



MPLAB® Harmony Help - OSAL Library

MPLAB Harmony Integrated Software Framework v1.11

OSAL Library Help

This section describes the Operating System Abstraction Layer (OSAL) that is available in MPLAB Harmony.

Introduction

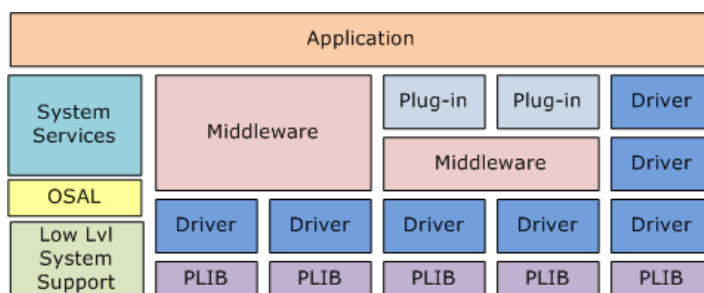
The Operating System Abstraction Layer (OSAL) provides a consistent interface to allow MPLAB Harmony-compliant libraries to take advantage of Operating System constructs when running in an OS environment or when operating without one. It is designed to take care of the underlying differences between the available OS Kernels or when no kernel is present.

Description

The OSAL provides the interface to commonly available Real-Time Operating Systems (RTOS) such that MPLAB Harmony libraries may be written using a single interface to a minimal set of OS features needed to provide thread safety. The OSAL interface can be implemented appropriately to support almost any desired RTOS. For systems where no RTOS is available, or desired, a bare version of the OSAL supports either polled or interrupt-driven environments running directly on the hardware. This allows applications designed using MPLAB Harmony libraries to be executed in all three common embedded environments: polled (shared multi-tasking), interrupt-driven, or RTOS-based.



Note: It is possible to make RTOS independent applications using the OSAL. However, as explained in the following section, that is not its purpose. Use and selection of an RTOS is usually determined by the availability of its unique features. And, utilizing those features will, of course, make an application OS-specific.



Scope

By design, the OSAL is a minimal API intended only to enable thread-safe operation for MPLAB Harmony libraries. It only exposes a very small subset of the capabilities of an operating system so that MPLAB Harmony libraries can use semaphores, mutexes, and critical sections (and a few other things) necessary to protect shared resources (data structures, peripheral registers, and other memory objects) from corruption by unsynchronized access by multiple threads. This is done to allow MPLAB Harmony libraries to be made compatible with the largest variety of operating systems, by using a minimal subset of some of the most common OS features. The OSAL is not intended to provide a complete abstraction of an RTOS, which is what you would normally do to implement a complete application. Abstracting an entire operating system is a much more complex task that is roughly equivalent to defining your own RTOS.

The OSAL is not designed to replace a commercial kernel, and therefore, the user is encouraged to use any of the specific features of their chosen RTOS in order to achieve best performance. As such, the OSAL can be considered to be an Operating System Compatibility Layer offering MPLAB Harmony-compliant libraries the required common functions to ensure correct operation in both RTOS and non-RTOS environments.

The common interface presented by the OSAL is designed to offer a set of services typically found on micro-kernel and mini-scheduler systems. Because of this it has no aspirations to provide an equivalent set of capabilities as those found on large multi-tasking systems such as μ CLinux™. The common services are designed to allow MPLAB Harmony to implement thread-safe Drivers and Middleware. The design intention is that drivers will use the minimal set of OSAL features necessary to ensure that they can safely operate in a multi-threaded environment yet can also compile and run correctly when no underlying RTOS is present. The range of features used by a driver is typically limited to these OSAL features (see the [Library Interface](#) section):

- Semaphore Functions
- Mutex Functions
- Critical Section Functions

Supported RTOS

RTOS	Release Type
FreeRTOS v8.x.x	Production
FreeRTOS v7.x.x	Production
OPENRTOS v8.x.x	Production
OPENRTOS v7.x.x	Production
Micrium® μ C/OS-II™	Beta
Micrium® μ C/OS-III™	Production
Express Logic ThreadX	Beta
SEGGER embOS®	Beta

Using the Library

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

Description

Interface Header File: `osal.h`

The interface to the OSAL Library is defined in the `osal.h` header file. Any C language source (.c) file that uses the OSAL System Service library should include `osal.h`.

Library File: `osal_<vendor-specified RTOS name>.c` (i.e., `freertos`, `ucos3`, etc.)

The OSAL Library consists of a basic implementation and individual ports of the OSAL to target operating systems. The basic implementation is used when the macro `OSAL_USE_BASIC` is defined and ports of a third-party RTOS, which is used when the macro `OSAL_USE_RTOS` is defined.

When an RTOS is being used (the `OSAL_USE_RTOS` configuration) then an external implementation file which provides the required interface wrappers should be added to the project. For instance for the FreeRTOS operating system the file `osal_freertos.c` should be added, while for the Micrium μ C/OS-III operating system the file `osal_ucos3.c` should be added.

The basic implementation and some generic ports are provided with the Library, however, it is the responsibility of third-party vendors to supply an implementation file for operating systems that are not already supported.

Please refer to the What is MPLAB Harmony? section for how the OSAL Library interacts with the framework.

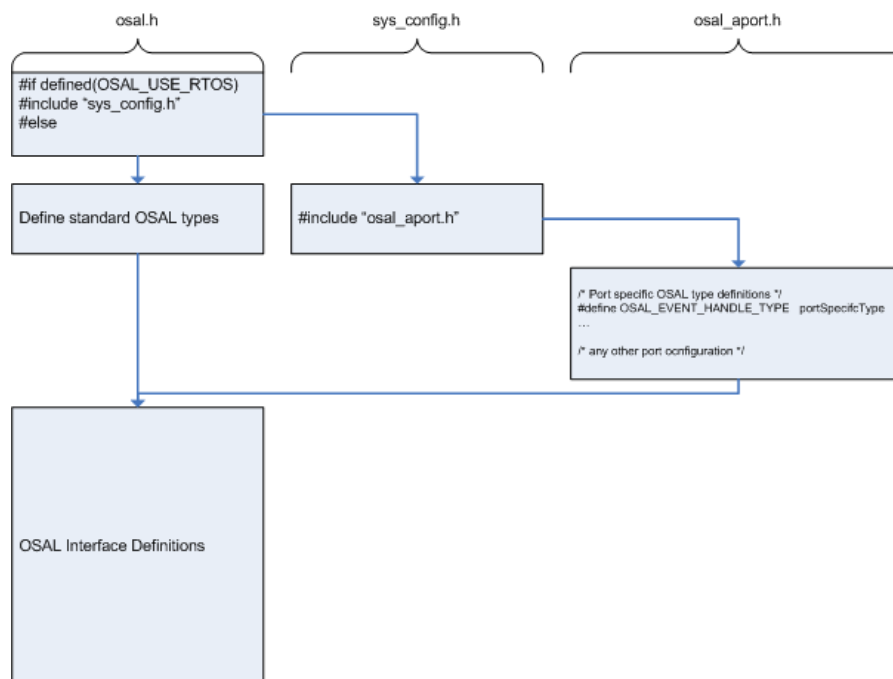
Abstraction Model

The OSAL Library provides a predefined set of functions and types that match common synchronization and communication services that an RTOS will typically provide. It is designed to be a lightweight abstraction model and deliberately excludes the much broader depth and breadth of services that a fully fledged RTOS provides. As such the interface defines only those core functions necessary for the MPLAB Harmony Drivers and middleware to operate in a multi-threaded environment.

Description

The common interface can easily be ported to many host Real-Time Operating Systems (RTOS) by third parties and the set of functions provides a basic level of RTOS compatibility. Where a specific RTOS does not implement a given architectural feature (e.g., events), the OSAL port for that RTOS should endeavor to imitate that feature using the constructs that are available. Although it is recognized that this may have a detrimental effect on the performance of that system it does allow MPLAB Harmony developers the broadest scope for using RTOS features in their designs.

Include Hierarchy of `osal.h`



Library Overview

This section provides an overview of the OSAL Library.

Description

Refer to the section System Services Introduction for how the system services operates in a system.

The OSAL Library provides a defined interface such that driver and middleware developers will be able to create MPLAB Harmony code that can safely operate in a multi-threaded environment when a supported RTOS is present yet will still compile and function correctly when MPLAB Harmony is being used in a non-RTOS environment with an interrupt or non-interrupt driven application model.

At the application layer, the developer is encouraged to use the specific features of a chosen RTOS once it has been selected since this is likely to provide a more effective and rich programming environment.

The OSAL Library is deliberately designed to be a thin layer over an underlying RTOS, which presents a predefined interface to the common features used by the majority of Real-Time Operating Systems, which includes:

Library Interface Section	Description
Semaphore Functions	Binary and counting semaphores.
Mutex Functions	Thread and resource locking mechanism.
Critical Section Functions	Application and scheduler locking mechanism.
Memory Allocation Functions	Memory allocation primitives or wrappers.
OSAL Control Functions	OSAL initialization.

One of the primary design guidelines is that a host operating system may not be present and so any operations that the OSAL presents are designed to compile out to safe default implementations if no RTOS is present. This can mean the following:

- Implementing a dummy function that mimics typical RTOS behavior
- Implementing a `#define` or inline function that returns a 'safe' generic return value, such as 'true' or a 'call succeeded' status
- Returning a `OSAL_RESULT_NOT_IMPLEMENTED` value to indicate an unsupported operation
- Throwing an `OSAL_ASSERT` failure to indicate a terminal error that prevents operation under specific circumstances

How the Library Works

This section provides information on how the OSAL Library works.

Core Functionality

The OSAL Library implements a lightweight abstraction layer on top of a host operating system. The OSAL functionality is designed to allow a programmer access to typical RTOS based functions than can reasonably be expected to exist on a small kernel or micro-scheduler typically found on the target Microchip Microcontrollers.

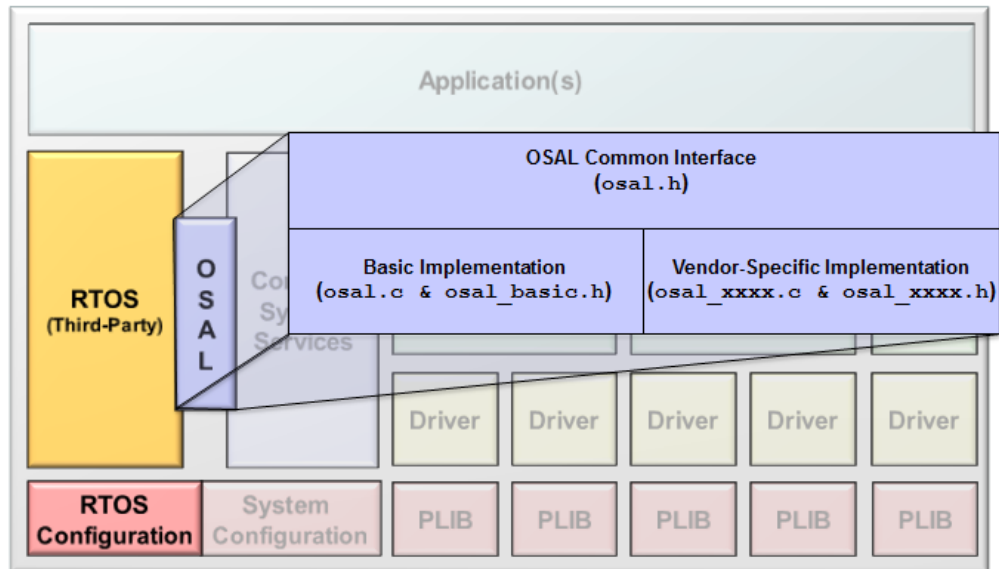
Description

The OSAL is designed to be compiled under three specific mutually exclusive operating modes:

Compilation Mode	Description
<code>OSAL_USE_BASIC</code>	Basic level of support. There is no underlying RTOS however certain facilities are replicated such as Semaphores and Mutexes.
<code>OSAL_USE_RTOS</code>	A full implementation of the OSAL interface is available and there is an underlying RTOS as part of the program.

In the absence of any one of these values being defined or passed into the compiler as a command line parameter then the OSAL will default to the `OSAL_USE_BASIC` mode of operation.

The following conceptual diagram shows the common interface and how the individual implementations are derived depending upon the configured mode of compilation.



The OSAL interface makes extensive use of object handles when passing RTOS parameters around the various functions. This has been done since depending on the operating mode and underlying RTOS (if any) there may be substantial differences between the C types of the variables. For instance, in the basic implementation mode a semaphore is implemented as a single 8-bit quantity to ensure atomic access on all architectures. On a third party RTOS the same semaphore may be implemented as a 16-bit index into an array of pointers while on another RTOS it could be represented as a `size_t` pointer (a pointer to the basic word type for that architecture). Because of these differences, the most convenient mechanism to maintain a regular interface is by the use of handles to implementation defined variables.

The majority of the OSAL objects also provide a `OSAL_XXX_Delete` function. This is provided for completeness however care should be taken in its use. Many RTOS implementations do not provide the ability to delete an object and excessive use of the delete can lead to memory fragmentation and unpredictable system behavior. It is recommended that it is not used at this time, or if used, the return value should be checked for a possible failure to delete the object.

In the following sections the basic behavior of the supported functions are documented. For the purposes of documentation the assumed operation mode will be that a full RTOS is present and so the behavior is that which will be observed in that scenario.

Semaphores

The semaphore implements a method for thread synchronization. This synchronization can be either between one thread and another or between an ISR and a thread. A semaphore once signalled will unblock the highest priority thread currently pending on it.

Description

A semaphore can be used to lock a shared resource, although it is more normal to use a mutex for such an activity. Once obtained a semaphore should be posted back to enable it to be retaken at a later time or in another thread.

```
/* mainline code prior to OS start */
/* declare a variable of type semaphore handle */
OSAL_SEM_DECLARE(semSync);
/* create the semaphore */
OSAL_SEM_Create(&semSync, OSAL_SEM_TYPE_BINARY, 0, 0);

/* thread one */
...
/* take the semaphore without waiting */
OSAL_SEM_Pend(semSync, 0);
... perform some actions
/* return the semaphore */
OSAL_SEM_Post(semSync);
...

/* thread two must not execute until thread one has finished its operations*/
...
/* block on the semaphore */
OSAL_SEM_Pend(semSync, OSAL_WAIT_FOREVER);
... perform some more actions
/* return the semaphore */
OSAL_SEM_Post(semSync);
```

A semaphore can be signalled multiple times and so provides a method for an ISR to release a thread waiting on it. Even though the blocked

thread never returns the semaphore, because the asynchronous ISR repeatedly posts it the next time the thread wants to pend on the semaphore it will be available. By moving the majority of interrupt service processing from the ISR to a high priority thread the system response time is improved and the eventual processing can take advantage of OSAL features such as mutexes and queues which would normally be harder to implement inside the ISR. This technique is known as deferred interrupt processing.

```
/* an example interrupt handler called from an ISR that performs task synchronization using a semaphore */
void _ISRTasksRX(void) /* N.B. pseudo-code ISR */
{
    ...

    _DRV_USART_InterruptSourceStatusClear(_DRV_USART_GET_INT_SRC_RX(_DRV_USART_OBJ(dObj),
rxInterruptSource));

    /* Release the receive semaphore unblocking any tasks */
    OSAL_SEM_PostISR(_DRV_USART_OBJ(dObj), rxSemID);

} /* DRV_USART_TasksRX */
```

Mutex Operation

A mutex or mutual exclusion is used to protect a shared resource from access by multiple threads at the same time. A shared resource may be a common data structure in RAM or it may be a hardware peripheral. In either case a mutex can be used to ensure the integrity of the entire resource by only allowing one thread to access it at a time.

Description

The library must be written in such a way that before the shared resources is accessed the mutex has to be obtained. Once obtained the accesses should occur, and once complete the mutex should then be released. While no restrictions are enforced the sequence of operations between the lock and unlock should ideally take as few lines of code as possible to ensure good system performance.

The mutex may be implemented as a form of binary semaphore but an underlying RTOS will often add other features. It is normal to add the restriction that a mutex may only be unlocked from the thread that originally obtained the lock in the first place. The RTOS may also provide features to mitigate priority inversion problems (where a high priority thread blocks on a lower priority one holding a mutex) by providing priority inheritance allowing lower priority threads to be temporarily raised to complete and release a locked mutex.

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

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

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

Critical Section Operation

This section describes how critical sections are used.

Description

Critical sections are used to form sequences of code that must operate in an atomic manner. The interface allows for the possibility of two types of critical section.

- When the critical section is entered all interrupts on the microcontroller are disabled. This prevents the protected sequence of code from being interrupted and ensures the complete atomicity of the operation. This is denoted by the OSAL_CRIT_TYPE_HIGH value
- When the critical section is entered the RTOS scheduler is disabled. In this second case other threads are prevented from running however interrupts can still occur which allows any asynchronous events to still be received and for the temporal accuracy of the RTOS scheduler to be maintained. This is denoted by the OSAL_CRIT_TYPE_LOW value

Since the behavior in the two cases is different the type of critical section must be identified in both the call to enter and leave.

```
/* enter and leave a critical section disabling interrupts */
OSAL_CRIT_Enter(OSAL_CRIT_TYPE_HIGH);
/* perform an atomic sequence of code */
```

```
...  
/* leave the critical section */  
OSAL_CRIT_Leave(OSAL_CRIT_TYPE_HIGH);
```

The underlying RTOS may not support the second scenario, in which case the OSAL implementation will default to disabling all interrupts.

Memory Operation

This section describes the memory operation using the OSAL Library.

Description

The OSAL Library provides an interface to a memory allocation mechanism. The memory required for dynamic instantiation of variables is normally provided by allocating it from the heap. However the standard C library implementation of `malloc` and `free` are not considered thread safe and so OSAL specific functions must be used if MPLAB Harmony or the application requires dynamic memory during operation.

When operating without an underlying RTOS the OSAL memory allocators default to using standard `malloc` and `free` functions. However, when operating with an RTOS the calls will defer to the specific scheme used by the RTOS. This may involve multiple memory pools or it may simply involve adding a critical section around calls to `malloc` and `free`. It is left to the implementation to define the most appropriate scheme.

```
/* allocate a large buffer */  
uint8_t* buffer;  
  
buffer = OSAL_Malloc(8000);  
if (buffer != NULL)  
{  
    ... manipulate the buffer  
    /* free the buffer */  
    OSAL_Free(buffer);  
    buffer = NULL;  
}
```

OSAL Operation

This section describes OSAL control features.

Description

When the OSAL is using an underlying RTOS it may be necessary to allow the RTOS to perform one-time initialization before any calls to it are made. For instance, the RTOS might implement multiple memory pools for managing queues and semaphores, and it must be given the chance to create these pools before any of the objects are created. For this reason the application program should call [OSAL_Initialize](#) early on and certainly before any MPLAB Harmony drivers or middleware is initialized (since these may also create OSAL objects at creation time).

Once the OSAL is initialized and any other remaining parts of the system are configured correctly, the specific RTOS can be started.

Configuring the Library

The configuration of the OSAL Library is based on the file `system_config.h`.

This header file contains the configuration selection for the OSAL Library. Based on the selections made, the OSAL may support the selected features.

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.

Description

To configure the OSAL the compilation mode needs to be selected. Three options are currently supported and these define the implementation details for the OSAL. In all cases the same interface is present; however, the functionality at the varying levels will be different or not supported. Only one of these `#define` statements should be specified, and this can be either done in a common location (such as `system_config.h`) or on the build tool command line.

Compilation Mode	Description
OSAL_USE_BASIC	Basic level of support. There is no underlying RTOS; however, certain facilities are replicated such as Semaphores and Mutexes.
OSAL_USE_RTOS	A full implementation of the OSAL interface is available and there is an underlying RTOS as part of the program.

Initialization Overrides

This section provides information for overriding OSAL initialization.

Description

The OSAL requires minimal configuration. In the case of the `OSAL_USE_BASIC`, all of the required configuration is provided by the standard `osal.h` include file. When a particular RTOS is being used then additional OSAL configuration parameters should be placed in `system_config.h`.

The primary requirement is to include the relevant interface implementation for the selected RTOS being used. This file will contain the mappings from OSAL interface types to types relevant for the chosen RTOS.

```
/* included the name of the port specific interface file */
#include "osal_<port name>.h"
```

Examples - Sample Functionality

Example `system_config.h` fragment:

```
// *****
// *****
// Section: Application Configuration
// *****
// *****
/* These definitions select the configuration options for the application */

/* Include the port specific OS implementation files. The file is required to
 * create an implementation layer for the selected OS. This include will be
 * changed depending on the required OS being used */
#include "osal_freertos.h"
```

Building the Library

This section lists the files that are available in the OSAL 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/osal.

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 for the selected RTOS.

Source File Name	Description
/osal.h	This file provides the interface definitions of the OSAL Library.

Required File(s)


MHC

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 be included in the MPLAB X IDE project to build this library.

Source File Name	Description
/src/osal.c /src/osal_impl_basic.h	Source files added to the project if using the "bare-metal" OSAL basic implementation used when no RTOS is present.
/src/osal_embos.c /src/osal_embos.h	Source files added to the project if using the embOS implementation for compatibility with the embOS operating system from SEGGER.
/src/osal_freertos.c /src/osal_freertos.h	Source files added to the project if using the FreeRTOS implementation for compatibility with the current version 8 of the FreeRTOS operating system from Real Time Engineers Ltd.
/src/osal_freertos_v7xx.c /src/osal_freertos_v7xx.h	Source files added to the project if using the FreeRTOS v7 implementation for compatibility with the version 7 of the FreeRTOS operating system from Real Time Engineers Ltd.
/src/osal_openrtos.c /src/osal_openrtos.h	Source files added to the project if using the OPENRTOS implementation for compatibility with the current version 8 of the OPENRTOS operating system from Wittenstein,
/src/osal_openrtos_v7xx.c /src/osal_openrtos_v7xx.h	Source files added to the project if using the OPENRTOS implementation for compatibility with the version 7 of the OPENRTOS operating system from Wittenstein.
/src/osal_threadx.c /src/osal_threadx.h	Source files added to the project if using the ThreadX implementation for compatibility with the ThreadX operating system from Express Logic.
/src/osal_ucos2.c /src/osal_ucos2.h	Source files added to the project if using the μ C/OS-II implementation for compatibility with the μ C/OS-II operating system from Micrium.
/src/osal_ucos3.c /src/osal_ucos3.h	Source files added to the project if using the μ C/OS-III implementation for compatibility with the μ C/OS-III operating system from Micrium.
/src/osal_impl_none.h	This header file should not be used for new designs, as it will be eliminated in a future release.

Optional File(s)







There are no optional files for the OSAL Library.

Module Dependencies





When OSAL_USE_RTOS is defined, the OSAL is dependent on the selected RTOS.

Library Interface



a) Semaphore Functions

	Name	Description
	OSAL_SEM_Create	Creates an OSAL Semaphore.
	OSAL_SEM_Delete	Deletes an OSAL Semaphore.
	OSAL_SEM_GetCount	Returns the current value of a counting semaphore.
	OSAL_SEM_Pend	Waits on a semaphore. Returns true if the semaphore was obtained within the time limit.
	OSAL_SEM_Post	Posts a semaphore or increments a counting semaphore.
	OSAL_SEM_PostISR	Posts a semaphore or increments a counting semaphore from within an Interrupt Service Routine (ISR).



b) Mutex Functions

	Name	Description
	OSAL_MUTEX_Create	Creates a mutex.
	OSAL_MUTEX_Delete	Deletes a mutex.
	OSAL_MUTEX_Lock	Locks a mutex.
	OSAL_MUTEX_Unlock	Unlocks a mutex.



c) Critical Section Functions

	Name	Description
	OSAL_CRIT_Enter	Enters a critical section with the specified severity level.
	OSAL_CRIT_Leave	Leaves a critical section with the specified severity level.




d) Memory Allocation Functions

	Name	Description
	OSAL_Free	Deallocates a block of memory and return to the default pool.
	OSAL_Malloc	Allocates memory using the OSAL default allocator.

e) OSAL Control Functions

	Name	Description
	OSAL_Initialize	Performs OSAL initialization.
	OSAL_Name	Obtains the name of the underlying RTOS.

f) Data Types and Constants

	Name	Description
	OSAL_RESULT	Enumerated type representing the general return value from OSAL functions.
	OSAL_SEM_TYPE	Enumerated type representing the possible types of semaphore.
	OSAL_CRIT_TYPE	Enumerated type representing the possible types of critical section.
	OSAL_MUTEX_DECLARE	Declares an OSAL mutex.
	OSAL_SEM_DECLARE	Declares an OSAL semaphore.

Description

This section describes the APIs of the OSAL Library. Refer to each section for a description.

a) Semaphore Functions

OSAL_SEM_Create Function

Creates an OSAL Semaphore.

File

help_osal.h

C

```
OSAL_RESULT OSAL_SEM_Create(OSAL_SEM_HANDLE_TYPE* semID, OSAL_SEM_TYPE type, uint8_t maxCount, uint8_t initialCount);
```

Returns

- OSAL_RESULT_TRUE - Semaphore created
- OSAL_RESULT_FALSE - Semaphore creation failed
- semID - Updated with valid semaphore handle if call was successful

Description

This function creates an OSAL binary or counting semaphore. If OSAL_SEM_TYPE_BINARY is specified, the maxcount and initialCount values are ignored.

Remarks

None.

Preconditions

Semaphore must have been declared.

Example

```
OSAL_SEM_Create(&mySemID, OSAL_SEM_TYPE_COUNTING, 10, 5);
```

Parameters

Parameters	Description
semID	Pointer to the Semaphore ID
type	If OSAL_SEM_TYPE_BINARY, create a binary semaphore. If OSAL_SEM_TYPE_COUNTING, create a counting semaphore with the specified count values.
maxCount	Maximum value for a counting semaphore (ignored for a BINARY semaphore). This parameter is ignored for Express Logic ThreadX, SEGGER embOS, and Micrium/OS-III as these RTOS libraries do not support this parameter.
initialCount	Starting count value for the semaphore (ignored for a BINARY semaphore) This should be less than or equal to maxCount when used with a RTOS library that supports the maxCount parameter (i.e., FreeRTOS and OPENRTOS)

Function

```
OSAL_RESULT OSAL_SEM_Create(OSAL_SEM_HANDLE_TYPE* semID, OSAL_SEM_TYPE type,
uint8_t maxCount, uint8_t initialCount)
```

OSAL_SEM_Delete Function

Deletes an OSAL Semaphore.

File

help_osal.h

C

```
OSAL_RESULT OSAL_SEM_Delete(OSAL_SEM_HANDLE_TYPE* semID);
```

Returns

- OSAL_RESULT_TRUE - Semaphore deleted
- OSAL_RESULT_FALSE - Semaphore deletion failed

Description

This function deletes an OSAL semaphore.

Remarks

None.

Preconditions

Semaphore must have been created.

Example

```
OSAL_SEM_Delete(&mySemID);
```

Parameters

Parameters	Description
semID	Pointer to the Semaphore ID

Function

```
OSAL_RESULT OSAL_SEM_Delete(OSAL_SEM_HANDLE_TYPE* semID)
```

OSAL_SEM_GetCount Function

Returns the current value of a counting semaphore.

File

help_osal.h

C

```
uint8_t OSAL_SEM_GetCount(OSAL_SEM_HANDLE_TYPE* semID);
```

Returns

- 0 - Semaphore is unavailable
- 1-255 - Current value of the counting semaphore

Description

This function returns the current value of a counting semaphore. The value returned is assumed to be a single value ranging from 0-255.

Remarks

None.

Preconditions

Semaphore must have been created.

Example

```
uint8_t semCount;

semCount = OSAL_SEM_GetCount(semUART);

if (semCount > 0)
{
    // obtain the semaphore
    if (OSAL_SEM_Pend(&semUART) == OSAL_RESULT_TRUE)
    {
        // perform processing on the comm channel
        ...
    }
}
else
{
    // no comm channels available
    ...
}
```

Parameters

Parameters	Description
semID	Pointer to the Semaphore ID

Function

```
uint8_t OSAL_SEM_GetCount(OSAL_SEM_HANDLE_TYPE* semID)
```

OSAL_SEM_Pend Function

Waits on a semaphore. Returns true if the semaphore was obtained within the time limit.

File

help_osal.h

C

```
OSAL_RESULT OSAL_SEM_Pend(OSAL_SEM_HANDLE_TYPE* semID, uint16_t waitMS);
```

Returns

- OSAL_RESULT_TRUE - Semaphore obtained
- OSAL_RESULT_FALSE - Semaphore not obtained or time-out occurred

Description

This function is a blocking function call that waits (i.e., pends) on a semaphore. The function will return true if the semaphore has been obtained, or false if it was not available or the time limit was exceeded.

Remarks

None.

Preconditions

Semaphore must have been created.

Example

```
if (OSAL_SEM_Pend(&semUARTRX, 50) == OSAL_RESULT_TRUE)
{
    // character available
    c = DRV_USART_ReadByte(drvID);
    ...
}
else
{
    // character not available, resend prompt
    ...
}
```

Parameters

Parameters	Description
semID	Pointer to the Semaphore ID
waitMS	Time limit to wait in milliseconds: <ul style="list-style-type: none">• 0 - do not wait• OSAL_WAIT_FOREVER - return only when semaphore is obtained• Other values - time-out delay

Function

```
OSAL_RESULT OSAL_SEM_Pend(OSAL_SEM_HANDLE_TYPE* semID, uint16_t waitMS)
```

OSAL_SEM_Post Function

Posts a semaphore or increments a counting semaphore.

File

help_osal.h

C

```
OSAL_RESULT OSAL_SEM_Post(OSAL_SEM_HANDLE_TYPE* semID);
```

Returns

- OSAL_RESULT_TRUE - Semaphore posted
- OSAL_RESULT_FALSE - Semaphore not posted

Description

This function posts a binary semaphore or increments a counting semaphore. The highest priority task currently blocked on the semaphore will be released and made ready to run.

Remarks

None.

Preconditions

Semaphore must have been created.

Example

```
OSAL_SEM_Post(&semSignal);
```

Parameters

Parameters	Description
semID	The semID

Function

```
OSAL_RESULT OSAL_SEM_Post(OSAL_SEM_HANDLE_TYPE* semID)
```

OSAL_SEM_PostISR Function

Posts a semaphore or increments a counting semaphore from within an Interrupt Service Routine (ISR).

File

help_osal.h

C

```
OSAL_RESULT OSAL_SEM_PostISR(OSAL_SEM_HANDLE_TYPE* semID);
```

Returns

- OSAL_RESULT_TRUE - Semaphore posted
- OSAL_RESULT_FALSE - Semaphore not posted

Description

This function posts a binary semaphore or increments a counting semaphore. The highest priority task currently blocked on the semaphore will be released and made ready to run. This form of the post function should be used inside an ISR.

Remarks

This version of the [OSAL_SEM_Post](#) function should be used if the program is, or may be, operating inside an ISR. The OSAL will take the necessary steps to ensure correct operation possibly disabling interrupts or entering a critical section. The exact requirements will depend upon the particular RTOS being used.

Preconditions

Semaphore must have been created.

Example

```
void __ISR(UART_2_VECTOR) _UART2RXHandler()  
{  
    char c;  
  
    // read the character  
    c = U2RXREG;  
    // clear the interrupt flag  
    IFS1bits.U2IF = 0;  
    // post a semaphore indicating a character has been received  
    OSAL_SEM_PostISR(&semSignal);  
}
```

Parameters

Parameters	Description
semID	Pointer to the Semaphore ID

Function

[OSAL_RESULT](#) OSAL_SEM_PostISR(OSAL_SEM_HANDLE_TYPE* semID)

b) Mutex Functions

OSAL_MUTEX_Create Function

Creates a mutex.

File

help_osal.h

C

```
OSAL_RESULT OSAL_MUTEX_Create(OSAL_MUTEX_HANDLE_TYPE* mutexID);
```

Returns

- OSAL_RESULT_TRUE - Mutex successfully created
- OSAL_RESULT_FALSE - Mutex failed to be created

Description

This function creates a mutex, allocating storage if required and placing the mutex handle into the passed parameter.

Remarks

None.

Preconditions

Mutex must have been declared.

Example

```
OSAL_MUTEX_HANDLE_TYPE mutexData;  
  
OSAL_MUTEX_Create(&mutexData);  
...  
if (OSAL_MUTEX_Lock(&mutexData, 1000) == OSAL_RESULT_TRUE)  
{  
    // manipulate the shared data  
    ...  
}
```

Parameters

Parameters	Description
mutexID	Pointer to the mutex handle

Function

[OSAL_RESULT](#) OSAL_MUTEX_Create(OSAL_MUTEX_HANDLE_TYPE* mutexID)

OSAL_MUTEX_Delete Function

Deletes a mutex.

File

help_osal.h

C

```
OSAL_RESULT OSAL_MUTEX_Delete(OSAL_MUTEX_HANDLE_TYPE* mutexID);
```


Returns

- OSAL_RESULT_TRUE - Mutex successfully deleted
- OSAL_RESULT_FALSE - Mutex failed to be deleted

Description

This function deletes a mutex and frees associated storage if required.

Remarks

None.

Preconditions

None.

Example

```
OSAL_MUTEX_Delete(&mutexData);
```

Parameters

Parameters	Description
mutexID	Pointer to the mutex handle

Function

```
OSAL_RESULT OSAL_MUTEX_Delete(OSAL_MUTEX_HANDLE_TYPE* mutexID)
```

OSAL_MUTEX_Lock Function

Locks a mutex.

File

help_osal.h

C

```
OSAL_RESULT OSAL_MUTEX_Lock(OSAL_MUTEX_HANDLE_TYPE* mutexID, uint16_t waitMS);
```

Returns

- OSAL_RESULT_TRUE - Mutex successfully obtained
- OSAL_RESULT_FALSE - Mutex failed to be obtained or time-out occurred

Description

This function locks a mutex, waiting for the specified time-out. If it cannot be obtained or the time-out period elapses 'false' is returned.

Remarks

None.

Preconditions

Mutex must have been created.

Example

```
...
if (OSAL_MUTEX_Lock(&mutexData, 1000) == OSAL_RESULT_TRUE)
{
    // manipulate the shared data
    ...

    // unlock the mutex
    OSAL_MUTEX_Unlock(&mutexData);
}
```

Parameters

Parameters	Description
mutexID	Pointer to the mutex handle

waitMS

Time-out value in milliseconds:

- 0, do not wait return immediately
- OSAL_WAIT_FOREVER, wait until mutex is obtained before returning
- Other values, time-out delay

Function

OSAL_RESULT OSAL_MUTEX_Lock(OSAL_MUTEX_HANDLE_TYPE* mutexID, uint16_t waitMS)

OSAL_MUTEX_Unlock Function

Unlocks a mutex.

File

help_osal.h

C

OSAL_RESULT OSAL_MUTEX_Unlock(OSAL_MUTEX_HANDLE_TYPE* **mutexID**);

Returns

- OSAL_RESULT_TRUE - Mutex released
- OSAL_RESULT_FALSE - Mutex failed to be released or error occurred

Description

This function unlocks a previously obtained mutex.

Remarks

None.

Preconditions

Mutex must have been created.

Example

```
...
if (OSAL_MUTEX_Lock(&mutexData, 1000) == OSAL_RESULT_TRUE)
{
    // manipulate the shared data
    ...

    // unlock the mutex
    OSAL_MUTEX_Unlock(&mutexData);
}
```

Parameters

Parameters	Description
mutexID	Pointer to the mutex handle

Function

OSAL_RESULT OSAL_MUTEX_Unlock(OSAL_MUTEX_HANDLE_TYPE* mutexID)

c) Critical Section Functions**OSAL_CRIT_Enter Function**

Enters a critical section with the specified severity level.

File

help_osal.h

C

OSAL_CRITSECT_DATA_TYPE OSAL_CRIT_Enter(OSAL_CRIT_TYPE **severity**);

Returns

A data type of `OSAL_CRITSECT_DATA_TYPE`, this value represents the state of interrupts before entering the critical section.

Description

This function enters a critical section of code. It is assumed that the sequence of operations bounded by the enter and leave critical section operations is treated as one atomic sequence that will not be disturbed. This function should be paired with [OSAL_CRIT_Leave\(\)](#).

Remarks

The sequence of operations bounded by the `OSAL_CRIT_Enter` and [OSAL_CRIT_Leave](#) form a critical section. The severity level defines whether the RTOS should perform task locking or completely disable all interrupts.

Preconditions

None.

Example

```
OSAL_CRITSECT_DATA_TYPE IntState;
// prevent other tasks preempting this sequence of code
IntState = OSAL_CRIT_Enter(OSAL_CRIT_TYPE_HIGH);
// modify the peripheral
DRV_USART_Reinitialize( objUSART, &initData);
OSAL_CRIT_Leave(OSAL_CRIT_TYPE_HIGH, IntState);
```

Parameters

Parameters	Description
severity	OSAL_CRIT_TYPE_LOW, The RTOS should disable all other running tasks effectively locking the scheduling mechanism. OSAL_CRIT_TYPE_HIGH, The RTOS should disable all possible interrupts sources including the scheduler ensuring that the sequence of code operates without interruption. The state of interrupts are returned to the user before they are disabled.

Function

`OSAL_CRITSECT_DATA_TYPE void OSAL_CRIT_Enter(OSAL_CRIT_TYPE severity)`

OSAL_CRIT_Leave Function

Leaves a critical section with the specified severity level.

File

help_osal.h

C

```
void OSAL_CRIT_Leave(OSAL_CRIT_TYPE severity, OSAL_CRITSECT_DATA_TYPE status);
```

Returns

None.

Description

This function leaves a critical section of code. It is assumed that the sequence of operations bounded by the enter and leave critical section operations is treated as one atomic sequence that will not be disturbed. The severity should match the severity level used in the corresponding [OSAL_CRIT_Enter](#) call to ensure that the RTOS carries out the correct action.

Remarks

The sequence of operations bounded by the [OSAL_CRIT_Enter](#) and `OSAL_CRIT_Leave` form a critical section. The severity level defines whether the RTOS should perform task locking or completely disable all interrupts.

Preconditions

None.

Example

```
OSAL_CRITSECT_DATA_TYPE IntState;
// prevent other tasks preempting this sequence of code
IntState = OSAL_CRIT_Enter(OSAL_CRIT_TYPE_LOW);
// modify the peripheral
```

```
DRV_USART_Reinitialize( objUSART, &initData);
OSAL_CRIT_Leave(OSAL_CRIT_TYPE_LOW, IntState);
```

Parameters

Parameters	Description
severity	OSAL_CRIT_TYPE_LOW, The scheduler will be unlocked, if no other nested calls to OSAL_CRIT_ENTER have been made. OSAL_CRIT_TYPE_HIGH, Interrupts are returned to the state passed into this function. The state should of been saved by an earlier call to OSAL_CRIT_Enter .
status	The value which will be used to set the state of global interrupts, if OSAL_CRIT_TYPE_HIGH is passed in.

Function

```
void OSAL_CRIT_Leave( OSAL_CRIT_TYPE severity, OSAL_CRITSECT_DATA_TYPE status)
```

d) Memory Allocation Functions

OSAL_Free Function

Deallocates a block of memory and return to the default pool.

File

help_osal.h

C

```
void OSAL_Free(void* pData);
```

Returns

None.

Description

This function deallocates memory and returns it to the default pool. In an RTOS-based application, the memory may have been allocated from multiple pools or simply from the heap. In non-RTOS applications, this function calls the C standard function free.

Remarks

None.

Preconditions

None.

Example

```
// create a working array
uint8_t* pData;

pData = OSAL_Malloc(32);
if (pData != NULL)
{
    ...

    // deallocate the memory
    OSAL_Free(pData);
    // and prevent it accidentally being used again
    pData = NULL;
}
```

Parameters

Parameters	Description
pData	Pointer to the memory block to be set free

Function

```
void OSAL_Free(void* pData)
```

OSAL_Malloc Function

Allocates memory using the OSAL default allocator.

File

help_osal.h

C

```
void* OSAL_Malloc(size_t size);
```

Returns

Pointer to the block of allocated memory. NULL is returned if memory could not be allocated.

Description

This function allocates a block of memory from the default allocator from the underlying RTOS. If no RTOS is present, it defaults to malloc. Many operating systems incorporate their own memory allocation scheme, using pools, blocks or by wrapping the standard C library functions in a critical section. Since a MPLAB Harmony application may not know what target OS is being used (if any), this function ensures that the correct thread-safe memory allocator will be used.

Remarks

None.

Preconditions

None.

Example

```
// create a working array
uint8_t* pData;

pData = OSAL_Malloc(32);
if (pData != NULL)
{
    ...
}
```

Parameters

Parameters	Description
size	Size of the requested memory block in bytes

Function

```
void* OSAL_Malloc(size_t size)
```

e) OSAL Control Functions

OSAL_Initialize Function

Performs OSAL initialization.

File

help_osal.h

C

```
OSAL_RESULT OSAL_Initialize();
```

Returns

OSAL_RESULT_TRUE - Initialization completed successfully.

Description

This function performs OSAL initialization. This function should be called near the start of main in an application that will use an underlying RTOS. This permits the RTOS to perform any one time initialization before the application attempts to create drivers or other items that may use the

RTOS. Typical actions performed by OSAL_Initialize would be to allocate and prepare any memory pools for later use.

Remarks

None.

Preconditions

None.

Example

```
int main()  
{  
    OSAL_Initialize();  
  
    App_Init();  
    OSAL_Start();  
}
```

Function

OSAL_RESULT OSAL_Initialize()

OSAL_Name Function

Obtains the name of the underlying RTOS.

File

help_osal.h

C

const char* OSAL_Name();

Returns

const char* - Name of the underlying RTOS or NULL

Description

This function returns a const char* to the textual name of the RTOS. The name is a NULL terminated string.

Remarks

None.

Preconditions

None.

Example

```
// get the RTOS name  
const char* sName;  
  
sName = OSAL_Name();  
sprintf(buff, "RTOS: %s", sName);
```

Function

const char* OSAL_Name()

f) Data Types and Constants

OSAL_RESULT Enumeration

Enumerated type representing the general return value from OSAL functions.

File

help_osal.h

C

```
enum OSAL_RESULT {  
    OSAL_RESULT_NOT_IMPLEMENTED = -1,  
    OSAL_RESULT_FALSE = 0,  
    OSAL_RESULT_TRUE = 1  
};
```

Description

OSAL Result type

This enum represents possible return types from OSAL functions.

Remarks

These enum values are the possible return values from OSAL functions where a standard success/fail type response is required. The majority of OSAL functions will return this type with a few exceptions.

OSAL_SEM_TYPE Enumeration

Enumerated type representing the possible types of semaphore.

File

help_osal.h

C

```
enum OSAL_SEM_TYPE {  
    OSAL_SEM_TYPE_BINARY,  
    OSAL_SEM_TYPE_COUNTING  
};
```

Description

OSAL Semaphore Type

This enum represents possible semaphore types.

OSAL_SEM_TYPE_BINARY - Simple binary type that can be taken once

OSAL_SEM_TYPE_COUNTING - Complex type that can be taken set number of times defined at creation time

Remarks

Binary and counting semaphore type.

OSAL_CRIT_TYPE Enumeration

Enumerated type representing the possible types of critical section.

File

help_osal.h

C

```
enum OSAL_CRIT_TYPE {  
    OSAL_CRIT_TYPE_LOW,  
    OSAL_CRIT_TYPE_HIGH  
};
```

Description

OSAL Critical Type

This enum represents possible critical section types.

OSAL_CRIT_TYPE_LOW - Low priority critical section, can be formed by locking the scheduler (if supported by RTOS) OSAL_CRIT_TYPE_HIGH

- High priority critical section, will be formed by disabling all interrupts.

Remarks

Critical section types.

OSAL_MUTEX_DECLARE Macro

Declares an OSAL mutex.

File

help_osal.h

C

```
#define OSAL_MUTEX_DECLARE(mutexID) OSAL_MUTEX_HANDLE_TYPE mutexID
```

Description

OSAL_MUTEX_Declare(mutexID)

This function declares a data item of type OSAL_MUTEX_HANDLE_TYPE.

Remarks

None.

// DOM-IGNORE-BEGIN

Place the definition of the OSAL_MUTEX_Declare macro inside each specific implementation file. Not all implementation files use the same definition. The type for each individual RTOS is different, and to handle this properly in the code, each implementation must define its own declaration. ****REMOVE THIS NOTE AND THE DOM STATEMENTS IF USING THIS API IN A FILE THAT IS INCLUDED IN THE MPLAB HARMONY INSTALLATION**** // DOM-IGNORE-END

OSAL_SEM_DECLARE Macro

Declares an OSAL semaphore.

File

help_osal.h

C

```
#define OSAL_SEM_DECLARE(semID) OSAL_SEM_HANDLE_TYPE semID
```

Description

OSAL_SEM_Declare(semID)

This function declares a data item of type OSAL_SEM_HANDLE_TYPE.

Remarks

None.

Files

Files

Name	Description
osal.h	Common interface definitions for the Operating System Abstraction Layer (OSAL).

Description

This section lists the source and header files used by the OSAL Library.

osal.h

Common interface definitions for the Operating System Abstraction Layer (OSAL).

Description

Descriptive File Name: Operating System Abstraction Layer

This file defines the common interface to the Operating System Abstraction Layer. It defines the common types used by the OSAL and defines the function prototypes. Depending upon the OSAL mode, a support level specific implementation file is included by this file to give the required level of compatibility. The available support levels include, OSAL_USE_NONE, OSAL_USE_BASIC, and OSAL_USE_RTOS.

File Name

osal.h

Company

Microchip Technology Inc.

Index

A

Abstraction Model 4
 OSAL Library 4

B

Building the Library 10
 OSAL Library 10

C

Configuring the Library 9
 OSAL Library 9
Core Functionality 5
Critical Section Operation 7

E

Examples - Sample Functionality 9

F

Files 25
 OSAL Library 25

H

How the Library Works 5
 OSAL Library 5

I

Initialization Overrides 9
Introduction 3
 OSAL Library 3

L

Library Interface 11
 OSAL Library 11
Library Overview 4
 OSAL Library 4

M

Memory Operation 8
Mutex Operation 7

O

OSAL Library Help 2
OSAL Operation 8
osal.h 25
OSAL_CRIT_Enter function 18
OSAL_CRIT_Leave function 19
OSAL_CRIT_TYPE enumeration 23
OSAL_CRIT_TYPE_HIGH enumeration member 23
OSAL_CRIT_TYPE_LOW enumeration member 23
OSAL_Free function 20
OSAL_Initialize function 21
OSAL_Malloc function 21
OSAL_MUTEX_Create function 16
OSAL_MUTEX_DECLARE macro 23
OSAL_MUTEX_Delete function 16
OSAL_MUTEX_Lock function 17
OSAL_MUTEX_Unlock function 18
OSAL_Name function 22

OSAL_RESULT enumeration 22
OSAL_RESULT_FALSE enumeration member 22
OSAL_RESULT_NOT_IMPLEMENTED enumeration member 22
OSAL_RESULT_TRUE enumeration member 22
OSAL_SEM_Create function 11
OSAL_SEM_DECLARE macro 24
OSAL_SEM_Delete function 12
OSAL_SEM_GetCount function 13
OSAL_SEM_Pend function 14
OSAL_SEM_Post function 14
OSAL_SEM_PostISR function 15
OSAL_SEM_TYPE enumeration 23
OSAL_SEM_TYPE_BINARY enumeration member 23
OSAL_SEM_TYPE_COUNTING enumeration member 23

S

Semaphores 6

U

Using the Library 4
 OSAL Library 4