough free volume available
SYS_FS_ERROR_FS_NOT_SUPPORTED	(21) Requested native file system is not supported
SYS_FS_ERROR_FS_NOT_MATCH_WITH_VOLUME	(22) Requested native file system does not match the format of volume
SYS_FS_ERROR_NOT_SUPPORTED_IN_NATIVE_FS	(23) Function not supported in native file system layer

Description

File Error enumeration

This enumeration lists the various error cases. When the application calls for a file system function which has a return type of [SYS_FS_RESULT](#) and if the return value is [SYS_FS_RES_FAILURE](#), the application can know the specific reason for failure by calling the [SYS_FS_FileError](#) function. The return value of [SYS_FS_FileError](#) function will be one of the enumeration of type [SYS_FS_ERROR](#).

Remarks

None.

SYS_FS_FILE_OPEN_ATTRIBUTES Enumeration

Lists the various attributes (modes) in which a file can be opened.

File

[sys_fs.h](#)

C

```

typedef enum {
    SYS_FS_FILE_OPEN_READ = 0,
    SYS_FS_FILE_OPEN_WRITE,
    SYS_FS_FILE_OPEN_APPEND,
    SYS_FS_FILE_OPEN_READ_PLUS,
    SYS_FS_FILE_OPEN_WRITE_PLUS,
    SYS_FS_FILE_OPEN_APPEND_PLUS
} SYS_FS_FILE_OPEN_ATTRIBUTES;

```

Members

Members	Description
SYS_FS_FILE_OPEN_READ = 0	reading the file = possible, if file exists. reading the file = file open returns error, if file does not exist. writing to the file = not possible. Write operation returns error
SYS_FS_FILE_OPEN_WRITE	reading the file = not possible. Read operation returns error. writing to the file = possible. If file exists, write happens from the beginning of the file, overwriting the existing content of the file. writing to the file = If file does not exist, a new file will be created and data will be written into the newly created file.
SYS_FS_FILE_OPEN_APPEND	reading the file = not possible. Read operation returns error writing to the file = possible. If file exists, write happens from the end of the file, preserving the existing content of the file. writing to the file = If file does not exist, a new file will be created and data will be written into the newly created file.
SYS_FS_FILE_OPEN_READ_PLUS	reading the file = possible, if file exists. reading the file = file open returns error, if file does not exist. writing to the file = possible, if file exists, starting from the beginning of the file (overwriting). writing to the file = file open returns error, if file does not exist.
SYS_FS_FILE_OPEN_WRITE_PLUS	reading the file = possible, if file exists. reading the file = If file does not exist, a new file will be created. writing to the file = possible. If file exists, write happens from the beginning of the file, overwriting the existing content of the file. writing to the file = If file does not exist, a new file will be created and data will be written into the newly created file.
SYS_FS_FILE_OPEN_APPEND_PLUS	reading the file = possible, if file exists. File read pointer will be moved to end of the file in this mode. reading the file = If file does not exist, a new file will be created. writing to the file = possible. If file exists, write happens from the end of the file, preserving the existing content of the file. writing to the file = If file does not exist, a new file will be created and data will be written into the newly created file.

Description

File open attributes

This enumeration lists the various attributes (modes) in which a file can be opened.

Remarks

None.

SYS_FS_FILE_SEEK_CONTROL Enumeration

Lists the various modes of file seek.

File

[sys_fs.h](#)

C

```
typedef enum {
    SYS_FS_SEEK_SET,
    SYS_FS_SEEK_CUR,
    SYS_FS_SEEK_END
} SYS_FS_FILE_SEEK_CONTROL;
```

Members

Members	Description
SYS_FS_SEEK_SET	Set file offset to input number of bytes from the start of file
SYS_FS_SEEK_CUR	Set file offset to its current location plus input number of bytes
SYS_FS_SEEK_END	Set file offset to size of the file plus input number of bytes

Description

File Seek control

This enumeration lists the various modes of file seek. When the application calls the [SYS_FS_FileSeek](#) function, it specifies the kind of seek that needs to be performed.

Remarks

None.

SYS_FS_FILE_SYSTEM_TYPE Enumeration

Enumerated data type identifying native file systems supported.

File

[sys_fs.h](#)

C

```
typedef enum {
    UNSUPPORTED_FS = 0,
    FAT,
    MPFS2
} SYS_FS_FILE_SYSTEM_TYPE;
```

Members

Members	Description
UNSUPPORTED_FS = 0	Unsupported File System
FAT	FAT FS native File system
MPFS2	MPFS2 native File system

Description

File System type

These enumerated values identify the native file system supported by the SYS FS.

Remarks

None.

SYS_FS_FSTAT Structure

File System status

File

[sys_fs.h](#)

C

```
typedef struct {
    uint32_t fsize;
    uint16_t fdate;
    uint16_t ftime;
    uint8_t fattrib;
    char fname[13];
    char * lfname;
    uint32_t lfsize;
} SYS_FS_FSTAT;
```

Members

Members	Description
uint32_t fsize;	File size
uint16_t fdate;	Last modified date
uint16_t ftime;	Last modified time
uint8_t fattrib;	Attribute
char fname[13];	Short file name (8.3 format)
char * lfname;	Pointer to the LFN buffer
uint32_t lfsize;	Size of LFN buffer in TCHAR

Description

SYS FS File status structure

This structure holds the various status of a file. The [SYS_FS_FileStat](#) () populates the contents of this structure.

Remarks

None.

SYS_FS_FUNCTIONS Structure

SYS FS Function signature structure for native file systems.

File

[sys_fs.h](#)

C

```
typedef struct {
    int (* mount)(uint8_t vol);
    int (* unmount)(uint8_t vol);
    int (* open)(uintptr_t handle, const char* path, uint8_t mode);
    int (* read)(uintptr_t fp, void* buff, uint32_t btr, uint32_t *br);
    int (* write)(uintptr_t fp, const void* buff, uint32_t btw, uint32_t* bw);
    int (* close)(uintptr_t fp);
    int (* seek)(uintptr_t handle, uint32_t offset);
    uint32_t (* tell)(uintptr_t handle);
    bool (* eof)(uintptr_t handle);
    uint32_t (* size)(uintptr_t handle);
    int (* fstat)(const char* path, uintptr_t fno);
    int (* mkdir)(const char *path);
    int (* chdir)(const char *path);
    int (* remove)(const char *path);
    int (* getlabel)(const char *path, char *buff, uint32_t *sn);
    int (* setlabel)(const char *label);
    int (* truncate)(uintptr_t handle);
    int (* currWD)(char* buff, uint32_t len);
    int (* chdrive)(uint8_t drive);
    int (* chmode)(const char* path, uint8_t attr, uint8_t mask);
    int (* ctime)(const char* path, uintptr_t ptr);
    int (* rename)(const char *oldPath, const char *newPath);
    int (* sync)(uintptr_t fp);
    char * (* getstrn)(char* buff, int len, uintptr_t handle);
    int (* putchar)(char c, uintptr_t handle);
    int (* putstrn)(const char* str, uintptr_t handle);
    int (* formattedprint)(uintptr_t handle, const char *str, ...);
    bool (* testerror)(uintptr_t handle);
    int (* formatDisk)(uint8_t vol, uint8_t sfd, uint32_t au);
    int (* opendir)(uintptr_t handle, const char *path);
    int (* readDir)(uintptr_t handle, uintptr_t stat);
    int (* closeDir)(uintptr_t handle);
    int (* partitionDisk)(uint8_t pdrv, const uint32_t szt[], void* work);
    int (* getCluster)(const char *path, uint32_t *tot_sec, uint32_t *free_sec);
} SYS_FS_FUNCTIONS;
```

Members

Members	Description
int (* mount)(uint8_t vol);	Function pointer of native file system for mounting a volume
int (* unmount)(uint8_t vol);	Function pointer of native file system for unmounting a volume
int (* open)(uintptr_t handle, const char* path, uint8_t mode);	Function pointer of native file system for opening a file
int (* read)(uintptr_t fp, void* buff, uint32_t btr, uint32_t *br);	Function pointer of native file system for reading a file
int (* write)(uintptr_t fp, const void* buff, uint32_t btw, uint32_t* bw);	Function pointer of native file system for writing to a file
int (* close)(uintptr_t fp);	Function pointer of native file system for closing a file
int (* seek)(uintptr_t handle, uint32_t offset);	Function pointer of native file system for moving the file pointer by a <ul style="list-style-type: none"> desired offset
uint32_t (* tell)(uintptr_t handle);	Function pointer of native file system for finding the position of the <ul style="list-style-type: none"> file pointer
bool (* eof)(uintptr_t handle);	Function pointer of native file system to check if the end of file is <ul style="list-style-type: none"> reached
uint32_t (* size)(uintptr_t handle);	Function pointer of native file system to know the size of file
int (* fstat)(const char* path, uintptr_t fno);	Function pointer of native file system to know the status of file
int (* mkdir)(const char *path);	Function pointer of native file system to create a directory
int (* chdir)(const char *path);	Function pointer of native file system to change a directory
int (* remove)(const char *path);	Function pointer of native file system to remove a file or directory
int (* getlabel)(const char *path, char *buff, uint32_t *sn);	Function pointer of native file system to get the volume label
int (* setlabel)(const char *label);	Function pointer of native file system to set the volume label

int (* truncate)(uintptr_t handle);	Function pointer of native file system to truncate the file
int (* currWD)(char* buff, uint32_t len);	Function pointer of native file system to obtain the current working <ul style="list-style-type: none"> • directory
int (* chdrive)(uint8_t drive);	Function pointer of native file system to set the current drive
int (* chmode)(const char* path, uint8_t attr, uint8_t mask);	Function pointer of native file system to change the attribute for file <ul style="list-style-type: none"> • or directory
int (* ctime)(const char* path, uintptr_t ptr);	Function pointer of native file system to change the time for a file or <ul style="list-style-type: none"> • directory
int (* rename)(const char *oldPath, const char *newPath);	Function pointer of native file system to rename a file or directory
int (* sync)(uintptr_t fp);	Function pointer of native file system to flush file
char * (* getstrn)(char* buff, int len, uintptr_t handle);	Function pointer of native file system to read a string from a file
int (* putchar)(char c, uintptr_t handle);	Function pointer of native file system to write a character into a file
int (* putstrn)(const char* str, uintptr_t handle);	Function pointer of native file system to write a string into a file
int (* formattedprint)(uintptr_t handle, const char *str, ...);	Function pointer of native file system to print a formatted string to <ul style="list-style-type: none"> • file
bool (* testerror)(uintptr_t handle);	Function pointer of native file system to test an error in a file
int (* formatDisk)(uint8_t vol, uint8_t sfd, uint32_t au);	Function pointer of native file system to format a disk
int (* opendir)(uintptr_t handle, const char *path);	Function pointer of native file system to open a directory
int (* readDir)(uintptr_t handle, uintptr_t stat);	Function pointer of native file system to read a directory
int (* closeDir)(uintptr_t handle);	Function pointer of native file system to close an opened directory
int (* partitionDisk)(uint8_t pdrv, const uint32_t szt[], void* work);	Function pointer of native file system to partition a physical drive
int (* getCluster)(const char *path, uint32_t *tot_sec, uint32_t *free_sec);	Function pointer of native file system to get total sectors and free <ul style="list-style-type: none"> • sectors

Description

SYS FS Function signature structure for native file systems

The SYS FS layer supports functions from each native file system layer. This structure specifies the signature for each function from native file system (parameter that needs to be passed to each function and return type for each function). If a new native file system is to be integrated with the SYS FS layer, the functions should follow the signature.

The structure of function pointer for the two native file systems: FAT FS and MPFS2 is already provided in the respective source files for the native file system. Hence the following structure is not immediately useful for the user. But the explanation for the structure is still provided for advanced users who would wish to integrate a new native file system to the MPLAB Harmony File System framework.

Remarks

None.

SYS_FS_HANDLE Type

This type defines the file handle.

File

[sys_fs.h](#)

C

```
typedef uintptr_t SYS_FS_HANDLE;
```

Description

SYS FS File Handle

This type defines the file handle. File handle is returned by the File Open function on successful operation.

Remarks

None.

SYS_FS_REGISTRATION_TABLE Structure

The sys_fs layer has to be initialized by passing this structure with suitably initialized members.

File

[sys_fs.h](#)

C

```
typedef struct {  
    SYS_FS_FILE_SYSTEM_TYPE nativeFileSystemType;  
    const SYS_FS_FUNCTIONS * nativeFileSystemFunctions;  
} SYS_FS_REGISTRATION_TABLE;
```

Members

Members	Description
SYS_FS_FILE_SYSTEM_TYPE nativeFileSystemType;	Native file system of type SYS_FS_FILE_SYSTEM_TYPE
const SYS_FS_FUNCTIONS * nativeFileSystemFunctions;	Pointer to the structure of type SYS_FS_FUNCTIONS which has the list of

Description

SYS_FS_REGISTRATION_TABLE structure

When the SYS FS layer is initialized, it has to know the type of native file system it has to support and the list of functions for native file system. The members of this structure can be initialized with suitable values and then passed on to [SYS_FS_Initialize](#) initialization function. Please refer to the example code provided for [SYS_FS_Initialize](#).

Remarks

None.

SYS_FS_RESULT Enumeration

Lists the various results of a file operation.

File

[sys_fs.h](#)

C

```
typedef enum {  
    SYS_FS_RES_SUCCESS = 0,  
    SYS_FS_RES_FAILURE = -1  
} SYS_FS_RESULT;
```

Members

Members	Description
SYS_FS_RES_SUCCESS = 0	Operation succeeded
SYS_FS_RES_FAILURE = -1	Operation failed

Description

File operation result enum

This enumeration lists the various results of a file operation. When a file operation function is called from the application, and if the return type of the function is SYS_FS_RESULT, then the enumeration below specifies the possible values returned by the function.

Remarks

None.

FAT_FS_MAX_LFN Macro

Maximum length of the Long File Name.

File

[sys_fs.h](#)

C

```
#define FAT_FS_MAX_LFN 255
```

Description

FAT File System LFN (Long File Name) max length

Defines the maximum length of file name during LFN selection. Set the value to 255.

Remarks

None.

FAT_FS_MAX_SS Macro

Lists the definitions for FAT file system sector size.

File

[sys_fs.h](#)

C

```
#define FAT_FS_MAX_SS 512
```

Description

FAT File System Sector size

Maximum sector size to be handled. Always set the value of sector size to 512

Remarks

None.

FAT_FS_USE_LFN Macro

Lists the definitions for FAT file system LFN selection.

File

[sys_fs.h](#)

C

```
#define FAT_FS_USE_LFN 1
```

Description

FAT File System LFN (long file name) selection

The FAT_FS_USE_LFN option switches the LFN support. Set the value to 1.

Remarks

None.

SYS_FS_HANDLE_INVALID Macro

Invalid file handle

File

[sys_fs.h](#)

C

```
#define SYS_FS_HANDLE_INVALID ((SYS_FS_HANDLE)(-1))
```

Description

SYS FS File Invalid Handle

This value defines the invalid file handle. Invalid file handle is returned on an unsuccessful File Open operation.

Remarks

None.

SYS_FS_FILE_DIR_ATTR Enumeration

Enumerated data type identifying the various attributes for file/directory.

File

[sys_fs.h](#)

C

```
typedef enum {
    SYS_FS_ATTR_RDO = 0x01,
    SYS_FS_ATTR_HID = 0x02,
    SYS_FS_ATTR_SYS = 0x04,
    SYS_FS_ATTR_VOL = 0x08,
    SYS_FS_ATTR_LFN = 0x0F,
    SYS_FS_ATTR_DIR = 0x10,
    SYS_FS_ATTR_ARC = 0x20,
    SYS_FS_ATTR_MASK = 0x3F
} SYS_FS_FILE_DIR_ATTR;
```

Members

Members	Description
SYS_FS_ATTR_RDO = 0x01	Read only
SYS_FS_ATTR_HID = 0x02	Hidden
SYS_FS_ATTR_SYS = 0x04	System
SYS_FS_ATTR_VOL = 0x08	Volume label
SYS_FS_ATTR_LFN = 0x0F	LFN entry
SYS_FS_ATTR_DIR = 0x10	Directory
SYS_FS_ATTR_ARC = 0x20	Archive
SYS_FS_ATTR_MASK = 0x3F	Mask of defined bits

Description

File or directory attribute

These enumerated values are the possible attributes for a file or directory.

Remarks

None.

SYS_FS_TIME Union

The structure to specify the time for a file or directory.

File

[sys_fs.h](#)

C

```
typedef union {
    struct discreteTime {
        unsigned second : 5;
        unsigned minute : 6;
        unsigned hour : 5;
        unsigned day : 5;
        unsigned month : 4;
        unsigned year : 7;
    }
    struct timeDate {
        uint16_t time;
        uint16_t date;
    }
    uint32_t packedTime;
} SYS_FS_TIME;
```

Members

Members	Description
unsigned second : 5;	Second / 2 (0..29)
unsigned minute : 6;	Minute (0..59)
unsigned hour : 5;	Hour (0..23)
unsigned day : 5;	Day in month(1..31)
unsigned month : 4;	Month (1..12)
unsigned year : 7;	Year from 1980 (0..127)
uint16_t time;	Time (hour, min, seconds)
uint16_t date;	Date (year, month, day)
uint32_t packedTime;	Combined time information in a 32-bit value

Description

SYS FS File time structure

This structure holds the date and time to be used to set for a file or directory.

bits 31-25: Year from 1980 (0..127) bits 24-21: Month (1..12) bits 20-16: Day in month(1..31) bits 15-11: Hour (0..23) bits 10-5 : Minute (0..59) bits 4-0 : Seconds / 2 (0..29)

Remarks

None.

SYS_FS_FORMAT Enumeration

Specifies the partitioning rule.

File

[sys_fs.h](#)

C

```
typedef enum {  
    SYS_FS_FORMAT_FDISK = 0,  
    SYS_FS_FORMAT_SFD = 1  
} SYS_FS_FORMAT;
```

Members

Members	Description
SYS_FS_FORMAT_FDISK = 0	Format disk with multiple partition
SYS_FS_FORMAT_SFD = 1	Format disk with single partition

Description

File formatting partition rule

This type specifies the partitioning rule. When SYS_FS_FORMAT_FDISK format is specified, a primary partition occupying the entire disk space is created and then an FAT volume is created on the partition. When SYS_FS_FORMAT_SFD format is specified, the FAT volume starts from the first sector of the physical drive.

The SYS_FS_FORMAT_FDISK partitioning is usually used for hard disk, MMC, SDC, CFC and U Disk. It can divide a physical drive into one or more partitions with a partition table on the MBR. However Windows does not support multiple partition on the removable media. The SYS_FS_FORMAT_SFD is non-partitioned method. The FAT volume starts from the first sector on the physical drive without partition table. It is usually used for floppy disk, micro drive, optical disk, and super-floppy media.

SYS_FS_EVENT Enumeration

Identifies the possible file system events.

File

[sys_fs.h](#)

C

```
typedef enum {  
    SYS_FS_EVENT_MOUNT,  
    SYS_FS_EVENT_UNMOUNT,
```

```
SYS_FS_EVENT_ERROR
} SYS_FS_EVENT;
```

Members

Members	Description
SYS_FS_EVENT_MOUNT	Media has been mounted successfully.
SYS_FS_EVENT_UNMOUNT	Media has been unmounted successfully.
SYS_FS_EVENT_ERROR	There was an error during the operation

Description

SYS FS Media Events

This enumeration identifies the possible events that can result from a file system.

Remarks

One of these values is passed in the "event" parameter of the event handling callback function that client registered with the file system by setting the event handler when media mount or unmount is completed.

SYS_FS_EVENT_HANDLER Type

Pointer to the File system Handler function.

File

[sys_fs.h](#)

C

```
typedef void (* SYS_FS_EVENT_HANDLER)(SYS_FS_EVENT event, void* eventData, uintptr_t context);
```

Returns

None.

Description

File System Event Handler function pointer

This data type defines the required function signature for the file system event handling callback function. A client must register a pointer to an event handling function whose function signature (parameter and return value types) match the types specified by this function pointer in order to receive event call backs from the file system.

Remarks

None.

Parameters

Parameters	Description
event	Identifies the type of event
eventData	Handle returned from the media operation requests
context	Value identifying the context of the application that registered the event handling function

Files

Files

Name	Description
sys_fs.h	Functions and type declarations required to interact with the MPLAB Harmony File System Service.
sys_fs_config_template.h	File System Service configuration templates.

Description

This section lists the source and header files used by the library.

sys_fs.h



Functions and type declarations required to interact with the MPLAB Harmony File System Service.

Enumerations

	Name	Description
	SYS_FS_ERROR	Lists the various error cases.
	SYS_FS_EVENT	Identifies the possible file system events.
	SYS_FS_FILE_DIR_ATTR	Enumerated data type identifying the various attributes for file/directory.
	SYS_FS_FILE_OPEN_ATTRIBUTES	Lists the various attributes (modes) in which a file can be opened.
	SYS_FS_FILE_SEEK_CONTROL	Lists the various modes of file seek.
	SYS_FS_FILE_SYSTEM_TYPE	Enumerated data type identifying native file systems supported.
	SYS_FS_FORMAT	Specifies the partitioning rule.
	SYS_FS_RESULT	Lists the various results of a file operation.

Functions

	Name	Description
	SYS_FS_CurrentDriveGet	Gets the current drive
	SYS_FS_CurrentDriveSet	Sets the drive.
	SYS_FS_CurrentWorkingDirectoryGet	Gets the current working directory
	SYS_FS_DirClose	Closes an opened directory.
	SYS_FS_DirectoryChange	Changes to a the directory specified.
	SYS_FS_DirectoryMake	Makes a directory.
	SYS_FS_DirOpen	Open a directory
	SYS_FS_DirRead	Reads the specified directory.
	SYS_FS_DirRewind	Rewinds to the beginning of the directory.
	SYS_FS_DirSearch	Searches for a file or directory.
	SYS_FS_DriveFormat	Formats a drive.
	SYS_FS_DriveLabelGet	Gets the drive label.
	SYS_FS_DriveLabelSet	Sets the drive label
	SYS_FS_DrivePartition	Partitions a physical drive (media).
	SYS_FS_DriveSectorGet	Obtains total number of sectors and number of free sectors for the specified drive.
	SYS_FS_Error	Returns the last error.
	SYS_FS_EventHandlerSet	Allows a client to identify an event handling function for the file system to call back when mount/unmount operation has completed.
	SYS_FS_FileCharacterPut	Writes a character to a file.
	SYS_FS_FileClose	Closes a file.
	SYS_FS_FileDirectoryModeSet	Sets the mode for the file or directory.
	SYS_FS_FileDirectoryRemove	Removes a file or directory.
	SYS_FS_FileDirectoryRenameMove	Renames or moves a file or directory.
	SYS_FS_FileDirectoryTimeSet	Sets or changes the time for a file or directory.
	SYS_FS_FileEOF	Checks for end of file.
	SYS_FS_FileError	Returns the file specific error.
	SYS_FS_FileNameGet	Reads the file name.
	SYS_FS_FileOpen	Opens a file.
	SYS_FS_FilePrintf	Writes a formatted string into a file.
	SYS_FS_FileRead	Read data from the file.
	SYS_FS_FileSeek	Moves the file pointer by the requested offset.
	SYS_FS_FileSize	Returns the size of the file in bytes.
	SYS_FS_FileStat	Gets file status.
	SYS_FS_FileStringGet	Reads a string from the file into a buffer.
	SYS_FS_FileStringPut	Writes a string to a file.
	SYS_FS_FileSync	Flushes the cached information when writing to a file.
	SYS_FS_FileTell	Obtains the file pointer position.
	SYS_FS_FileTestError	Checks for errors in the file.
	SYS_FS_FileTruncate	Truncates a file
	SYS_FS_FileWrite	Writes data to the file.
	SYS_FS_Initialize	Initializes the file system abstraction layer (sys_fs layer).
	SYS_FS_Mount	Mounts the file system.

	SYS_FS_Tasks	Maintains the File System tasks and functionalities.
	SYS_FS_Unmount	Unmounts the file system.

Macros

	Name	Description
	FAT_FS_MAX_LFN	Maximum length of the Long File Name.
	FAT_FS_MAX_SS	Lists the definitions for FAT file system sector size.
	FAT_FS_USE_LFN	Lists the definitions for FAT file system LFN selection.
	SYS_FS_HANDLE_INVALID	Invalid file handle

Structures

	Name	Description
	SYS_FS_FSTAT	File System status
	SYS_FS_FUNCTIONS	SYS FS Function signature structure for native file systems.
	SYS_FS_REGISTRATION_TABLE	The sys_fs layer has to be initialized by passing this structure with suitably initialized members.

Types

	Name	Description
	SYS_FS_EVENT_HANDLER	Pointer to the File system Handler function.
	SYS_FS_HANDLE	This type defines the file handle.

Unions

	Name	Description
	SYS_FS_TIME	The structure to specify the time for a file or directory.

Description

File System Service Library Interface Declarations and Types

This file contains function and type declarations required to interact with the MPLAB Harmony File System Service.

File Name

sys_fs.h

Company

Microchip Technology Inc.

sys_fs_config_template.h

File System Service configuration templates.

Macros

	Name	Description
	SYS_FS_AUTOMOUNT_ENABLE	Enable/Disable Auto Mount Feature of File system
	SYS_FS_MAX_FILE_SYSTEM_TYPE	Number of file systems used in the application
	SYS_FS_MAX_FILES	Number of simultaneous files access
	SYS_FS_MEDIA_MAX_BLOCK_SIZE	Media Sector Size information
	SYS_FS_MEDIA_NUMBER	Number of media used in the application
	SYS_FS_VOLUME_NUMBER	This macro defines number of volumes used in the application

Description

File System Service Configuration Templates

This file contains constants to configure the File System Service.

File Name

sys_fs_config_template.h

Company

Microchip Technology Inc.

Interrupt System Service Library

This section describes the Interrupt System Service Library.

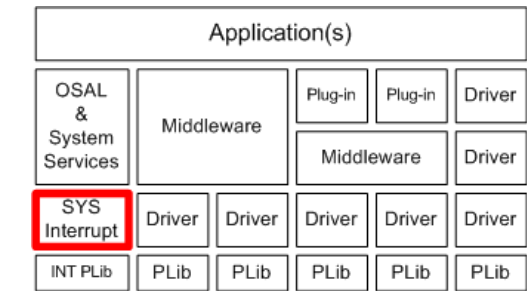
Introduction

Interrupt System Service for Microchip Microcontrollers

This library provides a low-level abstraction of the Interrupt System Service Library that is available on the Microchip family of PIC32 microcontrollers with a convenient C language interface. It can be used to simplify low-level access to the module without the necessity of interacting directly with the module's registers, thereby hiding differences from one microcontroller variant to another.

Description

This library implements the Interrupt System Service. It is part of the system services that provides support for processing interrupts. The Interrupt System provides support for initializing the processor's interrupt controller, registering Interrupt Service Routines (ISRs) and managing interrupts. These features enable making efficient and dynamic applications, drivers, and middleware that respond to external events as they occur in real time.



Using the Library

This topic describes the basic architecture of the Interrupt System Service Library and provides information and examples on its use.

Description

Interface Header File: [sys_int.h](#)

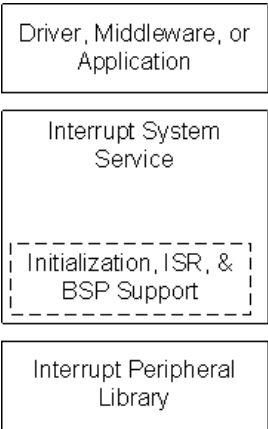
The interface to the Interrupt System Service library is defined in the [sys_int.h](#) header file, which is included by the `sys.h` system service header file. Any C language source (`.c`) file that uses the Interrupt System Service library should include `sys.h`.

Please refer to the What is MPLAB Harmony? section for how the library interacts with the framework.

Abstraction Model

This library provides an abstraction of the interrupt subsystem that is used by device drivers, middleware libraries and applications to receive and control interrupts in real time.

Description



Interrupt System Service

The interrupt system services provide support for initializing the processor's interrupt controller, managing Interrupt Service Routines (ISRs) and managing interrupts.

Initialization

Each software module (device driver, library, or application) that needs to receive an interrupt must enable that interrupt itself. This is normally done in the module's initialization routine which is called by the [SYS_Initialize](#) service.

A module that intends to use an interrupt must first register the Tasks function that is to be called when the desired source causes an interrupt. Then, it must enable that source, once it is ready to start receiving interrupts.

If the interrupt system service is configured for static usage, the routine that dynamically registers the Tasks function will be nulled out by a macro (generating no run-time code) and, instead, the Tasks routine must be called statically from the function that implements the raw ISR vector. How this is done is different for each processor family, as explained in the following section.

Interrupt Service Routine (ISR)

Each software module (device driver, library, or application) that needs to receive an interrupt must implement a Tasks routine to handle that interrupt. In order for the module to operate in an interrupt-driven mode, the Tasks routine must be called from within the appropriate "raw" Interrupt Service Routine (ISR).

How the raw ISR is implemented is highly dependent upon the specific processor being used. Libraries are available that implement raw ISRs for each processor family in a way that allows dynamic registration and deregistration of Tasks routines. These libraries maintain tables that associate the Tasks routine registered by the SYS INIT service with each interrupt source in the system.

Alternately, in a statically-linked system implementation, the ISR may be implemented by the system designer or integrator (in the configuration-specific `system_interrupt.c` file). Such "static" ISR implementations must identify the source of the interrupt then directly call the appropriate module's Tasks routine. This requires knowledge of the modules that have been included in the system and cannot be implemented in advance as a library.



Note: It is also possible, in a highly optimized system (or to support highly resource-restricted parts), to implement the logic of the module's Tasks routine directly in the raw ISR. However, this method is not recommended unless absolutely necessary to meet system timing requirements.

Board Support Packages (BSPs)

If the processor is affixed directly to the board, the BSP may also implement any required "raw" ISRs, eliminating the need for the system designer or integrator to implement the ISR(s) himself. Refer to the documentation for the BSP in use for details on what initialization and ISR support it provides. This support is not implemented by the Interrupt System Services library.

Library Overview

Refer to the [System Service Introduction](#) section for how the system services operates in a system.

The library interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the Interrupt System Service.

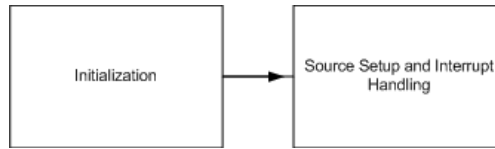
Library Interface Section	Description
Interrupt System Setup Functions	Provides processor specific initialization of the interrupt system
Global Interrupt Management Functions	Provide interface routines to enable/disable all interrupts on the system
Interrupt Source Management Functions	Provides setup and status routines for: <ul style="list-style-type: none">Setting up the source enable configurationQuerying the source interrupt status

How the Library Works

The Interrupt System Service Library can be used by a device driver, middleware layer, or application to provide access to, and control over, interrupts to the processor.

Description

The following diagram describes the major components of the usage model.



Note: Not all modes are available on all devices. Please refer to the specific device data sheet to determine the modes supported for your device.

Interrupt System Setup

The Interrupt System Service library must be initialized by calling the [SYS_INT_Initialize](#) function. If the MPLAB Harmony dynamic initialization service is used, the [SYS_INT_Initialize](#) function will be called automatically when the [SYS_Initialize](#) function is called. In a statically initialized system, the system designer or integrator must implement the [SYS_Initialize](#) function and that function must call [SYS_INT_Initialize](#) before initializing any modules that might require use of the interrupt system service. Once the library has been initialized, call the function [SYS_INT_Enable](#) to enable interrupts to the processor. However, before enabling the generation of interrupts to the processor, each individual module (driver, library, or application) must have a "Tasks" routine in place (either registered with [SYS_INT_DynamicRegister](#) or statically linked to the raw ISR) to handle the interrupt before it enables its own interrupt.

Example: Initializing the System Interrupt Library

```

SYS_INT_Initialize();
// Initialize all interrupt-aware software modules
SYS_INT_Enable();
  
```

Critical Sections

Critical Sections

Critical sections of code are small sections of code that must execute atomically, with no possibility of being interrupted. To support this, the following technique can be used.

Global Interrupt Management provides routines to create a global critical section of code.

Global Critical Section

If no interrupts of any kind can be allowed within a critical section of code, the following routines can be used to ensure this.

- [SYS_INT_Disable](#): To start a critical section, all interrupts are disabled with the call of this function
- [SYS_INT_Enable](#): To end a critical section, interrupts are enabled from the interrupt controller to the core
- [SYS_INT_IsEnabled](#): Status to indicate if whether or interrupts are currently enabled

Example: Global Critical Section

```

bool flag;
flag = SYS_INT_Disable();

// Do something critical

if (flag)
{
    SYS_INT_Enable();
}
  
```

Source Interrupt Management provides interface routines to create local critical sections.

Local Critical Sections

Normally, it is not necessary to globally disable all possible interrupts. For example, in a driver for a specific device, it is not normally important if an unrelated interrupt occurs in the middle of a critical section of code. However, if the interrupt for the source that the driver manages must not occur within a critical section of code, it can be protected using the following technique.

Example: Local Critical Section

```

bool flag;
// interrupt source enable status before disable is called
flag = SYS_INT_SourceDisable(MY_DRIVER_INTERRUPT_SOURCE);

// Do something critical

if (flag)
{
    SYS_INT_SourceEnable(MY_DRIVER_INTERRUPT_SOURCE);
}
  
```



Note: These methods of protecting critical sections is usually implemented as part of an Operating System Abstraction Layer (OSAL), so it is not normally necessary to use these examples explicitly. Normally, the OSAL will provide single functions or macros that implement this functionality. So, if available, an OSAL method is preferred over implementing the critical section code as shown in the previous examples.

Source Interrupt Management

The driver, middleware, or application's interrupt-handling Tasks routine must do two things at a minimum, in the following order.

1. Remove the cause of the interrupt.
2. Clear the interrupt source by calling the function [SYS_INT_SourceStatusClear](#).

Exactly what actions are necessary to remove the cause of an interrupt is completely dependent on the source of the interrupt. This is normally the main purpose of the driver itself and is beyond the scope of this section. Refer to the documentation for the peripheral being managed.



Warning

The cause of the interrupt must be removed before clearing the interrupt source or the interrupt may reoccur immediately after the source is cleared potentially causing an infinite loop. An infinite loop may also occur if the source is not cleared before the interrupt-handler returns.

Example: Handling Interrupts

```
void DRV_MYDEV_Tasks( SYS_MODULE_OBJ object )
{
    // Remove the cause of the interrupt
    //...

    // Clear Interrupt source
    SYS_INT_SourceStatusClear(myIntSourceID);
}
```



Note: The value of `myIntSourceID` is usually either a static or dynamic configuration option. Refer to the documentation for the specific device driver to identify how to define the interrupt source ID.

Testing Interrupt

Sometimes it is necessary to cause an interrupt in software, possibly for testing purposes. To support this, the function [SYS_INT_SourceStatusSet](#) is provided.

Example: Causing an Interrupt in Software

```
SYS_INT_SourceStatusSet(MY_DRIVER_INTERRUPT_SOURCE);
```



Note: This feature is not available for all interrupt sources on all Microchip microcontrollers. Refer to the specific device data sheet to determine whether it is possible for software to set a specific interrupt source.

Configuring the Library

This section provides information on configuring the Interrupt System Service Library.

Description

To use the Interrupt System Service Library, the following must be correctly configured:

- Select the Appropriate Processor
- Initialize the Interrupt System Service
- Configure the Raw ISR Support

Select the Appropriate Processor

The following data types are dependent on the processor selection and are actually defined in the Interrupt Peripheral Library for the specific microcontroller being used.

- INT_SOURCE
- INT_PRIORITY
- INT_SUBPRIORITY

These data types are configured by selecting the appropriate processor in MPLAB X IDE, which adds the "mprocessor" option to the compiler command line to identify the correct processor and processor-specific implementation of the peripheral library to use. Since the Interrupt System Service Library is part of the Microchip Firmware Framework, it will be built with the correct definition of these data types.

Initialize the Interrupt System Service

There are two ways to initialize the interrupt system service, depending on whether you are using a static configuration or a dynamic configuration.

For a Dynamic configuration the constant `SYS_INT_DYNAMIC` needs to be defined. This makes the `SYS_INT_DynamicRegister` and `SYS_INT_DynamicDeregister` functions available. The required driver tasks routines need to be registered using `SYS_INT_DynamicRegister` function.

For a Static configuration, the system designer or integrator must implement the `SYS_INT_Initialize` function. This function's purpose is to perform any actions necessary to initialize the interrupt subsystem and interrupt controller on the specific processor and system, usually interacting directly with the Interrupt Peripheral Library to accomplish these tasks.

Configure the Raw ISR Support

In some systems, there may only be a single actual (raw) ISR to handle all interrupts. In this sort of system, most of the Interrupt System Service Library may be implemented in software, with only the highest level interrupt being supported by hardware. In other systems, all interrupts may be supported by separate ISRs and vector selection and prioritization will be supported by hardware.

ISRs may be dynamically linked to specific interrupt sources or they may be statically linked at build time. If a dynamic interrupt library is used (by defining the constant `SYS_INT_DYNAMIC`), the calls to the `SYS_INT_DynamicRegister` function will register a pointer to the given Tasks routine for each registered interrupt source in an internal table. The dynamic library will then determine the source of the interrupt and call the given Tasks routine.

If a static configuration is desired, the "raw" ISR support must be implemented so that it directly calls (using static, build-time linkage) the appropriate module's Tasks routine. This requires the system implementer or integrator to implement the raw ISR, but it reduces the amount of overhead necessary to handle interrupts, reducing both interrupt latency and code size.

Static Configuration

When statically configuring raw ISR support, the system implementer or integrator must directly implement the raw ISRs in an appropriate manner for the selected processor. The raw ISR, must then call the appropriate Tasks routine to properly handle and clear the interrupt source.

Description

A static configuration of the raw ISR support for MPLAB Harmony requires processor-family-specific knowledge. Or, more accurately, it requires compiler-specific knowledge. The following example shows how to implement a raw ISR for the PIC32 family of devices. Refer to the compiler manual for details of how to implement an ISR.

Raw ISR Responsibilities:

- Identify the interrupt source
- Call the appropriate module's Tasks routine

The first thing a raw ISR must do is identify the source of the interrupt. Each interrupt source has its own interrupt "vector". This means that the only time a specific ISR is called is when a specific source has caused an interrupt. Therefore, the raw ISR can assume that every time it is called, its source has caused an interrupt. Once the raw ISR has identified the interrupt source, it must call the appropriate module's Tasks routine to service and clear the interrupt.

Example: PIC32 Timer1 Raw ISR

```
void __ISR ( _TIMER_1_VECTOR ) _InterruptHandler_TMR_1_stub( void )
{
    /* Call the timer driver's "Tasks" routine */
    DRV_TMR_Tasks ( gTMRObject );
}
```

In the example, `gTMRObject` holds the return value from the `DRV_TMR_Initialize` function.

The `SYS_INT_DynamicRegister` and `SYS_INT_DynamicDeregister` functions are macro switched to compile away to nothing if a static configuration is chosen.

Dynamic Configuration

When dynamically configuring raw ISR support, the system implementer or integrator must register each interrupt-driven driver or module's Tasks routine with the dynamic system interrupt service for the appropriate interrupt source. The dynamic SYS INT service will then ensure that the appropriate Tasks routine is called when an interrupt occurs.

Description

When using the dynamic system interrupt (SYS INT) service, it is not necessary to implement raw ISRs for interrupt-driven modules. The processor-family-specific, dynamic SYS INT implementation provided with MPLAB Harmony implements the raw ISRs so the system developer or integrator does not have to. Instead, the system developer must register the module's Tasks' routine using the `SYS_ModuleRegister` function after registering the module in the module registration routine (described in the SYS INIT documentation). The following example shows how a module must register its ISR Tasks routine.

Example: Dynamic Registration of an Interrupt-Driven Module

```
// Register the TMR driver's "Tasks" routine with the SYS INT service
SYS_INT_DynamicRegister(object, DRV_TMR_Tasks, PLIB_INT_SOURCE_TIMER_1);
```

The module init routine must register the module's Tasks routine with the SYS INT service instead of the SYS TASKS service. To do this, it calls the `SYS_INT_DynamicRegister` function, passing in the object (the same object handle returned by the module's initialization routine), along with a

pointer to the module's Tasks routine and the interrupt source with which it will be associated.



Note: Dynamic interrupt registration functionality is currently not supported in the Interrupt System Service.

Building the Library

This section lists the files that are available in the Interrupt System Service Library.

Description

The following three tables list and describe the header (.h) and source (.c) files that implement this library. The parent folder for these files is <install-dir>/framework/system/int.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using #include) by any code that uses this library.

Source File Name	Description
sys_int.h	Interrupt System Service Library API header file.

Required File(s)



All of the required files listed in the following table are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

Source File Name	Description
/src/sys_int_pic32.c	Interrupt System Service Library implementation.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
N/A	No optional files are available for this library.

Module Dependencies

The Interrupt System Service does not depend on any other modules.

Library Interface









a) Interrupt System Setup Functions

	Name	Description
	SYS_INT_Initialize	Configures and initializes the interrupt subsystem.
	SYS_INT_DynamicDeregister	Deregisters the current ISR from the given interrupt source.
	SYS_INT_DynamicRegister	Registers an Interrupt "Tasks" Routine for the specified interrupt source or trap).
	SYS_INT_ShadowRegisterAssign	Assigns a shadow register set for an interrupt priority level.
	SYS_INT_ShadowRegisterGet	Gets the shadow register set assigned for an interrupt priority level.
	SYS_INT_StatusGetAndDisable	Disables interrupts to the processor and return the previous status.
	SYS_INT_StatusRestore	Restores the processor status.

b) Global Interrupt Management Functions

	Name	Description
	SYS_INT_Disable	Disables interrupts to the processor.
	SYS_INT_Enable	Enables global interrupts to the processor.
	SYS_INT_IsEnabled	Identifies if interrupts are currently enabled or disabled at the top level.
	SYS_INT_ExternalInterruptTriggerSet	Sets the external interrupt trigger type.

c) Interrupt Source Management Functions

	Name	Description
	SYS_INT_SourceDisable	Disables the specified source from generating interrupts to the processor.
	SYS_INT_SourceEnable	Enables the specified source to generate interrupts to the processor.
	SYS_INT_SourceIsEnabled	Identifies if the specified source is enabled or disabled.
	SYS_INT_SourceStatusClear	Clears the interrupt request for the specified interrupt source.
	SYS_INT_SourceStatusGet	Determines the status of the specified interrupt source.
	SYS_INT_SourceStatusSet	Sets the specified interrupt source.
	SYS_INT_VectorPrioritySet	Sets the given interrupt vector to the specified priority.
	SYS_INT_VectorSubprioritySet	Sets the specified interrupt vector to the given sub priority.

d) Data Types and Constants

	Name	Description
	SYS_INT_TASKS_POINTER	Pointer to an interrupt-handling "Tasks" routine.
	INT_EXTERNAL_EDGE_TRIGGER	Lists the available external interrupt trigger options.
	SYS_INT_PROCESSOR_STATUS	CPU Processor status

Description

This section describes the APIs of the Interrupt System Service Library.

Refer to each section for a detailed description.

a) Interrupt System Setup Functions

SYS_INT_Initialize Function

Configures and initializes the interrupt subsystem.

File

[sys_int.h](#)

C

```
void SYS_INT_Initialize();
```

Returns

None.

Description

This function configures and initializes the interrupt subsystem appropriately for the current system design.

Remarks

None.

The System Interrupt library must be initialized by calling the SYS_INT_Initialize routine. This is normally done in the "SYS_Initialize" routine before any interrupt support is used. If the dynamic interrupt system service is not used, the "SYS_Initialize" routine must be implemented by the system designer or integrator as required by the system design and is not implemented by the System Interrupt library. The global interrupts are enabled as a part the call to "SYS_INT_Initialize". However, before enabling the generation of interrupts to the processor, each individual module (driver, library, or application) must have a "Tasks" routine to in place (statically linked to the raw ISR) to handle the interrupt before it enables it's own interrupt.

Example: Initializing the System Interrupt Library

```
SYS_INT_Initialize();
```

Preconditions

None.

Example

```
// Initialize the interrupt system. This needs to done in the initialization
// code.
SYS_INT_Initialize();
```

Function

```
void SYS_INT_Initialize ( void )
```

SYS_INT_DynamicDeregister Function

Deregisters the current ISR from the given interrupt source.

File

[sys_int.h](#)

C

```
void SYS_INT_DynamicDeregister( INT_SOURCE source );
```

Returns

None.

Description

This function deregisters the current Interrupt Service Routine (ISR), if any, from the specified interrupt source.

Remarks

It is safe to call this routine, even if no ISR has been registered for the given interrupt source.

Calling this routine is optional. If the system is designed such that the given ISR is expected to always be available once the system has been initialized, this routine does not need to be called.

Preconditions

[SYS_INT_Initialize](#) must have been called.

Example

```
SYS_INT_DynamicDeregister( SYS_INT_TIMER_1 );
```

Parameters

Parameters	Description
source	Identifier for the desired interrupt source

Function

```
void SYS_INT_DynamicDeregister ( INT_SOURCE source )
```

SYS_INT_DynamicRegister Function

Registers an Interrupt "Tasks" Routine for the specified interrupt source or trap).

File

[sys_int.h](#)

C

```
void SYS_INT_DynamicRegister( INT_SOURCE source, SYS_INT_TASKS_POINTER tasks, SYS_MODULE_OBJ object );
```

Returns

None.

Description

This function registers an Interrupt "Tasks" Routine for the specified interrupt source or trap).

Remarks

This routine only generates executable code when a driver is configured to register dynamically its "Tasks" routine with the system interrupt service. However, it should be called even if the ISR-to-source association is defined statically at compile time to maintain source code compatibility. A device driver normally registers its own ISR from its initialization routine.

In the example code, the macros MY_DRIVER_INTERRUPT_SOURCE would be defined in the appropriate configuration header, which would be included by the driver source file where the "DRV_MYDEV_Tasks" routine and the MyParam data would be defined.

It is safe to call this routine without first calling [SYS_INT_DynamicDeregister](#), even if a previous ISR has been registered. The effect will be that the new ISR supplants the old one.

The System Interrupt library must be initialized by calling the [SYS_INT_Initialize](#) routine. This is normally done in the "[SYS_Initialize](#)" routine before any interrupt support is used.

If the dynamic interrupt system service is not used, the "[SYS_Initialize](#)" routine must be implemented by the system designer or integrator as required by the system design and is not implemented by the System Interrupt library.

Once the library has been initialized, call the function [SYS_INT_Enable](#) to enable interrupts to the processor. However, before enabling the generation of interrupts to the processor, each individual module (driver, library, or application) must have a "Tasks" routine in place (either registered with [SYS_INT_DynamicRegister](#) or statically linked to the raw ISR) to handle the interrupt before it enables it's own interrupt.

Example: Initializing the System Interrupt Library

```
// Initialize the interrupt system.
SYS_INT_Initialize();
// Initialize all interrupt-aware software modules
SYS_INT_Enable();
```

Preconditions

[SYS_INT_Initialize](#) must have been called.

Example

```
SYS_INT_Initialize();
SYS_INT_DynamicRegister(MY_DRIVER_INTERRUPT_SOURCE, DRV_MYDEV_Tasks, MyObject);
```

Parameters

Parameters	Description
source	Identifier for the desired interrupt source
tasks	Pointer to the tasks routine
object	Handle to the module instance

Function

```
void SYS_INT_DynamicRegister( INT_SOURCE      source,
                             SYS_INT_TASKS_POINTER tasks,
                             SYS_MODULE_OBJ object )
```

SYS_INT_ShadowRegisterAssign Function

Assigns a shadow register set for an interrupt priority level.

File

[sys_int.h](#)

C

```
void SYS_INT_ShadowRegisterAssign(INT_PRIORITY_LEVEL priority, INT_SHADOW_REGISTER shadowRegister);
```

Returns

None.

Description

The function assigns a shadow register set for an interrupt priority level.

Remarks

This feature may not be available on all devices.

Preconditions

None.

Example

```
SYS_INT_ShadowRegisterAssign( INT_PRIORITY_LEVEL5, INT_SHADOW_REGISTER_5 );
```

Parameters

Parameters	Description
priority	Interrupt priority level for which the shadow register set has to be assigned.

shadowRegister	Shadow register set number.
----------------	-----------------------------

Function

```
void SYS_INT_ShadowRegisterAssign ( INT_PRIORITY_LEVEL priority,  
INT_SHADOW_REGISTER shadowRegister )
```

SYS_INT_ShadowRegisterGet Function

Gets the shadow register set assigned for an interrupt priority level.

File

[sys_int.h](#)

C

```
INT_SHADOW_REGISTER SYS_INT_ShadowRegisterGet( INT_PRIORITY_LEVEL priority );
```

Returns

None.

Description

The function gets the shadow register set assigned for an interrupt priority level.

Remarks

This feature may not be available on all devices. Refer to the specific device data sheet to determine availability.

Preconditions

None.

Example

```
INT_SHADOW_REGISTER shadowReg;  
  
shadowReg = SYS_INT_ShadowRegisterGet( INT_PRIORITY_LEVEL5 );
```

Parameters

Parameters	Description
priority	Interrupt priority level for which the shadow register set has to be assigned.

Function

```
INT_SHADOW_REGISTER SYS_INT_ShadowRegisterGet ( INT_MODULE_ID index,  
INT_PRIORITY_LEVEL priority )
```

SYS_INT_StatusGetAndDisable Function

Disables interrupts to the processor and return the previous status.

File

[sys_int.h](#)

C

```
SYS_INT_PROCESSOR_STATUS SYS_INT_StatusGetAndDisable ( );
```

Returns

- [SYS_INT_PROCESSOR_STATUS](#) - 32 bit value holding the processor status before disabling the interrupt

Description

This function disables interrupts to the processor at the top level. This function can be called to prevent any source from being able to generate an interrupt. It returns the processor status (which includes global interrupt status and Interrupt Priority status) before disabling the interrupts.

Remarks

Previous Master interrupt status can be found by checking 0th bit of the returned value.

This function should be paired with the use of [SYS_INT_StatusRestore\(\)](#). The value returned from this function should be passed into

[SYS_INT_StatusRestore\(\)](#) function.

Preconditions

[SYS_INT_Initialize](#) must have been called.

Example

```
SYS_INT_PROCESSOR_STATUS  processorStatus;

// Save the processor status (which includes global interrupt status)
// and Disable the global interrupts
processorStatus = SYS_INT_StatusGetAndDisable();

{
    // Do something critical
}

// set the processor status back to the one which was there before
// disabling the global interrupt
SYS_INT_StatusRestore(processorStatus);
```

Function

[SYS_INT_PROCESSOR_STATUS](#) [SYS_INT_StatusGetAndDisable](#) (void)

SYS_INT_StatusRestore Function

Restores the processor status.

File

[sys_int.h](#)

C

```
void SYS_INT_StatusRestore( SYS_INT_PROCESSOR_STATUS  processorStatus );
```

Returns

None.

Description

This function sets the processor status based on the 32 bit value passed as a parameter. 0th bit of the status is for Master Interrupt status.

Remarks

This function should be paired with the use of [SYS_INT_GetStateAndDisable\(\)](#). The value returned from [SYS_INT_GetStateAndDisable\(\)](#) should be passed into this function.

Preconditions

None.

Example

```
SYS_INT_PROCESSOR_STATUS  processorStatus;

processorStatus = SYS_INT_GetStateAndDisable();
{
    // do some critical work
}
SYS_INT_StatusRestore(processorStatus);
```

Parameters

Parameters	Description
processorStatus	value returned from previous call to SYS_INT_StatusGetAndDisable() .

Function

void [SYS_INT_StatusRestore](#) ([SYS_INT_PROCESSOR_STATUS](#) processorStatus)

b) Global Interrupt Management Functions

SYS_INT_Disable Function

Disables interrupts to the processor.

File

[sys_int.h](#)

C

```
bool SYS_INT_Disable();
```

Returns

true - Global Interrupts are enabled (before the call to disable) false - Global Interrupts are disabled (before the call to disable)

Description

This function disables interrupts to the processor at the top level. This function can be called to prevent any source from being able to generate an interrupt. It returns the global interrupt status before disabling the interrupts.

Remarks

This API will be deprecated. Use "[SYS_INT_StatusGetAndDisable](#)" instead.

Preconditions

[SYS_INT_Initialize](#) must have been called.

Example

```
// Interrupt enable status
bool flag;
// Disable the global interrupts
flag = SYS_INT_Disable();
// Do something critical
// Check if interrupts were disabled/enabled
if (flag)
{
    // enable the global interrupts if they were enabled before the
    // call to SYS_INT_Disable()
    SYS_INT_Enable();
}
```

Function

```
bool SYS_INT_Disable ( void )
```

SYS_INT_Enable Function

Enables global interrupts to the processor.

File

[sys_int.h](#)

C

```
void SYS_INT_Enable();
```

Returns

None.

Description

This function enables interrupts to the processor at the top level, allowing any currently enabled source to generate an interrupt. This function must be called before any source will be able to generate an interrupt.

Remarks

[SYS_INT_Enable](#) is called from the [SYS_INT_Initialize\(\)](#) function.

Preconditions

None.

Example

```
// Check if global interrupts are enabled
if( !SYS_INT_IsEnabled() )
{
    // Enable the global interrupts.
    SYS_INT_Enable();
}
```

Function

void SYS_INT_Enable (void)

SYS_INT_IsEnabled Function

Identifies if interrupts are currently enabled or disabled at the top level.

File

sys_int.h

C

```
bool SYS_INT_IsEnabled();
```

Returns

- true - If the interrupts are currently enabled
- false - If the interrupts are currently disabled

Description

This function identifies if interrupts are enabled or disabled at the top level.

Remarks

None.

Preconditions

SYS_INT_Initialize must have been called.

Example

```
// Check if global interrupts are enabled
if ( SYS_INT_IsEnabled() )
{
    // Interrupt enable status
    bool flag;
    // Disable the global interrupts.
    flag = SYS_INT_Disable();
}
```

Function

bool SYS_INT_IsEnabled (void)

SYS_INT_ExternalInterruptTriggerSet Function

Sets the external interrupt trigger type.

File

sys_int.h

C

```
void SYS_INT_ExternalInterruptTriggerSet( INT_EXTERNAL_SOURCES source, INT_EXTERNAL_EDGE_TRIGGER
edgeTrigger );
```

Returns

None.

Description

This function sets the External interrupt trigger type. User can set for multiple sources in a single call.

Remarks

Not supported for all interrupt sources. Check the specific data sheet to know the supported interrupt sources.

Preconditions

None.

Example

```
SYS_INT_ExternalInterruptTriggerSet (    INT_EXTERNAL_INT_SOURCE0 | INT_EXTERNAL_INT_SOURCE0 ,  
                                         INT_EDGE_TRIGGER_RISING );
```

Parameters

Parameters	Description
source	Identifier for the desired interrupt source

Function

```
void SYS_INT_ExternalInterruptTriggerSet (    INT_EXTERNAL_SOURCES source,  
                                             INT_EXTERNAL_EDGE_TRIGGER edgeTrigger )
```

c) Interrupt Source Management Functions

SYS_INT_SourceDisable Function

Disables the specified source from generating interrupts to the processor.

File

[sys_int.h](#)

C

```
bool SYS_INT_SourceDisable( INT_SOURCE source );
```

Returns

- true - The Interrupt source is enabled (before the call to SYS_INT_SourceDisable)
- false - The Interrupt source is disabled (before the call to SYS_INT_SourceDisable)

Description

This function disables the given source from generating interrupts the processor when events occur. It returns the interrupt source enable/disable status before disabling the interrupt source.

Remarks

None.

Preconditions

[SYS_INT_Initialize](#) must have been called.

Example

```
// interrupt source enable/disable status.  
bool flag  
// Initialize the interrupt system. This needs to be done in the initialization  
// code.  
SYS_INT_Initialize();  
// Disable the interrupt source  
flag = SYS_INT_SourceDisable(INT_SOURCE_PARALLEL_PORT);  
// before enabling the source check the enable/disable status  
if(flag)  
{  
    SYS_INT_SourceEnable( INT_SOURCE_PARALLEL_PORT );  
}
```


Parameters

Parameters	Description
source	Identifier for the desired interrupt source.

Function

bool SYS_INT_SourceDisable (INT_SOURCE source)

SYS_INT_SourceEnable Function

Enables the specified source to generate interrupts to the processor.

File

[sys_int.h](#)

C

```
void SYS_INT_SourceEnable( INT_SOURCE source );
```

Returns

None.

Description

This function enables the specified source to generate interrupts to the processor when events occur.

Remarks

An Interrupt Service Routine (ISR) for the given interrupt source must be ready to receive the call before the source is enabled.

Preconditions

[SYS_INT_Initialize](#) must have been called and an ISR must have been registered for the source.

Example

```
// Initialize the interrupt system. This needs to be done in the initialization
// code.
SYS_INT_Initialize();
// Enable the interrupt source
SYS_INT_SourceEnable( INT_SOURCE_PARALLEL_PORT );
```

Parameters

Parameters	Description
source	Identifier for the desired interrupt source

Function

void SYS_INT_SourceEnable (INT_SOURCE source)

SYS_INT_SourceIsEnabled Function

Identifies if the specified source is enabled or disabled.

File

[sys_int.h](#)

C

```
bool SYS_INT_SourceIsEnabled( INT_SOURCE source );
```

Returns

- true - If the given source is currently enabled.
- false - If the given source is currently disabled.

Description

This function identifies if the specified source is currently enabled or is currently disabled.

Remarks

None.

Preconditions

`SYS_INT_Initialize` must have been called.

Example

```
// Initialize the interrupt system. This needs to be done in the initialization
// code.
SYS_INT_Initialize();
// Check if the required interrupt source is enabled
if ( SYS_INT_SourceIsEnabled(INT_SOURCE_PARALLEL_PORT) )
{
    // App code
}
```

Parameters

Parameters	Description
source	Identifier for the desired interrupt source.

Function

`bool SYS_INT_SourceIsEnabled (INT_SOURCE source)`

SYS_INT_SourceStatusClear Function

Clears the interrupt request for the specified interrupt source.

File

`sys_int.h`

C

```
void SYS_INT_SourceStatusClear ( INT_SOURCE source );
```

Returns

None.

Description

This function clears the interrupt request for the specified interrupt source.

Remarks

None.

Preconditions

`SYS_INT_Initialize` must have been called.

Example

```
// Initialize the interrupt system. This needs to be done in the initialization
// code.
SYS_INT_Initialize();
// Check if the interrupt source flag is set
if ( SYS_INT_SourceStatusGet(INT_SOURCE_PARALLEL_PORT) )
{
    // Clear the interrupt flag
    SYS_INT_SourceStatusClear(INT_SOURCE_PARALLEL_PORT);
}
```

Parameters

Parameters	Description
source	Identifier for the desired interrupt source

Function

`void SYS_INT_SourceStatusClear (INT_SOURCE source)`

SYS_INT_SourceStatusGet Function

Determines the status of the specified interrupt source.

File

[sys_int.h](#)

C

```
bool SYS_INT_SourceStatusGet( INT_SOURCE source );
```

Returns

- true - If the given interrupt source is currently set
- false - If the given interrupt source is not currently set

Description

This function determines the current status of the interrupt source.

Remarks

Works even if the interrupt source or interrupts in general have not been enabled, so it can be used for polling implementations.

Preconditions

[SYS_INT_Initialize](#) must have been called.

Example

```
// Initialize the interrupt system.This needs to done in the initialization
// code.
SYS_INT_Initialize();
// Check if the required interrupt source is set
if ( SYS_INT_SourceStatusGet( INT_SOURCE_PARALLEL_PORT ) )
{
    // Handle interrupt
}
```

Parameters

Parameters	Description
source	Identifier for the desired interrupt source

Function

```
bool SYS_INT_SourceStatusGet ( INT_SOURCE source )
```

SYS_INT_SourceStatusSet Function

Sets the specified interrupt source.

File

[sys_int.h](#)

C

```
void SYS_INT_SourceStatusSet( INT_SOURCE source );
```

Returns

None.

Description

This function sets the specified interrupt source, causing the processor to be interrupted if interrupts are enabled, the source has been enabled, and the priority is higher than the current priority.

Remarks

Not supported for all interrupt sources. Check the specific data sheet for software clear only interrupt sources.

The driver, middleware, or application's interrupt-handling "Tasks" routine must do two things at a minimum, in the following order.

1. Remove the cause of the interrupt

2. Clear the interrupt source by calling the function [SYS_INT_SourceStatusClear](#)

Exactly what actions are necessary to remove the cause of an interrupt is completely dependent on the source of the interrupt. This is normally the main purpose of the driver itself and is beyond the scope of this section. Refer to the documentation for the peripheral being managed.

WARNING! The cause of the interrupt must be removed before clearing the interrupt source or the interrupt may re-occur immediately after the source is cleared potentially causing an infinite loop. An infinite loop may also occur if the source is not cleared before the interrupt-handler returns.

Example: Handling Interrupts

```
void DRV_MYDEV_Tasks( SYS_MODULE_OBJ object )
{
    // Remove the cause of the interrupt
    //...

    // Clear Interrupt source
    SYS_INT_SourceStatusClear(myIntSourceID);
}
```

Note: the value of "myIntSourceID" is usually either a static or dynamic configuration option. Refer to the documentation for the specific device driver to identify how to define the interrupt source ID.

Testing Interrupt

Some times it is necessary to cause an interrupt in software, possibly for testing purposes. To support this, the function [SYS_INT_SourceStatusSet](#) is provided.

Example: Causing an Interrupt in Software

```
SYS_INT_SourceStatusSet(MY_DRIVER_INTERRUPT_SOURCE);
```

Note: This feature is not available for all interrupt sources on all Microchip microcontrollers. Refer to the data sheet for the microcontroller being used to determine if it is possible for software to set a specific interrupt source.

Preconditions

[SYS_INT_Initialize](#) must have been called and an ISR must have been registered for the source (if interrupts and the source are enabled).

Example

```
// Initialize the interrupt system. This needs to done in the initialization
// code.
SYS_INT_Initialize();
// Check if interrupt source flag is set
if ( !SYS_INT_SourceStatusGet( INT_SOURCE_PARALLEL_PORT ) )
{
    // Set the interrupt source flag
    SYS_INT_SourceStatusSet( INT_SOURCE_PARALLEL_PORT );
}
```

Parameters

Parameters	Description
source	Identifier for the desired interrupt source

Function

```
void SYS_INT_SourceStatusSet ( INT_SOURCE source )
```

SYS_INT_VectorPrioritySet Function

Sets the given interrupt vector to the specified priority.

File

[sys_int.h](#)

C

```
void SYS_INT_VectorPrioritySet( INT_VECTOR vector, INT_PRIORITY_LEVEL priority );
```

Returns

None.

Description

This routine sets the given interrupt vector to the specified priority.

Remarks

This feature is not supported on all devices. Refer to the specific device data sheet or family reference manual to determine whether this feature is

supported.

In the example code, the macros MY_DRIVER_INTERRUPT_VECTOR, MY_DRIVER_ISR_PRIORITY would be defined appropriately during configuration.

Preconditions

`SYS_INT_Initialize` must have been called.

Example

```
#define MY_DRIVER_INTERRUPT_VECTOR  INT_VECTOR_T1
#define MY_DRIVER_ISR_PRIORITY      INT_PRIORITY_LEVEL2
// Initialize the interrupt system.This needs to done in the initialization
// code.
SYS_INT_Initialize();
// Assign priority to the interrupt vector
SYS_INT_VectorPrioritySet(MY_DRIVER_INTERRUPT_VECTOR, MY_DRIVER_ISR_PRIORITY);
```

Parameters

Parameters	Description
vector	Identifier for the desired interrupt vector
priority	Priority (if supported)

Function

void SYS_INT_VectorPrioritySet (INT_VECTOR vector, INT_PRIORITY_LEVEL priority)

SYS_INT_VectorSubprioritySet Function

Sets the specified interrupt vector to the given sub priority.

File

`sys_int.h`

C

```
void SYS_INT_VectorSubprioritySet(INT_VECTOR vector, INT_SUBPRIORITY_LEVEL subpriority);
```

Returns

None.

Description

This function sets the specified interrupt vector to the specified sub-priority.

Remarks

This feature is not supported on all devices. Refer to the specific device data sheet or family reference manual to determine whether this feature is supported.

In the example code, the macros MY_DRIVER_INTERRUPT_VECTOR, MY_DRIVER_ISR_SUB_PRIORITY would be defined appropriately during configuration.

Preconditions

`SYS_INT_Initialize` must have been called.

Example

```
#define MY_DRIVER_INTERRUPT_VECTOR  INT_VECTOR_T1
#define MY_DRIVER_ISR_PRIORITY      INT_PRIORITY_LEVEL2
#define MY_DRIVER_ISR_SUB_PRIORITY INT_SUBPRIORITY_LEVEL1
// Initialize the interrupt system.This needs to done in the initialization
// code.
SYS_INT_Initialize();
// Assign priority to the interrupt vector
SYS_INT_VectorPrioritySet(MY_DRIVER_INTERRUPT_VECTOR, MY_DRIVER_ISR_PRIORITY);
// Assign sub-priority to the interrupt vector
SYS_INT_VectorSubprioritySet(MY_DRIVER_INTERRUPT_VECTOR, MY_DRIVER_ISR_SUB_PRIORITY);
```

Parameters

Parameters	Description
vector	Identifier for the desired interrupt vector
subpriority	Subpriority (if supported)

Function

```
void SYS_INT_VectorSubprioritySet( INT_VECTOR vector,  
INT_SUBPRIORITY_LEVEL subpriority )
```

d) Data Types and Constants

SYS_INT_TASKS_POINTER Type

Pointer to an interrupt-handling "Tasks" routine.

File

[sys_int.h](#)

C

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

Description

Interrupt Tasks Routine Pointer

This data type defines a pointer to an interrupt-handling "Tasks" routine. The form of a tasks routine is as follows:

```
void My_Tasks ( SYS_MODULE_OBJ object );
```

Where "MyTasks" is the name of the tasks routine and object is a Handle to the module instance.

Remarks

"Tasks" is normally defined by a device driver, middleware, or system layer.

The term Interrupt Service Routine (ISR) is used for the "raw" ISR code that is either located directly at the interrupt vector address or whose address is loaded from the interrupt vector. The term "Tasks" routine is used to identify the driver-specific routine that is called by the actual ISR to perform the tasks necessary to handle and clear the interrupt.

INT_EXTERNAL_EDGE_TRIGGER Enumeration

Lists the available external interrupt trigger options.

File

[sys_int.h](#)

C

```
typedef enum {  
    INT_EDGE_TRIGGER_FALLING,  
    INT_EDGE_TRIGGER_RISING  
} INT_EXTERNAL_EDGE_TRIGGER;
```

Members

Members	Description
INT_EDGE_TRIGGER_FALLING	External interrupt trigger on falling edge
INT_EDGE_TRIGGER_RISING	External interrupt trigger on rising edge

Description

Interrupt external edge selection.

This enumeration lists all of the available external interrupt trigger options.

Remarks

None.

SYS_INT_PROCESSOR_STATUS Type

CPU Processor status

File

[sys_int.h](#)

C

```
typedef uint32_t SYS_INT_PROCESSOR_STATUS;
```

Description

Interrupt Processor Status

This data type holds the status of CPU register 2

Remarks

None.

Files

Files

Name	Description
sys_int.h	Interrupt System Service.

Description

This section lists the source and header files used by the library.




















sys_int.h

Interrupt System Service.

Enumerations

Name	Description
INT_EXTERNAL_EDGE_TRIGGER	Lists the available external interrupt trigger options.

Functions

Name	Description
 SYS_INT_Disable	Disables interrupts to the processor.
 SYS_INT_DynamicDeregister	Deregisters the current ISR from the given interrupt source.
 SYS_INT_DynamicRegister	Registers an Interrupt "Tasks" Routine for the specified interrupt source or trap).
 SYS_INT_Enable	Enables global interrupts to the processor.
 SYS_INT_ExternalInterruptTriggerSet	Sets the external interrupt trigger type.
 SYS_INT_Initialize	Configures and initializes the interrupt subsystem.
 SYS_INT_IsEnabled	Identifies if interrupts are currently enabled or disabled at the top level.
 SYS_INT_ShadowRegisterAssign	Assigns a shadow register set for an interrupt priority level.
 SYS_INT_ShadowRegisterGet	Gets the shadow register set assigned for an interrupt priority level.
 SYS_INT_SourceDisable	Disables the specified source from generating interrupts to the processor.
 SYS_INT_SourceEnable	Enables the specified source to generate interrupts to the processor.
 SYS_INT_SourceIsEnabled	Identifies if the specified source is enabled or disabled.
 SYS_INT_SourceStatusClear	Clears the interrupt request for the specified interrupt source.
 SYS_INT_SourceStatusGet	Determines the status of the specified interrupt source.
 SYS_INT_SourceStatusSet	Sets the specified interrupt source.
 SYS_INT_StatusGetAndDisable	Disables interrupts to the processor and return the previous status.
 SYS_INT_StatusRestore	Restores the processor status.
 SYS_INT_VectorPrioritySet	Sets the given interrupt vector to the specified priority.
 SYS_INT_VectorSubprioritySet	Sets the specified interrupt vector to the given sub priority.

Types

	Name	Description
	SYS_INT_PROCESSOR_STATUS	CPU Processor status
	SYS_INT_TASKS_POINTER	Pointer to an interrupt-handling "Tasks" routine.

Description

Interrupt System Service Library Interface Definition

This file contains the interface definition for the Interrupt System Service. It provides a way to interact with the interrupt subsystem to manage the occurrence of interrupts for sources supported by the system.

File Name

sys_int.h

Company

Microchip Technology Inc.

Memory System Service Library

This section describes the Memory System Service Library.

Introduction

Introduction to the Memory System Service.

Description

The Memory System Service provides support for the initialization of external memory and external memory controllers. It is configured to run in static mode by the MPLAB Harmony Configurator (MHC). When an external memory interface is enabled by MHC, the initialization code for the controller is generated and added to the system initialization sequence in the application. The specific timing parameters for the external memory to be used are selected by the user via MHC.

Using the Library

This topic describes the basic architecture of the Memory System Service Library and provides information and examples on its use.

Description

Interface Header File: [sys_memory.h](#)

The interface to the Memory System Service library is defined in the [sys_memory.h](#) header file, which is included by the [system.h](#) system service header file. Any C language source (.c) file that uses the Memory System Service library should include [system.h](#).

Please refer to the What is MPLAB Harmony? section for how the library interacts with the framework.

Abstraction Model

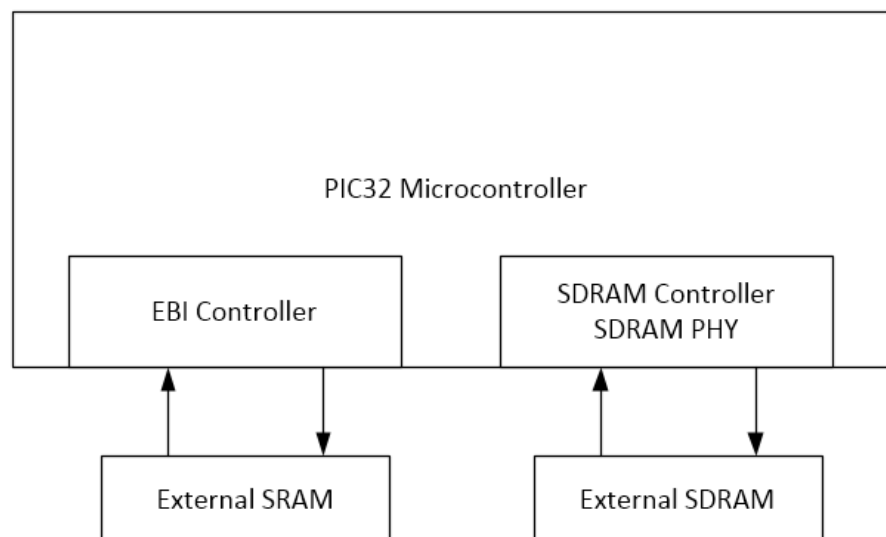
Provides information on the abstraction model for the Memory System Service.

Description

The Memory System Service manages the initialization of interfaces to external memory devices on some PIC32 microcontrollers. The initialization is performed during the system initialization sequence, so the memory is available to the application when the application tasks are initiated. The memory devices accessed via the interfaces initialized by this service are mapped into the virtual address space, and once initialized, are accessed directly just like internal SRAM. The type and number of external memory devices, as well as the virtual address mapping may vary between devices. Please refer to the specific device data sheet for details of a particular device.

The following figure provides the abstraction model for the Memory System Service.

Memory System Service Abstraction Model



Library Overview

Please refer to the [System Service Introduction](#) for a detailed description of MPLAB Harmony system services.

The library interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the Memory System Service.

How the Library Works

The Memory System Service Library can be used by a device driver, middleware layer, or application to provide access to, and control over, interrupts to the processor.

Description

Initialization

Once the service is initialized, there is no further run-time configuration or operation required. The external memory is simply accessed via virtual address, just like internal SRAM. Therefore, there is a single API that manages the initialization of the selected external memory devices: [SYS_MEMORY_Initialize](#).

The [SYS_MEMORY_Initialize](#) API will invoke the initialization function(s) of external memory devices enabled via MHC. There are presently two external memory controllers supported on some PIC32 microcontrollers, the External Bus Interface (EBI) and the SDRAM controller. The EBI can interface with external asynchronous SRAM or NOR Flash, while the SDRAM controller can interface with external DDR2 SDRAM.

```
SYS_MEMORY_EBI_Initialize();  
SYS_MEMORY_DDR_Initialize();
```

Configuring the Library

The configuration of the Memory System Service is based on the file `system_config.h`.

This header file contains the configuration selection for the Memory System service. Based on the selections made, the Memory System Service may support the selected features. These configuration settings will apply to all instances of the Memory System Service.

This header can be placed anywhere; however, the path of this header needs to be present in the include search path for a successful build. Refer to the Applications Help section for more details.

Building the Library

This section lists the files that are available in the Memory System Service Library.

Description

The following three tables list and describe the header (.h) and source (.c) files that implement this library. The parent folder for these files is `<install-dir>/framework/system/memory`.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using `#include`) by any code that uses this library.

Source File Name	Description
sys_memory.h	Memory System Service Library API header file.

Required File(s)



All of the required files listed in the following table are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

Source File Name	Description
N/A	No source files are available for this library.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
N/A	No optional files are available for this library.

Module Dependencies

The Memory System Service is not dependent upon other modules.

Library Interface

a) Functions

	Name	Description
⇒	SYS_MEMORY_DDR_Initialize	Initializes and Enables the DDR External Memory Controller.
⇒	SYS_MEMORY_EBI_Initialize	Initializes and Enables the EBI External Memory Controller.
⇒	SYS_MEMORY_Initialize	Initializes and Enables the External Memory Controller(s).

Description

This section describes the APIs of the Memory System Service Library.

a) Functions

SYS_MEMORY_DDR_Initialize Function

Initializes and Enables the DDR External Memory Controller.

File

[sys_memory.h](#)

C

```
void SYS_MEMORY_DDR_Initialize();
```

Returns

None.

Description

This function Enables the external DDR memory controller module.

Remarks

This routine must be called before any attempt to access external DDR memory.

Not all features are available on all devices. Refer to the specific device data sheet to determine availability.

Preconditions

None.

Example

```
SYS_MEMORY_DDR_Initialize(NULL);
```

Parameters

Parameters	Description
data	Pointer to the data structure containing any data necessary to initialize the hardware. This pointer may be null if no data is required and default initialization is to be used.

Function

```
void SYS_MEMORY_DDR_Initialize ( void * data)
```

SYS_MEMORY_EBI_Initialize Function

Initializes and Enables the EBI External Memory Controller.

File

[sys_memory.h](#)

C

```
void SYS_MEMORY_EBI_Initialize();
```

Returns

None.

Description

This function Enables the external EBI memory controller module.

Remarks

This routine must be called before any attempt to access external EBI memory.

Not all features are available on all devices. Refer to the specific device data sheet to determine availability.

Preconditions

None.

Example

```
SYS_MEMORY_EBI_Initialize(NULL);
```

Parameters

Parameters	Description
data	Pointer to the data structure containing any data necessary to initialize the hardware. This pointer may be null if no data is required and default initialization is to be used.

Function

```
void SYS_MEMORY_EBI_Initialize ( void * data)
```

SYS_MEMORY_Initialize Function

Initializes and Enables the External Memory Controller(s).

File

[sys_memory.h](#)

C

```
void SYS_MEMORY_Initialize();
```

Returns

None.

Description

This function Enables the external memory controller module(s).

Remarks

This routine must be called before any attempt to access external memory.

Not all features are available on all devices. Refer to the specific device data sheet to determine availability.

Preconditions

None.

Example

```
SYS_MEMORY_Initialize(NULL);
```

Parameters

Parameters	Description
data	Pointer to the data structure containing any data necessary to initialize the hardware. This pointer may be null if no data is required and default initialization is to be used.

Function

```
void SYS_MEMORY_Initialize ( void * data)
```

Files

Files

Name	Description
sys_memory.h	Memory System Service Implementation.




Description

This section lists the source and header files used by the library.

sys_memory.h

Memory System Service Implementation.

Functions

	Name	Description
	SYS_MEMORY_DDR_Initialize	Initializes and Enables the DDR External Memory Controller.
	SYS_MEMORY_EBI_Initialize	Initializes and Enables the EBI External Memory Controller.
	SYS_MEMORY_Initialize	Initializes and Enables the External Memory Controller(s).

Description

Memory System Service Implementation

The Memory System Service provides a simple interface to manage the memory controllers. This file implements the core interface routines for the Memory System Service.

File Name

sys_memory.h

Company

Microchip Technology Inc.

Messaging System Service Library

This section describes the Messaging System Service Library.

Introduction

This library provides intra-process and inter-process communication by the sending and receiving of simple messages. The format of these messages is under developer control, providing flexibility to tune message format for each application. The number and size of message queues is under developer control, with a message priority scheme implemented by multiple queues. Each priority queue has an configurable size. The number of message types and the number of receiving mailboxes is also configurable.

Description

This library provides intra-process and inter-process communication by the sending and receiving of simple messages. The format of these messages is under developer control, providing flexibility to tune message format for each application. The number and size of message queues is under developer control, with a message priority scheme implemented by multiple queues. Each priority queue has an configurable size. The number of message types and the number of receiving mailboxes is also configurable.

Consider an application that combines graphics on a display screen, with a touch overlay on top of the screen, and several buttons. System messages can be used by touch software to alert the graphics part of the application to update the screen as well as alert other parts of the application to perhaps change the audio volume whenever a slider value has changed. This is easily supported by the Messaging System Services Library.

Using the Library

This topic describes the basic architecture of the Messaging System Service Library and provides information and examples on its use.

Description

Interface Header File: [sys_msg.h](#)

The interface to the Messaging System Service library is defined in the [sys_msg.h](#) header file, which is included by the [system.h](#) system service header file. Any C language source (.c) file that uses the Messaging System Service library should include [system.h](#).

Please refer to the What is MPLAB Harmony? section for how the library interacts with the framework.

Abstraction Model

This library provides an abstraction of the messaging subsystem that is used by device drivers, middleware libraries and applications to receive and control interrupts in real time.

This library uses `calloc` to allocate memory for:

- Message queues for each priority as part of [SYS_MSG_Initialize](#)
- Mailbox definition objects
- Message type objects

If calls to `calloc` fails to allocate the needed memory then object handles are returned with a value of [SYS_OBJ_HANDLE_INVALID](#).

Library Overview

Please refer to the [System Service Introduction](#) for a detailed description of MPLAB Harmony system services.

The library interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the Messaging System Service module


Library Interface Section	Description
Initialization, Tasks, and Versioning Functions	Provides configuration and control functions.
Mailbox Functions	Provides mailbox APIs.
Message Type Functions	Provides messaging type APIs.
Message Send/Receive Functions	Provides messaging send and receive APIs.
Utility Functions	Provides messaging utility APIs.

How the Library Works

The Messaging System Service Library can be used by a device driver, middleware layer, or application to provide access to, and control over,

interrupts to the processor.

Description

 **Note:** Not all modes are available on all devices. Please refer to the specific device data sheet to determine the modes supported for your device.

The following example can be run on any PIC32 starter kit:

```
#include <sys/appio.h>
#include <GenericTypeDefs.h>

#include "system/common/sys_module.h"
#include "system/msg/sys_msg.h"
#include "system/msg/src/sys_msg_local.h"

// Strawman callback functions for three mailboxes
void myCallBack0( SYS_MSG_OBJECT *pMessage )
{
    DBPRINTF("CallBack0:: Message Type: %d, nSource: %d\r\n"
            "           param 0: %d, param1: %d, param2: %d \r\n",
            pMessage->nMessageTypeID,
            pMessage->nSource, pMessage->param0,
            pMessage->param1, pMessage->param2
            );
}

void myCallBack1( SYS_MSG_OBJECT *pMessage )
{
    DBPRINTF("CallBack1:: Message Type: %d, nSource: %d\r\n"
            "           param 0: %d, param1: %d, param2: %d \r\n",
            pMessage->nMessageTypeID,
            pMessage->nSource, pMessage->param0,
            pMessage->param1, pMessage->param2
            );
}

void myCallBack2( SYS_MSG_OBJECT *pMessage )
{
    DBPRINTF("CallBack2:: Message Type: %d, nSource: %d\r\n"
            "           param 0: %d, param1: %d, param2: %d \r\n",
            pMessage->nMessageTypeID,
            pMessage->nSource, pMessage->param0,
            pMessage->param1, pMessage->param2
            );
}

int main(void)
{
    SYS_MSG_MESSAGING_OBJECT oSysMsg;
    SYS_OBJ_HANDLE hSysMsg, hMsgType[5], hMailbox[3];
    SYS_MSG_INSTANCE iSysMsg = SYS_MSG_0;
    SYS_MSG_OBJECT myMessage[5];

    { // SYS_MSG_Initialize
        uint8_t nQSizes[] = SYS_MSG_BUFFER_SIZES;

        // Initialize the messaging system. This needs to be done in the initialization code.
        hSysMsg = SYS_MSG_Initialize(iSysMsg, SYS_MSG_MAX_PRIORITY+1, nQSizes);

        SYS_ASSERT( SYS_OBJ_HANDLE_INVALID != hSysMsg, "Bad hSysMsg!" );
    }

    // Create the message types to be used
    //                                     ID:      :Priority
    hMsgType[0] = SYS_MSG_TypeCreate(iSysMsg, 1<<0, 0);
    hMsgType[1] = SYS_MSG_TypeCreate(iSysMsg, 1<<1, 1);
    hMsgType[2] = SYS_MSG_TypeCreate(iSysMsg, 1<<2, 2);
```

```

hMsgType[3] = SYS_MSG_TypeCreate(iSysMsg,1<<3,3);
hMsgType[4] = SYS_MSG_TypeCreate(iSysMsg,1<<4,4);

// Create the mailboxes to be used
hMailbox[0] = SYS_MSB_MailboxOpen( iSysMsg, &myCallBack0 );
hMailbox[1] = SYS_MSB_MailboxOpen( iSysMsg, &myCallBack1 );
hMailbox[2] = SYS_MSB_MailboxOpen( iSysMsg, &myCallBack2 );

// Identify which messages are of interest for each mailbox.
SYS_MSG_MailboxMsgAdd(hMailbox[0],hMsgType[0]);
SYS_MSG_MailboxMsgAdd(hMailbox[0],SYS_MSG_ID2hMsgType(iSysMsg,1<<1));
SYS_MSG_MailboxMsgAdd(hMailbox[0],SYS_MSG_ID2hMsgType(iSysMsg,1<<2));
SYS_MSG_MailboxMsgAdd(hMailbox[0],SYS_MSG_ID2hMsgType(iSysMsg,1<<3));
SYS_MSG_MailboxMsgAdd(hMailbox[0],SYS_MSG_ID2hMsgType(iSysMsg,1<<4));
SYS_MSG_MailboxMsgAdd(hMailbox[1],hMsgType[0]);
SYS_MSG_MailboxMsgAdd(hMailbox[2],hMsgType[0]);

{ //Send and receive messages
    SYS_MSGQ_ELEMENT *pQElement;
    SYS_MSG_OBJECT *pMessage;
    SYS_MSG_RESULTS myResult;
    SYS_MSG_QUEUE_STATUS qStatus[5];
    uint16_t iPriority;

    for (iPriority=0;iPriority<=SYS_MSG_MAX_PRIORITY;iPriority++)
    {
        myMessage[iPriority].nSource = SYS_MSG_MAX_PRIORITY-iPriority;
        myMessage[iPriority].nMessageTypeID = 1<<iPriority;
        myMessage[iPriority].param0 = iPriority;
        myMessage[iPriority].param1 = 0;
        myMessage[iPriority].param2 = 0;
    }

    myMessage[4].param1 = 0;
    myResult = SYS_MSG_MessageSend( iSysMsg, &myMessage[4] );
    SYS_ASSERT(myResult == SYS_MSG_SENT,"Message not sent!");

    myMessage[4].param1++;
    myResult = SYS_MSG_MessageSend( iSysMsg, &myMessage[4] );
    SYS_ASSERT(myResult == SYS_MSG_SENT,"Message not sent!");

    myMessage[4].param1++;
    myResult = SYS_MSG_MessageSend( iSysMsg, &myMessage[4] );
    SYS_ASSERT(myResult == SYS_MSG_SENT,"Message not sent!");

    myMessage[4].param1++;
    myResult = SYS_MSG_MessageSend( iSysMsg, &myMessage[4] );
    SYS_ASSERT(myResult == SYS_MSG_SENT,"Message not sent!");

    myResult = SYS_MSG_MessageSend( iSysMsg, &myMessage[4] ); // Should fail, queue full!
    SYS_ASSERT(myResult == SYS_MSG_NOT_SENT,"Result should NOT Be Sent!");

    myMessage[3].param1 = myMessage[4].param1;
    myMessage[3].param1++;
    myResult = SYS_MSG_MessageSend( iSysMsg, &myMessage[3] );
    SYS_ASSERT(myResult == SYS_MSG_SENT,"Message not sent!");

    myMessage[3].param1++;
    myResult = SYS_MSG_MessageSend( iSysMsg, &myMessage[3] );
    SYS_ASSERT(myResult == SYS_MSG_SENT,"Message not sent!");

    myMessage[2].param1 = myMessage[3].param1;
    myMessage[2].param1++;
    myResult = SYS_MSG_MessageSend( iSysMsg, &myMessage[2] );
    SYS_ASSERT(myResult == SYS_MSG_SENT,"Message not sent!");

    myMessage[2].param1++;
    myResult = SYS_MSG_MessageSend( iSysMsg, &myMessage[2] );
    SYS_ASSERT(myResult == SYS_MSG_SENT,"Message not sent!");

```



```

myMessage[1].param1 = myMessage[2].param1;
myMessage[1].param1++;
myResult = SYS_MSG_MessageSend( iSysMsg, &myMessage[1] );
SYS_ASSERT(myResult == SYS_MSG_SENT,"Message not sent!");

myMessage[1].param1++;
myResult = SYS_MSG_MessageSend( iSysMsg, &myMessage[1] );
SYS_ASSERT(myResult == SYS_MSG_SENT,"Message not sent!");

myMessage[0].param1 = myMessage[1].param1;
myMessage[0].param1++;
myResult = SYS_MSG_MessageSend( iSysMsg, &myMessage[0] );
SYS_ASSERT(myResult == SYS_MSG_SENT,"Message not sent!");

myMessage[0].param1++;
myResult = SYS_MSG_MessageSend( iSysMsg, &myMessage[0] );
SYS_ASSERT(myResult == SYS_MSG_SENT,"Message not sent!");

myMessage[0].param1++;
myResult = SYS_MSG_MessageSend( iSysMsg, &myMessage[0] );
SYS_ASSERT(myResult == SYS_MSG_SENT,"Message not sent!");

// Preemptively read all Mailbox Zero messages
while ( NULL != (pMessage = SYS_MSG_MailboxMessagesGet(hMailbox[0])) )
{
    // "Process" message.
    DBPRINTF("Mailbox Zero: Message Type: %d, nSource: %d\r\n"
            "           param 0: %d, param1: %d, param2: %d \r\n",
            pMessage->nMessageTypeID,
            pMessage->nSource, pMessage->param0,
            pMessage->param1, pMessage->param2
            );
}

// Deliver messages for all mailboxes.
DBPRINTF("\r\nGot Messages: %d\r\n",SYS_MSG_GotMessages(iSysMsg));
for (iPriority=0;iPriority<=SYS_MSG_MAX_PRIORITY;iPriority++)
{
    qStatus[iPriority] =
        SYS_MSG_QueueStatus((SYS_OBJ_HANDLE)iSysMsg,iPriority);
}
DBPRINTF("Queue Status (4:-1:0): %d, %d, %d, %d, %d\r\n\r\n",
        qStatus[4],qStatus[3],qStatus[2],qStatus[1],qStatus[0]);
while ( (pQElement = SYS_MSG_MessageReceive(iSysMsg)) != NULL )
{
    SYS_MSG_MessageDeliver(iSysMsg,pQElement);
    for (iPriority=0;iPriority<=SYS_MSG_MAX_PRIORITY;iPriority++)
    {
        qStatus[iPriority] =
            SYS_MSG_QueueStatus((SYS_OBJ_HANDLE)iSysMsg,iPriority);
    }
    DBPRINTF("Queue Status (4:-1:0): %d, %d, %d, %d, %d\r\n\r\n",
            qStatus[4],qStatus[3],qStatus[2],qStatus[1],qStatus[0]);
}
DBPRINTF("Got Messages: %d\r\n",SYS_MSG_GotMessages(iSysMsg));
}

return 0;
}

```

Configuring the Library

The file [sys_msg_config.h](#) file provides configuration parameters that the implementer can use to adapt the Messaging System Service Library to a particular application.

Macros

	Name	Description
	SYS_MSG_BUFFER_SIZES	define SYS_MSG_BUFFER_SIZES { 4, 4, 4, 4, 1
	SYS_MSG_MAX_MAILBOXES	Specifies the maximum number of mailboxes possible.
	SYS_MSG_MAX_MSGS_DELIVERED	Specifies the maximum number of messages delivered per each call to SYS_MSG_Tasks .
	SYS_MSG_MAX_PRIORITY	Specifies the maximum message priority.
	SYS_MSG_MAX_TYPES	Specifies the maximum number of message types possible.
	_SYS_MSG_CONFIG_TEMPLATE__H	This is macro _SYS_MSG_CONFIG_TEMPLATE__H .

Description

By default, the system message format provides 64 bits of information in a message:

```
typedef union
{
    struct
    {
        uint16_t      param0; // Message parameter zero
        uint16_t      param1; // Message parameter one
        uint16_t      param2; // Message parameter two
        uint16_t nMessageTypeID; // Message type identifier
    };
    struct
    {
        uint16_t      nSource; // Message source identifier
        uintptr_t      * pData; // Pointer to additional message data
    };
} SYS_MSG_OBJECT;
```

The only required field in the message format definition is `nMessageTypeID`. However, it does not need to be 16 bits long.

The maximum number of mailboxes is defined by:

```
#define SYS_MSG_MAX_MAILBOXES (32)
```

The minimum number of mailboxes is one.

The maximum number of message types is defined by:

```
#define SYS_MSG_MAX_TYPES (32)
```

The minimum number of message types is one.

Message priorities run from zero to [SYS_MSG_MAX_PRIORITY](#), which is defined by:

```
#define SYS_MSG_MAX_PRIORITY (4)
```

For each message priority from 0,1, to [SYS_MSG_MAX_PRIORITY](#) a queue is created. The size of each queue is defined by:

```
// Message Priority: 0 1 2 3 4
#define SYS_MSG_BUFFER_SIZES { 64, 32, 16, 8, 4 }
```

In this example there are five priorities, 0, 1, ..4, and the sizes of each message queue is provided in the definition of [SYS_MSG_BUFFER_SIZES](#).

Then, this information is used when initializing the Messaging System Service in the application start-up code:

```
SYS_OBJ_HANDLE hSysMsg;
//SYS_MSG_Initialize
uint16_t nQSizes[] = SYS_MSG_BUFFER_SIZES;
hSysMsg = SYS_MSG_Initialize(SYS_MSG_MAX_PRIORITY+1,nQSizes);
SYS_ASSERT( SYS_OBJ_HANDLE_INVALID != hSysMsg,"Bad hSysMsg!" );
}
```

SYS_MSG_BUFFER_SIZES Macro

File

[sys_msg_config.h](#)

C

```
#define SYS_MSG_BUFFER_SIZES { 16 }
```

Description

```
define SYS_MSG_BUFFER_SIZES { 4, 4, 4, 4, 1
```

SYS_MSG_MAX_MAILBOXES Macro

Specifies the maximum number of mailboxes possible.

File

[sys_msg_config.h](#)

C

```
#define SYS_MSG_MAX_MAILBOXES (2)
```

Description

System Messaging Max Number of Mailboxes

Specifies the maximum number of mailboxes possible.

Remarks

Minimum number is 1 mailbox.

SYS_MSG_MAX_MSGS_DELIVERED Macro

Specifies the maximum number of messages delivered per each call to [SYS_MSG_Tasks](#).

File

[sys_msg_config.h](#)

C

```
#define SYS_MSG_MAX_MSGS_DELIVERED (1)
```

Description

System Messaging Maximum Number of Messages Delivered per call to [SYS_MSG_Tasks](#)

Specifies the maximum number of messages delivered per each call to [SYS_MSG_Tasks](#). If zero then all message queues are emptied before the tasks routine finishes execution.

Remarks

0 implies all queues are empty after [SYS_MSG_Tasks](#) is done.

SYS_MSG_MAX_PRIORITY Macro

Specifies the maximum message priority.

File

[sys_msg_config.h](#)

C

```
#define SYS_MSG_MAX_PRIORITY (0)
```

Description

System Messaging Maximum Priority

Specifies the maximum message priority.

Remarks

Message priorities run from 0 to SYS_MSG_MAX_PRIORITIES. The number of message queues is SYS_MSG_MAX_PRIORITIES+1.

SYS_MSG_MAX_TYPES Macro

Specifies the maximum number of message types possible.

File

[sys_msg_config.h](#)

C

```
#define SYS_MSG_MAX_TYPES (2)
```

Description

System Messaging Max Number of Message Types
Specifies the maximum number of message types possible.

Remarks

Minimum number is 1.

`_SYS_MSG_CONFIG_TEMPLATE__H` Macro**File**

`sys_msg_config.h`

C

```
#define _SYS_MSG_CONFIG_TEMPLATE__H
```

Description

This is macro `_SYS_MSG_CONFIG_TEMPLATE__H`.

Building the Library

This section lists the files that are available in the Messaging System Service Library.

Description

The following three tables list and describe the header (.h) and source (.c) files that implement this library. The parent folder for these files is `<install-dir>/framework/system/msg`.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using `#include`) by any code that uses this library.

Source File Name	Description
<code>sys_msg.h</code>	Messaging System Service Library API header file.
<code>/src/sys_msg_local.h</code>	System messaging local declarations and definitions.
<code>/config/sys_msg_config.h</code>	System messaging configuration.

Required File(s)

All of the required files listed in the following table are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

Source File Name	Description
<code>/src/sys_msg.c</code>	Messaging System Service Library implementation.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
N/A	No optional files are available for this library.

Module Dependencies

The Messaging System Service is not dependent upon other modules.

Library Interface

a) Initialization, Tasks, and Versioning Functions

	Name	Description
⇒	SYS_MSG_Deinitialize	Deinitializes System Messaging Instance.
⇒	SYS_MSG_Initialize	Configures and initializes the messaging subsystem.
⇒	SYS_MSG_Tasks	System Messaging Service Tasks routine.

b) Mailbox Functions

	Name	Description
⇒	SYS_MSG_MailboxMessagesGet	Gets queued messages for a mailbox.
⇒	SYS_MSG_MailboxMsgAdd	Adds a message type to the list of messages received by a mailbox.
⇒	SYS_MSG_MailboxMsgRemove	Removes a message type from the list of messages received by a mailbox.
⇒	SYS_MSG_MailboxClose	Closes (destroys) a mailbox previously opened with SYS_MSG_MailboxOpen .
⇒	SYS_MSG_MailboxOpen	Opens a system messaging mailbox.
⇒	SYS_MSG_MailboxReinit	Reinitializes a previously opened mailbox.

c) Message Type Functions

	Name	Description
⇒	SYS_MSG_TypeCreate	Creates a new message type.
⇒	SYS_MSG_TypeRemove	Removes an existing message type.

d) Message Send/Receive Functions

	Name	Description
⇒	SYS_MSG_GotMessages	Returns true if system messaging has undelivered messages, false otherwise.
⇒	SYS_MSG_MessageDeliver	Delivers messages to mailboxes.
⇒	SYS_MSG_MessageReceive	Receives the next message in the message queues.
⇒	SYS_MSG_MessageSend	Sends a message, as defined by a message structure.

e) Utility Functions

	Name	Description
⇒	SYS_MSG_ID2hMsgType	Translates message type identifier into handle of corresponding message type object.
⇒	SYS_MSG_QueueStatus	Returns the message queue status for a given message priority.

f) Data Types and Constants

	Name	Description
	SYS_MSG_INIT	Contains all the data necessary to initialize an instance of the System Messaging Service.
	SYS_MSG_INSTANCE	System Messaging instances numbering is from 0,1, to SYS_MSG_MAX_INSTANCE.
	SYS_MSG_OBJECT	This is type SYS_MSG_OBJECT.
	SYS_MSG_QUEUE_STATUS	Messaging queue status enumeration.
	SYS_MSG_RECEIVE_CALLBACK	Pointer to the System message received callback function.
	SYS_MSG_RESULTS	Enumeration of message send results.
	SYS_MSGQ_ELEMENT	Defines queue element for message queue belonging to each priority.
	SYS_MSG_MAILBOXES_ADDONE	don't need to round up the number of bitmaps
	SYS_MSG_NUM_MAILBOX_BITMAPS	This is macro SYS_MSG_NUM_MAILBOX_BITMAPS.
	SYS_OBJ_HANDLE_INVALID	This is macro SYS_OBJ_HANDLE_INVALID.
	SYS_OBJ_HANDLE_STATIC	This is macro SYS_OBJ_HANDLE_STATIC.
	SYS_OBJ_HANDLE	SYS_MODULE_OBJ was poorly named. It should be SYS_MODULE_OBJ_HANDLE or something shorter. For brevity, it was renamed to SYS_OBJ_HANDLE.

Description

This section describes the APIs of the Messaging System Service Library.

Refer to each section for a detailed description.

a) Initialization, Tasks, and Versioning Functions

SYS_MSG_Deinitialize Function

Deinitializes System Messaging Instance.

File

[sys_msg.h](#)

C

```
void SYS_MSG_Deinitialize(SYS_OBJ_HANDLE handleSysMsg);
```

Returns

None.

Description

This function deinitializes the System Messaging Instance and frees up allocated memory for it.

Remarks

None.

Preconditions

None.

Example

```
SYS_OBJ_HANDLE hSysMsg;
uint16_t nQSizes[] = SYS_MSG_BUFFER_SIZES;

// Initialize the messaging system. This needs to be done in the initialization code.
// Choose System Messaging instance that supports 8 byte messages
hSysMsg = SYS_MSG_Initialize(SYS_MSG_8Bytes, SYS_MSG_MAX_PRIORITY+1, nQSizes);

if (SYS_OBJ_HANDLE_INVALID == hSysMsg)
{
    // Handle error
}
.
.
.
// Remove this instance.
SYS_MSG_Deinitialize( hSysMsg );
```

Parameters

Parameters	Description
hSysMsg	handle to System Messaging Object for instance to be removed.

Function

```
void SYS_MSG_Deinitialize ( SYS_OBJ_HANDLE hSysMsg )
```

SYS_MSG_Initialize Function

Configures and initializes the messaging subsystem.

File

[sys_msg.h](#)

C

```
SYS_OBJ_HANDLE SYS_MSG_Initialize(const SYS_MSG_INSTANCE iSysMsg, SYS_OBJ_HANDLE pInitializeSysMsg);
```

Returns

Handle to the System Messaging object created. Returns [SYS_OBJ_HANDLE_INVALID](#) if allocation of data structure fails. Returns [SYS_OBJ_HANDLE_INVALID](#) if pointer to initialization data structure is NULL.

Description

This function configures and initializes the messaging subsystem appropriately for the current system design.

Remarks

None.

Preconditions

None.

Example

```
SYS_OBJ_HANDLE hSysMsg;
SYS_MSG_INIT sInitSysMsg = { 0, (SYS_MSG_MAX_PRIORITY+1), { SYS_MSG_BUFFER_SIZES } };
//uint16_t nQSizes[] = SYS_MSG_BUFFER_SIZES;

// Initialize the messaging system. This needs to be done in the initialization code.
// Choose System Messaging instance that supports 8 byte messages
hSysMsg = SYS_MSG_Initialize(SYS_MSG_8Bytes,&sInitSysMsg);

if (SYS_OBJ_HANDLE_INVALID == hSysMsg)
{
    // Handle error
}
```

Parameters

Parameters	Description
iSysMsg	Index of System Messaging Service to be initialized.
pInitSysMsg	Pointer to System Messaging initialization data structure. If NULL default config values are used.

Function

```
SYS_OBJ_HANDLE SYS_MSG_Initialize ( const SYS_MSG_INSTANCE iSysMsg,
                                   SYS_OBJ_HANDLE pInitSysMsg )
```

SYS_MSG_Tasks Function

System Messaging Service Tasks routine.

File

```
sys_msg.h
```

C

```
void SYS_MSG_Tasks(SYS_OBJ_HANDLE handleSysMsg);
```

Returns

None.

Description

This function is the System Messaging Service Tasks routine.

Remarks

None.

Preconditions

hSysMsg must have been returned from a call to [SYS_MSG_Initialize](#).

Example

```
while ( SYS_MSG_GotMessages( iSysMsg) )
{
    SYS_MSG_Tasks( hSysMsg );
}
```

Parameters

Parameters	Description
hSysMsg	handle to System Messaging Object.

Function

```
void SYS_MSG_Tasks ( SYS_OBJ_HANDLE hSysMsg )
```

b) Mailbox Functions

SYS_MSG_MailboxMessagesGet Function

Gets queued messages for a mailbox.

File

[sys_msg.h](#)

C

```
SYS_MSG_OBJECT * SYS_MSG_MailboxMessagesGet( SYS_OBJ_HANDLE hMailbox );
```

Returns

Pointer to next message in the queue(s) that is of interest to the mailbox. Function returns NULL if no messages are found in the queue.

Description

This function gets the queued messages for a mailbox. Messages returned by this function will not be received via the mailbox's callback function.

Remarks

None.

Preconditions

hMailbox provided by call to [SYS_MSG_MailboxOpen](#).

Example

```
SYS_MSG_OBJECT *pNextMessage;  
while ( NULL != (pNextMessage = SYS_MSG_MailboxMessagesGet(hMyMailbox)) )  
{  
    // Process message at *pNextMessage.  
}
```

Parameters

Parameters	Description
hMailbox	Object handle to mailbox

Function

```
SYS_MSG_OBJECT *SYS_MSG_MailboxMessagesGet( SYS_OBJ_HANDLE hMailbox )
```

SYS_MSG_MailboxMsgAdd Function

Adds a message type to the list of messages received by a mailbox.

File

[sys_msg.h](#)

C

```
void SYS_MSG_MailboxMsgAdd( SYS_OBJ_HANDLE hMailbox, SYS_OBJ_HANDLE hMsgType );
```

Returns

None.

Description

This function adds a message type to the list of messages received by a mailbox.

Remarks

When the message type handle is unknown but the message ID is known use [SYS_MSG_ID2hMsgType](#) to provide the message type handle. See code example.

Preconditions

hMailbox provided by call to [SYS_MSG_MailboxOpen](#).

Example

```
const SYS_MSG_INSTANCE iSysMsg;
SYS_OBJ_HANDLE hMsgType[5], hMailbox[3];

// Create three mailboxes
hMailbox[0] = SYS_MSG_MailboxOpen( iSysMsg, &myCallBack0 );
hMailbox[1] = SYS_MSG_MailboxOpen( iSysMsg, &myCallBack1 );
hMailbox[2] = SYS_MSG_MailboxOpen( iSysMsg, &myCallBack2 );

// Create five message types
//                                     Message ID:  :Priority
hMsgType[0] = SYS_MSG_TypeCreate(iSysMsg,1<<0,0);
hMsgType[1] = SYS_MSG_TypeCreate(iSysMsg,1<<1,1);
hMsgType[2] = SYS_MSG_TypeCreate(iSysMsg,1<<2,2);
hMsgType[3] = SYS_MSG_TypeCreate(iSysMsg,1<<3,3);
hMsgType[4] = SYS_MSG_TypeCreate(iSysMsg,1<<4,4);

// Add messages to each mailbox
SYS_MSG_MailboxMsgAdd(hMailbox[0],hMsgType[0]);
SYS_MSG_MailboxMsgAdd(hMailbox[1],hMsgType[1]);
SYS_MSG_MailboxMsgAdd(hMailbox[2],hMsgType[2]);

SYS_MSG_MailboxMsgAdd(hMailbox[0],SYS_MSG_ID2hMsgType(iSysMsg,1<<1));
SYS_MSG_MailboxMsgAdd(hMailbox[0],SYS_MSG_ID2hMsgType(iSysMsg,1<<2));
SYS_MSG_MailboxMsgAdd(hMailbox[0],SYS_MSG_ID2hMsgType(iSysMsg,1<<3));
SYS_MSG_MailboxMsgAdd(hMailbox[0],SYS_MSG_ID2hMsgType(iSysMsg,1<<4));
```

Parameters

Parameters	Description
hMailbox	Object handle to mailbox
hMsgType	Handle to message type of interest for this mailbox.

Function

```
void SYS_MSG_MailboxMsgAdd( SYS\_OBJ\_HANDLE hMailbox, SYS\_OBJ\_HANDLE hMsgType )
```

SYS_MSG_MailboxMsgRemove Function

Removes a message type from the list of messages received by a mailbox.

File

[sys_msg.h](#)

C

```
void SYS_MSG_MailboxMsgRemove(SYS_OBJ_HANDLE hMailbox, SYS_OBJ_HANDLE hMsgType);
```

Returns

None.

Description

This function removes a message type from the list of messages received by a mailbox.

Remarks

When the message type handle is unknown but the message ID is known use [SYS_MSG_ID2hMsgType](#) to provide the message type handle. See

code example.

Preconditions

hMailbox provided by call to [SYS_MSG_MailboxOpen](#).

Example

```
const SYS_MSG_INSTANCE iSysMsg;
SYS_OBJ_HANDLE hMsgType[5], hMailbox[3];

// Create three mailboxes
hMailbox[0] = SYS_MSG_MailboxOpen( iSysMsg, &myCallBack0 );
hMailbox[1] = SYS_MSG_MailboxOpen( iSysMsg, &myCallBack1 );
hMailbox[2] = SYS_MSG_MailboxOpen( iSysMsg, &myCallBack2 );

// Create five message types
//                                     Message ID:  :Priority
hMsgType[0] = SYS_MSG_TypeCreate(iSysMsg,1<<0,0);
hMsgType[1] = SYS_MSG_TypeCreate(iSysMsg,1<<1,1);
hMsgType[2] = SYS_MSG_TypeCreate(iSysMsg,1<<2,2);
hMsgType[3] = SYS_MSG_TypeCreate(iSysMsg,1<<3,3);
hMsgType[4] = SYS_MSG_TypeCreate(iSysMsg,1<<4,4);

// Add messages to each mailbox
SYS_MSG_MailboxMsgAdd(hMailbox[0],hMsgType[0]);
SYS_MSG_MailboxMsgAdd(hMailbox[1],hMsgType[1]);
SYS_MSG_MailboxMsgAdd(hMailbox[2],hMsgType[2]);

SYS_MSG_MailboxMsgAdd(hMailbox[0],SYS_MSG_ID2hMsgType(iSysMsg,1<<1));
SYS_MSG_MailboxMsgAdd(hMailbox[0],SYS_MSG_ID2hMsgType(iSysMsg,1<<2));
SYS_MSG_MailboxMsgAdd(hMailbox[0],SYS_MSG_ID2hMsgType(iSysMsg,1<<3));
SYS_MSG_MailboxMsgAdd(hMailbox[0],SYS_MSG_ID2hMsgType(iSysMsg,1<<4));
```

Parameters

Parameters	Description
hMailbox	Object handle to mailbox
hMsgType	Handle to message type to be ignored by this mailbox.

Function

void SYS_MSG_MailboxMsgRemove([SYS_OBJ_HANDLE](#) hMailbox, [SYS_OBJ_HANDLE](#) hMsgType)

SYS_MSG_MailboxClose Function

Closes (destroys) a mailbox previously opened with [SYS_MSG_MailboxOpen](#).

File

[sys_msg.h](#)

C

```
void SYS_MSG_MailboxClose( SYS_OBJ_HANDLE hMailbox );
```

Returns

None.

None.

Description

This function closes (destroys) a mailbox previously opened with [SYS_MSG_MailboxOpen](#).

Preconditions

hMailbox provided by call to [SYS_MSG_MailboxOpen](#).

Example

```
SYS_OBJ_HANDLE hMyMailbox;

hMyMailbox = SYS_MSG_MailboxOpen( iSysMsg, &myCallBackFunction );
```

```
SYS_MSG_MailboxClose(hMyMailbox);
```

Parameters

Parameters	Description
hMailbox	Handle to mailbox that is to be closed (destroyed).

Function

```
void SYS_MSG_MailboxClose( SYS_OBJ_HANDLE hMailbox )
```

SYS_MSG_MailboxOpen Function

Opens a system messaging mailbox.

File

[sys_msg.h](#)

C

```
SYS_OBJ_HANDLE SYS_MSG_MailboxOpen(const SYS_MSG_INSTANCE iSysMsg, SYS_MSG_RECEIVE_CALLBACK  
msgCallBackFunction);
```

Returns

Handle to new system messaging mailbox.

Description

This function opens a system messaging mailbox, providing a message callback function that is called whenever a message is received.

Remarks

A null callback function disables messaging callbacks.

Preconditions

iSysMsg must have been used in a call to [SYS_MSG_Initialize](#). Message callback function will not be called until SYS_MSG_MailboxSignUp has been used to sign up the mailbox for messages of interest.

Example

```
SYS_OBJ_HANDLE hMyMailbox;  
  
hMyMailbox = SYS_MSG_MailboxOpen( iSysMsg, &myCallBackFunction );  
if (SYS_OBJ_HANDLE_INVALID == SYS_MSG_TypeRemove)  
{  
    // Handle error  
}
```

Parameters

Parameters	Description
iSysMsg	Index of System Messaging Service instance.
msgCallBackFunction	pointer to message callback function

Function

```
SYS_OBJ_HANDLE SYS_MSG_MailboxOpen( const SYS_MSG_INSTANCE iSysMsg,  
SYS_MSG_RECEIVE_CALLBACK msgCallBackFunction );
```

SYS_MSG_MailboxReinit Function

Reinitializes a previously opened mailbox.

File

[sys_msg.h](#)

C

```
void SYS_MSG_MailboxReinit(SYS_OBJ_HANDLE hMailbox, SYS_MSG_RECEIVE_CALLBACK msgCallBackFunction);
```

Returns

None.

Description

This function reinitializes a previously opened mailbox by providing a new callback function and clearing all message type assignments.

Remarks

A null callback function disables messaging callbacks.

Preconditions

hMailbox provided by call to [SYS_MSG_MailboxOpen](#).

Example

```
SYS_OBJ_HANDLE hMyMailbox;

hMyMailbox = SYS_MSG_MailboxOpen( iSysMsg, &myCallBackFunction );
SYS_MSG_MailboxReinit(hMyMailbox,&anotherCallBackFunction );
```

Parameters

Parameters	Description
hMailbox	Object handle to mailbox
msgCallBackFunction	pointer to new message callback function

Function

```
void SYS_MSG_MailboxReinit( SYS_OBJ_HANDLE hMailbox,
                           SYS_MSG_RECEIVE_CALLBACK msgCallBackFunction );
```

c) Message Type Functions

SYS_MSG_TypeCreate Function

Creates a new message type.

File

[sys_msg.h](#)

C

```
SYS_OBJ_HANDLE SYS_MSG_TypeCreate(const SYS_MSG_INSTANCE iSysMsg, uint8_t nMessageTypeID, uint8_t
nMessagePriority);
```

Returns

Handle to new message type definition.

Description

This function creates a new message type, defining an integer message type and priority.

Remarks

None.

Preconditions

iSysMsg must have been used in a call to [SYS_MSG_Initialize](#).

Example

```
SYS_OBJ_HANDLE hMsgType;
hMsgType = SYS_MSG_TypeCreate( iSysMsg, 1, 3 );
if (SYS_OBJ_HANDLE_INVALID == hMsgType)
{
    // Handle error
}
```

Parameters

Parameters	Description
iSysMsg	Index of System Messaging Service instance.
nMessageTypeID	Integer message type identifier
nMessagePriority	Message priority, between 0 and SYS_MSG_MAX_PRIORITIES

Function

```
SYS_OBJ_HANDLE SYS_MSG_TypeCreate( const SYS_MSG_INSTANCE iSysMsg,  
uint8_t nMessageTypeID,  
uint8_t nMessagePriority )
```

SYS_MSG_TypeRemove Function

Removes an existing message type.

File

[sys_msg.h](#)

C

```
void SYS_MSG_TypeRemove( SYS_OBJ_HANDLE hMsgType );
```

Returns

None.

Description

This function removes an existing message type.

Remarks

When the message type handle is unknown but the message ID is known use [SYS_MSG_ID2hMsgType](#) to provide the message type handle. See code example.

Preconditions

None.

Example

```
SYS_OBJ_HANDLE hMsgType;  
hMsgType = SYS_MSG_TypeCreate( iSysMsg, 1, 3 );  
SYS_MSG_TypeRemove( hMsgType );
```

Alternately:

```
SYS_OBJ_HANDLE hMsgType;  
hMsgType = SYS_MSG_TypeCreate( iSysMsg, 1, 3 );  
SYS_MSG_TypeRemove( SYS_MSG_ID2hMsgType( iSysMsg, 1 ) );
```

Parameters

Parameters	Description
hMsgType	Handle to message type that is to be removed

Function

```
void SYS_MSG_TypeRemove( SYS_OBJ_HANDLE hMsgType )
```

d) Message Send/Receive Functions

SYS_MSG_GotMessages Function

Returns true if system messaging has undelivered messages, false otherwise.

File

[sys_msg.h](#)

C

```
bool SYS_MSG_GotMessages(const SYS_MSG_INSTANCE iSysMsg);
```

Returns

- true - Undelivered system messages exist
- false - No undelivered system messages exist

Description

This function returns the status when system messaging has undelivered messages.

Remarks

None.

Preconditions

iSysMsg must have been used in a call to [SYS_MSG_Initialize](#).

Example

```
DBPRINTF("rnGot Messages: %drn",SYS_MSG_GotMessages(iSysMsg));
while ( (pNextMessage = SYS_MSG_MessageReceive(iSysMsg)) != NULL )
{
    SYS_MSG_MessageDeliver(iSysMsg,pNextMessage);
    DBPRINTF("rn");
}
DBPRINTF("Got Messages: %drn",SYS_MSG_GotMessages(ihSysMsg));
```

Parameters

Parameters	Description
iSysMsg	Index of System Messaging Service instance.

Function

```
bool SYS_MSG_GotMessages(const SYS_MSG_INSTANCE iSysMsg)
```

SYS_MSG_MessageDeliver Function

Delivers messages to mailboxes.

File

[sys_msg.h](#)

C

```
void SYS_MSG_MessageDeliver(const SYS_MSG_INSTANCE iSysMsg, SYS_MSGQ_ELEMENT * pQElement);
```

Returns

None.

Description

This function delivers messages to mailboxes, and removes the message from the queue when done.

Remarks

None.

Preconditions

iSysMsg must have been used in a call to [SYS_MSG_Initialize](#).

Example

```
SYS_MSGQ_ELEMENT *pQElement;
SYS_MSG_OBJECT nextMessage;
while ( (pQElement = SYS_MSG_MessageReceive(iSysMsg)) != NULL )
{
    // In case you desire to examine message before delivering it.
    nextMessage = pQElement->sMessage;
```

```

    // Deliver message to all interested mailboxes
    SYS_MSG_MessageDeliver( iSysMsg, pQElement );
}

```

Parameters

Parameters	Description
iSysMsg	Index of System Messaging Service instance.
pQElement	pointer to queue element to be delivered.

Function

```
void SYS_MSG_MessageDeliver( const SYS_MSG_INSTANCE iSysMsg, SYS_MSGQ_ELEMENT *pQElement );
```

SYS_MSG_MessageReceive Function

Receives the next message in the message queues.

File

[sys_msg.h](#)

C

```
SYS_MSGQ_ELEMENT * SYS_MSG_MessageReceive( const SYS_MSG_INSTANCE iSysMsg );
```

Returns

Pointer to next message, as found in a message queue element, NULL if there are no messages.

Description

This function receives the next message in the message queues, returning NULL if the queues are empty.

Remarks

None.

Preconditions

iSysMsg must have been used in a call to [SYS_MSG_Initialize](#).

Example

```

SYS_MSGQ_ELEMENT *pQElement;
SYS_MSG_OBJECT nextMessage;
while ( (pQElement = SYS_MSG_MessageReceive(iSysMsg)) != NULL )
{
    // In case you desire to examine message before delivering it.
    nextMessage = pQElement->sMessage;

    // Deliver message to all interested mailboxes
    SYS_MSG_MessageDeliver( iSysMsg, pQElement );
}

```

Parameters

Parameters	Description
iSysMsg	Index of System Messaging Service instance.

Function

```
SYS_MSGQ_ELEMENT *SYS_MSG_MessageReceive( const SYS_MSG_INSTANCE iSysMsg )
```

SYS_MSG_MessageSend Function

Sends a message, as defined by a message structure.

File

[sys_msg.h](#)

C

```
SYS_MSG_RESULTS SYS_MSG_MessageSend( const SYS_MSG_INSTANCE iSysMsg, SYS_MSG_OBJECT * pMessage );
```

Returns

Message result from [SYS_MSG_RESULTS](#) enumeration.

Description

This function sends a message, as defined by a message structure.

Remarks

None.

Preconditions

iSysMsg must have been used in a call to [SYS_MSG_Initialize](#).

Example

```
SYS_MSG_OBJECT myMessage;
SYS_MSG_RESULTS myMessageStatus;
SYS_OBJ_HANDLE hMyMsgType;

hMyMsgType = SYS_MSG_TypeCreate( iSysMsg, 1, 3 );
if (SYS_OBJ_HANDLE_INVALID == hMyMsgType)
{
    // Handle error
}

myMessage.hMsgType = hMyMsgType;
myMessage.nSource = myMsgSource;
myMessage.param1 = parameterOneValue;
myMessage.param1 = parameterTwoValue;

myMessageStatus = SYS_MSG_MessageSend( iSysMsg, &myMessage );
SYS_ASSERT( myMessageStatus > 0, "Bad message status!" );
```

Parameters

Parameters	Description
iSysMsg	Index of System Messaging Service instance.
pMessage	Pointer to message definition.

Function

```
SYS_MSG_RESULTS SYS_MSG_MessageSend( const SYS_MSG_INSTANCE iSysMsg,
                                     SYS_MSG_OBJECT *pMessage )
```

e) Utility Functions

SYS_MSG_ID2hMsgType Function

Translates message type identifier into handle of corresponding message type object.

File

[sys_msg.h](#)

C

```
SYS_OBJ_HANDLE SYS_MSG_ID2hMsgType( const SYS_MSG_INSTANCE iSysMsg, uint8_t nMessageTypeID );
```

Returns

Handle to message type definition object corresponding to the message type identifier. Returns NULL if the message type is not defined.

Description

This function translates the message type identifier into the handle of the corresponding message type object.

Remarks

This function is useful in situations where the message type identifier is known but the message type handle is not. This allows applications to statically define message type IDs and use them in code instead of having to wait until message handles are known and then dynamically sharing

message handles via global variables.

Preconditions

iSysMsg must have been used in a call to [SYS_MSG_Initialize](#).

Example

```
#define MY_MESSAGE_TYPE_ID 1;

// Create message type
SYS_OBJ_HANDLE hMsgType;
uint8_t nPriority = 3;
hMsgType = SYS_MSG_TypeCreate( iSysMsg, MY_MESSAGE_TYPE_ID, nPriority );
if (SYS_OBJ_HANDLE_INVALID == hMsgType)
{
    // Handle error
}
.
.
.
// Remove message type without knowing message type handle
SYS_MSG_TypeRemove( iSysMsg, SYS_MSG_ID2hMsgType( iSysMsg, MY_MESSAGE_TYPE_ID ) );
```

Parameters

Parameters	Description
iSysMsg	Index of System Messaging Service instance.
nMessageTypeID	Integer message type identifier.

Function

```
SYS_OBJ_HANDLE SYS_MSG_ID2hMsgType( const SYS_MSG_INSTANCE iSysMsg,
uint8_t nMessageTypeID )
```

SYS_MSG_QueueStatus Function

Returns the message queue status for a given message priority.

File

[sys_msg.h](#)

C

```
SYS_MSG_QUEUE_STATUS SYS_MSG_QueueStatus( const SYS_MSG_INSTANCE iSysMsg, uint8_t nMessagePriority );
```

Returns

Number of messages in queue if not full or SYS_MSG_QUEUE_FULL if full. If message priority is not legal, returns SYS_MSG_QUEUE_BAD.

Description

This function returns the message queue status for a given message priority.

Remarks

None.

Preconditions

iSysMsg must have been used in a call to [SYS_MSG_Initialize](#).

Example

Parameters

Parameters	Description
iSysMsg	Index of System Messaging Service instance.
nMessagePriority	message priority of interest, from zero to SYS_MSG_MAX_PRIORITIES.

Function

```
SYS_MSG_QUEUE_STATUS SYS_MSG_QueueStatus( const SYS_MSG_INSTANCE iSysMsg, uint8_t nMessagePriority )
```

f) Data Types and Constants

SYS_MSG_INIT Structure

Contains all the data necessary to initialize an instance of the System Messaging Service.

File

[sys_msg.h](#)

C

```
typedef struct {
    uint8_t nMaxMsgsDelivered;
    uint8_t nMessagePriorities;
    uint16_t * nQSizes;
} SYS_MSG_INIT;
```

Members

Members	Description
uint8_t nMaxMsgsDelivered;	Maximum number of messages delivered per call to SYS_MSG_Tasks
uint8_t nMessagePriorities;	Number of message priorities desired
uint16_t * nQSizes;	Array of queue sizes for priorities 0,1,... SYS_MSG_MAX_PRIORITY

Description

System Messaging Service Initialization Data

This data type contains all of the data necessary to initialize an instance of the System Messaging Service.

Remarks

A pointer to a structure of this format containing the desired initialization data must be passed into the [SYS_MSG_Initialize](#) routine. If nMaxMsgsDelivered == 0. ALL messages in priority queues are delivered each time [SYS_MSG_Tasks](#) is called.

SYS_MSG_INSTANCE Enumeration

System Messaging instances numbering is from 0,1, to SYS_MSG_MAX_INSTANCE.

File

[sys_msg.h](#)

C

```
typedef enum {
    SYS_MSG_0,
    SYS_MSG_1,
    SYS_MSG_2,
    SYS_MSG_3,
    SYS_MSG_4,
    SYS_MSG_NUM_INSTANCES
} SYS_MSG_INSTANCE;
```

Description

Enumeration of the Allowable of System Messaging Instances

System Messaging instances numbering is from 0,1, to SYS_MSG_MAX_INSTANCE.

Remarks

None.

SYS_MSG_OBJECT Structure

File

[sys_msg.h](#)

C

```
typedef struct {
    union {
        struct {
            uint8_t nMessageTypeID;
            uint8_t nSource;
            uint16_t param0;
            uint16_t param1;
            uint16_t param2;
        }
        struct {
            uint16_t dummy;
            uint16_t nSizeData;
            uintptr_t * pData;
        }
    }
} SYS_MSG_OBJECT;
```

Members

Members	Description
uint8_t nMessageTypeID;	Message type identifier
uint8_t nSource;	Message source identifier
uint16_t param0;	Message parameter zero
uint16_t param1;	Message parameter one
uint16_t param2;	Message parameter two
uint16_t nSizeData;	Size of data that pData identifies
uintptr_t * pData;	Pointer to additional message data

Description

This is type SYS_MSG_OBJECT.

SYS_MSG_QUEUE_STATUS Enumeration

Messaging queue status enumeration.

File

[sys_msg.h](#)

C

```
typedef enum {
    SYS_MSG_QUEUE_BAD,
    SYS_MSG_QUEUE_FULL,
    SYS_MSG_QUEUE_EMPTY
} SYS_MSG_QUEUE_STATUS;
```

Members

Members	Description
SYS_MSG_QUEUE_BAD	QUEUE Status: full
SYS_MSG_QUEUE_FULL	QUEUE Status: full
SYS_MSG_QUEUE_EMPTY	QUEUE Status: empty

Description

System Messaging Queue Status Enumeration

This enumeration provides the messaging queue status. Positive values indicate the number of messages in the queue.

Remarks

None.

SYS_MSG_RECEIVE_CALLBACK Type

Pointer to the System message received callback function.

File

[sys_msg.h](#)

C

```
typedef void (* SYS_MSG_RECEIVE_CALLBACK) (SYS_MSG_OBJECT *pMessage);
```

Description

Pointer to the System Message Received Callback Function

This data type is a pointer to the function provided for each system messaging mailbox that is called when a system message is received for each mailbox.

SYS_MSG_RESULTS Enumeration

Enumeration of message send results.

File

[sys_msg.h](#)

C

```
typedef enum {  
    SYS_MSG_NOT_SENT_QFULL,  
    SYS_MSG_BAD_PRIORITY,  
    SYS_MSG_BAD_MSGTYPE,  
    SYS_MSG_NOT_SENT,  
    SYS_MSG_SENT  
} SYS_MSG_RESULTS;
```

Members

Members	Description
SYS_MSG_NOT_SENT_QFULL	Message could not be sent, no room available in priority queues
SYS_MSG_BAD_PRIORITY	Message could not be sent, Message Message priority bad
SYS_MSG_BAD_MSGTYPE	Message could not be sent, Message type bad
SYS_MSG_NOT_SENT	Message could not be sent, no other information available
SYS_MSG_SENT	Message sent

Description

System Messaging Results Enumeration

This enumeration provides message send results.

Remarks

SYS_MSG_SENT aligns with SYS_MSGQ_Success. SYS_MSG_NOT_SENT aligns with SYS_MSGQ_Failure

SYS_MSGQ_ELEMENT Structure

Defines queue element for message queue belonging to each priority.

File

[sys_msg.h](#)

C

```
typedef struct {  
    SYS_MSG_OBJECT sMessage;  
    uint16_t mailboxInterestBitMap[SYS_MSG_NUM_MAILBOX_BITMAPS];  
} SYS_MSGQ_ELEMENT;
```

Members

Members	Description
SYS_MSG_OBJECT sMessage;	System Message Bit map for mailboxes interested in this message type, modified as each mailbox is notified.

Description

System Messaging Queues Element

This data type defines the queue element for the message queue belonging to each priority.

Remarks

None.

SYS_MSG_MAILBOXES_ADDONE Macro

File

[sys_msg.h](#)

C

```
#define SYS_MSG_MAILBOXES_ADDONE 0
```

Description

don't need to round up the number of bitmaps

SYS_MSG_NUM_MAILBOX_BITMAPS Macro

File

[sys_msg.h](#)

C

```
#define SYS_MSG_NUM_MAILBOX_BITMAPS (SYS_MSG_MAX_MAILBOXES/16 + SYS_MSG_MAILBOXES_ADDONE)
```

Description

This is macro SYS_MSG_NUM_MAILBOX_BITMAPS.

SYS_OBJ_HANDLE_INVALID Macro

File

[sys_msg.h](#)

C

```
#define SYS_OBJ_HANDLE_INVALID ( (SYS_OBJ_HANDLE) -1 )
```

Description

This is macro SYS_OBJ_HANDLE_INVALID.

SYS_OBJ_HANDLE_STATIC Macro

File

[sys_msg.h](#)

C

```
#define SYS_OBJ_HANDLE_STATIC ( (SYS_OBJ_HANDLE) 0 )
```

Description

This is macro SYS_OBJ_HANDLE_STATIC.

SYS_OBJ_HANDLE Type

[SYS_MODULE_OBJ](#) was poorly named. It should be SYS_MODULE_OBJ_HANDLE or something shorter. For brevity, it was renamed to SYS_OBJ_HANDLE.

File

[sys_msg.h](#)

C

```
typedef uintptr_t SYS_OBJ_HANDLE;
```

Description

[SYS_MODULE_OBJ](#) Rename

[SYS_MODULE_OBJ](#) was poorly named. It should be [SYS_MODULE_OBJ_HANDLE](#) or something shorter. For brevity, it was renamed to [SYS_OBJ_HANDLE](#).

Remarks

None.

Files**Files**

Name	Description
sys_msg.h	System Service for the messaging module.
sys_msg_config.h	System Messaging Configuration definitions file

Description

This section lists the source and header files used by the library.


















sys_msg.h

System Service for the messaging module.

Enumerations

	Name	Description
	SYS_MSG_INSTANCE	System Messaging instances numbering is from 0,1, to SYS_MSG_MAX_INSTANCE .
	SYS_MSG_QUEUE_STATUS	Messaging queue status enumeration.
	SYS_MSG_RESULTS	Enumeration of message send results.

Functions

	Name	Description
	SYS_MSG_Deinitialize	Deinitializes System Messaging Instance.
	SYS_MSG_GotMessages	Returns true if system messaging has undelivered messages, false otherwise.
	SYS_MSG_ID2hMsgType	Translates message type identifier into handle of corresponding message type object.
	SYS_MSG_Initialize	Configures and initializes the messaging subsystem.
	SYS_MSG_MailboxClose	Closes (destroys) a mailbox previously opened with SYS_MSG_MailboxOpen .
	SYS_MSG_MailboxMessagesGet	Gets queued messages for a mailbox.
	SYS_MSG_MailboxMsgAdd	Adds a message type to the list of messages received by a mailbox.
	SYS_MSG_MailboxMsgRemove	Removes a message type from the list of messages received by a mailbox.
	SYS_MSG_MailboxOpen	Opens a system messaging mailbox.
	SYS_MSG_MailboxReinit	Reinitializes a previously opened mailbox.
	SYS_MSG_MessageDeliver	Delivers messages to mailboxes.
	SYS_MSG_MessageReceive	Receives the next message in the message queues.
	SYS_MSG_MessageSend	Sends a message, as defined by a message structure.
	SYS_MSG_QueueStatus	Returns the message queue status for a given message priority.
	SYS_MSG_Tasks	System Messaging Service Tasks routine.
	SYS_MSG_TypeCreate	Creates a new message type.
	SYS_MSG_TypeRemove	Removes an existing message type.

Macros

	Name	Description
	SYS_MSG_MAILBOXES_ADDONE	don't need to round up the number of bitmaps
	SYS_MSG_NUM_MAILBOX_BITMAPS	This is macro SYS_MSG_NUM_MAILBOX_BITMAPS .
	SYS_OBJ_HANDLE_INVALID	This is macro SYS_OBJ_HANDLE_INVALID .

	SYS_OBJ_HANDLE_STATIC	This is macro <code>SYS_OBJ_HANDLE_STATIC</code> .
--	---------------------------------------	--

Structures

	Name	Description
	SYS_MSG_INIT	Contains all the data necessary to initialize an instance of the System Messaging Service.
	SYS_MSG_OBJECT	This is type <code>SYS_MSG_OBJECT</code> .
	SYS_MSGQ_ELEMENT	Defines queue element for message queue belonging to each priority.

Types

	Name	Description
	SYS_MSG_RECEIVE_CALLBACK	Pointer to the System message received callback function.
	SYS_OBJ_HANDLE	SYS_MODULE_OBJ was poorly named. It should be <code>SYS_MODULE_OBJ_HANDLE</code> or something shorter. For brevity, it was renamed to <code>SYS_OBJ_HANDLE</code> .

Description

Messaging System Service Library Interface Definitions

This file contains the interface definition for the messaging system service. It provides a way to interact with the messaging subsystem.

File Name

sys_msg.h

Company

Microchip Technology Inc.

sys_msg_config.h

System Messaging Configuration definitions file

Macros

	Name	Description
	_SYS_MSG_CONFIG_TEMPLATE__H	This is macro <code>_SYS_MSG_CONFIG_TEMPLATE__H</code> .
	SYS_MSG_BUFFER_SIZES	define <code>SYS_MSG_BUFFER_SIZES { 4, 4, 4, 4, 1</code>
	SYS_MSG_MAX_MAILBOXES	Specifies the maximum number of mailboxes possible.
	SYS_MSG_MAX_MSGS_DELIVERED	Specifies the maximum number of messages delivered per each call to SYS_MSG_Tasks .
	SYS_MSG_MAX_PRIORITY	Specifies the maximum message priority.
	SYS_MSG_MAX_TYPES	Specifies the maximum number of message types possible.

Description

System Messaging Configuration Definitions

These definitions statically define the operation of the System Messaging service.

File Name

sys_msg_config.h

Company

Microchip Technology Inc.

Ports System Service Library

This section describes the Ports System Service Library.

Introduction

This library provides an interface to manage and control general purpose input or output ports controlled by the Ports modules on the Microchip families of microcontrollers.

Description

One challenge designers of general purpose microcontroller devices face is to provide a large set of available peripheral features on parts with a limited number of I/O pins. To help meet this challenge and to provide flexibility for board designers, many devices provide an ability to route I/O signals for selected peripherals to different I/O pins. However, in some cases, general purpose I/O (GPIO) pins must be used to implement the desired functionality under direct software control. The purpose of the ports system service is to provide direct control of general purpose I/O pins and to support the selection of the desired peripheral functionality for supported I/O pins and ports.

General purpose I/O pins can be controlled individually, but they are also grouped together in sets and can be controlled together as a unit. These groups of GPIO pins are called "ports" on Microchip microcontrollers and this library provides the ability to read and write data patterns to or from them in sets or as individual pins. In addition peripheral IO routing and pin/port control, this library provides the ability to select and configure several other features of the I/O pins and ports available on Microchip microcontrollers, as follows.

Other Features

- Individual output pin/port open-drain enable/disable
- Individual input pin/port pull-up enable/disable
- Monitor select inputs and generate interrupt on mismatch condition [Change Notification]
- Operate during Sleep and Idle modes
- Port line Analog/Digital Selection
- Port slew rate control



Note: Trademarks and Intellectual Property are property of their respective owners. Customers are responsible for obtaining appropriate licensing or rights before using this software. Refer to the MPLAB Harmony *Software License Agreement* for complete licensing information. A copy of this agreement is available in the <install-dir>/doc folder of your MPLAB Harmony installation.

Using the Library

This topic describes the basic architecture of the Ports System Service Library and provides information and examples on its use.

Description

Interface Header File: [sys_ports.h](#)

The interface to the Ports System Service Library is defined in the [sys_ports.h](#) header file, which is included by the `sys.h` header file.

Any C language source (.c) file that uses the Ports System Service must include `sys.h`.

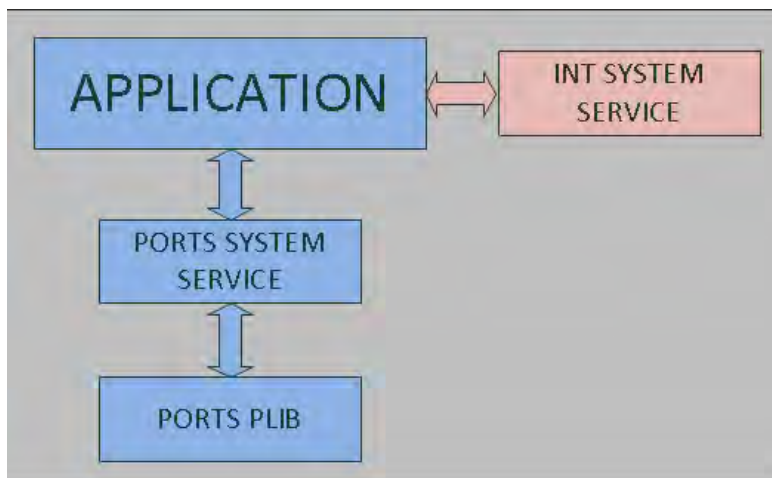
Please refer to the What is MPLAB Harmony? section for how the library interacts with the framework.

Abstraction Model

This library provides a low-level abstraction of the Ports System Service Library. This topic describes how that abstraction is modeled in software and introduces the library's interface.

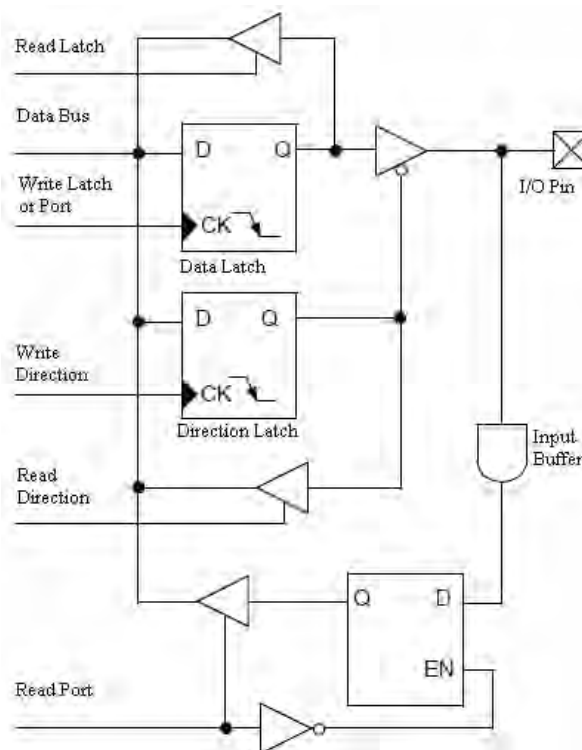
Description

This model explains how the system interfaces with the Ports System Service and the application as illustrated in the following diagram.



General Purpose I/O

All port pins have three registers directly associated with their operation as digital I/O. The Data Direction register determines whether the pin is an input or an output. If the data direction bit is a '1', the pin is an input. All port pins are defined as inputs after a Reset. Reads from the Output Latch register, read the latch, while writes to the latch, write the latch. Reads from the port, read the port pins, while writes to the port pins, write the latch.



The *pull-ups* act as a current source or sink source connected to the pin, and eliminates the need for external resistors when push-button or keypad devices are connected. These *pull-ups* prevent floating state of the pins by providing voltage to it. These features are available on some pins and some parts. Please refer to the specific device data sheet for further information.

The *open-drain* feature allows the generation of outputs higher than VDD (e.g., 5V) on any desired digital only pins by using external pull-up resistors. The maximum open-drain voltage allowed is the same as the maximum VIH specification.

The *input change notification* function of the I/O ports allows the microcontrollers to generate interrupt requests to the processor in response to a change of state on selected input pins. This feature is capable of detecting input change of states even in Sleep mode, when the clocks are disabled.

The *Alternate Pin Function selections* are used to steer specific peripheral input and output functions between different pins.

The *output slew rate* of each port is programmable to select either the standard transition rate or a reduced transition rate of x times the standard to minimize EMI. The reduced transition time is the default slew rate for all ports.

Peripheral Pin Select

Available Pins:

The number of available pins is dependent on the particular device and its pin count. Pins that support the peripheral pin select feature include the

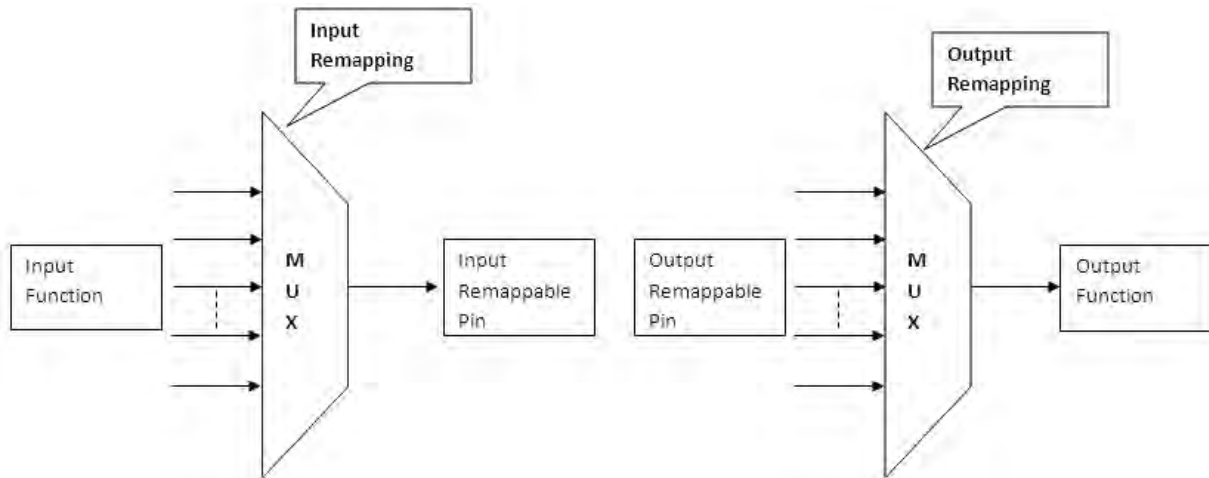
designation "RPn" in their full pin designation, where "RP" designates a remappable peripheral and "n" is the remappable port number.

Available Peripherals:

The peripherals managed by the peripheral pin select are all digital-only peripherals. These include general serial communications (UART and SPI), general purpose timer clock inputs, timer-related peripherals (Input Capture and Output Compare) and interrupt-on-change inputs.

In comparison, some digital-only peripheral modules are never included in the peripheral pin select feature. This is because the peripheral's function requires special I/O circuitry on a specific port and cannot be easily connected to multiple pins. These modules include I2C among others. A similar requirement excludes all modules with analog inputs, such as the A/D converter. A key difference between remappable and non-remappable peripherals is that remappable peripherals are not associated with a default I/O pin. The peripheral must always be assigned to a specific I/O pin before it can be used. In contrast, non-remappable peripherals are always available on a default pin, assuming that the peripheral is active and not conflicting with another peripheral.

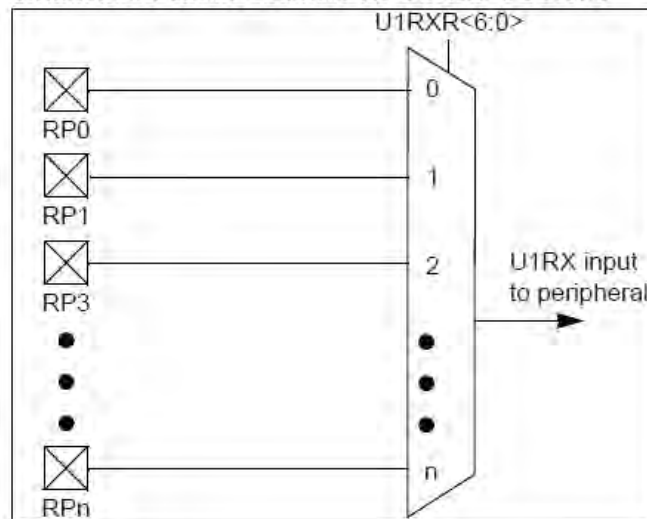
When a remappable peripheral is active on a given I/O pin, it takes priority over all other digital I/O and digital communication peripherals associated with the pin. Priority is given regardless of the type of peripheral that is mapped. Remappable peripherals never take priority over any analog functions associated with the pin.



Input Mapping:

The inputs of the peripheral pin select options are mapped on the basis of the peripheral. That is, a control register associated with a peripheral dictates to which pin it will be mapped.

Remappable Input Example for USART 1 Receive

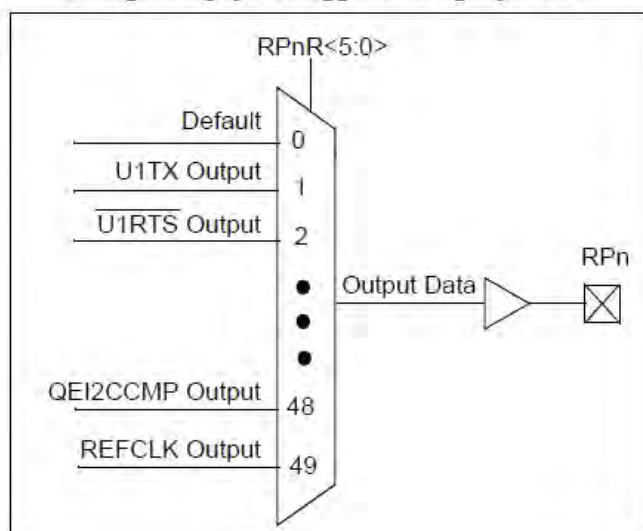


Note: For input only, peripheral pin select functionality does not have priority over I/O port settings. Therefore, when configuring Remappable Pin for input, the corresponding bit in the I/O port register must also be configured for input (set to '1').

Output Mapping:

In contrast to inputs, the outputs of the peripheral pin select options are mapped on the basis of the pin. In this case, a control register associated with a particular pin dictates the peripheral output to be mapped.

Multiplexing of Remappable Output for RPN



Mapping Limitations:

The control schema of the peripheral select pins is not limited to a small range of fixed peripheral configurations. There are no mutual or hardware-enforced lockouts between any of the peripheral mapping SFRs. Literally any combination of peripheral mappings across any or all of the RPN pins is possible. This includes both many-to-one and one-to-many mappings of peripheral inputs and outputs to pins. While such mappings may be technically possible from a configuration point of view, they may not be supportable from an electrical point of view.

Library Overview

Please refer to the [System Service Introduction](#) for a detailed description of MPLAB Harmony system services.

The library interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the Ports System Service Library.

Library Interface Section	Description
Pin Control Functions	Port bit/pin read/write/toggle/clear/set interfaces.
Ports Control Functions	Port access read/write/toggle/clear/set interfaces.
Change Notification Functions	Interface routines for Port line change notification.
Peripheral Pin Select Functions	Interface routines for mapping the digital input/output to a specific PPS Remappable input/output pin.

How the Library Works

Pin Control

Pins Functions Usage

- **Pin Read:** Port pin can be read at bit/pin level using [SYS_PORTS_PinRead](#) with appropriate parameters
- **Pin Write:** Port pin can be written at bit/pin level using [SYS_PORTS_PinWrite](#) with appropriate parameters
- **Pin Clear:** Port pin can be cleared at bit/pin level using [SYS_PORTS_PinClear](#) with appropriate parameters
- **Pin Set:** Port pin can be set at bit/pin level using [SYS_PORTS_PinSet](#) with appropriate parameters
- **Pin Direction Control:** Port pin direction can be set at bit/pin level using [SYS_PORTS_PinDirectionSelect](#) with appropriate parameters. The Direction information can be obtained through the interface [SYS_PORTS_DirectionGet](#).
- **Pin Toggle:** Port pin can be toggled at bit/pin level using [SYS_PORTS_PinToggle](#) with appropriate parameters
- **Pin Open Drain:** Port pin can be enabled open drain functionality at bit/pin level using [SYS_PORTS_PinOpenDrainEnable](#) with appropriate parameters. Similarly, the Port pin can be disabled open drain functionality at bit/pin level using [SYS_PORTS_PinOpenDrainDisable](#) with appropriate parameters.

Example:

```
// PORT Direction setting for output
SYS_PORTS_PinDirectionSelect(MY_PORTS_INSTANCE, SYS_PORTS_DIRECTION_OUTPUT, MY_CHANNEL, MY_PINNUM);
```

```
// PORT Direction setting for input
SYS_PORTS_PinDirectionSelect(MY_PORTS_INSTANCE, SYS_PORTS_DIRECTION_INPUT, MY_CHANNEL, MY_PINNUM);
// Writing a value into a PORT
SYS_PORTS_PinWrite(MY_PORTS_INSTANCE, MY_CHANNEL, MY_PINNUM, MY_VALUE);
// Reading back the previously written value
bool readData = SYS_PORTS_PinRead(MY_PORTS_INSTANCE, MY_CHANNEL, MY_PINNUM);
// Clearing the PORT
SYS_PORTS_PinClear(MY_PORTS_INSTANCE, MY_CHANNEL, MY_PINNUM);
// Setting the port
SYS_PORTS_PinSet(MY_PORTS_INSTANCE, MY_CHANNEL, MY_PINNUM);
// Toggling a PORT
SYS_PORTS_PinToggle(MY_PORTS_INSTANCE, MY_CHANNEL, MY_PINNUM);
```



Note: Not all features are available on all devices. Please refer to the specific device data sheet to determine availability.

Ports Control

Port Functions Usage

- **Port Read:** Ports can be read at byte/word level using the interface [SYS_PORTS_Read](#) with appropriate parameters
- **Port Write:** Ports can be written to at byte/word level using the interface [SYS_PORTS_Write](#) appropriate parameters
- **Port Clear:** Ports can be cleared at byte/word level using the interface [SYS_PORTS_Clear](#) with appropriate parameters
- **Port Set:** Ports can be set at byte/word level using the interface [SYS_PORTS_Set](#) with appropriate parameters
- **Port Direction Control:** Ports direction can be set at byte/word level using the interface [SYS_PORTS_DirectionSelect](#) with appropriate parameters. The Direction information can be obtained through the interface [SYS_PORTS_DirectionGet](#).
- **Port Toggle:** Ports can be toggled at byte/word level using the interface [SYS_PORTS_Toggle](#) with appropriate parameters
- **Port Open Drain:** Ports can be enabled open drain functionality at byte/word level using the interface [SYS_PORTS_OpenDrainEnable](#) with appropriate parameters. Similarly, the Ports can be disabled open drain functionality at byte/word level using the interface [SYS_PORTS_OpenDrainDisable](#) with appropriate parameters.

Example:

```
// PORT Direction setting for output
SYS_PORTS_DirectionSelect(MY_PORTS_INSTANCE, SYS_PORTS_DIRECTION_OUTPUT, MY_CHANNEL,
(PORTS_DATA_MASK)0xFFFF);
// PORT Direction setting for input
SYS_PORTS_DirectionSelect(MY_PORTS_INSTANCE, SYS_PORTS_DIRECTION_INPUT, MY_CHANNEL,
(PORTS_DATA_MASK)0xFFFF);
// Writing a value into a PORT
SYS_PORTS_Write(MY_PORTS_INSTANCE, MY_CHANNEL, (PORTS_DATA_TYPE)0x1234)
// Reading back the previously written value
PORTS_DATA_TYPE readData = SYS_PORTS_Read(MY_PORTS_INSTANCE, MY_CHANNEL);
// Clearing the PORT
SYS_PORTS_Clear(MY_PORTS_INSTANCE, MY_CHANNEL, (PORTS_DATA_MASK)0x00FF);
// Setting the port
SYS_PORTS_Set(MY_PORTS_INSTANCE, MY_CHANNEL, 0x1234, (PORTS_DATA_MASK)0x00FF);
// Toggling a PORT
SYS_PORTS_Toggle(MY_PORTS_INSTANCE, MY_CHANNEL, (PORTS_DATA_MASK)0x00FF);
```



Note: Not all features are available on all devices. Please refer to the specific device data sheet to determine availability.

Change Notification

Change Notification Feature Usage

The change notification feature can be enabled using "[SYS_PORTS_ChangeNotificationEnable](#)". This routine performs the following operations:

- Change notification can be disabled after the successful usage using the interface [SYS_PORTS_ChangeNotificationDisable](#)
- Certain microcontrollers support the global control over the change notification feature using the following interfaces [SYS_PORTS_ChangeNotificationGlobalEnable](#) and [SYS_PORTS_ChangeNotificationGlobalDisable](#)
- If there are any requirements to control the pull-ups [SYS_PORTS_ChangeNotificationPullUpEnable](#) and [SYS_PORTS_ChangeNotificationPullUpDisable](#) could be used

Change Notification Operation in Sleep and Idle Modes

The change notification module continues to operate during Sleep or Idle mode. Its operation can be enabled and disabled using the interfaces [SYS_PORTS_ChangeNotificationIdleModeEnable](#) and [SYS_PORTS_ChangeNotificationIdleModeDisable](#), respectively.

Example:

```
// Enabling the global change notification
SYS_PORTS_ChangeNotificationGlobalEnable(MY_PORTS_INSTANCE);
// Enabling weak pull-ups for the change notification PIN 10
SYS_PORTS_ChangeNotificationPullUpEnable(MY_PORTS_INSTANCE, PORTS_CHANGE_NOTICE_PIN_10);
// Enabling change notification on PIN 10
SYS_PORTS_ChangeNotificationEnable(MY_PORTS_INSTANCE, PORTS_CHANGE_NOTICE_PIN_10, SYS_PORTS_PULLUP_ENABLE);
// Enabling the change notification in idle mode.
SYS_PORTS_ChangeNotificationInIdleModeEnable(MY_PORTS_INSTANCE);
```



Note: Not all features are available on all devices. Please refer to the specific device data sheet to determine availability.

Peripheral Pin Select

Ports Remapping or Peripheral Pin Select Usage

Input/Output Function Remapping:

The `SYS_PORTS_RemapInput` and `SYS_PORTS_RemapOutput` functions with appropriate parameters can be used to remap a particular port pin as input/output for a peripheral.

Example:

```
// Remapping input function 'Input Capture 1' to the Remappable input pin 'RPD2'
SYS_PORTS_RemapInput (PORTS_ID_0, INPUT_FUNC_IC1, INPUT_PIN_RPD2 );
// Remapping output function 'UART1 Transmit' to the Remappable output pin 'RPA14'
SYS_PORTS_RemapInputOutput(PORTS_ID_0, OUTPUT_FUNC_U1TX, OUTPUT_PIN_RPA14);
```



Note: Not all features are available on all devices. Please refer to the specific device data sheet to determine availability.

Miscellaneous

Other Usage

Slew Rate:

Slew rate of a particular port can be controlled through the interfaces `SYS_PORTS_SlewRateSetReduced` and `SYS_PORTS_SlewRateSetStandard`.

Open Drain:

Peripheral based open drain can be controlled through the interfaces `SYS_PORTS_PeripheralOpenDrainEnable` and `SYS_PORTS_PeripheralOpenDrainDisable`.



Note: Not all features are available on all devices. Please refer to the specific device data sheet to determine availability.

Special Considerations

Note on Ports Usage:

1. Setting a port pin as an analog input also requires that the corresponding direction be set. If the direction is set to output, the digital output level (VOH or VOL) will be outputted.
2. When reading the port register, all pins configured as analog input channels will read as cleared.
3. Pins configured as digital inputs will not output an analog input. Analog levels on any pin that is defined as a digital input may cause the input buffer to consume current that exceeds the device specifications.
4. Pull-ups and pull-downs on change notification pins should always be disabled when the port pin is configured as a digital output.

Considerations for the Peripheral Pin Select:

The ability to control Peripheral Pin Select options introduces several considerations into application design that could be overlooked. This is particularly true for several common peripherals that are available only as remappable peripherals.

The main consideration is that the Peripheral Pin Selects are not available on default pins in the device's default (Reset) state, all Peripheral Pin Select inputs are tied to Vss and all Peripheral Pin Select outputs are disconnected. This situation requires the user to initialize the device with the proper peripheral configuration before any other application code is executed. For application safety, however, it is best to lock the configuration after writing to the control registers.

A final consideration is that Peripheral Pin Select functions neither override analog inputs, nor reconfigure pins with analog functions for digital I/O. If a pin is configured as an analog input on device Reset, it must be explicitly reconfigured as digital I/O when used with a Peripheral Pin Select.



Note: Not all features are available on all devices. Please refer to the specific device data sheet to determine availability.

Configuring the Library

The configuration of the Ports System Service is based on the file `system_config.h`.

This header file contains the configuration selection for the Ports System Service. Based on the selections made, the Ports System Service may support the selected features. These configuration settings will apply to all instances of the Ports System Service.

This header can be placed anywhere, the path of this header needs to be present in the include search path for a successful build. Refer to the Applications Help section for more details.

Building the Library

This section lists the files that are available in the Ports System Service Library.

Description

The following three tables list and describe the header (.h) and source (.c) files that implement this library. The parent folder for these files is `<install-dir>/framework/system/ports`.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using `#include`) by any code that uses this library.

Source File Name	Description
<code>sys_ports.h</code>	Ports System Service Library API header file.

Required File(s)



All of the required files listed in the following table are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

Source File Name	Description
<code>/src/sys_ports.c</code>	Ports System Service Library implementation.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
N/A	There are no optional files for this library.

Module Dependencies





The Reset System Service Library depends on the following modules:

- Ports Peripheral Library
- Device Control Peripheral Library











Library Interface

a) Pin Control Functions









	Name	Description
	<code>SYS_PORTS_PinModeSelect</code>	Enables the selected pin as analog or digital. Implementation: Dynamic
	<code>SYS_PORTS_PinOpenDrainDisable</code>	Disables the open-drain functionality for the selected pin. Implementation: Dynamic
	<code>SYS_PORTS_PinOpenDrainEnable</code>	Enables the open-drain functionality for the selected pin. Implementation: Dynamic
	<code>SYS_PORTS_PinRead</code>	Reads the selected digital pin. Implementation: Dynamic
	<code>SYS_PORTS_PinSet</code>	Sets the selected digital pin/latch. Implementation: Dynamic

	SYS_PORTS_PinToggle	Toggles the selected digital pin. Implementation: Dynamic
	SYS_PORTS_PinWrite	Writes the selected digital pin. Implementation: Dynamic
	SYS_PORTS_PinClear	Clears the selected digital pin. Implementation: Dynamic
	SYS_PORTS_PinDirectionSelect	Enables the direction for the selected pin. Implementation: Dynamic



b) Ports Control Functions

	Name	Description
	SYS_PORTS_Clear	Clears the selected digital port. Implementation: Dynamic
	SYS_PORTS_DirectionGet	Reads the direction for the selected port. Implementation: Dynamic
	SYS_PORTS_DirectionSelect	Enables the direction for the selected port. Implementation: Dynamic
	SYS_PORTS_OpenDrainDisable	Disables the open-drain functionality for the selected port. Implementation: Dynamic
	SYS_PORTS_OpenDrainEnable	Enables the open-drain functionality for the selected port. Implementation: Dynamic
	SYS_PORTS_Read	Reads the data from the I/O port. Implementation: Dynamic
	SYS_PORTS_Set	Sets the selected digital port/latch based on the mask. Implementation: Dynamic
	SYS_PORTS_Toggle	Toggles the selected digital port pins. Implementation: Dynamic
	SYS_PORTS_Write	Writes the data to the I/O port. Implementation: Dynamic
	SYS_PORTS_Initialize	Initializes PORT Pins/Channels. Implementation: Static/Dynamic

c) Change Notification Functions

	Name	Description
	SYS_PORTS_ChangeNotificationDisable	Disables the change notification for the selected port. Implementation: Dynamic
	SYS_PORTS_ChangeNotificationEnable	Enables the change notification for the selected port. Implementation: Dynamic
	SYS_PORTS_ChangeNotificationGlobalDisable	Globally disables the change notification for the selected port. Implementation: Dynamic
	SYS_PORTS_ChangeNotificationGlobalEnable	Globally enables the change notification for the selected port. Implementation: Dynamic
	SYS_PORTS_ChangeNotificationIdleModeDisable	Disables the change notification for the selected port in Sleep or Idle mode. Implementation: Dynamic
	SYS_PORTS_ChangeNotificationIdleModeEnable	Enables the change notification for the selected port in Sleep or Idle mode. Implementation: Dynamic
	SYS_PORTS_ChangeNotificationPullUpDisable	Disables a weak pull-up on the change notification pin. Implementation: Dynamic
	SYS_PORTS_ChangeNotificationPullUpEnable	Enables a weak pull-up on the change notification pin. Implementation: Dynamic

d) Peripheral Pin Select Functions

	Name	Description
	SYS_PORTS_RemapInput	Input/Output (I/O) function remapping. Implementation: Dynamic
	SYS_PORTS_RemapOutput	Input/Output (I/O) function remapping. Implementation: Dynamic

e) Data Types and Constants

	Name	Description
	SYS_PORTS_PIN_DIRECTION	Defines the direction of the port pins.
	SYS_PORTS_PULLUP_PULLDOWN_STATUS	Provides the pull-up and pull-down status.

Description

This section describes the APIs of the Ports System Service Library.

Refer to each section for a detailed description.

a) Pin Control Functions

SYS_PORTS_PinModeSelect Function

Enables the selected pin as analog or digital.

Implementation: Dynamic

File

[sys_ports.h](#)

C

```
void SYS_PORTS_PinModeSelect(PORTS_MODULE_ID index, PORTS_ANALOG_PIN pin, PORTS_PIN_MODE mode);
```

Returns

None.

Description

This function enables the selected pin as analog or digital.

Remarks

Not all features are available on all devices. Refer to the specific device data sheet for availability.

Preconditions

None.

Example

```
// Where MY_PORTS_INSTANCE, is the ports instance selected for use by the
// application developer.
// MY_PIN      - PORTS_ANALOG_PIN_AN0
// MY_PIN_MODE - PORTS_PIN_MODE_ANALOG
SYS_PORTS_PinModSYS_PORTS_PinModeSelecteSelect( MY_PORTS_INSTANCE, MY_PIN, MY_PIN_MODE );
```

Parameters

Parameters	Description
index	Identifier for the device instance to be configured
pin	Possible values of PORTS_ANALOG_PIN
mode	Possible values of PORTS_PIN_MODE

Function

```
void SYS_PORTS_PinModeSelect ( PORTS_MODULE_ID index,
PORTS_ANALOG_PIN pin,
PORTS_PIN_MODE mode)
```

SYS_PORTS_PinOpenDrainDisable Function

Disables the open-drain functionality for the selected pin.

Implementation: Dynamic

File

[sys_ports.h](#)

C

```
void SYS_PORTS_PinOpenDrainDisable(PORTS_MODULE_ID index, PORTS_CHANNEL channel, PORTS_BIT_POS bitPos);
```

Returns

None.

Description

This function disables the open-drain functionality for the selected pin.

Remarks

Not all features are available on all devices. Refer to the specific device data sheet for availability.

Preconditions

None.

Example

```
// Where MY_PORTS_INSTANCE, is the ports instance selected for use by the
// application developer.
// MY_PINNUM - PORTS_PIN_10
SYS_PORTS_PinOpenDrainDisable(MY_PORTS_INSTANCE, MY_PINNUM);
```

Parameters

Parameters	Description
index	Identifier for the device instance to be configured
channel	Identifier for the PORT channel: A, B, C, etc.
bitPos	Possible values of PORTS_BIT_POS

Function

```
void SYS_PORTS_PinOpenDrainDisable ( PORTS_MODULE_ID index,
PORTS_CHANNEL channel,
PORTS_BIT_POS bitPos )
```

SYS_PORTS_PinOpenDrainEnable Function

Enables the open-drain functionality for the selected pin.

Implementation: Dynamic

File

[sys_ports.h](#)

C

```
void SYS_PORTS_PinOpenDrainEnable(PORTS_MODULE_ID index, PORTS_CHANNEL channel, PORTS_BIT_POS bitPos);
```

Returns

None.

Description

This function enables the open-drain functionality for the selected pin.

Remarks

Not all features are available on all devices. Refer to the specific device data sheet for availability.

Preconditions

None.

Example

```
// Where MY_PORTS_INSTANCE, is the ports instance selected for use by the
// application developer.
```

```
// MY_PINNUM - PORTS_PIN_10
SYS_PORTS_PinOpenDrainEnable( MY_PORTS_INSTANCE, MY_CHANNEL, MY_PINNUM );
```

Parameters

Parameters	Description
index	Identifier for the device instance to be configured
channel	Identifier for the PORT channel: A, B, C, etc.
bitPos	Possible values of PORTS_BIT_POS

Function

```
void SYS_PORTS_PinOpenDrainEnable ( PORTS_MODULE_ID index,
PORTS_CHANNEL channel,
PORTS_BIT_POS bitPos )
```

SYS_PORTS_PinRead Function

Reads the selected digital pin.

Implementation: Dynamic

File

[sys_ports.h](#)

C

```
bool SYS_PORTS_PinRead(PORTS_MODULE_ID index, PORTS_CHANNEL channel, PORTS_BIT_POS bitPos);
```

Returns

The status of the port pin.

Description

This function reads the selected digital pin, not the Latch.

Remarks

Not all features are available on all devices. Refer to the specific device data sheet for availability.

Preconditions

None.

Example

```
// Where MY_PORTS_INSTANCE, is the ports instance selected for use by the
// application developer.
// MY_PINNUM - PORTS_PIN_10
bool bitStatus = SYS_PORTS_PinRead(MY_PORTS_INSTANCE, MY_CHANNEL, MY_PINNUM);
```

Parameters

Parameters	Description
index	Identifier for the device instance to be configured
channel	Identifier for the PORT channel: A, B, C, etc.
bitPos	Possible values of PORTS_BIT_POS

Function

```
bool SYS_PORTS_PinRead ( PORTS_MODULE_ID index,
PORTS_CHANNEL channel,
PORTS_BIT_POS bitPos )
```

SYS_PORTS_PinSet Function

Sets the selected digital pin/latch.

Implementation: Dynamic

File

[sys_ports.h](#)

C

```
void SYS_PORTS_PinSet(PORTS_MODULE_ID index, PORTS_CHANNEL channel, PORTS_BIT_POS bitPos);
```

Returns

None.

Description

This function sets the selected digital pin/latch.

Remarks

Not all features are available on all devices. Refer to the specific device data sheet for availability.

Preconditions

None.

Example

```
// Where MY_PORTS_INSTANCE, is the ports instance selected for use by the
// application developer.
// MY_PINNUM - PORTS_PIN_10
SYS_PORTS_PinSet(MY_PORTS_INSTANCE, MY_CHANNEL, MY_PINNUM);
```

Parameters

Parameters	Description
index	Identifier for the device instance to be configured
channel	Identifier for the PORT channel: A, B, C, etc.
bitPos	Possible values of PORTS_BIT_POS

Function

```
void SYS_PORTS_PinSet( PORTS_MODULE_ID index,
PORTS_CHANNEL channel,
PORTS_BIT_POS bitPos )
```

SYS_PORTS_PinToggle Function

Toggles the selected digital pin.

Implementation: Dynamic

File

[sys_ports.h](#)

C

```
void SYS_PORTS_PinToggle(PORTS_MODULE_ID index, PORTS_CHANNEL channel, PORTS_BIT_POS bitPos);
```

Returns

None.

Description

This function toggles the selected digital pin.

Remarks

Not all features are available on all devices. Refer to the specific device data sheet for availability.

Preconditions

None.

Example

```
// Where MY_PORTS_INSTANCE, is the ports instance selected for use by the
// application developer.
```

```
// MY_PINNUM - PORTS_PIN_10
SYS_PORTS_PinToggle( MY_PORTS_INSTANCE, MY_CHANNEL, MY_PINNUM );
```

Parameters

Parameters	Description
index	Identifier for the device instance to be configured
channel	Identifier for the PORT channel: A, B, C, etc.
bitPos	Possible values of PORTS_BIT_POS

Function

```
void SYS_PORTS_PinToggle ( PORTS_MODULE_ID index,
PORTS_CHANNEL channel,
PORTS_BIT_POS bitPos )
```

SYS_PORTS_PinWrite Function

Writes the selected digital pin.

Implementation: Dynamic

File

[sys_ports.h](#)

C

```
void SYS_PORTS_PinWrite(PORTS_MODULE_ID index, PORTS_CHANNEL channel, PORTS_BIT_POS bitPos, bool value);
```

Returns

None.

Description

This function writes the selected digital pin.

Remarks

Not all features are available on all devices. Refer to the specific device data sheet for availability.

Preconditions

None.

Example

```
// Where MY_PORTS_INSTANCE, is the ports instance selected for use by the
// application developer.
// MY_PINNUM - PORTS_PIN_10
SYS_PORTS_PinWrite(MY_PORTS_INSTANCE, MY_CHANNEL, MY_PINNUM, value);
```

Parameters

Parameters	Description
index	Identifier for the device instance to be configured
channel	Identifier for the PORT channel: A, B, C, etc.
bitPos	Possible values of PORTS_BIT_POS
value	Value to be written to the specific pin/latch: <ul style="list-style-type: none">true - Sets the bitfalse - Clears the bit

Function

```
void SYS_PORTS_PinWrite ( PORTS_MODULE_ID index,
PORTS_CHANNEL channel,
PORTS_BIT_POS bitPos
bool value )
```

SYS_PORTS_PinClear Function

Clears the selected digital pin.

Implementation: Dynamic

File

[sys_ports.h](#)

C

```
void SYS_PORTS_PinClear(PORTS_MODULE_ID index, PORTS_CHANNEL channel, PORTS_BIT_POS bitPos);
```

Returns

None.

Description

This function clears the selected digital pin.

Remarks

Not all features are available on all devices. Refer to the specific device data sheet for availability.

Preconditions

None.

Example

```
// Where MY_PORTS_INSTANCE, is the ports instance selected for use by the
// application developer.
// MY_PINNUM - PORTS_IO_PIN_10
SYS_PORTS_PinClear( MY_PORTS_INSTANCE, MY_CHANNEL, MY_PINNUM );
```

Parameters

Parameters	Description
index	Identifier for the device instance to be configured
channel	Identifier for the PORT channel: A, B, C, etc.
bitPos	Possible values of PORTS_BIT_POS

Function

```
void SYS_PORTS_PinClear ( PORTS_MODULE_ID index,
PORTS_CHANNEL channel,
PORTS_BIT_POS bitPos )
```

SYS_PORTS_PinDirectionSelect Function

Enables the direction for the selected pin.

Implementation: Dynamic

File

[sys_ports.h](#)

C

```
void SYS_PORTS_PinDirectionSelect(PORTS_MODULE_ID index, SYS_PORTS_PIN_DIRECTION pinDir, PORTS_CHANNEL
channel, PORTS_BIT_POS bitPos);
```

Returns

None.

Description

This function enables the direction for the selected pin.

Remarks

Not all features are available on all devices. Refer to the specific device data sheet for availability.

Preconditions

None.

Example

```
// Where MY_PORTS_INSTANCE, is the ports instance selected for use by the
// application developer.
// MY_PINNUM - PORTS_PIN_10
SYS_PORTS_PIN_DIRECTION pinDir;
pinDir = SYS_PORTS_DIRECTION_INPUT;
SYS_PORTS_PinDirectionSelect(MY_PORTS_INSTANCE, pinDir, MY_CHANNEL, MY_PINNUM);
```

Parameters

Parameters	Description
index	Identifier for the device instance to be configured
pinDir	Pin direction
channel	Identifier for the PORT channel: A, B, C, etc.
bitPos	Possible values of PORTS_BIT_POS

Function

```
void SYS_PORTS_PinDirectionSelect ( PORTS_MODULE_ID index,
                                   SYS_PORTS_PIN_DIRECTION pinDir,
                                   PORTS_CHANNEL channel,
                                   PORTS_BIT_POS bitPos )
```

b) Ports Control Functions

SYS_PORTS_Clear Function

Clears the selected digital port.

Implementation: Dynamic

File

[sys_ports.h](#)

C

```
void SYS_PORTS_Clear(PORTS_MODULE_ID index, PORTS_CHANNEL channel, PORTS_DATA_MASK clearMask);
```

Returns

None.

Description

This function clears the selected digital port.

Remarks

Not all features are available on all devices. Refer to the specific device data sheet for availability.

Preconditions

None.

Example

```
// Where MY_PORTS_INSTANCE, is the ports instance selected for use by the
// application developer.
PORTS_DATA_MASK clearMask = (PORTS_DATA_MASK)0x00FF;
SYS_PORTS_Clear( MY_PORTS_INSTANCE, MY_CHANNEL, clearMask );
```

Parameters

Parameters	Description
index	Identifier for the device instance to be configured
channel	Identifier for the PORT channel: A, B, C, etc.

clearMask	Identifies the bits to be cleared
-----------	-----------------------------------

Function

```
void SYS_PORTS_Clear ( PORTS_MODULE_ID index, PORTS_CHANNEL channel,  
PORTS_DATA_MASK clearMask )
```

SYS_PORTS_DirectionGet Function

Reads the direction for the selected port.

Implementation: Dynamic

File

[sys_ports.h](#)

C

```
PORTS_DATA_MASK SYS_PORTS_DirectionGet( PORTS_MODULE_ID index, PORTS_CHANNEL channel );
```

Returns

Direction of the port.

Description

This function reads the direction for the selected port.

Remarks

None.

Preconditions

None.

Example

```
// Where MY_PORTS_INSTANCE, is the ports instance selected for use by the  
// application developer.  
PORTS_DATA_MASK value;  
value = SYS_PORTS_DirectionGet( MY_PORTS_INSTANCE, MY_CHANNEL );
```

Parameters

Parameters	Description
index	Identifier for the device instance to be configured
channel	Identifier for the PORT channel: A, B, C, etc.

Function

```
PORTS_DATA_MASK SYS_PORTS_DirectionGet( PORTS_MODULE_ID index, PORTS_CHANNEL channel )
```

SYS_PORTS_DirectionSelect Function

Enables the direction for the selected port.

Implementation: Dynamic

File

[sys_ports.h](#)

C

```
void SYS_PORTS_DirectionSelect( PORTS_MODULE_ID index, SYS_PORTS_PIN_DIRECTION pinDir, PORTS_CHANNEL  
channel, PORTS_DATA_MASK mask );
```

Returns

None

Description

This function enables the direction for the selected port.

Remarks

None.

Preconditions

None.

Example

```
// Where MY_PORTS_INSTANCE, is the ports instance selected for use by the
// application developer.
SYS_PORTS_PIN_DIRECTION pinDir;
pinDir = SYS_PORTS_DIRECTION_INPUT;
PORTS_DATA_MASK myMask = (PORTS_DATA_MASK)0x00FF;
SYS_PORTS_DirectionSelect(MY_PORTS_INSTANCE, pinDir, MY_CHANNEL, myMask );
```

Parameters

Parameters	Description
index	Identifier for the device instance to be configured
pinDir	Pin direction
channel	Identifier for the PORT channel: A, B, C, etc.
mask	Mask for the direction of width PORTS_DATA_MASK

Function

```
void SYS_PORTS_DirectionSelect( PORTS_MODULE_ID index,
                               SYS_PORTS_PIN_DIRECTION pinDir,
                               PORTS_CHANNEL channel,
                               PORTS_DATA_MASK mask )
```

SYS_PORTS_OpenDrainDisable Function

Disables the open-drain functionality for the selected port.

Implementation: Dynamic

File

[sys_ports.h](#)

C

```
void SYS_PORTS_OpenDrainDisable(PORTS_MODULE_ID index, PORTS_CHANNEL channel, PORTS_DATA_MASK mask);
```

Returns

None.

Description

This function disables the open-drain functionality for the selected port.

Remarks

Not all features are available on all devices. Refer to the specific device data sheet for availability.

Preconditions

None.

Example

```
// Where MY_PORTS_INSTANCE, is the ports instance selected for use by the
// application developer.
SYS_PORTS_PeripheralOpenDrainDisable( MY_PORTS_INSTANCE, MY_CHANNEL, (PORTS_DATA_MASK)0x00FF );
```

Parameters

Parameters	Description
index	Identifier for the device instance to be configured
channel	Identifier for the PORT channel: A, B, C, etc.
mask	Mask of type PORTS_DATA_MASK

Function

```
void SYS_PORTS_OpenDrainDisable( PORTS_MODULE_ID index, PORTS_CHANNEL channel,  
PORTS_DATA_MASK mask )
```

SYS_PORTS_OpenDrainEnable Function

Enables the open-drain functionality for the selected port.

Implementation: Dynamic

File

[sys_ports.h](#)

C

```
void SYS_PORTS_OpenDrainEnable(PORTS_MODULE_ID index, PORTS_CHANNEL channel, PORTS_DATA_MASK mask);
```

Returns

None.

Description

This function enables the open-drain functionality for the selected port.

Remarks

Not all features are available on all devices. Refer to the specific device data sheet for availability.

Preconditions

None.

Example

```
// Where MY_PORTS_INSTANCE, is the ports instance selected for use by the  
// application developer.  
SYS_PORTS_OpenDrainEnable( MY_PORTS_INSTANCE, MY_CHANNEL, (PORTS_DATA_MASK)0x00FF );
```

Parameters

Parameters	Description
index	Identifier for the device instance to be configured
channel	Identifier for the PORT channel: A, B, C, etc.
mask	Mask of type PORTS_DATA_MASK

Function

```
void SYS_PORTS_OpenDrainEnable( PORTS_MODULE_ID index, PORTS_CHANNEL channel,  
PORTS_DATA_MASK mask )
```

SYS_PORTS_Read Function

Reads the data from the I/O port.

Implementation: Dynamic

File

[sys_ports.h](#)

C

```
PORTS_DATA_TYPE SYS_PORTS_Read(PORTS_MODULE_ID index, PORTS_CHANNEL channel);
```

Returns

Returns the data read from the port.

Description

This function reads the data from the I/O port.

Remarks

None.

Preconditions

The direction of the port to be set as input.

Example

```
// Where MY_PORTS_INSTANCE, is the ports instance selected for use by the
// application developer.
PORTS_DATA_TYPE readData;
readData = SYS_PORTS_Read( MY_PORTS_INSTANCE, MY_CHANNEL );
```

Parameters

Parameters	Description
index	Identifier for the device instance to be configured
channel	Identifier for the PORT channel: A, B, C, etc.

Function

PORTS_DATA_TYPE SYS_PORTS_Read(PORTS_MODULE_ID index, PORTS_CHANNEL channel)

SYS_PORTS_Set Function

Sets the selected digital port/latch based on the mask.

Implementation: Dynamic

File

[sys_ports.h](#)

C

```
void SYS_PORTS_Set( PORTS_MODULE_ID index, PORTS_CHANNEL channel, PORTS_DATA_TYPE value, PORTS_DATA_MASK
mask );
```

Returns

None.

Description

This function Sets the selected digital port/latch relative to the mask.

This function "AND" value and mask parameters and then set the bits in the port channel that were set in the result of the ANDing operation.

Remarks

Not all features are available on all devices. Refer to the specific device data sheet for availability.

Preconditions

None.

Example

```
// Where MY_PORTS_INSTANCE, is the ports instance selected for use by the
// application developer.
// MY_VALUE - 0x1234
PORTS_DATA_MASK myMask = ( PORTS_DATA_MASK ) 0x00FF;

// Set the MY_CHANNEL bit positions 2,4 and 5 (0x0034 = b0000 0000 0011 0100)
SYS_PORTS_Set( MY_PORTS_INSTANCE, MY_CHANNEL, MY_VALUE, myMask );
```

Parameters

Parameters	Description
index	Identifier for the device instance to be configured
channel	Identifier for the PORT channel: A, B, C, etc.
value	Consists of information about which port bit has to be set
mask	Identifies the bits which could be intended for setting

Function

```
void SYS_PORTS_Set( PORTS_MODULE_ID index, PORTS_CHANNEL channel,  
PORTS_DATA_TYPE value,  
PORTS_DATA_MASK mask )
```

SYS_PORTS_Toggle Function

Toggles the selected digital port pins.

Implementation: Dynamic

File

[sys_ports.h](#)

C

```
void SYS_PORTS_Toggle( PORTS_MODULE_ID index, PORTS_CHANNEL channel, PORTS_DATA_MASK toggleMask );
```

Returns

None.

Description

This function toggles the selected digital port pins.

Remarks

None.

Preconditions

None.

Example

```
// Where MY_PORTS_INSTANCE, is the ports instance selected for use by the  
// application developer.  
PORTS_DATA_MASK toggleMask = (PORTS_DATA_MASK)0x00FF;  
SYS_PORTS_Toggle( MY_PORTS_INSTANCE, MY_CHANNEL, toggleMask );
```

Parameters

Parameters	Description
index	Identifier for the device instance to be configured
channel	Identifier for the PORT channel: A, B, C, etc.
toggleMask	Identifies the bits to be toggled

Function

```
void SYS_PORTS_Toggle( PORTS_MODULE_ID index, PORTS_CHANNEL channel,  
PORTS_DATA_MASK toggleMask )
```

SYS_PORTS_Write Function

Writes the data to the I/O port.

Implementation: Dynamic

File

[sys_ports.h](#)

C

```
void SYS_PORTS_Write( PORTS_MODULE_ID index, PORTS_CHANNEL channel, PORTS_DATA_TYPE value );
```

Returns

None.

Description

This function writes the data to the I/O port.

Remarks

None.

Preconditions

The direction of the port to be set as output.

Example

```
// Where MY_PORTS_INSTANCE, is the ports instance selected for use by the
// application developer.
PORTS_DATA_TYPE writeData;
SYS_PORTS_Write( MY_PORTS_INSTANCE, MY_CHANNEL, MY_VALUE );
```

Parameters

Parameters	Description
index	Identifier for the device instance to be configured
channel	Identifier for the PORT channel: A, B, C, etc.
value	Value to be written into a port of width PORTS_DATA_TYPE

Function

```
void SYS_PORTS_Write( PORTS_MODULE_ID index,
PORTS_CHANNEL channel,
PORTS_DATA_TYPE value )
```

SYS_PORTS_Initialize Function

Initializes PORT Pins/Channels.

Implementation: Static/Dynamic

File

[sys_ports.h](#)

C

```
void SYS_PORTS_Initialize();
```

Returns

None.

Description

This function initializes different port pins/channels to the desired state. It also remaps the pins to the desired specific function.

Remarks

This API must be called at the time of system initialization to initialize the ports pins.

Preconditions

None.

Example

```
SYS_PORTS_Initialize();
```

Function

```
void SYS_PORTS_Initialize()
```

c) Change Notification Functions

SYS_PORTS_ChangeNotificationDisable Function

Disables the change notification for the selected port.

Implementation: Dynamic

File

[sys_ports.h](#)

C

```
void SYS_PORTS_ChangeNotificationDisable(PORTS_MODULE_ID index, PORTS_CHANGE_NOTICE_PIN pinNum);
```

Returns

None.

Description

This function disables the change notification for the selected port.

Remarks

Not all features are available on all devices. Refer to the specific device data sheet for availability.

Preconditions

None.

Example

```
// Where MY_PORTS_INSTANCE, is the ports instance selected for use by the
// application developer.
PORTS_CHANGE_NOTICE_PIN pinNum;
SYS_PORTS_ChangeNotificationDisable( index, pinNum );
```

Parameters

Parameters	Description
index	Identifier for the device instance to be configured
pinNum	Possible values of PORTS_CHANGE_NOTICE_PIN

Function

```
void SYS_PORTS_ChangeNotificationDisable( PORTS_MODULE_ID index,
PORTS_CHANGE_NOTICE_PIN pinNum )
```

SYS_PORTS_ChangeNotificationEnable Function

Enables the change notification for the selected port.

Implementation: Dynamic

File

[sys_ports.h](#)

C

```
void SYS_PORTS_ChangeNotificationEnable(PORTS_MODULE_ID index, PORTS_CHANGE_NOTICE_PIN pinNum,
SYS_PORTS_PULLUP_PULLDOWN_STATUS value);
```

Returns

None.

Description

This function enables the change notification for the selected port.

Remarks

Not all features are available on all devices. Refer to the specific device data sheet for availability.

Preconditions

None.

Example

```
// Where MY_PORTS_INSTANCE, is the ports instance selected for use by the
// application developer.
SYS_PORTS_PULLUP_PULLDOWN_STATUS value;
```

```
PORTS_CHANGE_NOTICE_PIN pinNum;  
SYS_PORTS_ChangeNotificationEnable( index, pinNum, value );
```

Parameters

Parameters	Description
index	Identifier for the device instance to be configured
value	Pull-up enable or disable value
pinNum	Possible values of PORTS_CHANGE_NOTICE_PIN

Function

```
void SYS_PORTS_ChangeNotificationEnable( PORTS_MODULE_ID index,  
PORTS_CHANGE_NOTICE_PIN pinNum,  
SYS_PORTS_PULLUP_PULLDOWN_STATUS value )
```

SYS_PORTS_ChangeNotificationGlobalDisable Function

Globally disables the change notification for the selected port.

Implementation: Dynamic

File

[sys_ports.h](#)

C

```
void SYS_PORTS_ChangeNotificationGlobalDisable( PORTS_MODULE_ID index );
```

Returns

None.

Description

This function globally disables the change notification for the selected port.

Remarks

Not all features are available on all devices. Refer to the specific device data sheet for availability.

Preconditions

None.

Example

```
// Where MY_PORTS_INSTANCE, is the ports instance selected for use by the  
// application developer.  
SYS_PORTS_ChangeNotificationGlobalDisable( MY_PORTS_INSTANCE );
```

Function

```
void SYS_PORTS_ChangeNotificationGlobalDisable( PORTS_MODULE_ID index );
```

SYS_PORTS_ChangeNotificationGlobalEnable Function

Globally enables the change notification for the selected port.

Implementation: Dynamic

File

[sys_ports.h](#)

C

```
void SYS_PORTS_ChangeNotificationGlobalEnable( PORTS_MODULE_ID index );
```

Returns

None.

Description

This function globally enables the change notification for the selected port.

Remarks

Not all features are available on all devices. Refer to the specific device data sheet for availability.

Preconditions

None.

Example

```
// Where MY_PORTS_INSTANCE, is the ports instance selected for use by the
// application developer.
SYS_PORTS_ChangeNotificationGlobalEnable( MY_PORTS_INSTANCE );
```

Function

void SYS_PORTS_ChangeNotificationGlobalEnable(PORTS_MODULE_ID index)

SYS_PORTS_ChangeNotificationInIdleModeDisable Function

Disables the change notification for the selected port in Sleep or Idle mode.

Implementation: Dynamic

File

sys_ports.h

C

```
void SYS_PORTS_ChangeNotificationInIdleModeDisable( PORTS_MODULE_ID index );
```

Returns

None.

Description

This function disables the change notification for the selected port in Sleep or Idle mode.

Remarks

Not all features are available on all devices. Refer to the specific device data sheet for availability.

Preconditions

None.

Example

```
// Where MY_PORTS_INSTANCE, is the ports instance selected for use by the
// application developer.
SYS_PORTS_ChangeNotificationInIdleModeDisable( MY_PORTS_INSTANCE );
```

Function

void SYS_PORTS_ChangeNotificationInIdleModeDisable(PORTS_MODULE_ID index);

SYS_PORTS_ChangeNotificationInIdleModeEnable Function

Enables the change notification for the selected port in Sleep or Idle mode.

Implementation: Dynamic

File

sys_ports.h

C

```
void SYS_PORTS_ChangeNotificationInIdleModeEnable( PORTS_MODULE_ID index );
```

Returns

None.

Description

This function enables the change notification for the selected port in Sleep or Idle mode.

Remarks

Not all features are available on all devices. Refer to the specific device data sheet for availability.

Preconditions

None.

Example

```
// Where MY_PORTS_INSTANCE, is the ports instance selected for use by the
// application developer.
SYS_PORTS_ChangeNotificationInIdleModeEnable( MY_PORTS_INSTANCE );
```

Function

```
void SYS_PORTS_ChangeNotificationInIdleModeEnable( PORTS_MODULE_ID index );
```

SYS_PORTS_ChangeNotificationPullUpDisable Function

Disables a weak pull-up on the change notification pin.

Implementation: Dynamic

File

[sys_ports.h](#)

C

```
void SYS_PORTS_ChangeNotificationPullUpDisable(PORTS_MODULE_ID index, PORTS_CHANGE_NOTICE_PIN pinNum);
```

Returns

None.

Description

This function Disables a weak pull-up on the change notification pin.

Remarks

Not all features are available on all devices. Refer to the specific device data sheet for availability.

Preconditions

None.

Example

```
// Where MY_PORTS_INSTANCE, is the ports instance selected for use by the
// application developer.
// MY_PINNUM - PORTS_CHANGE_NOTICE_PIN_10
SYS_PORTS_ChangeNotificationPullUpDisable( MY_PORTS_INSTANCE, MY_PINNUM );
```

Parameters

Parameters	Description
index	Identifier for the device instance to be configured
pinNum	Possible values of PORTS_CHANGE_NOTICE_PIN

Function

```
void SYS_PORTS_ChangeNotificationPullUpDisable ( PORTS_MODULE_ID index,
PORTS_CHANGE_NOTICE_PIN pinNum )
```

SYS_PORTS_ChangeNotificationPullUpEnable Function

Enables a weak pull-up on the change notification pin.

Implementation: Dynamic

File

[sys_ports.h](#)

C

```
void SYS_PORTS_ChangeNotificationPullUpEnable(PORTS_MODULE_ID index, PORTS_CHANGE_NOTICE_PIN pinNum);
```

Returns

None.

Description

This function enables a weak pull-up on the change notification pin.

Remarks

Not all features are available on all devices. Refer to the specific device data sheet for availability.

Preconditions

None.

Example

```
// Where MY_PORTS_INSTANCE, is the ports instance selected for use by the
// application developer.
// MY_PINNUM - PORTS_CHANGE_NOTICE_PIN_10
SYS_PORTS_ChangeNotificationPullUpEnable( MY_PORTS_INSTANCE, MY_PINNUM );
```

Parameters

Parameters	Description
index	Identifier for the device instance to be configured
pinNum	Possible values of PORTS_CHANGE_NOTICE_PIN

Function

```
void SYS_PORTS_ChangeNotificationPullUpEnable ( PORTS_MODULE_ID    index,
PORTS_CHANGE_NOTICE_PIN pinNum )
```

d) Peripheral Pin Select Functions

SYS_PORTS_RemapInput Function

Input/Output (I/O) function remapping.

Implementation: Dynamic

File

[sys_ports.h](#)

C

```
void SYS_PORTS_RemapInput(PORTS_MODULE_ID index, PORTS_REMAP_INPUT_FUNCTION function, PORTS_REMAP_INPUT_PIN
remapPin);
```

Returns

None.

Description

This function controls the I/O function remapping.

Remarks

This feature may not be available on all devices. Please refer to the specific device data sheet to determine availability or use SYS_PORTS_ExistsRemapInputOutput in your application to determine whether this feature is available.

Preconditions

None.

Example

```
// Where MY_PORTS_INSTANCE, is the ports instance selected for use by the
// application developer.
```

```
// Remapping input function 'Input Capture 1' to the Remappable pin 'RPD2'  
SYS_PORTS_RemapInput(MY_PORTS_INSTANCE, INPUT_FUNC_IC1, INPUT_PIN_RPD2 );
```

Parameters

Parameters	Description
index	Identifier for the device instance to be configured

Function

```
void SYS_PORTS_RemapInput( PORTS_MODULE_ID    index,  
PORTS_REMAP_INPUT_FUNCTION function,  
PORTS_REMAP_INPUT_PIN    remapPin )  
One of the possible values of PORTS_REMAP_INPUT_FUNCTION  
remapPin    - One of the possible values of PORTS_REMAP_INPUT_PIN
```

SYS_PORTS_RemapOutput Function

Input/Output (I/O) function remapping.

Implementation: Dynamic

File

[sys_ports.h](#)

C

```
void SYS_PORTS_RemapOutput( PORTS_MODULE_ID index, PORTS_REMAP_OUTPUT_FUNCTION function,  
PORTS_REMAP_OUTPUT_PIN remapPin );
```

Returns

None.

Description

This function controls the I/O function remapping.

Remarks

This feature may not be available on all devices. Please refer to the specific device data sheet to determine availability or use SYS_PORTS_ExistsRemapInputOutput in your application to determine whether this feature is available.

Preconditions

None.

Example

```
// Where MY_PORTS_INSTANCE, is the ports instance selected for use by the  
// application developer.  
// Remapping output function 'UART3 Transmit' to the Remappable pin 'RPA14'  
SYS_PORTS_RemapInputOutput(MY_PORTS_INSTANCE, OUTPUT_FUNC_U3TX, OUTPUT_PIN_RPA14);
```

Parameters

Parameters	Description
index	Identifier for the device instance to be configured

Function

```
void SYS_PORTS_RemapOutput( PORTS_MODULE_ID    index,  
PORTS_REMAP_OUTPUT_FUNCTION function,  
PORTS_REMAP_OUTPUT_PIN    remapPin )  
One of the possible values of PORTS_REMAP_OUTPUT_FUNCTION  
remapPin    - One of the possible values of PORTS_REMAP_OUTPUT_PIN
```

e) Data Types and Constants

SYS_PORTS_PIN_DIRECTION Enumeration

Defines the direction of the port pins.

File

[sys_ports.h](#)

C

```
typedef enum {  
    SYS_PORTS_DIRECTION_OUTPUT,  
    SYS_PORTS_DIRECTION_INPUT  
} SYS_PORTS_PIN_DIRECTION;
```

Members

Members	Description
SYS_PORTS_DIRECTION_OUTPUT	Direction as output
SYS_PORTS_DIRECTION_INPUT	Direction as input

Description

SYS_PORTS_PIN_DIRECTION

These constants provide the port pin direction definitions.

Remarks

None.

SYS_PORTS_PULLUP_PULLDOWN_STATUS Enumeration

Provides the pull-up and pull-down status.

File

[sys_ports.h](#)

C

```
typedef enum {  
    SYS_PORTS_PULLUP_DISABLE,  
    SYS_PORTS_PULLUP_ENABLE  
} SYS_PORTS_PULLUP_PULLDOWN_STATUS;
```

Members

Members	Description
SYS_PORTS_PULLUP_DISABLE	PULLUP Disable
SYS_PORTS_PULLUP_ENABLE	PULLUP Enable

Description

SYS_PORTS_PULLUP status

These constants provide the pull-up or pull-down status definitions.

Remarks

None.

Files

Files

Name	Description
sys_ports.h	Ports System Service interface definitions

Description

This section lists the source and header files used by the library.


sys_ports.h







Ports System Service interface definitions

Enumerations

	Name	Description
	SYS_PORTS_PIN_DIRECTION	Defines the direction of the port pins.
	SYS_PORTS_PULLUP_PULLDOWN_STATUS	Provides the pull-up and pull-down status.

Functions

	Name	Description
	SYS_PORTS_ChangeNotificationDisable	Disables the change notification for the selected port. Implementation: Dynamic
	SYS_PORTS_ChangeNotificationEnable	Enables the change notification for the selected port. Implementation: Dynamic
	SYS_PORTS_ChangeNotificationGlobalDisable	Globally disables the change notification for the selected port. Implementation: Dynamic
	SYS_PORTS_ChangeNotificationGlobalEnable	Globally enables the change notification for the selected port. Implementation: Dynamic
	SYS_PORTS_ChangeNotificationIdleModeDisable	Disables the change notification for the selected port in Sleep or Idle mode. Implementation: Dynamic
	SYS_PORTS_ChangeNotificationIdleModeEnable	Enables the change notification for the selected port in Sleep or Idle mode. Implementation: Dynamic
	SYS_PORTS_ChangeNotificationPullUpDisable	Disables a weak pull-up on the change notification pin. Implementation: Dynamic
	SYS_PORTS_ChangeNotificationPullUpEnable	Enables a weak pull-up on the change notification pin. Implementation: Dynamic
	SYS_PORTS_Clear	Clears the selected digital port. Implementation: Dynamic
	SYS_PORTS_DirectionGet	Reads the direction for the selected port. Implementation: Dynamic
	SYS_PORTS_DirectionSelect	Enables the direction for the selected port. Implementation: Dynamic
	SYS_PORTS_Initialize	Initializes PORT Pins/Channels. Implementation: Static/Dynamic
	SYS_PORTS_OpenDrainDisable	Disables the open-drain functionality for the selected port. Implementation: Dynamic
	SYS_PORTS_OpenDrainEnable	Enables the open-drain functionality for the selected port. Implementation: Dynamic
	SYS_PORTS_PinClear	Clears the selected digital pin. Implementation: Dynamic
	SYS_PORTS_PinDirectionSelect	Enables the direction for the selected pin. Implementation: Dynamic
	SYS_PORTS_PinModeSelect	Enables the selected pin as analog or digital. Implementation: Dynamic
	SYS_PORTS_PinOpenDrainDisable	Disables the open-drain functionality for the selected pin. Implementation: Dynamic
	SYS_PORTS_PinOpenDrainEnable	Enables the open-drain functionality for the selected pin. Implementation: Dynamic
	SYS_PORTS_PinRead	Reads the selected digital pin. Implementation: Dynamic
	SYS_PORTS_PinSet	Sets the selected digital pin/latch. Implementation: Dynamic
	SYS_PORTS_PinToggle	Toggles the selected digital pin. Implementation: Dynamic
	SYS_PORTS_PinWrite	Writes the selected digital pin. Implementation: Dynamic

	SYS_PORTS_Read	Reads the data from the I/O port. Implementation: Dynamic
	SYS_PORTS_RemapInput	Input/Output (I/O) function remapping. Implementation: Dynamic
	SYS_PORTS_RemapOutput	Input/Output (I/O) function remapping. Implementation: Dynamic
	SYS_PORTS_Set	Sets the selected digital port/latch based on the mask. Implementation: Dynamic
	SYS_PORTS_Toggle	Toggles the selected digital port pins. Implementation: Dynamic
	SYS_PORTS_Write	Writes the data to the I/O port. Implementation: Dynamic

Description

Ports System Service Interface Definition

This file contains the interface definition for the Ports system service. It provides a way to interact with the Ports subsystem to manage the timing requests supported by the system

File Name

sys_ports.h

Company

Microchip Technology Inc.

Random Number Generator (RNG) System Service Library

This section describes the Random Number Generator (RNG) System Service Library.

Introduction

This library provides a Random Number Generator (RNG) service with a convenient C language interface.

Description

The Random Number Generator (RNG) System Service provides a simple C-language interface for generating random numbers at both pseudo-random and cryptographic strengths.

Using the Library

This topic describes the basic architecture of the RNG System Service Library and provides information and examples on its use.

Description

Interface Header File: [sys_random.h](#)

The interface to the RNG System Service library is defined in the [sys_random.h](#) header file.

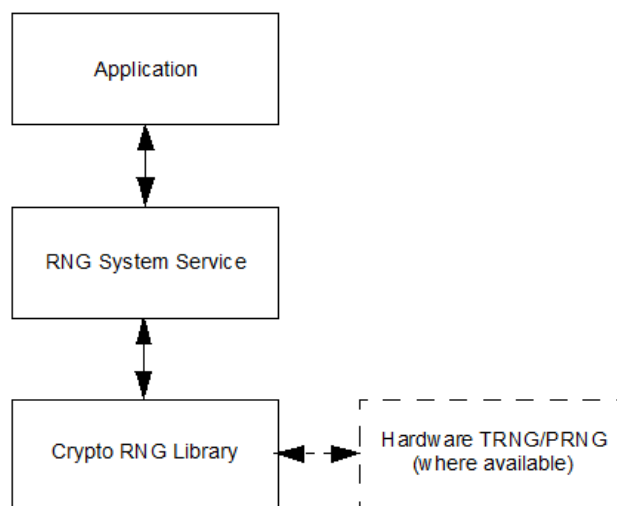
Please refer to the What is MPLAB Harmony? section for how the RNG System Service interacts with the framework.

Abstraction Model

This topic provides a description of the software abstraction for the RNG System Service.

Description

This model explains how the system interfaces with the RNG System Service and the application, as illustrated in the following diagram.



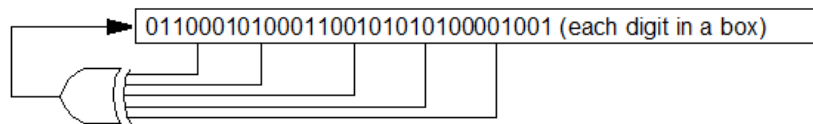
Random Numbers

Random numbers are often desired in systems in order to provide a change in the system that cannot necessarily be predicted. For example, a game might use random numbers for a dice roll or a playing card shuffle. Another example is when a collision occurs in an Ethernet network. A delay is inserted in the next attempt to communicate on the network, and the length of the delay is determined by a random number.

Pseudo Random Numbers (PRN)

Due to the deterministic nature of computer systems, it is actually very difficult to produce true randomness in number generation. For many systems, a pseudo-random number is sufficient. Typically, the pseudo-random number is generated via a Linear Feedback Shift Register (LFSR). The LFSR takes a seed value, and generates a new number typically by having taps on the LFSR in different places and XORing the bits together. This new bit is appended to the beginning of the number after the original number has been shifted one bit to the right.

And example of a LFSR is shown in the following figure.



Cryptographically-secure Pseudo Random Numbers (CSPRN)

Because pseudo-random numbers can be predicted if an attacker knows the setup of the LFSR, it is desired in certain applications to have a way of minimizing the predictability of the random numbers. To accomplish this, a sub-set of PRNGs are run through statistical tests to determine their security in cryptographic applications. A Cryptographically-secure Pseudo Random Number generator is one that has passed all such statistical tests.

True Random Number Generator (TRNG)

To provide seed values for the PRNG, and in applications that require true randomness, there are hardware options for a true random number generator (TRNG). These generators use ring oscillators and the inherent thermal noise within silicon to create the individual bits of a number. Because of the nature of these oscillators, it is not possible to predict the generated numbers.

Library Overview

Please refer to the [System Service Introduction](#) for a detailed description of MPLAB Harmony system services.

The library interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the Random Number Generator module.

Library Interface Section	Description
Random Number Generator Functions	Provides system module APIs. Device initialization, deinitialization, reinitialization and status functions.
Data Types and Constants	Device-specific enumerations, structures, and constants.

How the Library Works

Describes how the RNG Library works.

Description

Service Initialization

To begin using the Random Number System Service, it is necessary to initialize the service using the [SYS_RANDOM_Initialize](#) function.

Example:

```
// Initialization data for the RNG
SYS_RANDOM_INIT rngInit = { 0x12345678, NULL, 32 };
// Initialize the System Random Service
SYS_RANDOM_Initialize( 0, rngInit );
```

Pseudo-Random Numbers

The Pseudo-Random Number functions use the C-language functions `seed()` and `rand()` to provide the pseudo-random numbers.

Example:

```
// Seed the PRNG
SYS_RANDOM_PseudoSeedSet( 0x876543210 );
// Get a Pseudo-random Number
uint32_t myRandomNum = SYS_RANDOM_PseudoGet();
```

Cryptographic Pseudo-Random Numbers

The Cryptographic Random Number functions use the Crypto RNG library functions to seed and generate numbers. On PIC32 devices with a PRNG/TRNG module, it will use the hardware to generate the numbers.

Example:

```
// Get a 32-bit Crypto Random Number
uint32_t myRandomNum = SYS_RANDOM_CryptoGet();
// Fill an array of random numbers
uint32_t myRandomNumArray[10];
SYS_RANDOM_CryptoBlockGet(myRandomNumArray, 10);
// Get a 1-byte Random Number
```

```
uint8_t myRandomByte = SYS_RANDOM_CryptoByteGet();
```

Configuring the Library

Macros

	Name	Description
	SYS_RANDOM_CRYPTO_SEED_SIZE	

Description

The configuration of the RNG System Service is based on the file `system_config.h`.

This header file contains the configuration selection for the Random Number Generator System Service. Based on the selections made, the Random Number Generator System Service may support the selected features. These configuration settings will apply to all instances of the Random Number Generator System Service.

This header can be placed anywhere; however, the path of this header needs to be present in the include search path for a successful build. Refer to the Applications Help section for more details.

`SYS_RANDOM_CRYPTO_SEED_SIZE` Macro

File

[sys_random_config_template.h](#)

C

```
#define SYS_RANDOM_CRYPTO_SEED_SIZE (256/8)
```

Description

Cryptographic Quality Seed Value

Building the Library

This section lists the files that are available in the Random Number Generator System Service Library.

Description

The following three tables list and describe the header (.h) and source (.c) files that implement this library. The parent folder for these files is `<install-dir>/framework/system/random`.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using `#include`) by any code that uses this library.

Source File Name	Description
sys_random.h	Random Number Generator System Service Library API header file.

Required File(s)



All of the required files listed in the following table are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

Source File Name	Description
<code>/src/sys_random.c</code>	Random Number Generator System Service Library implementation.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
N/A	No optional files are available for this system service.











Module Dependencies

The Random Number Generator System Service is dependent upon the following modules:

- Cryptographic (Crypto) Library

Library Interface

a) Random Number Generator Functions

	Name	Description
	SYS_RANDOM_CryptoBlockGet	Generates a sequence of random bytes using the cryptographic quality Random Number Generator.
	SYS_RANDOM_CryptoByteGet	Returns a random byte from the cryptographic quality Random Number Generator.
	SYS_RANDOM_CryptoEntropyAdd	Adds randomness to the cryptographic quality Random Number Generator.
	SYS_RANDOM_CryptoGet	Returns a random 32 bit value from the cryptographic quality Random Number Generator.
	SYS_RANDOM_CryptoSeedSet	Reseeds the cryptographic quality Random Number Generator.
	SYS_RANDOM_CryptoSeedSizeGet	Reseeds the cryptographic quality Random Number Generator.
	SYS_RANDOM_Deinitialize	Deinitializes the Random Number Generator system service.
	SYS_RANDOM_Initialize	Initializes the Random Number Generator system service.
	SYS_RANDOM_PseudoGet	Returns a random value from the Pseudo-random Number Generator.
	SYS_RANDOM_PseudoSeedSet	Reseeds the Pseudo-random Number Generator.

b) Data Types and Constants

	Name	Description
	SYS_RANDOM_INIT	Random Number Generator Service Initialization Data

Description

This section describes the APIs of the RNG System Service Library.

Refer to each section for a detailed description.

a) Random Number Generator Functions

SYS_RANDOM_CryptoBlockGet Function

Generates a sequence of random bytes using the cryptographic quality Random Number Generator.

File

[sys_random.h](#)

C

```
void SYS_RANDOM_CryptoBlockGet(void * buffer, size_t size);
```

Returns

None.

Description

This function uses the Cryptographic-quality Random Number Generator to fill a block of data with random numbers.

Remarks

When run, it will fill the memory starting at buffer with size bytes of random numbers.

Preconditions

System service must be initialized by calling [SYS_RANDOM_Initialize](#) before this function is called.

Example

```
#define RNG_DATA_SIZE    32

uint8_t randomData[RNG_DATA_SIZE];
SYS_MODULE_OBJ rngObject;
```

```
rngObject = SYS_RANDOM_Initialize();  
...  
SYS_RANDOM_CryptoBlockGet((void *)randomData, RNG_DATA_SIZE);
```

Parameters

Parameters	Description
buffer	Pointer to the memory location to fill with random data.
size	The amount of random data, in bytes, to put in memory.

Function

```
void SYS_RANDOM_CryptoBlockGet( uint8_t buffer, size_t bufferSize );
```

SYS_RANDOM_CryptoByteGet Function

Returns a random byte from the cryptographic quality Random Number Generator.

File

[sys_random.h](#)

C

```
uint8_t SYS_RANDOM_CryptoByteGet();
```

Returns

Returns one byte of cryptographic-quality random data.

Description

This function returns a single byte from the Cryptographic-quality Random Number Generator.

Remarks

None.

Preconditions

System service must be initialized by calling [SYS_RANDOM_Initialize](#) before this function is called.

Example

```
uint8_t randomData;  
SYS_MODULE_OBJ rngObject;  
  
rngObject = SYS_RANDOM_Initialize();  
randomData = SYS_RANDOM_CryptoByteGet();
```

Function

```
uint8_t SYS_RANDOM_CryptoByteGet( void );
```

SYS_RANDOM_CryptoEntropyAdd Function

Adds randomness to the cryptographic quality Random Number Generator.

File

[sys_random.h](#)

C

```
void SYS_RANDOM_CryptoEntropyAdd(uint8_t data);
```

Returns

None.

Description

This function adds entropy to the Cryptographic-quality Random Number Generator in order to adjust the randomness of the generated numbers.

Remarks

This function currently does not function.

Preconditions

None.

Example

```
SYS_MODULE_OBJ rngObject;  
  
rngObject = SYS_RANDOM_Initialize();  
SYS_RANDOM_CryptoEntropyAdd(23);
```

Parameters

Parameters	Description
data	8-bit value to add to the entropy calculation.

Function

```
void SYS_RANDOM_CryptoEntropyAdd( uint8_t data );
```

SYS_RANDOM_CryptoGet Function

Returns a random 32 bit value from the cryptographic quality Random Number Generator.

File

[sys_random.h](#)

C

```
uint32_t SYS_RANDOM_CryptoGet();
```

Returns

32-bit random number.

Description

This function returns a 32-bit random number from the Cryptographic- quality Random Number Generator.

Remarks

On PIC32MZ devices with a hardware Random Number Generator, this function returns a value from that generator. Other devices will use a software library to generate the random number.

Preconditions

System service must be initialized by calling [SYS_RANDOM_Initialize](#) before this function is called.

Example

```
SYS_MODULE_OBJ rngObject;  
uint32_t rngData;  
  
rngObject = SYS_RANDOM_Initialize();  
...  
rngData = SYS_RANDOM_CryptoGet();
```

Function

```
uint32_t SYS_RANDOM_CryptoGet( void );
```

SYS_RANDOM_CryptoSeedSet Function

Reseeds the cryptographic quality Random Number Generator.

File

[sys_random.h](#)

C

```
void SYS_RANDOM_CryptoSeedSet(void * seed, size_t size);
```

Returns

None.

Description

This function inserts a seed value into the Cryptographic-quality Random Number Generator. The value will be used to generate the next random number.

Remarks

None.

Preconditions

System service must be initialized by calling [SYS_RANDOM_Initialize](#) before this function is called.

Example

```
SYS_MODULE_OBJ rngObject;
uint8_t rngSeed[SYS_RANDOM_CRYPT_SEED_SIZE];

rngObject = SYS_RANDOM_Initialize();
// Fill the rngSeed
SYS_RANDOM_CryptoSeedSet(rngSeed, sizeof(rngSeed));
```

Parameters

Parameters	Description
seed	Pointer to the memory location containing the new seed.
size	Must be less or equal to SYS_RANDOM_CRYPT_SEED_SIZE

Function

```
void SYS_RANDOM_CryptoSeedSet( void *seed, size_t size);
```

SYS_RANDOM_CryptoSeedSizeGet Function

Reseeds the cryptographic quality Random Number Generator.

File

[sys_random.h](#)

C

```
size_t SYS_RANDOM_CryptoSeedSizeGet();
```

Returns

Returns the size (in bytes) of the most recently set seed value for the cryptographic strength random number generator.

Description

This function returns the current size of the seed used in the Cryptographic- quality Random Number Generator.

Remarks

None.

Preconditions

System service must be initialized by calling [SYS_RANDOM_Initialize](#) before this function is called.

Example

```
SYS_MODULE_OBJ rngObject;
size_t seedSize;

rngObject = SYS_RANDOM_Initialize();
...
seedSize = SYS_RANDOM_CryptoSeedSizeGet();
```

Function

```
size_t SYS_RANDOM_CryptoSeedSizeGet( void );
```

SYS_RANDOM_Deinitialize Function

Deinitializes the Random Number Generator system service.

File

[sys_random.h](#)

C

```
void SYS_RANDOM_Deinitialize(SYS_MODULE_OBJ object);
```

Returns

None.

Description

This function deinitializes the Random Number Generator system service.

Remarks

None.

Preconditions

System service must be initialized by calling [SYS_RANDOM_Initialize](#) before this function is called.

Example

```
SYS_MODULE_OBJ rngObject;  
  
rngObject = SYS_RANDOM_Initialize();
```

Parameters

Parameters	Description
object	SYS RANDOM object handle, returned from SYS_RANDOM_Initialize

Function

```
void SYS_RANDOM_Deinitialize( SYS\_MODULE\_OBJ object );
```

SYS_RANDOM_Initialize Function

Initializes the Random Number Generator system service.

File

[sys_random.h](#)

C

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

Returns

If successful, returns a valid handle to an object. Otherwise, it returns [SYS_MODULE_OBJ_INVALID](#).

Description

This function initializes the Random Number System service, including seeding the Pseudo Random Number Generator, the Crypto RNG library, and setting the seed size.

Remarks

None.

Preconditions

None.

Example

```
SYS_MODULE_OBJ rngObject;  
  
rngObject = SYS_RANDOM_Initialize();
```

```
SYS_RANDOM_Deinitialize(rngObject);
```

Parameters

Parameters	Description
index	Module instance index.
init	initialization data for the random system service (cast of a pointer to a SYS_RANDOM_INIT structure to a SYS_MODULE_INDEX structure pointer).

Function

```
SYS_MODULE_OBJ SYS_RANDOM_Initialize( const SYS_RANDOM_INIT* const randomInit );
```

SYS_RANDOM_PseudoGet Function

Returns a random value from the Pseudo-random Number Generator.

File

[sys_random.h](#)

C

```
uint32_t SYS_RANDOM_PseudoGet();
```

Returns

A 32-bit pseudo-random number.

Description

This function returns one 32-bit Pseudo-random Number.

Remarks

None.

Preconditions

None.

Example

```
uint32_t rngNumber;  
  
rngNumber = SYS_RANDOM_PseudoGet();
```

Function

```
uint32_t SYS_RANDOM_PseudoGet( void );
```

SYS_RANDOM_PseudoSeedSet Function

Reseeds the Pseudo-random Number Generator.

File

[sys_random.h](#)

C

```
void SYS_RANDOM_PseudoSeedSet(uint32_t seed);
```

Returns

None.

Description

This function inserts a seed value into the Pseudo-Random Number Generator. The value will be used to generate the next random number.

Preconditions

None.

Example

```
uint32_t rngSeed = 0x12345678;
```

```
SYS_RANDOM_PseudoSeedSet ( rngSeed );
```

Remarks: None.

Parameters

Parameters	Description
seed	32-bit value to use as the seed for the Pseudo RNG.

Function

```
void SYS_RANDOM_PseudoSeedSet( uint32_t seed );
```

b) Data Types and Constants

SYS_RANDOM_INIT Structure

File

[sys_random.h](#)

C

```
typedef struct {
    uint32_t seedPseudo;
    void * seedCrypto;
    size_t seedCryptoSize;
} SYS_RANDOM_INIT;
```

Description

Random Number Generator Service Initialization Data

Files

Files

Name	Description
sys_random.h	Random Number Generator System Service interface definition.
sys_random_config_template.h	Configuration options template file for the Random Number Generator system service.

Description

This section lists the source and header files used by the library.

sys_random.h

Random Number Generator System Service interface definition.

Functions

	Name	Description
⇒	SYS_RANDOM_CryptoBlockGet	Generates a sequence of random bytes using the cryptographic quality Random Number Generator.
⇒	SYS_RANDOM_CryptoByteGet	Returns a random byte from the cryptographic quality Random Number Generator.
⇒	SYS_RANDOM_CryptoEntropyAdd	Adds randomness to the cryptographic quality Random Number Generator.
⇒	SYS_RANDOM_CryptoGet	Returns a random 32 bit value from the cryptographic quality Random Number Generator.
⇒	SYS_RANDOM_CryptoSeedSet	Reseeds the cryptographic quality Random Number Generator.
⇒	SYS_RANDOM_CryptoSeedSizeGet	Reseeds the cryptographic quality Random Number Generator.
⇒	SYS_RANDOM_Deinitialize	Deinitializes the Random Number Generator system service.
⇒	SYS_RANDOM_Initialize	Initializes the Random Number Generator system service.
⇒	SYS_RANDOM_PseudoGet	Returns a random value from the Pseudo-random Number Generator.
⇒	SYS_RANDOM_PseudoSeedSet	Reseeds the Pseudo-random Number Generator.

Structures

	Name	Description
	SYS_RANDOM_INIT	Random Number Generator Service Initialization Data

Description

Random Number Generator (RNG) System Service Interface Definition

This file contains the interface definition for the Random Number Generator System Service.

File Name

sys_random.h

Company

Microchip Technology Inc.

sys_random_config_template.h

Configuration options template file for the Random Number Generator system service.

Macros

	Name	Description
	SYS_RANDOM_CRYPT0_SEED_SIZE	

Description

Random Number Generator System Service Configuration Definitions Template

This file contains example definitions of configuration options for the random number generator system service. It is for documentation purposes only. Do not include it directly source files.

File Name

sys_random_config_template.h

Company

Microchip Technology Inc.

Reset System Service Library

This section describes the Reset System Service Library.

Introduction

This library provides an interface to manage the Reset module on the Microchip family of microcontrollers during different modes of operation.

Description

The Reset System Service Library is a MPLAB Harmony system service that can be used to identify the reason for the most recent processor reset, manage the system's reset reason status flags, and even cause a reset under software control if the processor in use supports that functionality.

Using the Library

This topic describes the basic architecture of the Reset System Service and provides information and examples on its use.

Description

Interface Header File: `sys_reset.h`

The interface to the Clock System Service is defined in the `sys_reset.h` header file, which is included by the `sys.h` header file.

Any C language source (.c) file that uses the Reset System Service must include `sys.h`.

Abstraction Model

This topic describes the basic abstraction model and functionality of the Reset System Service.

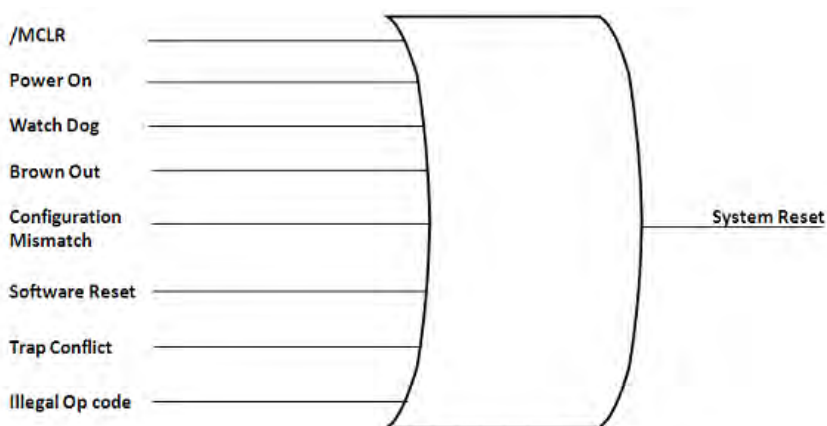
Description

The Reset System Service Library provides the following functionality:

- Microcontroller Reset Reason Management
- Triggering Software Reset

When one of the sources described in the following section asserts a reset request or the software requests a reset using this library, the microcontroller will be immediately reset.

The following figure illustrates the hardware abstraction model for the reset system service,



Reset Sources

- **/MCLR:** Whenever the /MCLR pin is driven low, the device asynchronously asserts reset. The processor continues to use the existing clock source that was in use before the /MCLR Reset occurred.
- **Power On [POR]:** A power-on event generates an internal power-on reset pulse when a VDD rise is detected above POR voltage. In particular, VDD must fall below POR voltage before a new POR is initiated.
- **Watch Dog Timer [WDT]:** Whenever a watchdog timer time-out occurs, the device asynchronously asserts system reset. The clock source remains unchanged. Note that a WDT time-out during Sleep or Idle mode will wake-up the processor, but will not reset the processor.
- **Brown Out:** If the voltage supplied to the internal regulator is inadequate to maintain a regulated level, the regulator reset circuitry will generate a Brown-Out Reset (BOR). Brown out reset is applicable only when the regulator is enabled.
- **Configuration Mismatch:** To maintain the integrity of the stored configuration values, all device configuration bits are implemented as a complementary set of register bits. For each bit, as the actual value of the register is written as '1', a complementary value, '0', is stored in its corresponding background register and vice versa. The bit pairs are compared every time, including sleep mode. During this comparison, if the

configuration bit values are not found opposite to each other, a configuration mismatch event is generated which causes a device reset.

- **Software Reset:** Whenever the reset instruction or a reset sequence is executed, the device asserts system reset. This reset state does not reinitialize the clock. The clock source that is in effect prior to the reset instruction or reset sequence execution remains in effect.
- **Trap Conflict:** A trap conflict reset occurs when a hard and a soft trap occur at the same time.
- **Illegal Opcode:** A device reset is generated if the device attempts to execute an illegal opcode value that was fetched from program memory.



Note: For more information on the electrical specifications, refer to the "**Electrical Characteristics**" chapter of the specific device data sheet.

Library Overview

Please refer to the [System Service Introduction](#) for a detailed description of MPLAB Harmony system services.

The library interface routines are divided into the following sub-sections, which address one of the blocks or the overall operation of the Reset System Service Library.

Library Interface Section	Description
Functions	Provides functions to read the reset reason and clear the respective status, to trigger the software reset, and reset the NMI delay count.

How the Library Works

Reset Reason Management

Describes how to use the library to identify and clear the reason a reset occurred.

Description

Reset Reason

Other software modules may identify the reason for the most recent system reset using the [SYS_RESET_ReasonGet](#) function. After obtaining the reset reason, its status flag can be cleared using the [SYS_RESET_ReasonClear](#) function.

Example: Identifying and Clearing Reset Reason

```
RESET_REASON reasonType;

reasonType = SYS_RESET_ReasonGet();
if( reasonType == RESET_REASON_MCLR )
{
    // Take some reset reason specific action
    // ...

    // Clear the reset reason status flag
    SYS_RESET_ReasonClear( RESET_REASON_MCLR );
}
```

Example: Clearing Multiple Reset Reasons

```
// Clear multiple reason flags
SYS_RESET_ReasonClear( RESET_REASON_MCLR|RESET_REASON_POWERON );

// Clear all current reason flags
SYS_RESET_ReasonClear( RESET_REASON_ALL );
```



Note: Not every reset reason is supported by every device, refer to the device specific information for more information.

Triggering a Software Reset

Describes how to use the Reset System Service Library to cause a system reset.

Description

Software Reset Trigger

Software can trigger the a system reset by using the `SYS_RESET_SoftwareReset` function.

Example

```
SYS_RESET_SoftwareReset();
```

**Note:**

On devices that support software reset functionality, this routine never returns. Execution continues from the reset vector, following the reset. On devices that do not support software reset, calling this routine will give a build warning and execution will continue immediately following this function.

Configuring the Library

Describes how to configure the system reset library.

Description

The configuration of the Reset System Service is affected by the following:

- Processor Selection
- System Configuration

Processor Selection

Different processors support different reset reasons. Not all reset reasons are defined for all processors. Refer to the specific device data sheet or family reference manual for the processor in use to identify which reset reasons it supports.

System Configuration

Many MPLAB Harmony libraries provide configuration options that can be selected by defining their desired values at build-time using the C-language preprocessor "#define" statement. To obtain these definitions, MPLAB Harmony libraries include a file named "system_config.h" that must be defined as part of the over-all system configuration and placed in the compiler's include file search path.

Refer to the Applications Help section for more details.

System Reset Library Configuration Options

This library does not support any build-time configuration items.

Building the Library

This section lists the files that are available in the Reset System Service Library.

Description

The following three tables list and describe the header (.h) and source (.c) files that implement this library. The parent folder for these files is `<install-dir>/framework/system/reset`.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using `#include`) by any code that uses this library.

Source File Name	Description
<code>sys_reset.h</code>	Reset System Service Library API header file.

Required File(s)



All of the required files listed in the following table are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

Source File Name	Description
<code>/src/sys_reset.c</code>	Reset System Service Library implementation.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
N/A	There are no optional files for this library.





Module Dependencies

The Reset System Service Library depends on the following modules:

- Reset Peripheral Library
- Device Control Peripheral Library

Library Interface

a) Functions

	Name	Description
	SYS_RESET_ReasonClear	Clears the status flag for the specified reset reason.
	SYS_RESET_ReasonGet	Returns the reason for the reset.
	SYS_RESET_SoftwareReset	Triggers a software reset.
	SYS_RESET_NMIDelayCountSet	Sets the Delay for WDT/DMT NMI reset events.

Description

This section describes the APIs of the Reset System Service Library.

Refer to each section for a detailed description.

a) Functions

SYS_RESET_ReasonClear Function

Clears the status flag for the specified reset reason.

File

[sys_reset.h](#)

C

```
void SYS_RESET_ReasonClear(RESET_REASON reason);
```

Returns

None.

Description

This function clears the specified status flag(s) that were previously set by the processor to identify the reason for the most recent reset.

Remarks

None.

Preconditions

Hardware should have set the reset reason.

Example

```
// To clear a single reset reason,
SYS_RESET_ReasonClear( RESET_REASON_MCLR );

// To clear more than one reason,
SYS_RESET_ReasonClear( RESET_REASON_MCLR | RESET_REASON_POWERON );

// To clear all the reasons,
SYS_RESET_ReasonClear( RESET_REASON_ALL );
```

Parameters

Parameters	Description
reason	One of the possible values of the enum RESET_REASON

Function

```
void SYS_RESET_ReasonClear( RESET_REASON reason )
```

SYS_RESET_ReasonGet Function

Returns the reason for the reset.

File

[sys_reset.h](#)

C

```
RESET_REASON SYS_RESET_ReasonGet ( ) ;
```

Returns

A value identifying the reason for the most recent reset.

Description

This function identifies the reason that the most recent reset occurred.

Remarks

None.

Preconditions

None.

Example

```
RESET_REASON reasonType;  
reasonType = SYS_RESET_ReasonGet ( ) ;
```

Function

```
RESET_REASON SYS_RESET_ReasonGet( void )
```

SYS_RESET_SoftwareReset Function

Triggers a software reset.

File

[sys_reset.h](#)

C

```
void SYS_RESET_SoftwareReset ( ) ;
```

Returns

None.

Description

This function triggers a processor.

Remarks

The software reset feature must be supported by the processor in use.

Preconditions

None.

Example

```
SYS_RESET_SoftwareReset ( ) ;
```

Function

```
void SYS_RESET_SoftwareReset( void )
```

SYS_RESET_NMIDelayCountSet Function

Sets the Delay for WDT/DMT NMI reset events.

File

[sys_reset.h](#)

C

```
void SYS_RESET_NMIDelayCountSet( RESET_NMI_COUNT_TYPE nmi_count );
```

Returns

None

Description

This function sets the NMI counter value, that is equivalent to the number of SYSCLK cycles needed for the device Reset to occur when a WDT/DMT NMI event is triggered. This NMI reset counter is only applicable to these two specific NMI events only.

Remarks

The NMI feature must be supported by the processor in use.

Preconditions

None.

Example

```
SYS_RESET_NMIDelayCountSet( 0x54 );
```

Parameters

Parameters	Description
nmi_count	NMI counter value.

Function

```
void SYS_RESET_NMIDelayCountSet( RESET_NMI_COUNT_TYPE nmi_count )
```

Files

Files





Name	Description
sys_reset.h	Reset System Service interface definition.

Description

sys_reset.h

Reset System Service interface definition.

Functions

	Name	Description
	SYS_RESET_NMIDelayCountSet	Sets the Delay for WDT/DMT NMI reset events.
	SYS_RESET_ReasonClear	Clears the status flag for the specified reset reason.
	SYS_RESET_ReasonGet	Returns the reason for the reset.
	SYS_RESET_SoftwareReset	Triggers a software reset.

Description

Reset System Service Interface Definition

This file contains the interface definition for the Reset System Service. It provides a way to interact with the Reset subsystem to manage the

system resets and their status.

File Name

sys_reset.h

Company

Microchip Technology Inc.

Timer System Service Library

This section describes the Timer System Service Library.

Introduction

This library provides interfaces to manage alarms and/or delays.

Description

The Timer System Service Library is capable of providing periodic or one-shot alarm delays to the user. It works as a client for the Timer driver and opens one of the instances of the driver to perform the activities. The periodicity or the delay is an integer multiple of the Timer driver alarm period.

Using the Library

This topic describes the basic architecture of the Timer System Service Library and provides information and examples on its use.

Description

Interface Header File: `sys_tmr.h`


The interface to the Timer System Service library is defined in the `sys_tmr.h` header file, which is included by the `sys.h` system service header file. Any C language source (.c) file that uses the Timer System Service library should include `sys.h`.

Please refer to the What is MPLAB Harmony? section for how the System Service interacts with the framework.

Abstraction Model

The Timer System Service module uses the Timer driver abstraction layer to provide the following functionalities,

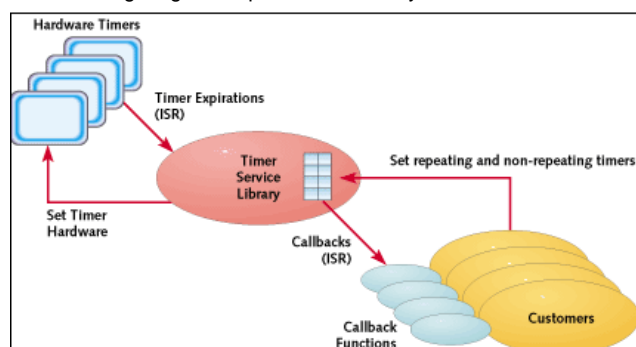
- Periodic Callback
- One Shot/Single Callback
- Delays

 **Note:** The Timer System Service will work in both the polling and interrupt driven environments; however, Interrupt mode is recommended.

Description

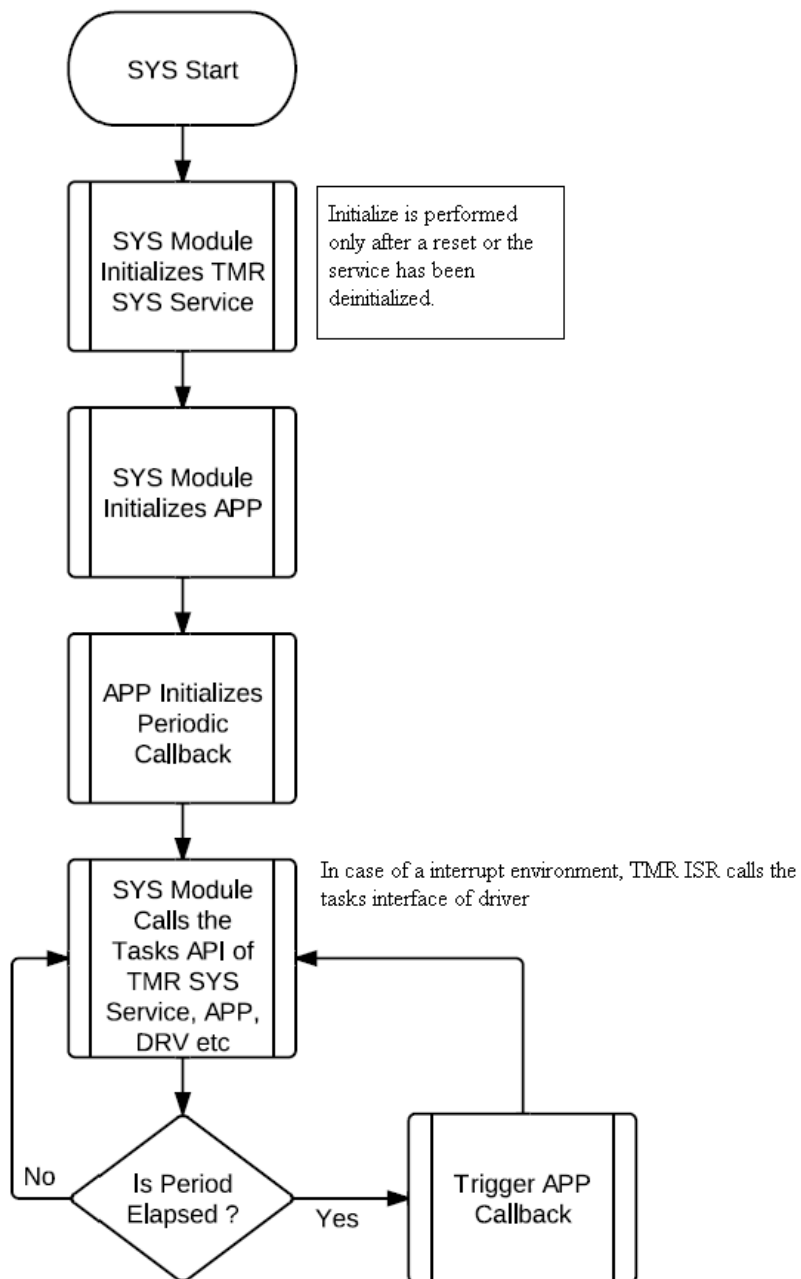
Abstraction Model

The following diagram depicts the Timer System Service abstraction model.



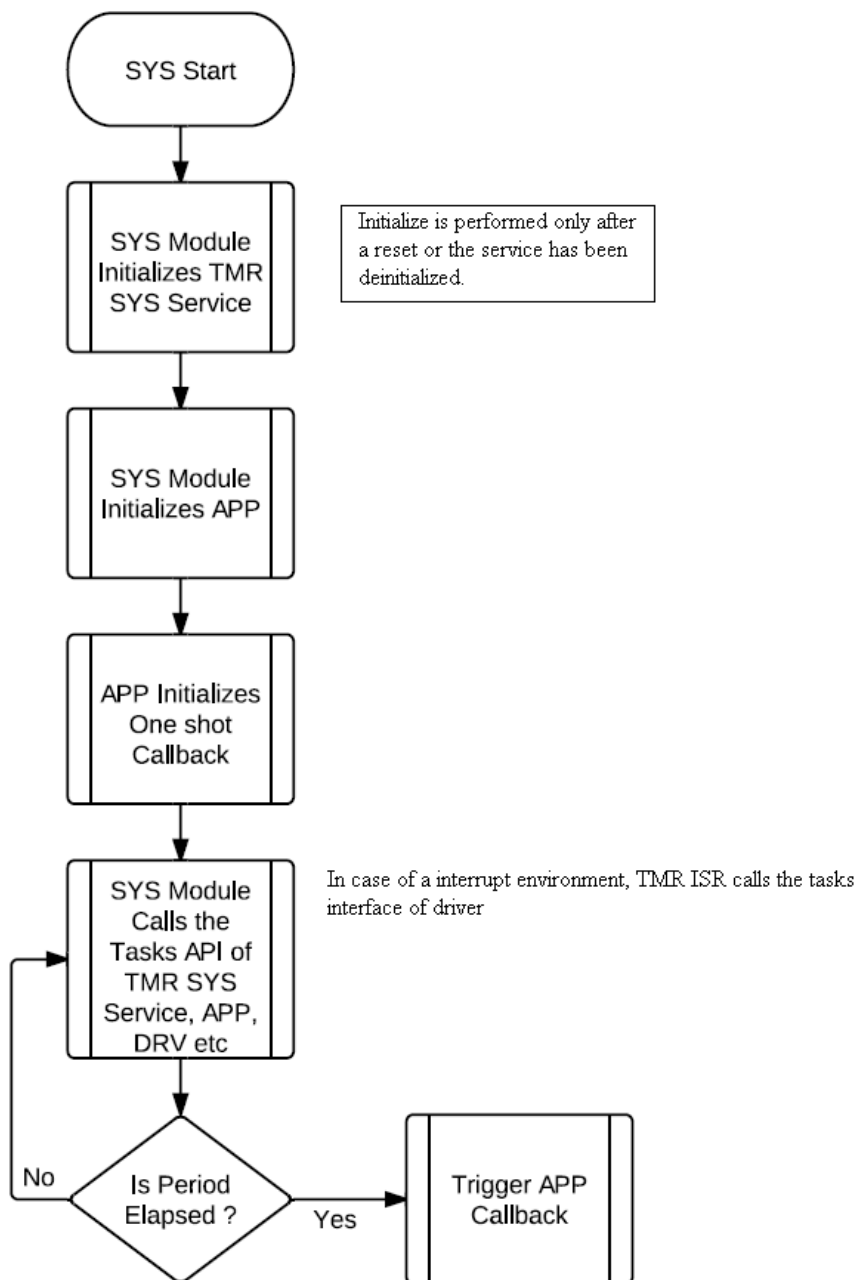
Periodic Callback

The following flow diagram depicts the Periodic Callback usage model.



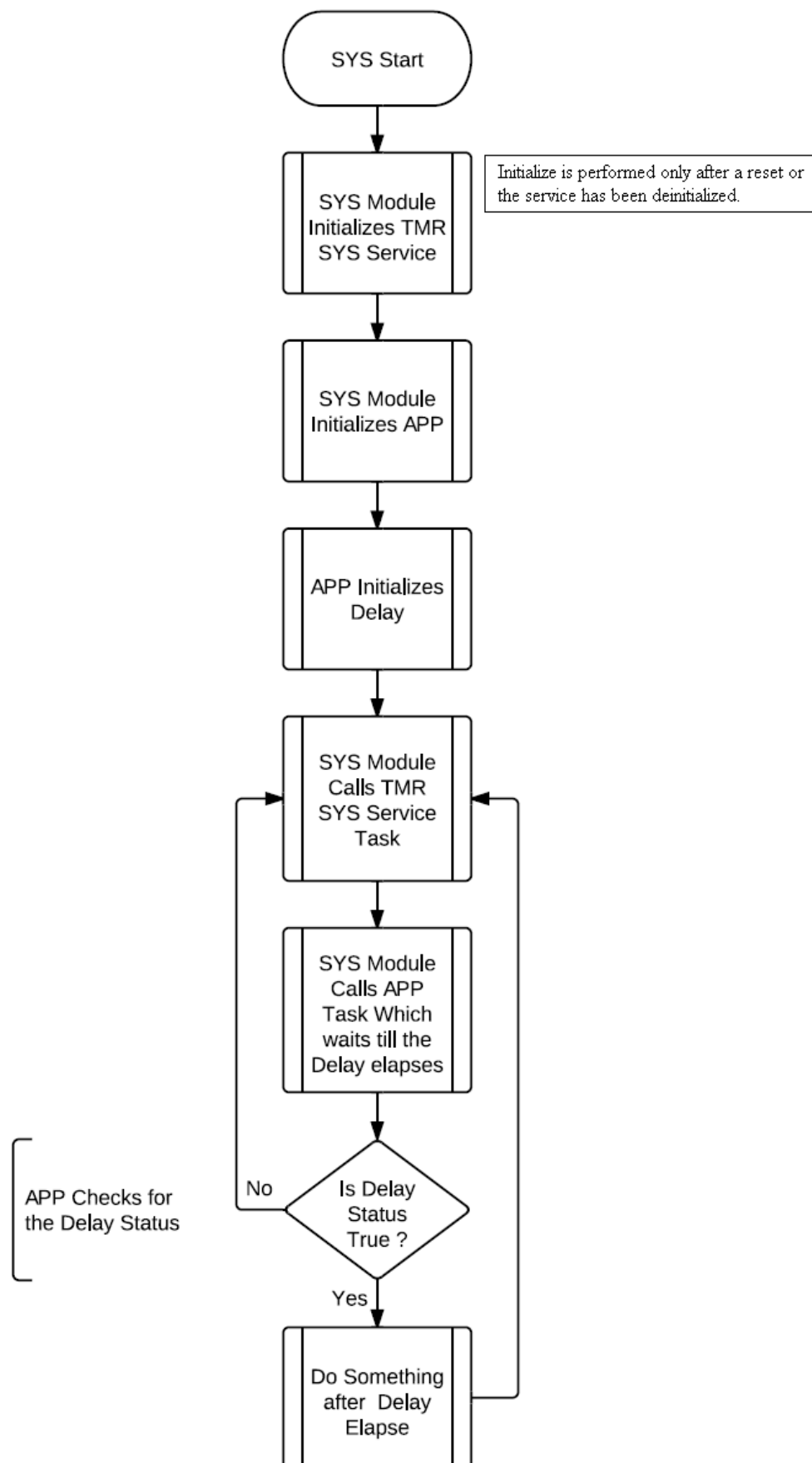
One-Shot Callback

The following flow diagram depicts the One-Shot Callback usage model.



Delay

The following flow diagram depicts the Delay usage model.



Library Overview

Please refer to the [System Service Introduction](#) for a detailed description of MPLAB Harmony system services.

The library interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the Timer System Service Library.

Library Interface Section	Description
System Level Interaction Functions	Provides system module APIs. Device initialization, deinitialization, reinitialization and status functions.
Timed Callback Functions	Provides interfaces to handle timed periodic or one shot callbacks
Timed Delay Functions	Provides interfaces to handle timed delays
Miscellaneous Functions	Provides interfaces for timer tick counts, etc.

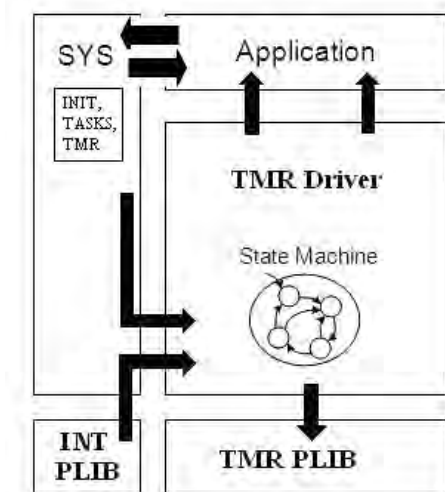
How the Library Works

The library provides interfaces to support:

- System Interaction
- Periodic Callback
- One shot/Single Callback
- Delays
- Tick count Information

Description

The following model gives information on interaction between various modules.



Note: The arrows in the diagram indicate the dependencies or support.

Module interaction occurs in the following order:

1. The initialization, reinitialization, deinitialization are handled by the SYS INIT module.
2. The Timer System Service Initialization function initializes the library's internal data structures.
3. After the initialization, Timer System Service Tasks function would be called by the SYS Tasks module to open the Timer Driver and do other configuration activities. Once the Timer driver is ready to be used the Timer System Service Tasks API makes the system ready to be used by setting the status of the Timer System Service module to SYS_STATUS_READY.
4. The application can now enable either periodic callback or one-shot callback or enable delays.

- Notes:**
1. It is possible to have multiple clients access the system timer service, each with different periodic or single callbacks or delays.
 2. If the underlying Timer Driver uses Interrupt mode, it is the responsibility of the user or the application to ensure that the Interrupt system module has been initialized and that priorities have been set accordingly.
 3. Ensure that the selected timer clock source is DRV_TMR_CLKSOURCE_INTERNAL, as the Timer System Service will not function if the timer clock source is external.

System Interaction

Initialization and Reinitialization:

The SYS module performs the initialization and the reinitialization of the Timer System Service. During initialization the following information is populated into the internal data structures from the init structure [SYS_TMR_INIT](#) passed as a parameter:

Initialization Member	Description
moduleInit	System module initialization of the power state
drvIndex	Timer driver module/instance index, the same index used to initialize the timer driver
tmrFreq	Required System Timer frequency in Hertz, which is the base frequency for all of the system time-keeping.

The [SYS_TMR_Initialize](#) API returns a handle of the type [SYS_MODULE_OBJ](#). After this, the object handle returned by the Initialize interface would be used by the other system interfaces such as [SYS_TMR_Reinitialize](#), [SYS_TMR_Deinitialize](#), [SYS_TMR_Status](#), and [SYS_TMR_Tasks](#).



Note: If an attempt is made to initialize the SYS TMR module if it is already initialized, will safely return without modifying the behavior of the system.

Example for Timer System Service initialization:

```
SYS_MODULE_OBJ  objectHandle;
SYS_TMR_INIT    initConfig;
```

```
// Populate the SYS TMR initialization structure
```

```
initConfig.moduleInit.value      = SYS_MODULE_POWER_RUN_FULL;
initConfig.drvIndex              = DRV_TMR_INDEX_0;
initConfig.tmrFreq               = 1000;
```

```
objectHandle = SYS_TMR_Initialize( SYS_TMR_INDEX_0, ( SYS_MODULE_INIT * )&initConfig );
if( SYS_MODULE_OBJ_INVALID == objectHandle )
{
    // Handle error
}
```

Sample initialization sequence in polling environment:

```
void SYS_Initialize( SYS_INIT_DATA *data )
{
    /* Application Initialization */
    App_Init();

    /* Initialize the Timer driver */
    drvTmrObject = DRV_TMR_Initialize( DRV_TMR_INDEX_0, (SYS_MODULE_INIT *)&tmrInitData );

    /* Initialize the SYS TMR Module */
    sysTmrObject = SYS_TMR_Initialize( SYS_TMR_INDEX_0, (SYS_MODULE_INIT *)&sysTmrInitData );
}
```

Sample initialization sequence in an interrupt driven environment:

```
void SYS_Initialize( SYS_INIT_DATA *data )
{
    /* Application Initialization */
    App_Init();

    /* Initialize the Timer driver */
    drvTmrObject = DRV_TMR_Initialize( DRV_TMR_INDEX_0, ( SYS_MODULE_INIT * )&tmrInitData );

    /* Initialize the SYS TMR Module */
    sysTmrObject = SYS_TMR_Initialize( SYS_TMR_INDEX_0, ( SYS_MODULE_INIT * )&sysTmrInitData );

    /* Set the Timer Interrupt priority, sub-priority if supported by the device */
    SYS_INT_VectorPrioritySet( INT_VECTOR_T1, INT_PRIORITY_LEVEL1 );
    SYS_INT_VectorSubprioritySet( INT_VECTOR_T1, INT_SUBPRIORITY_LEVEL1 );

    /* Initialize the interrupt sub system module */
    SYS_INT_Initialize();
}
```

Deinitialization:

The deinitialize operation ([SYS_TMR_Deinitialize](#)) places the module in inactive state. Once the initialize operation has been called, the deinitialize

operation must be called before the initialize is called again. If the operation requires time to allow the hardware to complete, this will be reported by the [SYS_TMR_Status](#) operation.

Status:

Status of the system timer module can be checked using [SYS_TMR_Status](#) interface. After the initialization, reinitialization or the deinitialization activities the System timer Service status can be retrieved and checked for further actions.

Tasks Routine:

The system will call [SYS_TMR_Tasks](#) from System Task Service. This interface gets the system timer module into running mode. This interface needs to be called after the timer driver is successfully initialized.

Sample call order:

```
void SYS_Tasks( void )
{
    /* Call Driver TMR Tasks API */
    DRV_TMR_Tasks( drvTmrObject );

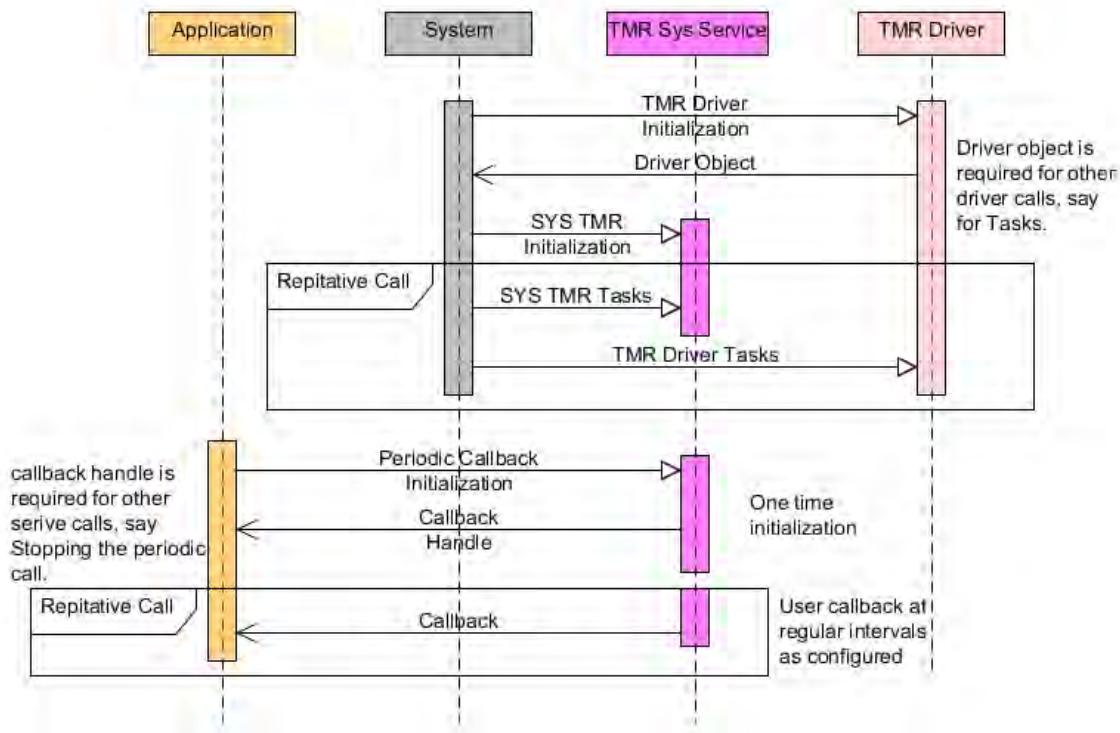
    /* Call SYS TMR Tasks API */
    SYS_TMR_Tasks( sysTmrObject );

    /* Call the Application Tasks */
    App_Tasks();
}
```

Periodic Callback

This section describes the periodic callback registration and its usage.

The following diagram depicts the interaction across various modules for the periodic callback feature,



Steps involved in registering and using the feature:

1. The Timer System Service should be in a ready state. Use the [SYS_TMR_STATUS](#) function to verify that the service is in this state.
2. Register the periodic callback feature using the [SYS_TMR_CallbackPeriodic](#) function with the period, a context pointer, and the pointer to the callback routine. The interface [SYS_TMR_CallbackPeriodic](#) returns a valid handle after successful registration.



Notes:

1. The handle can be later used by the user to stop the periodic callback by the user using the interface [SYS_TMR_CallbackStop](#). Based on the periodicity set by the user the respective callback would be triggered repetitively after it elapses.
2. The period parameter will be adjusted internally to match the Timer System running frequency.

Example:

```

void SYS_Initialize ( SYS_INIT_DATA *data )
{
    /* Initialize the Timer driver */
    drvTmrObject = DRV_TMR_Initialize ( SYS_INDEX, (SYS_MODULE_INIT *)&tmrInitData );

    /* Initialize the SYS TMR Module */
    sysTmrObject = SYS_TMR_Initialize ( SYS_INDEX, (SYS_MODULE_INIT *)&sysTmrInitData );

    /* Application Initialization */
    App_Initialize ();
}

void SYS_Tasks(void)
{
    /* Call Driver TMR Tasks API */
    DRV_TMR_Tasks ( drvTmrObject );

    /* Call SYS TMR Tasks API */
    SYS_TMR_Tasks ( sysTmrObject );

    /* Call the App Tasks */
    App_Tasks ();
}

void App_Tasks ( void )
{
    switch (testState)
    {
        /* Wait in Init state until the SYS TMR Module is in running mode */
        case TEST_STATE_Init:
            if (SYS_STATUS_READY == SYS_TMR_Status(sysTmrObject))
            {
                /* SYS TMR is in running mode */
                testState = TEST_STATE_Config_Periodic;
            }
            else
            {
                testState = TEST_STATE_Init;
            }
            break;

        case TEST_STATE_Config_Periodic:
            /* Activate periodic callback */
            handle = SYS_TMR_CallbackPeriodic (120, 0, &Test_Callback);
            testState = TEST_STATE_Count_Read;
            break;

        case TEST_STATE_Count_Read:
            testCount1 = SYS_TMR_TickCountGet ();
            break;
    }
}

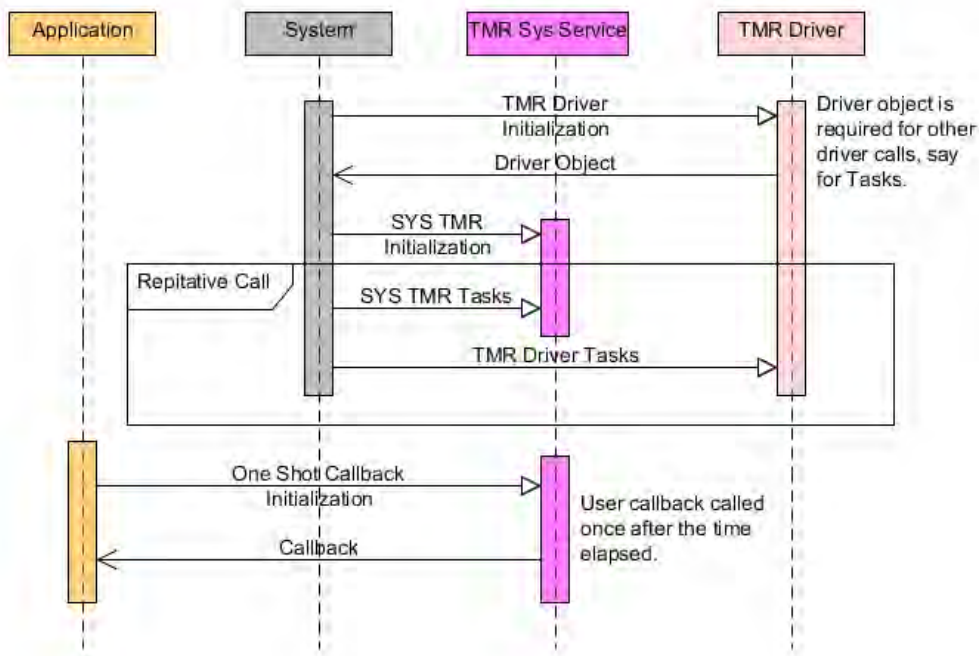
// Callback which will be called repetitively after the configured
// time elapses
void Test_Callback ( uintptr_t context, uint32_t currTick )
{
    PORTA = ~PORTA;
}

```

One Shot Callback

This section describes the one-shot/single callback registration and its usage.

The following diagram depicts the interaction across various modules for the one-shot callback feature,



Steps involved in registering and using the feature:

1. Initial requirement is to have the Timer System Service module in the SYS_STATUS_READY state. The user can call the function [SYS_TMR_Status](#) to get the current status information of the module.
2. One-shot/Single callback feature can be registered by calling the function [SYS_TMR_CallbackSingle](#) with the parameters periodicity of the callback, a context pointer, and a pointer to the user's callback routine.

Based on the periodicity set by the user, the respective callback would be triggered once after it elapses.



Notes:

1. The handle can be later used by the user to stop the one-shot callback by the user using the interface [SYS_TMR_CallbackStop](#).
2. The period parameter will be adjusted internally to match the Timer System running frequency.

Example:

```

void SYS_Initialize ( SYS_INIT_DATA *data )
{
    /* Initialize the Timer driver */
    drvTmrObject = DRV_TMR_Initialize ( SYS_INDEX, (SYS_MODULE_INIT *)&tmrInitData );

    /* Initialize the SYS TMR Module */
    sysTmrObject = SYS_TMR_Initialize ( SYS_INDEX, (SYS_MODULE_INIT *)&sysTmrInitData );

    /* Application Initialization */
    App_Initialize ();
}

void SYS_Tasks(void)
{
    /* Call Driver TMR Tasks API */
    DRV_TMR_Tasks ( drvTmrObject );

    /* Call SYS TMR Tasks API */
    SYS_TMR_Tasks ( sysTmrObject );

    /* Call the App Tasks */
    App_Tasks ();
}

void App_Tasks ( void )
{
    switch (testState)
    {

```



```

/* Wait in Init state until the SYS TMR Module is in running mode */
case TEST_STATE_Init:
    if (SYS_STATUS_READY == SYS_TMR_Status(sysTmrObject))
    {
        /* SYS TMR is in running mode */
        testState = TEST_STATE_Config_Single;
    }
    else
    {
        testState = TEST_STATE_Init;
    }
    break;

case TEST_STATE_Config_Single:
    /* Activate one shot/single callback */
    handle = SYS_TMR_CallbackSingle (120, 0, &Test_Callback);
    testState = TEST_STATE_Count_Read;
    break;

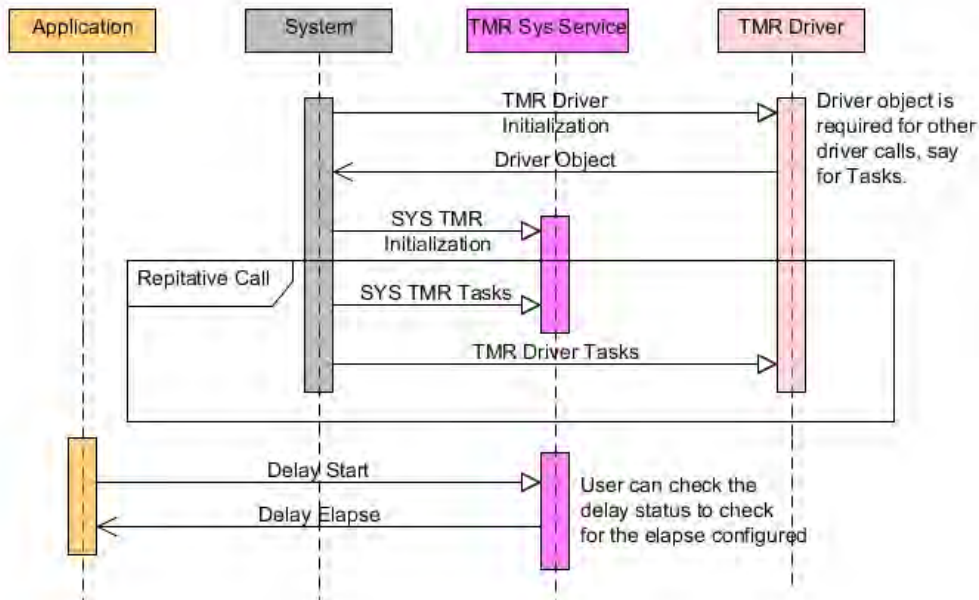
case TEST_STATE_Count_Read:
    testCount1 = SYS_TMR_TickCountGet ();
    break;
}

// Callback which will be called repetitively after the configured
// time elapses
void Test_Callback ( uintptr_t context, uint32_t currTick )
{
    PORTA = ~PORTA;
}

```

Delays

This section describes the delay registration and its usage. The mechanism depicted supports only non-blocking delays. The following diagram depicts the interaction across various modules for the delay feature.



Steps involved in registering and using the feature:

1. Initial requirement is to have the Timer System Service module in the SYS_STATUS_READY state. The user can call the function [SYS_TMR_Status](#) to get the current status information of the module.
2. Delay feature can be registered by calling the function [SYS_TMR_DelayMS](#) with the delay parameter.

**Notes:**

1. The API [SYS_TMR_DelayMS](#) returns a valid handle after successful registration. This handle can be later used by the user to check the status of the delay using the API [SYS_TMR_DelayStatusGet](#).
2. The period parameter will be adjusted internally to match the Timer System running frequency.

Example:

```

case DELAY_START_STATE:
    handle = SYS_TMR_DelayMS (500);
    state = DELAY_CHECKSTATE;
break;

case DELAY_CHECKSTATE:
    if (true == SYS_TMR_DelayStatusGet(handle))
    {
        state = DELAY_START_STATE;
    }
    else
    {
        state = DELAY_CHECKSTATE;
    }
break;

```

Miscellaneous

Tick Counts Information: The API [SYS_TMR_TickCountGet](#) provides the current tick information.

Examples**Sample Application Tasks****Periodic callback with only one period with 120 ms period:**

```

switch (state)
{
    case STATE_Init:
        if (SYS_STATUS_READY == SYS_TMR_Status(sysTmrObject))
        {
            state = STATE_Config_Periodic;
        }
        else
        {
            state = STATE_Init;
        }
        break;

    case STATE_Config_Periodic:
        handle = SYS_TMR_CallbackPeriodic (120, 0, &Test_Callback);
        state = STATE_Count_Read;
        break;

    case STATE_Count_Read:
        testCount = SYS_TMR_TickCountGet ();
        state = STATE_Count_Read;
        break;
}

```

Periodic callback with multiple periods 120 ms and 250 ms:

```

switch (state)
{
    case STATE_Init:
        if (SYS_STATUS_READY == SYS_TMR_Status(sysTmrObject))
        {
            state = STATE_Config_Periodic;
        }
        else
        {
            state = STATE_Init;
        }
        break;
}

```

```

case STATE_Config_Periodic:
    handle1 = SYS_TMR_CallbackPeriodic (120, 0, &Test_Callback1);
    handle2 = SYS_TMR_CallbackPeriodic (250, 0, &Test_Callback2);
    state = STATE_Count_Read;
break;

case STATE_Count_Read:
    testCount = SYS_TMR_TickCountGet ();
    state = STATE_Count_Read;
break;
}

void Test_Callback1 ( uintptr_t context, uint32_t currTick )
{
    /* Assume RA0 is connected to one LED */
    PORTAbits.RA0 = ~PORTAbits.RA0;
}

```

```

void Test_Callback2 ( uintptr_t context, uint32_t currTick )
{
    /* Assume RA1 is connected to another LED */
    PORTAbits.RA1 = ~PORTAbits.RA1;
}

```

One-Shot Callback:

```

switch (state)
{
    case STATE_Init:
        if (SYS_STATUS_READY == SYS_TMR_Status(sysTmrObject))
        {
            state = STATE_Config_Single;
        }
        else
        {
            state = STATE_Init;
        }
        break;

    case STATE_Config_Single:
        SYS_TMR_CallbackSingle (500, 0, &Test_Callback);
        state = STATE_Count_Read;
        break;

    case STATE_Count_Read:
        testCount = SYS_TMR_TickCountGet ();
        state = STATE_Count_Read;
        break;
}

```

Delay:

```

switch (state)
{
    case STATE_Init:
        if (SYS_STATUS_READY == SYS_TMR_Status(sysTmrObject))
        {
            state = STATE_DelayStart;
        }
        else
        {
            state = STATE_Init;
        }
        break;

    case STATE_DelayStart:
        dlHandle = SYS_TMR_DelayMS (500);
        state = STATE_DelayCheck;
        break;

    case STATE_DelayCheck:
        if (SYS_TMR_DELAY_EXPIRED == SYS_TMR_DelayStatusGet(dlHandle))

```

```

    {
        state = STATE_DelayStart;
    }
    else
    {
        state = STATE_DelayCheck;
    }
    break;
}

```

Configuring the Library

Macros

	Name	Description
	<code>_SYS_TMR_CONFIG_TEMPLATE_H</code>	This is macro <code>_SYS_TMR_CONFIG_TEMPLATE_H</code> .

Description

The configuration of the Timer System Service is based on the file `system_config.h`.

This header file contains the configuration selection for the Timer System Service build. Based on the selections made, the Timer System Service may support the selected features. These configuration settings will apply to all instances of the Timer System Service.

This header can be placed anywhere, the path of this header needs to be present in the include search path for a successful build. Refer to the Applications Help section for more details.

`_SYS_TMR_CONFIG_TEMPLATE_H` Macro

File

`sys_tmr_config_template.h`

C

```
#define _SYS_TMR_CONFIG_TEMPLATE_H
```

Description

This is macro `_SYS_TMR_CONFIG_TEMPLATE_H`.

Building the Library

This section lists the files that are available in the Timer System Service Library.

Description

The following three tables list and describe the header (.h) and source (.c) files that implement this library. The parent folder for these files is `<install-dir>/framework/system/timer`.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using `#include`) by any code that uses this library.

Source File Name	Description
<code>sys_tmr.h</code>	Timer System Service Library API header file.

Required File(s)



All of the required files listed in the following table are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

Source File Name	Description
<code>/src/dynamic/sys_tmr.c</code>	Timer System Service Library implementation.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

Source File Name	Description
N/A	There are no optional files for this library.





Module Dependencies

The Timer System Service Library depends on the following modules:




- Timer Driver Library

Library Interface



a) System Level Interaction Functions

	Name	Description
	SYS_TMR_Deinitialize	Deinitializes the specific module instance of the TMR module
	SYS_TMR_Initialize	Initializes hardware and data for the instance of the Timer module and opens the specific module instance.
	SYS_TMR_Status	Returns status of the specific instance of the Timer module.
	SYS_TMR_Tasks	Maintains the system timer's state machine and implements its ISR.





b) Timed Callback Functions

	Name	Description
	SYS_TMR_CallbackStop	Stops a periodic timer object.
	SYS_TMR_CallbackPeriodic	Creates a permanent timer object that calls back to the client periodically.
	SYS_TMR_CallbackSingle	Creates a one-shot/single callback timer object.







c) Timed Delay Functions

	Name	Description
	SYS_TMR_DelayStatusGet	Checks the status of the previously requested delay timer object.
	SYS_TMR_DelayMS	Creates a timer object that times out after the specified delay.

d) Timed Object Functions

	Name	Description
	SYS_TMR_ObjectCountGet	Returns the current millisecond count of a timer object.
	SYS_TMR_ObjectCreate	Creates a new timer object.
	SYS_TMR_ObjectDelete	Deletes an existent timer object.
	SYS_TMR_ObjectReload	Reloads an existing timer object.

e) Miscellaneous Functions

	Name	Description
	SYS_TMR_TickCountGet	Provides the current counter value.
	SYS_TMR_TickCountGetLong	Provides the current counter value with 64-bit resolution.
	SYS_TMR_TickCounterFrequencyGet	Provides the number of ticks per second
	SYS_TMR_SystemCountGet	Provides the current system time count value.
	SYS_TMR_SystemCountFrequencyGet	Provides the current system count frequency.
	SYS_TMR_ModuleStatusGet	Returns status of the specific instance of the Timer module.

f) Data Types and Constants

	Name	Description
	SYS_TMR_CALLBACK	This data type defines a pointer callback function.
	SYS_TMR_HANDLE	Identifies a client timer object.
	SYS_TMR_INIT	Identifies the system timer initialize structure.
	SYS_TMR_HANDLE_INVALID	Identifies the invalid handle of the system timer.
	SYS_TMR_INDEX_0	Timer System Service index definitions.
	SYS_TMR_FLAGS	Defines the types and flags supported by a client timer object.
	SYS_TMR_CLIENT_TOLERANCE	Sets the accepted client tolerance in %.
	SYS_TMR_DRIVER_INDEX	Sets the default timer driver object to be used by the system timer service.

	SYS_TMR_FREQUENCY	Specifies the operating frequency for the system timer, Hz.
	SYS_TMR_FREQUENCY_TOLERANCE	Sets the error tolerance in %.
	SYS_TMR_INTERRUPT_NOTIFICATION	Activates the notification mechanism from within the timer interrupt.
	SYS_TMR_MAX_CLIENT_OBJECTS	Sets the maximum number of timer clients that can be active at a given time.
	SYS_TMR_MODULE_INIT	Sets the default module init value by the system timer service.
	SYS_TMR_UNIT_RESOLUTION	Specifies the internal units for the time keeping.

Description

This section describes the APIs of the Timer System Service Library.

Refer to each section for a detailed description.

a) System Level Interaction Functions

SYS_TMR_Deinitialize Function

Deinitializes the specific module instance of the TMR module

File

[sys_tmr.h](#)

C

```
void SYS_TMR_Deinitialize(SYS_MODULE_OBJ object);
```

Returns

None.

Description

This function deinitializes the specific module instance disabling its operation (and any hardware for driver modules). Resets all of the internal data structures and fields for the specified instance to the default settings.

Remarks

Once the Initialize operation has been called, the De-initialize operation must be called before the Initialize operation can be called again.

Preconditions

The [SYS_TMR_Initialize](#) function should have been called before calling this function.

Example

```
SYS_MODULE_OBJ    object;    // Returned from SYS_TMR_Initialize
SYS_STATUS        status;

SYS_TMR_Deinitialize (object);

status = SYS_TMR_Status (object);

if (SYS_MODULE_UNINITIALIZED == status)
{
    // Check again later if you need to know
    // when the SYS TMR is deinitialized.
}
```

Parameters

Parameters	Description
object	SYS TMR object handle, returned from SYS_TMR_Initialize

Function

```
void SYS_TMR_Deinitialize ( SYS_MODULE_OBJ object )
```

SYS_TMR_Initialize Function

Initializes hardware and data for the instance of the Timer module and opens the specific module instance.

File

[sys_tmr.h](#)

C

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

Returns

If successful, returns a valid handle to an object. Otherwise, it returns [SYS_MODULE_OBJ_INVALID](#). The returned object must be passed as argument to [SYS_TMR_Reinitialize](#), [SYS_TMR_Deinitialize](#), [SYS_TMR_Tasks](#) and [SYS_TMR_Status](#) routines.

Description

This function initializes hardware for the instance of the Timer module, using the specified hardware initialization data. It also initializes any internal data structures.

Remarks

This routine should only be called once during system initialization.

Preconditions

None.

Example

```
SYS_MODULE_OBJ  objectHandle;
SYS_TMR_INIT    initConfig;

// Populate the timer initialization structure
initConfig.moduleInit.value      = SYS_MODULE_POWER_RUN_FULL;
initConfig.drvIndex              = DRV_TMR_INDEX_0;
initConfig.tmrFreq               = 1000;

objectHandle = SYS_TMR_Initialize (SYS_TMR_INDEX_0, (SYS_MODULE_INIT*)&initConfig);
if (SYS_MODULE_OBJ_INVALID == objectHandle)
{
    // Handle error
}
```

Parameters

Parameters	Description
index	Index for the instance to be initialized
init	Pointer to a data structure containing any data necessary to initialize the sys timer. This pointer may be null if no data is required because static overrides have been provided.

Function

```
SYS_MODULE_OBJ SYS_TMR_Initialize ( const SYS_MODULE_INDEX      index,
const          SYS_MODULE_INIT * const  init )
```

SYS_TMR_Status Function

Returns status of the specific instance of the Timer module.

File

[sys_tmr.h](#)

C

```
SYS_STATUS SYS_TMR_Status(SYS_MODULE_OBJ object);
```

Returns

- SYS_STATUS_READY - Indicates that the driver initialization is complete and ready to be used.

Description

This function returns the status of the specific module instance.

Remarks

Any value greater than `SYS_STATUS_READY` is also a normal running state in which the driver is ready to accept new operations

- `SYS_STATUS_ERROR` - Indicates that the driver is in an error state.

Any value less than `SYS_STATUS_ERROR` is also an error state.

- `SYS_MODULE_UNINITIALIZED` - Indicates that the driver has been deinitialized.

This value is less than `SYS_STATUS_ERROR`.

None.

Preconditions

The `SYS_TMR_Initialize` function should have been called before calling this function.

Example

```
SYS_MODULE_OBJ    object;    // Returned from SYS_TMR_Initialize
SYS_STATUS        tmrStatus;

tmrStatus = SYS_TMR_Status (object);

if ( SYS_STATUS_ERROR <= tmrStatus )
{
    // Handle error
}
```

Parameters

Parameters	Description
object	SYS TMR object handle, returned from SYS_TMR_Initialize

Function

`SYS_STATUS` `SYS_TMR_Status` (`SYS_MODULE_OBJ` object)

SYS_TMR_Tasks Function

Maintains the system timer's state machine and implements its ISR.

File

[sys_tmr.h](#)

C

```
void SYS_TMR_Tasks(SYS_MODULE_OBJ object);
```

Returns

None.

Description

This function is used to maintain the system timer's internal state machine. It should be always be called from 'Sys_Tasks' function.

Remarks

This function is normally not called directly by an application. It is called by the system's Tasks routine ([SYS_Tasks](#)) or by the appropriate raw ISR.

Preconditions

The `SYS_TMR_Initialize` function must have been called for the specified TMR driver instance.

Example

```
SYS_MODULE_OBJ    object;    // Returned from SYS_TMR_Initialize

while (true)
{
    SYS_TMR_Tasks ( object );

    // Do other tasks
}
```


Parameters

Parameters	Description
object	SYS TMR object handle, returned from SYS_TMR_Initialize

Function

void SYS_TMR_Tasks ([SYS_MODULE_OBJ](#) object)

b) Timed Callback Functions

SYS_TMR_CallbackStop Function

Stops a periodic timer object.

File

[sys_tmr.h](#)

C

```
void SYS_TMR_CallbackStop(SYS_TMR_HANDLE handle);
```

Returns

None.

Description

This function deletes a previously created periodic timer object. All the associated resources are released.

Remarks

This function is equivalent to [SYS_TMR_ObjectDelete](#)(handle);.

Preconditions

The [SYS_TMR_CallbackPeriodic](#) function should have been called to obtain a valid timer handle.

Example

```
int count, state;
void Test_Callback ( uintptr_t context, uint32_t currTick )
{
    count++;
}

void DelayTask ( )
{
    SYS_TMR_HANDLE handle;
    switch (state)
    {
        case 1:
            handle = SYS_TMR_CallbackPeriodic ( 20, 1, Test_Callback );
            state++;
            break;
        case 2:
            if ( count > 100 )
            {
                SYS_TMR_CallbackStop ( handle );
                state++;
            }
            count++;
            break;
    }
}
```

Parameters

Parameters	Description
handle	A valid periodic timer handle, returned by a SYS_TMR_CallbackPeriodic call.

Function

void SYS_TMR_CallbackStop ([SYS_TMR_HANDLE](#) handle)

SYS_TMR_CallbackPeriodic Function

Creates a permanent timer object that calls back to the client periodically.

File

[sys_tmr.h](#)

C

```
SYS_TMR_HANDLE SYS_TMR_CallbackPeriodic(uint32_t periodMs, uintptr_t context, SYS_TMR_CALLBACK callback);
```

Returns

A valid timer object handle of type [SYS_TMR_HANDLE](#) if the call succeeds. [SYS_TMR_HANDLE_INVALID](#) otherwise.

Description

This function creates a timer object that will continuously notify the client once the time-out occurs. The timer object will be deleted by a call to [SYS_TMR_CallbackStop](#)

Remarks

The created timer object is persistent and needs to be deleted with [SYS_TMR_CallbackStop](#) when no longer needed.

This function is equivalent to [SYS_TMR_ObjectCreate](#)(periodMs, context, callback, SYS_TMR_FLAG_PERIODIC);

Preconditions

The [SYS_TMR_Initialize](#) function should have been called before calling this function.

Example

```
SYS_TMR_HANDLE handle20ms, handle30ms;

void Test_Callback ( uintptr_t context, uint32_t currTick );

handle20ms = SYS_TMR_CallbackPeriodic ( 20, 1, Test_Callback );
handle30ms = SYS_TMR_CallbackPeriodic ( 20, 2, Test_Callback );

void Test_Callback ( uintptr_t context, uint32_t currTick )
{
    if ( context == 1 )
    {
        //20 ms
    }
    else
    {
        //30 ms
    }
}
```

Parameters

Parameters	Description
periodMs	Periodic delay in milliseconds
context	A client parameter that's passed in the callback function. This will help to identify the callback.
callback	Pointer to a callback routine that will be called periodically once the time-out occurs.

Function

```
SYS\_TMR\_HANDLE SYS_TMR_CallbackPeriodic ( uint32_t periodMs, uintptr_t context,
SYS\_TMR\_CALLBACK callback )
```

SYS_TMR_CallbackSingle Function

Creates a one-shot/single callback timer object.

File

[sys_tmr.h](#)

C

```
SYS_TMR_HANDLE SYS_TMR_CallbackSingle(uint32_t periodMs, uintptr_t context, SYS_TMR_CALLBACK callback);
```

Returns

A valid timer object handle of type [SYS_TMR_HANDLE](#) if the call succeeds. [SYS_TMR_HANDLE_INVALID](#) otherwise.

Description

This function creates a self-destroying one-shot/single timer object that will notify the client once the time-out occurs. Once the time-out occurs and the notification is sent the timer object will be invalidated.

Remarks

The timer object will delete itself and free the associated resources. The [SYS_TMR_HANDLE](#) will be no longer valid.

However, [SYS_TMR_CallbackStop\(\)](#) can be called to kill the timer object before its time-out expires, if not needed anymore.

This function is equivalent to [SYS_TMR_ObjectCreate](#)(periodMs, context, callback, SYS_TMR_FLAG_SINGLE | SYS_TMR_FLAG_AUTO_DELETE);

Preconditions

The [SYS_TMR_Initialize](#) function should have been called before calling this function.

Example

```
SYS_TMR_HANDLE handle;
void Test_Callback20ms ( uintptr_t context, uint32_t currTick );

handle = SYS_TMR_CallbackSingle ( 20, 1, Test_Callback20ms );
```

Parameters

Parameters	Description
periodMs	Periodic delay in milliseconds
context	A client parameter that's passed in the callback function. This will help to identify the callback.
callback	Pointer to a callback routine which will be called once

Function

```
SYS\_TMR\_HANDLE SYS_TMR_CallbackSingle ( uint32_t periodMs, uintptr_t context,
SYS\_TMR\_CALLBACK callback )
```

c) Timed Delay Functions**SYS_TMR_DelayStatusGet Function**

Checks the status of the previously requested delay timer object.

File

[sys_tmr.h](#)

C

```
bool SYS_TMR_DelayStatusGet (SYS_TMR_HANDLE handle);
```

Returns

- true - If the delay is elapsed
- false - If the delay is not elapsed

Description

This function checks the status of the previously requested delay timer object.

Remarks

Once the required delay is achieved and [SYS_TMR_DelayStatusGet](#) returns true the [SYS_TMR_HANDLE](#) will be no longer valid and the created timer object is deleted.

Preconditions

The [SYS_TMR_DelayMS](#) function should have been called before calling this function.

Example

```
SYS_TMR_HANDLE tmrHandle;

case APP_ADD_DELAY:
    tmrHandle = SYS_TMR_DelayMS ( 50 );
    state = APP_CHECK_DELAY;
    break;
case APP_CHECK_DELAY:
    if ( SYS_TMR_DelayStatusGet ( tmrHandle ) == true )
    {
        state = APP_DELAY_COMPLETE;
    }
    break;
```

Parameters

Parameters	Description
handle	A valid timer object handle, returned by SYS_TMR_DelayMS

Function

```
bool SYS_TMR_DelayStatusGet ( SYS\_TMR\_HANDLE handle )
```

SYS_TMR_DelayMS Function

Creates a timer object that times out after the specified delay.

File

[sys_tmr.h](#)

C

```
SYS_TMR_HANDLE SYS_TMR_DelayMS(uint32_t delayMs);
```

Returns

A valid timer handler of type [SYS_TMR_HANDLE](#) is returned if the call is successful. [SYS_TMR_HANDLE_INVALID](#) otherwise.

Description

This function creates a self-destroying timer object that times out after the required delay. The delay status can be checked using [SYS_TMR_DelayStatusGet\(\)](#);

Remarks

Once the required delay is achieved and [SYS_TMR_DelayStatusGet\(\)](#) returns true the [SYS_TMR_HANDLE](#) will be no longer valid and the created timer object is deleted.

However, [SYS_TMR_CallbackStop\(\)](#) can be called to kill the timer object before its time-out expires, if not needed anymore.

This function is equivalent to [SYS_TMR_ObjectCreate](#)(delayMs, 0, 0, SYS_TMR_FLAG_SINGLE | SYS_TMR_FLAG_AUTO_DELETE);

Preconditions

The [SYS_TMR_Initialize](#) function should have been called before calling this function.

Example

```
SYS_TMR_HANDLE tmrHandle;

case APP_ADD_DELAY:
    tmrHandle = SYS_TMR_DelayMS ( 50 );
    state = APP_CHECK_DELAY;
    break;
case APP_CHECK_DELAY:
    if ( SYS_TMR_DelayStatusGet (tmrHandle ) == true )
    {
        state = APP_DELAY_COMPLETE;
    }
    break;
```

Parameters

Parameters	Description
delayMs	Indicates the delay required in milliseconds

Function

[SYS_TMR_HANDLE](#) SYS_TMR_DelayMS (uint32_t delayMs)

d) Timed Object Functions

SYS_TMR_ObjectCountGet Function

Returns the current millisecond count of a timer object.

File

[sys_tmr.h](#)

C

```
uint32_t SYS_TMR_ObjectCountGet(SYS_TMR_HANDLE handle, uint32_t* pRateMs);
```

Returns

- > 0 - the current timer object millisecond count
- = 0 - time-out has occurred
- < 0 - if invalid timer object

Description

This function returns the current count and the time-out values of a timer object, in milliseconds.

Remarks

This function will destroy the auto-delete timer objects if the time-out has occurred.

Preconditions

The [SYS_TMR_ObjectCreate](#) function should have been called to get a valid timer object handle.

Parameters

Parameters	Description
handle	A valid timer handle, returned by a SYS_TMR_ObjectCreate call.
pRateMs	pointer to address to store the current object reload rate, in milliseconds. Could be NULL, if not needed.

Function

```
uint32_t SYS_TMR_ObjectCountGet ( SYS\_TMR\_HANDLE handle, uint32_t* pRateMs)
```

SYS_TMR_ObjectCreate Function

Creates a new timer object.

File

[sys_tmr.h](#)

C

```
SYS_TMR_HANDLE SYS_TMR_ObjectCreate(uint32_t periodMs, uintptr_t context, SYS_TMR_CALLBACK callback,  
SYS_TMR_FLAGS flags);
```

Returns

A valid timer object handle of type [SYS_TMR_HANDLE](#) if the call succeeds. [SYS_TMR_HANDLE_INVALID](#) otherwise.

Description

This function creates a new timer object of requested type. If a callback is supplied, it will notify the user once the time-out occurs. The timer object will be deleted by a call to [SYS_TMR_ObjectDelete](#)

Remarks

The created timer object needs to be deleted with [SYS_TMR_ObjectDelete](#) when no longer needed (unless is auto-delete type).

The auto-delete objects are deleted if the time-out has occurred and the client has been notified, meaning:

- The object has a notification/callback function, which has been called upon time-out.
- [SYS_TMR_ObjectCountGet\(\)](#)/[SYS_TMR_DelayStatusGet](#) has been called

Preconditions

The [SYS_TMR_Initialize](#) function should have been called before calling this function.

Example

```
SYS_TMR_HANDLE handle20ms, handle30ms;

void Test_Callback ( uintptr_t context, uint32_t currTick );

handle20ms = SYS_TMR_ObjectCreate ( 20, 1, Test_Callback, SYS_TMR_FLAG_PERIODIC );
handle30ms = SYS_TMR_ObjectCreate ( 30, 2, Test_Callback, SYS_TMR_FLAG_PERIODIC );

void Test_Callback ( uintptr_t context, uint32_t currTick )
{
    if ( context == 1 )
    {
        //20 ms
    }
    else
    {
        //30 ms
    }
}

// when done with the timer objects
SYS_TMR_ObjectDelete(handle20ms);
SYS_TMR_ObjectDelete(handle30ms);
```

Parameters

Parameters	Description
periodMs	Periodic delay in milliseconds
context	A client parameter that's passed in the callback function. This will help to identify the callback. Can be 0 if not needed.
callback	Pointer to a callback routine that will be called periodically once the time-out occurs. Can be NULL if not needed.
flags	multiple flags specifying what type of object is created: <ul style="list-style-type: none"> • SYS_TMR_FLAG_PERIODIC - a periodic timer object is requested • SYS_TMR_FLAG_SINGLE - a single timer object is requested • SYS_TMR_FLAG_AUTO_DELETE - auto-delete object (SYS_TMR_FLAG_SINGLE only)

Function

```
SYS\_TMR\_HANDLE SYS_TMR_ObjectCreate ( uint32\_t periodMs, uintptr\_t context,
SYS\_TMR\_CALLBACK callback, SYS\_TMR\_FLAGS flags)
```

SYS_TMR_ObjectDelete Function

Deletes an existent timer object.

File

[sys_tmr.h](#)

C

```
void SYS_TMR_ObjectDelete(SYS\_TMR\_HANDLE handle);
```

Returns

None.

Description

This function deletes a timer object and release the associated resources.

Remarks

The created timer objects need to be deleted when it is no longer needed (unless of auto-delete type).

Preconditions

The [SYS_TMR_ObjectCreate](#) function should have been called to get a valid timer object handle.

Parameters

Parameters	Description
handle	A valid timer handle, returned by a SYS_TMR_ObjectCreate call.

Function

```
void SYS_TMR_ObjectDelete ( SYS\_TMR\_HANDLE handle)
```

SYS_TMR_ObjectReload Function

Reloads an existing timer object.

File

[sys_tmr.h](#)

C

```
bool SYS_TMR_ObjectReload(SYS\_TMR\_HANDLE handle, uint32_t periodMs, uintptr_t context, SYS\_TMR\_CALLBACK callback);
```

Returns

- true - If the call succeeds
- false - If the call fails

Description

This function changes the parameters for an already existing timer object. The timer object should be valid.

Remarks

The function works for any kind of timer object. It will fail if the timer object was auto-delete type and the time-out has already expired.

Preconditions

The [SYS_TMR_ObjectCreate](#) function should have been called to get a valid timer object handle.

Parameters

Parameters	Description
handle	A valid timer handle, returned by a SYS_TMR_ObjectCreate call.
periodMs	Periodic delay in milliseconds
context	A client parameter that's passed in the callback function. This will help to identify the callback. Can be '0' if not needed.
callback	Pointer to a callback routine that will be called periodically once the time-out occurs. Can be NULL if not needed.

Function

```
bool SYS_TMR_ObjectReload ( SYS\_TMR\_HANDLE handle, uint32_t periodMs,  
uintptr_t context, SYS\_TMR\_CALLBACK callback)
```

e) Miscellaneous Functions

SYS_TMR_TickCountGet Function

Provides the current counter value.

File

[sys_tmr.h](#)

C

```
uint32_t SYS_TMR_TickCountGet();
```

Returns

The current system timer tick count value.

Description

This function provides the current tick counter value.

Remarks

None.

Preconditions

The [SYS_TMR_Initialize](#) function should have been called before calling this function.

Example

```
uint32_t count;
count = SYS_TMR_TickCountGet ();
```

Function

```
uint32_t SYS_TMR_TickCountGet( void )
```

SYS_TMR_TickCountGetLong Function

Provides the current counter value with 64-bit resolution.

File

[sys_tmr.h](#)

C

```
uint64_t SYS_TMR_TickCountGetLong();
```

Returns

The current system timer tick count value.

Description

This function provides the current tick counter value as a 64-bit value.

Remarks

None.

Preconditions

The [SYS_TMR_Initialize](#) function should have been called before calling this function.

Example

```
uint64_t lCount;
lCount = SYS_TMR_TickCountGetLong ();
```

Function

```
uint64_t SYS_TMR_TickCountGetLong( void )
```

SYS_TMR_TickCounterFrequencyGet Function

Provides the number of ticks per second

File

[sys_tmr.h](#)

C

```
uint32_t SYS_TMR_TickCounterFrequencyGet();
```

Returns

The current system timer ticks per second.

Description

This function provides the system tick resolution, the number of ticks per second

Remarks

None.

Preconditions

The [SYS_TMR_Initialize](#) function should have been called before calling this function.

Example

```
uint32_t tickFrequencyHz;  
  
tickFrequencyHz = SYS_TMR_TickCounterFrequencyGet ( );
```

Function

```
uint32_t SYS_TMR_TickCounterFrequencyGet ( void )
```

SYS_TMR_SystemCountGet Function

Provides the current system time count value.

File

[sys_tmr.h](#)

C

```
uint64_t SYS_TMR_SystemCountGet();
```

Returns

The current system count value.

Description

This function provides the current system time count value with the greatest possible resolution. It is the number of counts from the system start up as counted with the system count frequency ([SYS_TMR_SystemCountFrequencyGet](#)).

Remarks

None.

Preconditions

The [SYS_TMR_Initialize](#) function should have been called before calling this function.

Example

```
uint64_t sysCount;  
sysCount = SYS_TMR_SystemCountGet ( );  
// the system has been running for sysCount/SYS_TMR_SystemCountFrequencyGet seconds
```

Function

```
uint64_t SYS_TMR_SystemCountGet( void )
```

SYS_TMR_SystemCountFrequencyGet Function

Provides the current system count frequency.

File

[sys_tmr.h](#)

C

```
uint32_t SYS_TMR_SystemCountFrequencyGet();
```

Returns

The current system frequency value in Hertz.

Description

This function provides the current system count frequency in Hz. The tick count measurement is based on this value. This is the resolution of the system time base.

Remarks

None.

Preconditions

The [SYS_TMR_Initialize](#) function should have been called before calling this function.

Example

```
uint32_t sysCountFreq;  
sysCountFreq = SYS_TMR_SystemCountFrequencyGet();
```

Function

```
uint32_t SYS_TMR_SystemCountFrequencyGet( void )
```

SYS_TMR_ModuleStatusGet Function

Returns status of the specific instance of the Timer module.

File

[sys_tmr.h](#)

C

```
SYS_STATUS SYS_TMR_ModuleStatusGet(const SYS_MODULE_INDEX index);
```

Returns

- SYS_STATUS_READY - Indicates that the service initialization is complete and the timer is ready to be used.
- SYS_STATUS_ERROR - Indicates that the timer service is in an error state

Description

This function returns the status of the specific module instance. It uses the index of the instance rather than an object handle.

Remarks

Any value less than SYS_STATUS_ERROR is also an error state.

- SYS_MODULE_UNINITIALIZED - Indicates that the timer service has been deinitialized

None.

Preconditions

The [SYS_TMR_Initialize](#) function should have been called before calling this function.

Example

```
SYS_STATUS tmrStatus = SYS_TMR_ModuleStatusGet (SYS_TMR_INDEX_0);  
  
if ( SYS_STATUS_ERROR <= tmrStatus )  
{  
    // Handle error  
}
```

Parameters

Parameters	Description
object	SYS TMR object handle, returned from SYS_TMR_Initialize

Function

```
SYS_STATUS SYS_TMR_ModuleStatusGet ( const SYS_MODULE_INDEX index )
```

f) Data Types and Constants

SYS_TMR_CALLBACK Type

This data type defines a pointer callback function.

File

[sys_tmr.h](#)

C

```
typedef void (* SYS_TMR_CALLBACK)(uintptr_t context, uint32_t currTick);
```

Description

SYS TMR Callback Function

Defines a pointer to a client notification function that's called when a timer object times out.

Remarks

The client notification function should be kept as short as possible and non blocking. It is meant for some quick operation (like flag setting) and not for extensive operation. It can be called within interrupt context.

Parameters

Parameters	Description
context	A client parameter that's passed in the callback function. This will help to identify the callback.
currTick	The current system tick when the notification is called.

SYS_TMR_HANDLE Type

Identifies a client timer object.

File

[sys_tmr.h](#)

C

```
typedef uintptr_t SYS_TMR_HANDLE;
```

Description

SYS TMR Handle

This handle identifies a registered instance of a client timer object. Access to a timer object parameters and functionality has to use a valid client timer handle.

Remarks

None.

SYS_TMR_INIT Structure

Identifies the system timer initialize structure.

File

[sys_tmr.h](#)

C

```
typedef struct {  
    SYS_MODULE_INIT moduleInit;  
    SYS_MODULE_INDEX drvIndex;  
    uint32_t tmrFreq;  
} SYS_TMR_INIT;
```

Members

Members	Description
SYS_MODULE_INIT moduleInit;	System module initialization
SYS_MODULE_INDEX drvIndex;	Driver Module index
uint32_t tmrFreq;	System timer required frequency, Hz This is the base frequency for all time keeping

Description

SYS TMR Initialize structure

This structure identifies the system timer initialize structure.

Remarks

None.

SYS_TMR_HANDLE_INVALID Macro

Identifies the invalid handle of the system timer.

File

[sys_tmr.h](#)

C

```
#define SYS_TMR_HANDLE_INVALID ( ( uintptr_t ) -1 )
```

Description

SYS Timer invalid handle macro definition

This enumeration identifies the invalid handle of the system timer.

Remarks

None.

SYS_TMR_INDEX_0 Macro

Timer System Service index definitions.

File

[sys_tmr.h](#)

C

```
#define SYS_TMR_INDEX_0 0
```

Description

SYS Timer Module Index Numbers

These constants provide Timer System Service index definitions.

Remarks

These constants should be used in place of hard-coded numeric literals.

SYS_TMR_FLAGS Enumeration

Defines the types and flags supported by a client timer object.

File

[sys_tmr.h](#)

C

```
typedef enum {  
    SYS_TMR_FLAG_PERIODIC = 0x0001,  
    SYS_TMR_FLAG_SINGLE = 0x0000,  
    SYS_TMR_FLAG_AUTO_DELETE = 0x1000  
} SYS_TMR_FLAGS;
```

Members

Members	Description
SYS_TMR_FLAG_PERIODIC = 0x0001	the timer is of periodic type as the time-out is achieved the timer is reloaded so it will periodically provide notifications
SYS_TMR_FLAG_SINGLE = 0x0000	the timer is a single shot type it fires just once, when the time-out occurs
SYS_TMR_FLAG_AUTO_DELETE = 0x1000	as time-out occurs, the timer deletes itself only SINGLE and DELAY timer types support this feature

Description

SYS TMR Flags

This enumerated type describes the types and run time flags supported by a client timer object.

Remarks

Multiple flags can be ORed.

SYS_TMR_CLIENT_TOLERANCE Macro

Sets the accepted client tolerance in %.

File

[sys_tmr_config_template.h](#)

C

```
#define SYS_TMR_CLIENT_TOLERANCE (10)
```

Description

Client accepted Tolerance configuration

This macro sets the percentage of the accepted tolerance for the requested client delay. If the system timer cannot achieve a delay within the specified tolerance it will reject the client request.

Remarks

This error depends on the actual frequency that the system timer uses as a time base.

SYS_TMR_DRIVER_INDEX Macro

Sets the default timer driver object to be used by the system timer service.

File

[sys_tmr_config_template.h](#)

C

```
#define SYS_TMR_DRIVER_INDEX (DRV_TMR_INDEX_0)
```

Description

Default timer driver object configuration

This macro sets the default timer driver object to be used by the system timer service.

Remarks

This value can be overridden by a run time initialization value.

SYS_TMR_FREQUENCY Macro

Specifies the operating frequency for the system timer, Hz.

File

[sys_tmr_config_template.h](#)

C

```
#define SYS_TMR_FREQUENCY (1250)
```

Description

System timer running frequency

This macro sets the operating frequency for the system timer, in Hz. All the timing is maintained based on this frequency. This is actually the rate of the interrupt that's requested to the hardware timer that the system timer uses.

The higher the rate, the better time keeping but the operation of the CPU will be interrupted more frequently (if the underlying timer driver is working in interrupts) and the result will be a greater overhead. Therefore is not recommended to go higher than 1 KHz (below 1 ms) when the timer driver works in interrupts.

Remarks

The system timer cannot obtain timing with a finer resolution than this frequency.

This value can be overridden by a run time initialization value.

The actual running frequency depends on the timer driver running frequency. Select a value that makes sense to minimize the error (see [SYS_TMR_FREQUENCY_TOLERANCE](#) notes).

SYS_TMR_FREQUENCY_TOLERANCE Macro

Sets the error tolerance in %.

File

[sys_tmr_config_template.h](#)

C

```
#define SYS_TMR_FREQUENCY_TOLERANCE (10)
```

Description

Error Tolerance configuration

This macro sets the accepted tolerance percentage for the requested system timer frequency ([SYS_TMR_FREQUENCY](#)).

Remarks

This error depends on the time base that the system timer uses (i.e. the underlying timer driver timing frequency). The actual running frequency for the system timer may be slightly different from the requested [SYS_TMR_FREQUENCY](#).

SYS_TMR_INTERRUPT_NOTIFICATION Macro

Activates the notification mechanism from within the timer interrupt.

File

[sys_tmr_config_template.h](#)

C

```
#define SYS_TMR_INTERRUPT_NOTIFICATION (false)
```

Description

Interrupt Notification Mode configuration

This macro enables/disables the notification towards the timer clients to be delivered directly from the underlying time base interrupt routine (the routine that notifies the system timer itself that another system tick has elapsed).

Enabling interrupt notifications provide a more accurate and responsive callback towards the clients. However the client callbacks need to be non blocking and as short as possible (the client callback are meant for setting a flag to indicate that further processing is needed and not to execute lengthy processing inside the callback itself).

- true - Client notification in interrupt enabled
- false - Client notification in interrupt disabled

Remarks

None.

SYS_TMR_MAX_CLIENT_OBJECTS Macro

Sets the maximum number of timer clients that can be active at a given time.

File

[sys_tmr_config_template.h](#)

C

```
#define SYS_TMR_MAX_CLIENT_OBJECTS (5)
```

Description

Maximum client timer objects configuration

This macro sets the maximum number of clients that the Timer system service can support.

Remarks

None.

SYS_TMR_MODULE_INIT Macro

Sets the default module init value by the system timer service.

File

[sys_tmr_config_template.h](#)

C

```
#define SYS_TMR_MODULE_INIT (SYS_MODULE_POWER_RUN_FULL)
```

Description

Default module init object configuration

This macro sets the default module init value by the system timer service.

Remarks

This value can be overridden by a run time initialization value.

SYS_TMR_UNIT_RESOLUTION Macro

Specifies the internal units for the time keeping.

File

[sys_tmr_config_template.h](#)

C

```
#define SYS_TMR_UNIT_RESOLUTION (10000)
```

Description

System timer internal units resolution

This macro sets the system timer internal time units, in Hz. The higher the value, the better resolution but the range of the delays that can be obtained is shortened.

A value too low will limit the resolution of delays that can be obtained. In order to be able to obtain milliseconds delays the value should be greater than 1000 Hz.

The internal timing calculation is done with integer arithmetic so in order to do a precise enough calculation for ms values, a finer time unit needs to be used.

Remarks

10000 Hz (meaning 0.1 ms units) is the recommended value because the delays than need to be achieved are in the ms range.

The service tries to achieve enough resolution for delays in the range of milliseconds. That means that this value cannot be < 1000 and the service will fail to initialize if such a value is attempted.

The setting affects the range of delays that can be obtained. $\text{delay} < (2^{32} - 1) / \text{SYS_TMR_RESOLUTION}$ gives the maximum delay value, in seconds, that can be achieved. For example: $\text{delay} < 119.3$ hours (for $\text{SYS_TMR_RESOLUTION} == 10000$).

Files

Files

Name	Description
sys_tmr.h	Timer System Service interface definition.
sys_tmr_config_template.h	Contains configuration definitions that are common to timer system services and aggregates the configuration files for the system services.

Description

This section lists the source and header files used by the library.




















sys_tmr.h

Timer System Service interface definition.

Enumerations

	Name	Description
	SYS_TMR_FLAGS	Defines the types and flags supported by a client timer object.

Functions

	Name	Description
	SYS_TMR_CallbackPeriodic	Creates a permanent timer object that calls back to the client periodically.
	SYS_TMR_CallbackSingle	Creates a one-shot/single callback timer object.
	SYS_TMR_CallbackStop	Stops a periodic timer object.
	SYS_TMR_Deinitialize	Deinitializes the specific module instance of the TMR module
	SYS_TMR_DelayMS	Creates a timer object that times out after the specified delay.
	SYS_TMR_DelayStatusGet	Checks the status of the previously requested delay timer object.
	SYS_TMR_Initialize	Initializes hardware and data for the instance of the Timer module and opens the specific module instance.
	SYS_TMR_ModuleStatusGet	Returns status of the specific instance of the Timer module.
	SYS_TMR_ObjectCountGet	Returns the current millisecond count of a timer object.
	SYS_TMR_ObjectCreate	Creates a new timer object.
	SYS_TMR_ObjectDelete	Deletes an existent timer object.
	SYS_TMR_ObjectReload	Reloads an existing timer object.
	SYS_TMR_Status	Returns status of the specific instance of the Timer module.
	SYS_TMR_SystemCountFrequencyGet	Provides the current system count frequency.
	SYS_TMR_SystemCountGet	Provides the current system time count value.
	SYS_TMR_Tasks	Maintains the system timer's state machine and implements its ISR.
	SYS_TMR_TickCounterFrequencyGet	Provides the number of ticks per second
	SYS_TMR_TickCountGet	Provides the current counter value.
	SYS_TMR_TickCountGetLong	Provides the current counter value with 64-bit resolution.

Macros

	Name	Description
	SYS_TMR_HANDLE_INVALID	Identifies the invalid handle of the system timer.
	SYS_TMR_INDEX_0	Timer System Service index definitions.

Structures

	Name	Description
	SYS_TMR_INIT	Identifies the system timer initialize structure.

Types

	Name	Description
	SYS_TMR_CALLBACK	This data type defines a pointer callback function.
	SYS_TMR_HANDLE	Identifies a client timer object.

Description

Timer System Service Interface Definition

This file contains the interface definition for the Timer System Service. It provides a way to interact with the Timer subsystem to manage the timing requests supported by the system.

File Name

sys_tmr.h

Company

Microchip Technology Inc.

sys_tmr_config_template.h

Contains configuration definitions that are common to timer system services and aggregates the configuration files for the system services.

Macros

	Name	Description
	_SYS_TMR_CONFIG_TEMPLATE_H	This is macro _SYS_TMR_CONFIG_TEMPLATE_H .
	SYS_TMR_CLIENT_TOLERANCE	Sets the accepted client tolerance in %.
	SYS_TMR_DRIVER_INDEX	Sets the default timer driver object to be used by the system timer service.
	SYS_TMR_FREQUENCY	Specifies the operating frequency for the system timer, Hz.
	SYS_TMR_FREQUENCY_TOLERANCE	Sets the error tolerance in %.
	SYS_TMR_INTERRUPT_NOTIFICATION	Activates the notification mechanism from within the timer interrupt.
	SYS_TMR_MAX_CLIENT_OBJECTS	Sets the maximum number of timer clients that can be active at a given time.
	SYS_TMR_MODULE_INIT	Sets the default module init value by the system timer service.
	SYS_TMR_UNIT_RESOLUTION	Specifies the internal units for the time keeping.

Description

Timer System Service Configuration Definitions for the Template Version

This file contains configuration definitions that are common to timer system services and aggregates the configuration files for the system services.

File Name

sys_tmr_config_template.h

Company

Microchip Technology Inc.

Touch System Service Library

The Touch System Service is a thin layer specifically for abstracting input from various touch drivers. It is designed to interpret the data from the touch driver and translate it into single-touch behavior that can be used by other modules in the framework, such as the Graphics Library.

The Touch System Service supports both resistive touch driver (ADC10BIT) and capacitive touch drivers (MTCH6301 and MTCH36303).

Currently, the Touch System Service is only available to support non-gestural single finger touch screen input.

Introduction

This library provides a low-level abstraction of the Device Control System Service Library that is available on the Microchip family of PIC32 microcontrollers with a convenient C language interface. It can be used to simplify low-level access to the module without the necessity of interacting directly with the module's registers, there by hiding differences from one microcontroller variant to another.

Description

The Touch System Service is accessed via the Message System Service. For details on how to register to the Message System Service, please refer to its section.



Note: The MPLAB Harmony Graphics Composer is designed to automatically configure the Touch System Service and the Message System Service based on the user's request for touch screen input.

Using the Library

This topic describes the basic architecture of the Touch System Service Library and provides information and examples on its use.

Description

Interface Header File: [sys_touch.h](#)

The interface to the Touch System Service library is defined in the [sys_touch.h](#) header file.

Please refer to the What is MPLAB Harmony? section for how the Touch System Service interacts with the framework.

Library Overview

A client needs to register a callback to allow the Messaging System Service to send messages of type "TYPE_TOUCHSCREEN" to it. Since it is a dependency, the Messaging System Service is automatically enabled in the MPLAB Harmony Configurator configuration tree when the Touch System Service is enabled.

For detail with how to interface the Message System Service, please refer to the [Messaging System Service Library](#) section.

Configuring the Library

No configuration is required for the Touch System Service.

Building the Library

This section lists the files that are available in the Touch System Service Library.

Description

The following three tables list and describe the header (.h) and source (.c) files that implement this library. The parent folder for these files is `<install-dir>/framework/system/touch`.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using `#include`) by any code that uses this library.

Source File Name	Description
sys_touch.h	Touch System Service Library API header file.

Required File(s)



All of the required files listed in the following table are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

Source File Name	Description
/src/sys_touch.c	Touch System Service Library implementation.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.






Source File Name	Description
N/A	No optional files are available for the library.

Module Dependencies

The Touch System Service Library is not dependent upon other modules.

Library Interface

a) Functions

	Name	Description
	SYS_TOUCH_CalibrationSet	Sets the calibration values for Touch client specified by moduleIndex.
	SYS_TOUCH_Initialize	Initializes and Enables the Touch Controller.
	SYS_TOUCH_Open	Opens the touch service specified by the moduleIndex and returns a handle to it.
	SYS_TOUCH_TouchMsgGet	Populates the Graphics Object Library (GOL) message structure.
	SYS_TOUCH_Tasks	Maintains the system service's state machine.

b) Data Types and Constants

	Name	Description
	NVM_READ_FUNC	typedef for read function pointer
	NVM_SECTORERASE_FUNC	typedef for sector erase function pointer
	NVM_WRITE_FUNC	typedef for write function pointer
	SYS_TOUCH_HANDLE	Handle for requested Touch interface.
	SYS_TOUCH_INDEX	Identifies the Touch System Service module index definitions.
	SYS_TOUCH_INIT	Identifies the touch attributes supported
	SYS_TOUCH_STATUS	Identifies the current status/state of touch.
	SYS_TOUCH_HANDLE_INVALID	Invalid touch handle.
	TouchShowMessage	macro to draw repeating text
	SYS_INPUT_DEVICE_TYPE	Specifies the different user input devices supported in the library.
	SYS_INPUT_DEVICE_EVENT	Specifies the different user input device events supported in the library.
	_SYS_TOUCH_H	This is macro _SYS_TOUCH_H.

Description

This section describes the APIs of the Touch System Service Library.

a) Functions

SYS_TOUCH_CalibrationSet Function

Sets the calibration values for Touch client specified by moduleIndex.

File

[sys_touch.h](#)

C

```
void SYS_TOUCH_CalibrationSet(SYS_MODULE_INDEX moduleIndex, DRV_TOUCH_SAMPLE_POINTS * samplePoints);
```

Returns

none.

Description

This function sets the calibration values provided by samplePoints for the specified client.

Remarks

None.

Preconditions

Function [SYS_TOUCH_Open](#) must have been called before calling this function.

Example

```
SYS_MODULE_INDEX moduleIndex;
SYS_TOUCH_SAMPLE_POINTS samplePoints;
SYS_TOUCH_CalibrationSet( moduleIndex, samplePoints);
```

Parameters

Parameters	Description
moduleIndex	Identifier for the touch to be opened
samplePoints	sample points

Function

```
void SYS_TOUCH_CalibrationSet( SYS_MODULE_INDEX moduleIndex,
DRV_TOUCH_SAMPLE_POINTS * samplePoints)
```

SYS_TOUCH_Initialize Function

Initializes and Enables the Touch Controller.

File

[sys_touch.h](#)

C

```
SYS_MODULE_OBJ SYS_TOUCH_Initialize(const SYS_MODULE_INDEX moduleIndex, const SYS_MODULE_INIT * const init);
```

Returns

If successful, returns a valid handle to the DMA module object. Otherwise, it returns [SYS_MODULE_OBJ_INVALID](#).

Description

This function Enables the Touch module. Enable/Disable stop in idle mode feature based on the passed parameter value.

This routine initializes the Touch module making it ready for clients to open and use it. The initialization data is specified by the init parameter.

Remarks

This routine must be called before any other TOUCH systems service routines are called.

Not all features are available on all devices. Refer to the specific device data sheet to determine availability.

Preconditions

None.

Example

```
SYS_MODULE_OBJ objectHandle;
SYS_TOUCH_INIT touchInit;

objectHandle = SYS_TOUCH_Initialize(SYS_TOUCH_INDEX_1,
                                   (SYS_MODULE_INIT*)touchInit);
if (SYS_MODULE_OBJ_INVALID == objectHandle)
{
    // Handle error
}
```

Parameters

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

Function

[SYS_MODULE_OBJ](#) SYS_TOUCH_Initialize (const [SYS_MODULE_INIT](#) * const init)

SYS_TOUCH_Open Function

Opens the touch service specified by the moduleIndex and returns a handle to it.

File

[sys_touch.h](#)

C

```
SYS_TOUCH_HANDLE SYS_TOUCH_Open( SYS_MODULE_INDEX moduleIndex );
```

Returns

If successful, the routine returns a valid open-instance handle (a number identifying both the caller and the module instance).

If an error occurs, the return value is [SYS_TOUCH_HANDLE_INVALID](#).

Description

This function opens the touch service specified by the index and provides a handle that must be provided to all other client-level operations to identify the caller.

Remarks

The handle returned is valid until the SYS_TOUCH_lose routine is called.

Preconditions

The [SYS_TOUCH_Initialize](#) function must have been called before calling this function.

Example

```
SYS_TOUCH_HANDLE handle;

handle = SYS_TOUCH_Open(SYS_TOUCH_INDEX_0);
if (SYS_TOUCH_HANDLE_INVALID == handle)
{
    // Unable to open the service
    // May be the service is not initialized or the initialization
    // is not complete.
}
```

Parameters

Parameters	Description
moduleIndex	Identifier for the touch to be opened

Function

[SYS_TOUCH_HANDLE](#) SYS_TOUCH_Open([SYS_MODULE_INDEX](#) moduleIndex)

SYS_TOUCH_TouchMsgGet Function

Populates the Graphics Object Library (GOL) message structure.

File

[sys_touch.h](#)

C

```
void SYS_TOUCH_TouchMsgGet( SYS_MODULE_INDEX moduleIndex );
```

Returns

None.

Description

This function populates the GOL message structure.

Remarks

None.

Preconditions

None.

Function

void SYS_TOUCH_TouchMsgGet([SYS_MODULE_INDEX](#) moduleIndex)

SYS_TOUCH_Tasks Function

Maintains the system service's state machine.

File

[sys_touch.h](#)

C

`void SYS_TOUCH_Tasks (SYS_MODULE_OBJ object) ;`

Returns

None.

Description

This function is used to maintain the Touch system service's internal state machine. This function is specifically designed for non interrupt trigger implementations(polling mode), and should be used only in polling mode. this function should be called from the [SYS_Tasks\(\)](#) function.

Remarks

This function is normally not called directly by an application. It is called by the system's Tasks routine ([SYS_Tasks](#)).

Preconditions

Touch should have been initialized by calling SYS_Touch_Initialize.

Example

```
// 'object' Returned from SYS_Touch_Initialize

while (true)
{
    SYS_Touch_Tasks ((object) );

    // Do other tasks
}
```

Parameters

Parameters	Description
object	Object handle for the Touch module (returned from SYS_Touch_Initialize)

Function

void SYS_Touch_Tasks([SYS_MODULE_OBJ](#) object);

b) Data Types and Constants

NVM_READ_FUNC Type

File

[sys_touch.h](#)

C

```
typedef uint16_t (* NVM_READ_FUNC)(uint32_t);
```

Description

typedef for read function pointer

NVM_SECTORERASE_FUNC Type

File

[sys_touch.h](#)

C

```
typedef void (* NVM_SECTORERASE_FUNC)(uint32_t);
```

Description

typedef for sector erase function pointer

NVM_WRITE_FUNC Type

File

[sys_touch.h](#)

C

```
typedef void (* NVM_WRITE_FUNC)(uint16_t, uint32_t);
```

Description

typedef for write function pointer

SYS_TOUCH_HANDLE Type

Handle for requested Touch interface.

File

[sys_touch.h](#)

C

```
typedef uintptr_t SYS_TOUCH_HANDLE;
```

Description

Touch System Service handle

A Touch handle is returned by a call to the [SYS_TOUCH_Open](#) function. This handle is an identification for the associated touch interface.

The handle once assigned to a client expires when the client calls [SYS_TOUCH_Close](#).

Remarks

None.

SYS_TOUCH_INDEX Enumeration

Identifies the Touch System Service module index definitions.

File

[sys_touch.h](#)

C

```
typedef enum {  
    SYS_TOUCH_INDEX_0,  
    SYS_TOUCH_INDEX_1  
} SYS_TOUCH_INDEX;
```

Members

Members	Description
SYS_TOUCH_INDEX_0	First Touch service instance
SYS_TOUCH_INDEX_1	Second Touch service

Description

SYS TOUCH Module Index Numbers

These constants provide the Touch System Service index definitions.

Remarks

These constants should be used in place of hard-coded numeric literals.

These values should be passed into the [SYS_TOUCH_Initialize](#) and function to identify the driver instance in use.

SYS_TOUCH_INIT Structure

Identifies the touch attributes supported

File

[sys_touch.h](#)

C

```
typedef struct {  
    SYS_MODULE_INDEX driverModuleIndex;  
} SYS_TOUCH_INIT;
```

Members

Members	Description
SYS_MODULE_INDEX driverModuleIndex;	The Module index of the driver to which touch is attached.

Description

Touch System Service Initialization

This data type defines the touch id and relate with the driver that should be used to initialize touch system.

These functions help to align touch with hardware touch driver specific driver function calls.

Remarks

None.

SYS_TOUCH_STATUS Enumeration

Identifies the current status/state of touch.

File

[sys_touch.h](#)

C

```
typedef enum {  
    SYS_TOUCH_ERROR = -1,  
    SYS_TOUCH_UNINITIALIZED = 0,  
    SYS_TOUCH_BUSY = 1,  
    SYS_TOUCH_READY = 2  
} SYS_TOUCH_STATUS;
```

Members

Members	Description
SYS_TOUCH_ERROR = -1	An unspecified error has occurred.

SYS_TOUCH_UNINITIALIZED = 0	The module has not yet been initialized
SYS_TOUCH_BUSY = 1	An operation is currently in progress
SYS_TOUCH_READY = 2	Any previous operations have succeeded and the service is ready for additional operations

Description

Touch System Service status

Identifies the current status/state of touch.

Remarks

This enumeration is the return type for the status routine.

SYS_TOUCH_HANDLE_INVALID Macro

Invalid touch handle.

File

[sys_touch.h](#)

C

```
#define SYS_TOUCH_HANDLE_INVALID (((SYS_TOUCH_HANDLE) -1))
```

Description

Invalid Touch Handle

If the touch system service is unable to allow an additional clients to use it, it must then return the special value SYS_TOUCH_HANDLE_INVALID. Callers should check the handle returned for this value to ensure this value was not returned before attempting to call any other driver routines using the handle.

Remarks

None.

TouchShowMessage Macro

File

[sys_touch.h](#)

C

```
#define TouchShowMessage( pStr, color, x, y, width, height) \
    { \
        GFX_ColorSet(gfxIndex, color); \
        while(GFX_TextStringBoxDraw(gfxIndex, x,y,width,height,pStr, 0, GFX_ALIGN_LEFT) == \
        GFX_STATUS_FAILURE); \
    }
```

Description

macro to draw repeating text

SYS_INPUT_DEVICE_TYPE Enumeration

Specifies the different user input devices supported in the library.

File

[sys_touch.h](#)

C

```
typedef enum {
    TYPE_UNKNOWN = 0,
    TYPE_KEYBOARD,
    TYPE_TOUCHSCREEN
} SYS_INPUT_DEVICE_TYPE;
```

Members

Members	Description
TYPE_UNKNOWN = 0	Unknown device.
TYPE_KEYBOARD	Keyboard.
TYPE_TOUCHSCREEN	Touchscreen.

Description

Input Device Type: SYS_INPUT_DEVICE_TYPE

This enumeration specifies the different user input devices supported in the library.

Remarks

Beta. The enum will be relocated to SYS_INPUT when available.

SYS_INPUT_DEVICE_EVENT Enumeration

Specifies the different user input device events supported in the library.

File

[sys_touch.h](#)

C

```
typedef enum {  
    EVENT_INVALID = 0,  
    EVENT_MOVE,  
    EVENT_PRESS,  
    EVENT_STILLPRESS,  
    EVENT_RELEASE,  
    EVENT_KEYSCAN,  
    EVENT_CHARCODE,  
    EVENT_SET,  
    EVENT_SET_STATE,  
    EVENT_CLR_STATE  
} SYS_INPUT_DEVICE_EVENT;
```

Members

Members	Description
EVENT_INVALID = 0	Invalid event.
EVENT_MOVE	Move event.
EVENT_PRESS	Press event.
EVENT_STILLPRESS	Continuous press event.
EVENT_RELEASE	Release event.
EVENT_KEYSCAN	Key scan event, parameters for the object ID and keyboard scan code will be sent with this event in the GFX_GOL_MESSAGE as parameter.
EVENT_CHARCODE	Character code event. The actual character code will be sent with this event in the GFX_GOL_MESSAGE as parameter.
EVENT_SET	Generic set event.
EVENT_SET_STATE	Generic set state event.
EVENT_CLR_STATE	Generic clear state event.

Description

Input device event: INPUT_DEVICE_EVENT

This enumeration specifies the different user input device events supported in the graphics library.

Remarks

None.

_SYS_TOUCH_H Macro

File

[sys_touch.h](#)

C

```
#define _SYS_TOUCH_H
```

Description

This is macro `_SYS_TOUCH_H`.

Files

Files

Name	Description
sys_touch.h	Touch System Service Implementation.

Description

This section lists the source and header files used by the library.






sys_touch.h

Touch System Service Implementation.

Enumerations

	Name	Description
	SYS_INPUT_DEVICE_EVENT	Specifies the different user input device events supported in the library.
	SYS_INPUT_DEVICE_TYPE	Specifies the different user input devices supported in the library.
	SYS_TOUCH_INDEX	Identifies the Touch System Service module index definitions.
	SYS_TOUCH_STATUS	Identifies the current status/state of touch.

Functions

	Name	Description
	SYS_TOUCH_CalibrationSet	Sets the calibration values for Touch client specified by moduleIndex.
	SYS_TOUCH_Initialize	Initializes and Enables the Touch Controller.
	SYS_TOUCH_Open	Opens the touch service specified by the moduleIndex and returns a handle to it.
	SYS_TOUCH_Tasks	Maintains the system service's state machine.
	SYS_TOUCH_TouchMsgGet	Populates the Graphics Object Library (GOL) message structure.

Macros

	Name	Description
	_SYS_TOUCH_H	This is macro <code>_SYS_TOUCH_H</code> .
	SYS_TOUCH_HANDLE_INVALID	Invalid touch handle.
	TouchShowMessage	macro to draw repeating text

Structures

	Name	Description
	SYS_TOUCH_INIT	Identifies the touch attributes supported

Types

	Name	Description
	NVM_READ_FUNC	typedef for read function pointer
	NVM_SECTORERASE_FUNC	typedef for sector erase function pointer
	NVM_WRITE_FUNC	typedef for write function pointer
	SYS_TOUCH_HANDLE	Handle for requested Touch interface.

Description

Touch System Service Implementation

The Touch System Service provides a simple interface to manage the touch screen drivers. This file implements the core interface routines for the Touch System Service. This is a resistive touch screen driver that is using the Microchip Graphics Library. The calibration values are automatically checked (by reading a specific memory location on the non-volatile memory) when initializing the module if the function pointers to the read and write callback functions are initialized. If the read value is invalid calibration will automatically be executed. Otherwise, the calibration values will be

loaded and used. The driver assumes that the application side provides the read and write routines to a non-volatile memory. If the callback functions are not initialized, the calibration routine will always be called at startup to initialize the global calibration values. This driver assumes that the Graphics Library is initialized and will be using the default font of the library.

File Name

sys_touch.c

Company

Microchip Technology Inc.

Watchdog Timer System Service Library

This section describes the Watchdog Timer (WDT) System Service Library.

Introduction

This library provides a low-level abstraction of the Watchdog Timer System Service Library that is available on the Microchip family of microcontrollers with a convenient C language interface. It can be used to simplify low-level access to the module without the necessity of interacting directly with the module's registers, there by abstracting differences from one microcontroller variant to another.

Description

The primary function of the Watchdog Timer (WDT) is to reset the microcontroller, in the event of a software malfunction, by resetting the device if it has not been cleared by software. To ensure that application does not hang, the application is required to reset the timer periodically. It can also be used to wake the device from Sleep or Idle mode. The WDT is a free-running timer which uses the low-power RC oscillator and requires no external components. Therefore, the WDT will continue to operate even if the system's primary clock source is stopped.

Using the Library

This topic describes the basic architecture of the Watchdog Timer System Service Library and provides information and examples on its use.

Description

Interface Header File: `sys_wdt.h`

The interface to the Watchdog Timer System Service library is defined in the `sys_wdt.h` header file. This file is included by the `sys.h` file. Any C language source (.c) file that uses the Watchdog Timer System Service library should include `sys.h`.

Please refer to the What is MPLAB Harmony? section for how the System Service interacts with the framework.

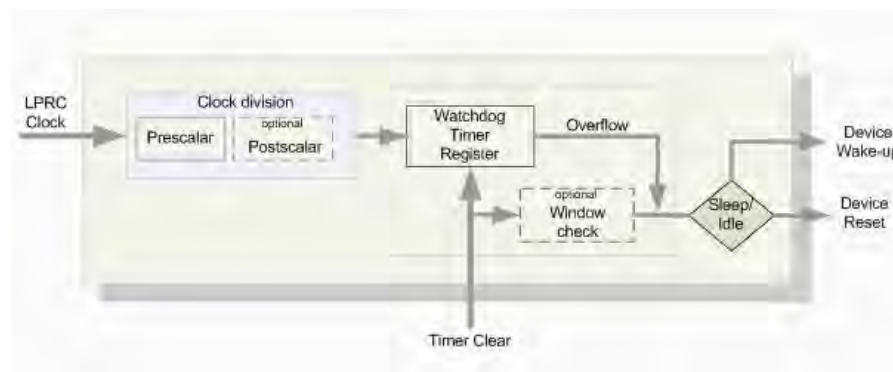
Abstraction Model

This library provides a low-level abstraction of the Watchdog Timer System Service Library. This topic describes how that abstraction is modeled in software and introduces the library's interface.

Description

The Watchdog Timer uses the internal Low-Power RC (LPRC) Oscillator as the source of clock. The clock is divided by the configured prescaler value. There may be one more postscaler divisors and these should be set through the Configuration bits. The divided clock is then used to increment a counter. If the software does not clear the counter in time, the counter overflows and that will result in reset in normal mode. In Sleep or Idle mode, the overflow will result in a device wake-up. In Windowed mode, resetting the counter when the count is not in the specified window will also lead to a reset.

Watchdog Timer Software Abstraction Block Diagram



Library Overview

Refer to the [System Service Introduction](#) section for how the system services operates in a system.

The library interface routines are divided into various sub-sections, which address one of the blocks of the overall operation of the Watchdog Timer module.

Library Interface Section	Description
Enable and Disable Functions	APIs to enable and disable the Watchdog Timer

Clearing the Timer Functions

API to clear the Watchdog Timer

How the Library Works

Provides information on how the library works.

Description

Watchdog Timer Initialization:

For devices that have Windowed mode, use the argument to control the mode. The argument will be discarded for devices that do not have Windowed mode.

```
//Enable the Watchdog Timer with Window mode enabled
```

```
SYS_WDT_Enable(true);
```

```
//Enable the Watchdog Timer with Window mode disabled
```

```
SYS_WDT_Enable(false);
```

```
//Some code
```

```
SYS_WDT_Disable();
```



Note: The previous code example will not work if the Watchdog Timer is controlled through the Configuration bits.

Service the Watchdog Timer:

To ensure the normal operation of the system, the software must periodically clear the Watchdog Timer.

```
SYS_WDT_TimerClear();
```



Note: Not all modes are available on all devices. Please refer to the specific device data sheet to determine the supported modes.

Configuring the Library

Macros

	Name	Description
	WDT_PLIB_ID	Configures the module's ID used by the peripheral library.

Description

The configuration of the Watchdog Timer System Service is based on the file `system_config.h`.

This header file contains the configuration selection for the Watchdog Timer System Service. Based on the selections made, the Watchdog Timer System Service may support the selected features. These configuration settings will apply to all instances of the Watchdog Timer System Service.

This header can be placed anywhere; however, the path of this header needs to be present in the include search path for a successful build. Refer to the Applications Help section for more details.

WDT_PLIB_ID Macro

Configures the module's ID used by the peripheral library.

File

[sys_wdt_config.h](#)

C

```
#define WDT_PLIB_ID WDT_ID_1
```

Description

Watchdog timer module ID used by the Peripheral Library

This macro configures the module's ID used by the peripheral library.

Remarks

None.

Building the Library

This section lists the files that are available in the Watchdog Timer System Service Library.

Description

The following three tables list and describe the header (.h) and source (.c) files that implement this library. The parent folder for these files is <install-dir>/framework/system/wdt.

Interface File(s)

This table lists and describes the header files that must be included (i.e., using #include) by any code that uses this library.

Source File Name	Description
sys_wdt.h	Watchdog Timer System Service Library API header file.

Required File(s)



All of the required files listed in the following table are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

Source File Name	Description
/src/sys_wdt.c	Watchdog Timer System Service Library implementation.

Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.



Source File Name	Description
N/A	No optional files are available for this library.

Module Dependencies

The Watchdog Timer System Service is not dependent upon other modules.

Library Interface

a) Enable and Disable Functions

	Name	Description
	SYS_WDT_Disable	Disables the WDT if it is enabled in software.
	SYS_WDT_Enable	Enables the WDT. The argument 'windowModeEnable' will be used only for those devices that support 'window mode'. Otherwise, it will be discarded.

b) Clearing the Timer Functions

	Name	Description
	SYS_WDT_TimerClear	Reset the WDT.

Description

This section describes the Application Programming Interface (API) functions of the Watchdog Timer System Service.

Refer to each section for a detailed description.

a) Enable and Disable Functions

SYS_WDT_Disable Function

Disables the WDT if it is enabled in software.

File

[sys_wdt.h](#)

C

```
void SYS_WDT_Disable();
```

Returns

None.

Description

This function disables the WDT if it is enabled in software. If the WDT is enabled through 'configuration bits' it cannot be disabled using this function.

Remarks

The example code doesn't include the settings that should be done through configuration bits.

Preconditions

The WDT should be disabled through 'configuration bits'.

Example

```
SYS_WDT_Disable();
```

Function

```
void SYS_WDT_Disable ( void )
```

SYS_WDT_Enable Function

Enables the WDT. The argument 'windowModeEnable' will be used only for those devices that support 'window mode'. Otherwise, it will be discarded.

File

[sys_wdt.h](#)

C

```
void SYS_WDT_Enable(bool windowModeEnable);
```

Returns

None.

Description

This function enables the WDT. The argument 'windowModeEnable' will be used only for those devices that support 'window mode'. Otherwise, the argument will be discarded. This function could be called multiple times to enable/disable the 'window mode'.

Remarks

Calling this function is not necessary if the WDT is enabled through Configuration bits.

Preconditions

None.

Example

```
bool windowModeEnable = true;

SYS_WDT_Enable(windowModeEnable);
```

Parameters

Parameters	Description
windowModeEnable	A flag indicates whether to enable/disable the 'window mode'. <ul style="list-style-type: none">true - Enable the 'window mode'false - Disable the 'window mode'

Function

```
void SYS_WDT_Enable ( bool windowModeEnable )
```


b) Clearing the Timer Functions

SYS_WDT_TimerClear Function

Reset the WDT.

File

[sys_wdt.h](#)

C

```
void SYS_WDT_TimerClear();
```

Returns

None.

Description

This function clears the WDT counter. The WDT should be cleared periodically before the count overflows and forces the device to Reset.

Remarks

Clearing the WDT before the count reaches the window, will cause a reset in Windowed mode.

The example code doesn't include the settings that should be done through Configuration bits.

This feature is not available on all devices. Please refer to the specific device data sheet to determine whether this feature is supported.

Preconditions

None.

Example

```
//Application loop
while(1)
{
    SYS_WDT_TimerClear();
    //user code
}
```

Function

```
void SYS_WDT_TimerClear ( void )
```

Files

Files

Name	Description
sys_wdt.h	Watchdog Timer (WDT) System Service interface definition.
sys_wdt_config.h	Watchdog Timer (WDT) System Service interface definition.




Description

This section lists the source and header files used by the library.

sys_wdt.h

Watchdog Timer (WDT) System Service interface definition.

Functions

	Name	Description
	SYS_WDT_Disable	Disables the WDT if it is enabled in software.
	SYS_WDT_Enable	Enables the WDT. The argument 'windowModeEnable' will be used only for those devices that support 'window mode'. Otherwise, it will be discarded.
	SYS_WDT_TimerClear	Reset the WDT.

Description

Watchdog Timer System Service Interface Definition

This file contains the interface definition for the WDT System Service. It provides a way to interact with the WDT subsystem to manage the timing requests supported by the system.

File Name

sys_wdt.h

Company

Microchip Technology Inc.

sys_wdt_config.h

Watchdog Timer (WDT) System Service interface definition.

Macros

	Name	Description
	WDT_PLIB_ID	Configures the module's ID used by the peripheral library.

Description

Watchdog Timer System Service Interface Definition

This file contains the interface definition for the WDT System Service. It provides a way to interact with the Watchdog timer subsystem to manage the timing requests supported by the system

File Name

sys_wdt_config.h

Company

Microchip Technology Inc.

Index

—
 _promptStr macro 58
 _SYS_DEBUG_MESSAGE macro 103
 _SYS_DEBUG_PRINT macro 104
 _SYS_MSG_CONFIG_TEMPLATE_H macro 284
 _SYS_TMR_CONFIG_TEMPLATE_H macro 364
 _SYS_TOUCH_H macro 394

A

Abstraction Model 23, 50, 70, 89, 107, 133, 186, 251, 273, 278, 304, 334, 345, 352, 397

- Clock System Service Library 23
- Command Processor System Service Library 50
- Console System Service Library 70
- Debug System Service Library 89
- Device Control System Service Library 107
- DMA System Service Library 133
- Interrupt System Service Library 251
- Memory System Service Library 273
- Messaging System Service Library 278
- Ports System Service Library 304
- RNG System Service Library 334
- Timer System Service Library 352
- Watchdog Timer System Service Library 397

Application I/O (AppIO) 72

Application Interaction 188

B

Building the Library 30, 51, 76, 91, 109, 143, 195, 256, 274, 284, 310, 336, 347, 364, 386, 399

- Clock System Service Library 30
- Command Processor System Service Library 51
- Console System Service Library 76
- Debug System Service Library 91
- Device Control System Service Library 109
- DMA System Service Library 143
- File System Service Library 195
- Interrupt System Service Library 256
- Memory System Service Library 274
- Messaging System Service Library 284
- Ports System Service Library 310
- Random Number Generator System Service Library 336
- Reset System Service Library 347
- Timer System Service Library 364
- Touch System Service Library 386
- Watchdog Timer System Service Library 399

C

Change Notification 308
 Changing the Clock 25
 Changing the System Error Level 91
 Channel Setup and Management 135
 CLK_BUSES_PERIPHERAL enumeration 42
 CLK_BUSES_REFERENCE enumeration 42
 CLK_SOURCES_PERIPHERAL enumeration 43
 CLK_SOURCES_REFERENCE enumeration 43
 CLK_SOURCES_SYSTEM enumeration 44

Clock System Service Library 23
 Closing a File 184
 Command AppIO Console Device Configuration Options 74
 Command Processor System Service Library 50
 COMMAND_HISTORY_DEPTH macro 58
 Comparison of API Names 177
 Configuring the Library 27, 51, 74, 91, 109, 143, 193, 254, 274, 281, 310, 336, 347, 364, 386, 398

- Clock System Service Library 27
- Command Processor System Service Library 51
- Console System Service Library 74
- Debug System Service Library 91
- Device Control System Service Library 109
- DMA System Service Library 143
- File System Service Library 193
- Interrupt System Service Library 254
- Memory System Service Library 274
- Messaging System Service Library 281
- Ports System Service Library 310
- Random Number Generator System Service Library 336
- Timer System Service Library 364
- Touch System Service Library 386
- Watchdog Timer System Service Library 398

Console Device Modules 72
 Console System Service Library 70
 CONSOLE_DEVICE_HANDLE_INVALID macro 87
 Core Functionality 51, 108
 Core Functions 72
 Critical Sections 253

D

Debug Messages and Error Reporting 90
 Debug System Initialization 90
 Debug System Service Library 89
 Delays 361
 Device Control System Service Library 107
 Direct Memory Access (DMA) System Service Library 133
 Dynamic Configuration 255

E

ESC_SEQ_SIZE macro 59
 Examples 362

F

FAT_FS_MAX_LFN macro 244
 FAT_FS_MAX_SS macro 245
 FAT_FS_USE_LFN macro 245
 File EOF 184
 File Seek 185
 File System Service Library 176
 File System Service Library Porting Guide 176
 File Tell 185
 Files 20, 47, 68, 87, 105, 131, 174, 248, 271, 277, 302, 331, 343, 350, 384, 395, 401

- Clock System Service Library 47
- Command Processor System Service Library 68
- Console System Service Library 87
- Debug System Service Library 105
- Device Control System Service Library 131

- DMA System Service Library 174
- File System Service Library 248
- Interrupt System Service Library 271
- Memory System Service Library 277
- Messaging System Service Library 302
- Ports System Service Library 331
- Random Number Generator System Service Library 343
- Reset System Service 350
- System Service Library Overview 20
- Timer System Service Library 384
- Touch System Service Library 395
- Watchdog Timer System Service Library 401

G

- Global Control and Status 140

H

- How the Library Works 24, 51, 71, 90, 107, 134, 188, 252, 274, 278, 307, 335, 346, 356, 398

- Clock System Service Library 24
- Command Processor System Service Library 51
- Console System Service Library 71
- Debug System Service Library 90
- Device Control System Service Library 107
- DMA System Service Library 134
- Interrupt System Service Library 252
- Memory System Service Library 274
- Messaging System Service Library 278
- Ports System Service Library 307
- Random Number Generator System Service Library 335
- Timer System Service Library 356
- Watchdog Timer System Service Library 398

I

- Initialization 25, 177
- Initialization and Tasks 134
- INT_EXTERNAL_EDGE_TRIGGER enumeration 270
- Interrupt System Service Library 251
- Interrupt System Setup 253
- Introduction 3, 23, 50, 70, 89, 107, 133, 176, 251, 273, 278, 304, 334, 345, 352, 386, 397
 - Clock System Service Library 23
 - Command Processor System Service Library 50
 - Console System Service Library 70
 - Debug System Service Library 89
 - Device Control System Service Library 107
 - DMA System Service Library 133
 - File System Service Library 176
 - Interrupt System Service Library 251
 - Memory System Service Library 273
 - Messaging System Service Library 278
 - Ports System Service Library 304
 - RNG System Service Library 334
 - System Service Library Overview 3
 - Timer System Service Library 352
 - Touch System Service Library 386
 - Watchdog Timer System Service Library 397

L

- L1 Cache 108

- Library Interface 7, 31, 52, 77, 92, 110, 143, 196, 256, 275, 285, 310, 337, 348, 365, 387, 399

- Clock System Service Library 31
- Command Processor System Service Library 52
- Console System Service Library 77
- Debug System Service Library 92
- Device Control System Service Library 110
- DMA System Service Library 143
- File System Service Library 196
- Interrupt System Service Library 256
- Memory System Service Library 275
- Messaging System Service Library 285
- Ports System Service Library 310
- Random Number Generator System Service Library 337
- System Service Library 7
- Timer System Service Library 365
- Touch System Service Library 387
- Watchdog Timer System Service Library 399

- Library Overview 24, 50, 71, 90, 107, 134, 187, 252, 273, 278, 307, 335, 346, 356, 386, 397

- Clock System Service Library 24
- Command Processor System Service Library 50
- Debug System Service Library 90
- Device Control System Service Library 107
- DMA System Service Library 134
- File System Service Library 188
- Interrupt System Service Library 252
- Memory System Service Library 273
- Messaging System Service Library 278
- Ports System Service Library 307
- Random Number Generator System Service Library 335
- System Console System Service Library 71
- Timer System Service Library 356
- Touch System Service Library 386
- Watchdog Timer System Service Library 397

- LINE_TERM macro 59

M

- MAIN_RETURN macro 8
- MAIN_RETURN_CODE macro 17
- MAIN_RETURN_CODES enumeration 12
- MAX_CMD_ARGS macro 59
- MAX_CMD_GROUP macro 59
- Memory System Service Library 273
- Memory to Memory Transfer 141
- Messaging System Service Library 278
- Miscellaneous 309, 362
- Mounting a Volume 181
- MPLAB Harmony Module System Interface 4

N

- NVM_READ_FUNC type 391
- NVM_SECTORERASE_FUNC type 391
- NVM_WRITE_FUNC type 391

O

- Obtaining System Version Information 7
- One Shot Callback 359
- Opening a File 182

Oscillator Tuning 27

P

Periodic Callback 358

Peripheral Pin Select 309

Pin Control 307

Ports Control 308

Ports System Service Library 304

R

Random Number Generator (RNG) System Service Library 334

Reading a File 183

Reset Reason Management 346

Reset System Service Library 345

S

Source Interrupt Management 254

Special Considerations 309

Static Configuration 255

STDERR_FILENO macro 85

STDIN_FILENO macro 85

STDOUT_FILENO macro 86

SYS_ASSERT macro 9

SYS_CACHE_COHERENCY enumeration 130

sys_clk.h 47

SYS_CLK_ClockFailureCallbackRegister function 37

SYS_CLK_CONFIG_EXTERNAL_CLOCK macro 30

SYS_CLK_CONFIG_FREQ_ERROR_LIMIT macro 28

SYS_CLK_CONFIG_PRIMARY_XTAL macro 28

SYS_CLK_CONFIG_SECONDARY_XTAL macro 28

SYS_CLK_CONFIG_SYSPLL_INP_DIVISOR macro 29

sys_clk_config_template.h 48

SYS_CLK_CONFIGBIT_USBPLL_DIVISOR macro 29

SYS_CLK_CONFIGBIT_USBPLL_ENABLE macro 29

SYS_CLK_ERROR_HANDLER type 45

SYS_CLK_FRC_TUNING_TYPE type 47

SYS_CLK_FRCTune function 46

SYS_CLK_INIT structure 41

SYS_CLK_Initialize function 39

SYS_CLK_PeripheralFrequencyGet function 33

SYS_CLK_PeripheralFrequencySet function 34

SYS_CLK_REFERENCE_SETUP structure 45

SYS_CLK_ReferenceClockSetup function 35

SYS_CLK_ReferenceFrequencyGet function 37

SYS_CLK_ReferenceFrequencySet function 36

SYS_CLK_SecondaryOscillatorDisable function 40

SYS_CLK_SecondaryOscillatorEnable function 40

SYS_CLK_SecondaryOscillatorIsEnabled function 41

SYS_CLK_STATUS enumeration 46

SYS_CLK_SystemFrequencyGet function 32

SYS_CLK_SystemFrequencySet function 32

SYS_CLK_TaskError function 38

SYS_CLK_UPLL_BEFORE_DIV2_FREQ macro 30

SYS_CMD_ADDGRP function 54

SYS_CMD_API structure 61

SYS_CMD_BUFFER_DMA_READY macro 66

SYS_CMD_CallbackFunction type 67

SYS_CMD_CONSOLE_IO_PARAM enumeration 66

SYS_CMD_DATA_RDY_FNC type 61

SYS_CMD_DELETE function 53

SYS_CMD_DESCRIPTOR structure 62

SYS_CMD_DESCRIPTOR_TABLE structure 62

SYS_CMD_DEVICE_LIST structure 62

SYS_CMD_DEVICE_MAX_INSTANCES macro 66

SYS_CMD_DEVICE_NODE structure 60

SYS_CMD_EVENT enumeration 67

SYS_CMD_FNC type 63

SYS_CMD_GETC_FNC type 63

SYS_CMD_HANDLE type 63

SYS_CMD_INIT structure 61

SYS_CMD_INIT_DATA structure 64

SYS_CMD_Initialize function 53

SYS_CMD_MAX_LENGTH macro 59

SYS_CMD_MESSAGE function 54

SYS_CMD_MESSAGE macro 67

SYS_CMD_MSG_FNC type 64

SYS_CMD_PRINT function 55

SYS_CMD_PRINT macro 67

SYS_CMD_PRINT_FNC type 64

SYS_CMD_PUTC_FNC type 65

SYS_CMD_READ_BUFFER_SIZE macro 60

SYS_CMD_READC_FNC type 65

SYS_CMD_READY_TO_READ function 55

SYS_CMD_READY_TO_WRITE function 56

SYS_CMD_RegisterCallback function 57

SYS_CMD_STATE enumeration 65

SYS_CMD_Tasks function 56

SYS_CMDIO_ADD function 58

SYS_CMDIO_GET_HANDLE function 57

sys_command.h 68

sys_common.h 20

sys_console.h 87

SYS_CONSOLE_BUFFER_DMA_READY macro 87

SYS_CONSOLE_Deinitialize function 79

SYS_CONSOLE_DEV_DESC structure 85

SYS_CONSOLE_DEVICE enumeration 84

SYS_CONSOLE_Flush function 83

SYS_CONSOLE_INDEX_0 macro 86

SYS_CONSOLE_INDEX_1 macro 86

SYS_CONSOLE_INDEX_2 macro 86

SYS_CONSOLE_INDEX_3 macro 87

SYS_CONSOLE_INIT structure 85

SYS_CONSOLE_Initialize function 77

SYS_CONSOLE_MESSAGE macro 66

SYS_CONSOLE_OBJECT_INSTANCE structure 84

SYS_CONSOLE_PRINT macro 66

SYS_CONSOLE_Read function 81

SYS_CONSOLE_RegisterCallback function 82

SYS_CONSOLE_Reinitialize function 78

SYS_CONSOLE_Status function 80

SYS_CONSOLE_Tasks function 80

SYS_CONSOLE_Write function 82

SYS_DEBUG macro 99

sys_debug.h 105

SYS_DEBUG_BreakPoint macro 103

SYS_DEBUG_Deinitialize function 92

SYS_DEBUG_ErrorLevelGet function 97

- SYS_DEBUG_ErrorLevelSet function 98
SYS_DEBUG_INDEX_0 macro 100
SYS_DEBUG_INIT structure 98
SYS_DEBUG_Initialize function 93
SYS_DEBUG_Message function 96
SYS_DEBUG_MESSAGE macro 101
SYS_DEBUG_Print function 96
SYS_DEBUG_PRINT macro 102
SYS_DEBUG_Reinitialize function 94
SYS_DEBUG_Status function 94
SYS_DEBUG_Tasks function 95
sys_devcon.h 131
SYS_DEVCON_CacheClean function 118
SYS_DEVCON_CacheCoherencyGet function 119
SYS_DEVCON_CacheCoherencySet function 119
SYS_DEVCON_CacheFlush function 120
SYS_DEVCON_CacheInit function 120
SYS_DEVCON_CacheSync function 121
SYS_DEVCON_DataCacheAssociativityGet function 124
SYS_DEVCON_DataCacheClean function 122
SYS_DEVCON_DataCacheFlush function 122
SYS_DEVCON_DataCacheInvalidate function 123
SYS_DEVCON_DataCacheLineSizeGet function 124
SYS_DEVCON_DataCacheLinesPerWayGet function 125
SYS_DEVCON_DataCacheLock function 123
SYS_DEVCON_DataCacheSizeGet function 125
SYS_DEVCON_Deinitialize function 111
SYS_DEVCON_HANDLE type 129
SYS_DEVCON_INDEX_0 macro 130
SYS_DEVCON_INIT structure 129
SYS_DEVCON_Initialize function 112
SYS_DEVCON_InstructionCacheAssociativityGet function 127
SYS_DEVCON_InstructionCacheFlush function 126
SYS_DEVCON_InstructionCacheInvalidate function 126
SYS_DEVCON_InstructionCacheLineSizeGet function 128
SYS_DEVCON_InstructionCacheLinesPerWayGet function 128
SYS_DEVCON_InstructionCacheLock function 127
SYS_DEVCON_InstructionCacheSizeGet function 128
SYS_DEVCON_JTAGDisable function 116
SYS_DEVCON_JTAGEnable function 116
SYS_DEVCON_PerformanceConfig function 117
SYS_DEVCON_PowerModeEnter function 118
SYS_DEVCON_Reinitialize function 112
SYS_DEVCON_Status function 113
SYS_DEVCON_SystemLock function 115
SYS_DEVCON_SystemUnlock function 115
SYS_DEVCON_Tasks function 114
SYS_DEVCON_TraceDisable function 116
SYS_DEVCON_TraceEnable function 117
sys_dma.h 174
SYS_DMA_CHANNEL_CHAIN_PRIO enumeration 169
SYS_DMA_CHANNEL_COUNT macro 173
SYS_DMA_CHANNEL_CRC_MODE enumeration 169
SYS_DMA_CHANNEL_HANDLE type 172
SYS_DMA_CHANNEL_HANDLE_INVALID macro 174
SYS_DMA_CHANNEL_IGNORE_MATCH enumeration 172
SYS_DMA_CHANNEL_OP_MODE enumeration 170
SYS_DMA_CHANNEL_OPERATION_MODE_CRC structure 170
SYS_DMA_ChannelAbortEventSet function 148
SYS_DMA_ChannelAllocate function 149
SYS_DMA_ChannelCRCGet function 161
SYS_DMA_ChannelCRCSet function 162
SYS_DMA_ChannelDestinationTransferredSizeGet function 163
SYS_DMA_ChannelDisable function 150
SYS_DMA_ChannelEnable function 151
SYS_DMA_ChannelErrorGet function 158
SYS_DMA_ChannelForceAbort function 151
SYS_DMA_ChannelForceStart function 153
SYS_DMA_ChannelsBusy function 154
SYS_DMA_ChannelRelease function 155
SYS_DMA_ChannelSetup function 163
SYS_DMA_ChannelSetupMatchAbortMode function 155
SYS_DMA_ChannelSourceTransferredSizeGet function 165
SYS_DMA_ChannelTransferAdd function 157
SYS_DMA_ChannelTransferEventHandlerSet function 160
SYS_DMA_ChannelTransferSet function 166
SYS_DMA_CRC_WRITE_ORDER enumeration 171
SYS_DMA_INIT enumeration 173
SYS_DMA_Initialize function 146
SYS_DMA_IsBusy function 167
SYS_DMA_Resume function 168
SYS_DMA_SIDL enumeration 173
SYS_DMA_Suspend function 168
SYS_DMA_Tasks function 144
SYS_DMA_TasksError function 145
SYS_DMA_TasksErrorISR function 146
SYS_DMA_TasksISR function 147
SYS_DMA_TRANSFER_EVENT enumeration 171
SYS_ERROR macro 100
SYS_ERROR_LEVEL enumeration 99
SYS_ERROR_PRINT macro 103
sys_fs.h 248
SYS_FS_AUTOMOUNT_ENABLE macro 193
sys_fs_config_template.h 250
SYS_FS_CurrentDriveGet function 229
SYS_FS_CurrentDriveSet function 230
SYS_FS_CurrentWorkingDirectoryGet function 216
SYS_FS_DirClose function 219
SYS_FS_DirectoryChange function 217
SYS_FS_DirectoryMake function 218
SYS_FS_DirOpen function 219
SYS_FS_DirRead function 220
SYS_FS_DirRewind function 223
SYS_FS_DirSearch function 221
SYS_FS_DriveFormat function 232
SYS_FS_DriveLabelGet function 230
SYS_FS_DriveLabelSet function 231
SYS_FS_DrivePartition function 234
SYS_FS_DriveSectorGet function 236
SYS_FS_ERROR enumeration 238
SYS_FS_Error function 213
SYS_FS_EVENT enumeration 247
SYS_FS_EVENT_HANDLER type 248
SYS_FS_EventHandlerSet function 237
SYS_FS_FILE_DIR_ATTR enumeration 246
SYS_FS_FILE_OPEN_ATTRIBUTES enumeration 239

SYS_FS_FILE_SEEK_CONTROL enumeration 240
SYS_FS_FILE_SYSTEM_TYPE enumeration 240
SYS_FS_FileCharacterPut function 210
SYS_FS_FileClose function 197
SYS_FS_FileDirectoryModeSet function 206
SYS_FS_FileDirectoryRemove function 206
SYS_FS_FileDirectoryRenameMove function 207
SYS_FS_FileDirectoryTimeSet function 208
SYS_FS_FileEOF function 198
SYS_FS_FileError function 213
SYS_FS_FileNameGet function 203
SYS_FS_FileOpen function 199
SYS_FS_FilePrintf function 204
SYS_FS_FileRead function 214
SYS_FS_FileSeek function 200
SYS_FS_FileSize function 201
SYS_FS_FileStat function 224
SYS_FS_FileStringGet function 211
SYS_FS_FileStringPut function 212
SYS_FS_FileSync function 222
SYS_FS_FileTell function 202
SYS_FS_FileTestError function 205
SYS_FS_FileTruncate function 209
SYS_FS_FileWrite function 215
SYS_FS_FORMAT enumeration 247
SYS_FS_FSTAT structure 241
SYS_FS_FUNCTIONS structure 241
SYS_FS_HANDLE type 243
SYS_FS_HANDLE_INVALID macro 245
SYS_FS_Initialize function 225
SYS_FS_MAX_FILE_SYSTEM_TYPE macro 194
SYS_FS_MAX_FILES macro 194
SYS_FS_MEDIA_MAX_BLOCK_SIZE macro 194
SYS_FS_MEDIA_NUMBER macro 195
SYS_FS_Mount function 226
SYS_FS_REGISTRATION_TABLE structure 243
SYS_FS_RESULT enumeration 244
SYS_FS_Tasks 186
SYS_FS_Tasks function 228
SYS_FS_TIME union 246
SYS_FS_Unmount function 228
SYS_FS_VOLUME_NUMBER macro 195
sys_init.h 22
SYS_Initialize function 10
SYS_INPUT_DEVICE_EVENT enumeration 394
SYS_INPUT_DEVICE_TYPE enumeration 393
sys_int.h 271
SYS_INT_Disable function 262
SYS_INT_DynamicDeregister function 258
SYS_INT_DynamicRegister function 258
SYS_INT_Enable function 262
SYS_INT_ExternalInterruptTriggerSet function 263
SYS_INT_Initialize function 257
SYS_INT_IsEnabled function 263
SYS_INT_PROCESSOR_STATUS type 271
SYS_INT_ShadowRegisterAssign function 259
SYS_INT_ShadowRegisterGet function 260
SYS_INT_SourceDisable function 264
SYS_INT_SourceEnable function 265
SYS_INT_SourceIsEnabled function 265
SYS_INT_SourceStatusClear function 266
SYS_INT_SourceStatusGet function 267
SYS_INT_SourceStatusSet function 267
SYS_INT_StatusGetAndDisable function 260
SYS_INT_StatusRestore function 261
SYS_INT_TASKS_POINTER type 270
SYS_INT_VectorPrioritySet function 268
SYS_INT_VectorSubprioritySet function 269
sys_memory.h 277
SYS_MEMORY_DDR_Initialize function 275
SYS_MEMORY_EBI_Initialize function 275
SYS_MEMORY_Initialize function 276
SYS_MESSAGE macro 100
sys_module.h 21
SYS_MODULE_DEINITIALIZE_ROUTINE type 12
SYS_MODULE_INDEX type 13
SYS_MODULE_INIT union 13
SYS_MODULE_INITIALIZE_ROUTINE type 14
SYS_MODULE_OBJ type 14
SYS_MODULE_OBJ_INVALID macro 18
SYS_MODULE_OBJ_STATIC macro 18
SYS_MODULE_POWER_IDLE_RUN macro 18
SYS_MODULE_POWER_IDLE_STOP macro 19
SYS_MODULE_POWER_OFF macro 19
SYS_MODULE_POWER_RUN_FULL macro 19
SYS_MODULE_POWER_SLEEP macro 20
SYS_MODULE_REINITIALIZE_ROUTINE type 15
SYS_MODULE_STATUS_ROUTINE type 15
SYS_MODULE_TASKS_ROUTINE type 16
sys_msg.h 302
SYS_MSG_BUFFER_SIZES macro 282
sys_msg_config.h 303
SYS_MSG_Deinitialize function 286
SYS_MSG_GotMessages function 293
SYS_MSG_ID2hMsgType function 296
SYS_MSG_INIT structure 298
SYS_MSG_Initialize function 286
SYS_MSG_INSTANCE enumeration 298
SYS_MSG_MailboxClose function 290
SYS_MSG_MAILBOXES_ADDONE macro 301
SYS_MSG_MailboxMessagesGet function 288
SYS_MSG_MailboxMsgAdd function 288
SYS_MSG_MailboxMsgRemove function 289
SYS_MSG_MailboxOpen function 291
SYS_MSG_MailboxReinit function 291
SYS_MSG_MAX_MAILBOXES macro 283
SYS_MSG_MAX_MSGS_DELIVERED macro 283
SYS_MSG_MAX_PRIORITY macro 283
SYS_MSG_MAX_TYPES macro 283
SYS_MSG_MessageDeliver function 294
SYS_MSG_MessageReceive function 295
SYS_MSG_MessageSend function 295
SYS_MSG_NUM_MAILBOX_BITMAPS macro 301
SYS_MSG_OBJECT structure 298
SYS_MSG_QUEUE_STATUS enumeration 299
SYS_MSG_QueueStatus function 297

SYS_MSG_RECEIVE_CALLBACK type 299
SYS_MSG_RESULTS enumeration 300
SYS_MSG_Tasks function 287
SYS_MSG_TypeCreate function 292
SYS_MSG_TypeRemove function 293
SYS_MSGQ_ELEMENT structure 300
SYS_OBJ_HANDLE type 301
SYS_OBJ_HANDLE_INVALID macro 301
SYS_OBJ_HANDLE_STATIC macro 301
sys_ports.h 332
SYS_PORTS_ChangeNotificationDisable function 324
SYS_PORTS_ChangeNotificationEnable function 325
SYS_PORTS_ChangeNotificationGlobalDisable function 326
SYS_PORTS_ChangeNotificationGlobalEnable function 326
SYS_PORTS_ChangeNotificationIdleModeDisable function 327
SYS_PORTS_ChangeNotificationIdleModeEnable function 327
SYS_PORTS_ChangeNotificationPullUpDisable function 328
SYS_PORTS_ChangeNotificationPullUpEnable function 328
SYS_PORTS_Clear function 318
SYS_PORTS_DirectionGet function 319
SYS_PORTS_DirectionSelect function 319
SYS_PORTS_Initialize function 324
SYS_PORTS_OpenDrainDisable function 320
SYS_PORTS_OpenDrainEnable function 321
SYS_PORTS_PIN_DIRECTION enumeration 331
SYS_PORTS_PinClear function 317
SYS_PORTS_PinDirectionSelect function 317
SYS_PORTS_PinModeSelect function 312
SYS_PORTS_PinOpenDrainDisable function 312
SYS_PORTS_PinOpenDrainEnable function 313
SYS_PORTS_PinRead function 314
SYS_PORTS_PinSet function 314
SYS_PORTS_PinToggle function 315
SYS_PORTS_PinWrite function 316
SYS_PORTS_PULLUP_PULLDOWN_STATUS enumeration 331
SYS_PORTS_Read function 321
SYS_PORTS_RemapInput function 329
SYS_PORTS_RemapOutput function 330
SYS_PORTS_Set function 322
SYS_PORTS_Toggle function 323
SYS_PORTS_Write function 323
SYS_POWER_MODE enumeration 130
SYS_PRINT macro 101
sys_random.h 343
sys_random_config_template.h 344
SYS_RANDOM_CRYPT0_SEED_SIZE macro 336
SYS_RANDOM_CryptoBlockGet function 337
SYS_RANDOM_CryptoByteGet function 338
SYS_RANDOM_CryptoEntropyAdd function 338
SYS_RANDOM_CryptoGet function 339
SYS_RANDOM_CryptoSeedSet function 339
SYS_RANDOM_CryptoSeedSizeGet function 340
SYS_RANDOM_Deinitialize function 341
SYS_RANDOM_INIT structure 343
SYS_RANDOM_Initialize function 341
SYS_RANDOM_PseudoGet function 342
SYS_RANDOM_PseudoSeedSet function 342
sys_reset.h 350
SYS_RESET_NMIDelayCountSet function 350
SYS_RESET_ReasonClear function 348
SYS_RESET_ReasonGet function 349
SYS_RESET_SoftwareReset function 349
SYS_STATUS enumeration 17
SYS_Tasks function 9
sys_tmr.h 384
SYS_TMR_CALLBACK type 379
SYS_TMR_CallbackPeriodic function 370
SYS_TMR_CallbackSingle function 370
SYS_TMR_CallbackStop function 369
SYS_TMR_CLIENT_TOLERANCE macro 381
sys_tmr_config_template.h 385
SYS_TMR_Deinitialize function 366
SYS_TMR_DelayMS function 372
SYS_TMR_DelayStatusGet function 371
SYS_TMR_DRIVER_INDEX macro 381
SYS_TMR_FLAGS enumeration 380
SYS_TMR_FREQUENCY macro 381
SYS_TMR_FREQUENCY_TOLERANCE macro 382
SYS_TMR_HANDLE type 379
SYS_TMR_HANDLE_INVALID macro 380
SYS_TMR_INDEX_0 macro 380
SYS_TMR_INIT structure 379
SYS_TMR_Initialize function 366
SYS_TMR_INTERRUPT_NOTIFICATION macro 382
SYS_TMR_MAX_CLIENT_OBJECTS macro 382
SYS_TMR_MODULE_INIT macro 383
SYS_TMR_ModuleStatusGet function 378
SYS_TMR_ObjectCountGet function 373
SYS_TMR_ObjectCreate function 373
SYS_TMR_ObjectDelete function 374
SYS_TMR_ObjectReload function 375
SYS_TMR_Status function 367
SYS_TMR_SystemCountFrequencyGet function 377
SYS_TMR_SystemCountGet function 377
SYS_TMR_Tasks function 368
SYS_TMR_TickCounterFrequencyGet function 376
SYS_TMR_TickCountGet function 375
SYS_TMR_TickCountGetLong function 376
SYS_TMR_UNIT_RESOLUTION macro 383
sys_touch.h 395
SYS_TOUCH_CalibrationSet function 387
SYS_TOUCH_HANDLE type 391
SYS_TOUCH_HANDLE_INVALID macro 393
SYS_TOUCH_INDEX enumeration 391
SYS_TOUCH_INIT structure 392
SYS_TOUCH_Initialize function 388
SYS_TOUCH_Open function 389
SYS_TOUCH_STATUS enumeration 392
SYS_TOUCH_Tasks function 390
SYS_TOUCH_TouchMsgGet function 389
SYS_VersionGet macro 11
SYS_VersionStrGet macro 11
sys_wdt.h 401
sys_wdt_config.h 402
SYS_WDT_Disable function 399
SYS_WDT_Enable function 400

- SYS_WDT_TimerClear function 401
- System Configuration 180
- System Functions 71
- System Interaction 51, 107, 357
- System Service Libraries Help 2
- System Service Overview 3
- System State Machine 3
- system.h 22

T

- Timer System Service Library 352
- Touch System Service Library 386
- TouchShowMessage macro 393
- Triggering a Software Reset 346

U

- UART Console Device 73
- UART Console Device Configuration Options 75
- USB-CDC Console Device 73
- USB-CDC Console Device Configuration Options 74
- Using Status Functions 26
- Using the File System 189
- Using the Library 23, 50, 70, 89, 107, 133, 186, 251, 273, 278, 304, 334, 345, 352, 386, 397
 - Clock System Service Library 23
 - Command Processor System Service Library 50
 - Console System Service Library 70
 - Debug System Service Library 89
 - Device Control System Service Library 107
 - DMA System Service Library 133
 - Interrupt System Service Library 251
 - Memory System Service Library 273
 - Messaging System Service Library 278
 - Port System Service Library 304
 - RNG System Service Library 334
 - Timer System Service Library 352
 - Touch System Service Library 386
 - Watchdog Timer System Service Library 397
- Using the SYS_ASSERT Macro 6

W

- Watchdog Timer System Service Library 397
- WDT_PLIB_ID macro 398
- Writing a File 183