HCC OS Abstraction Layer (Base) User Guide

Version 2.00

For use with OS Abstraction Layer versions 2.03 and above

Table of Contents

1. System Overview

This chapter contains the fundamental information for this module.

The component sections are as follows:

- Introduction – describes the main elements of the module.
- Feature Check – summarizes the main features of the module as bullet points.
- Packages and Documents – the *Packages* section lists the packages that you need in order to use this module. The *Documents* section lists the relevant user guides.
- Change History – lists the earlier versions of this manual, giving the software version that each manual describes.

1.1. Introduction

This guide is for those who wish to use HCC Embedded's OS Abstraction Layer (OAL) for their developments in embedded systems. Using the OAL facilitates development of embedded system software that is independent of a specific Real Time Operating System (RTOS) from an embedded software supplier.

> **Note:** This manual is only needed by those creating an OAL for their RTOS. For systems where HCC has already provided an OAL, this manual is only provided for reference and completeness.

All HCC systems and modules that require RTOS functionality use the OAL to provide it. The OAL allows these to run seamlessly with a wide variety of RTOSes, or without an RTOS.

The HCC OAL is an abstraction of a RTOS. It defines how HCC software requires an RTOS to behave and its Application Programming Interface (API) defines the functions it requires. It can be used in two situations:

- The RTOS you use provides the required functions, so "hooks" must be created to call its functions from the HCC abstractions. This porting to the RTOS is either undertaken by HCC and provided in a specific package, or implemented by yourself using an HCC template.
- There is no RTOS. A "No OS" package is available for this situation.

The OAL API defines functions for handling the following elements:

- Events – these are used as a signaling mechanism, both between tasks and from asynchronous sources such as Interrupt Service Routines to tasks.
- Interrupt Service Routines (ISRs) – depending on the RTOS, ISR functions may be platform-specific or RTOS-specific.
- Mutexes – these guarantee that, while one task is using a particular resource, no other task can pre-empt it and use the same resource.
- Tasks.

1.2. Feature Check

The main features of the module are the following:

- Fully MISRA-compliant.
- Conforms to the HCC Advanced Embedded Framework.
- Provides a standard interface for HCC tasks.
- Provides a standard interface for HCC mutexes.
- Provides a standard interface for HCC events.

1.3. Packages and Documents

Packages

The table below lists the packages that you need in order to use the OAL:

Package	Description
oal_base	The OAL base package described in this document. This may be the only OAL package you have, or you may have a package specific to your RTOS as well.
oal_os_<_your RTOS_>	A package specific to your RTOS. You will only have this if HCC has provided a package for your RTOS.
oal_os_template	A package template providing a set of empty functions for you to insert code into. You will only have this if you are implementing your own RTOS within this system.
psp_template_base	The base Platform Support Package (PSP).

For details of how packages combine, see [Using the Base Package with your RTOS](#).

Documents

For an overview of HCC RTOS software, see [Product Information](#) on the main HCC website.

Readers should note the points in the [HCC Documentation Guidelines](#) on the HCC documentation website.

HCC Firmware Quick Start Guide

This document describes how to install packages provided by HCC in the target development environment. Also follow the _Quick Start Guide_ when HCC provides package updates.

HCC Source Tree Guide

This document describes the HCC source tree. It gives an overview of the system to make clear the logic behind its organization.

HCC OS Abstraction Layer (Base) User Guide

This is this document.

HCC OAL for <RTOS> User Guide

There is a separate document for each RTOS that HCC has ported the OAL to. These complement this base OAL user guide.

1.4. Change History

This section describes past changes to this manual.

- To download this manual or a PDF describing an earlier software version, see OAL PDFs.
- For the history of changes made to the package code itself, see History: oal_base.

The current version of this manual is 2.00. The full list of versions is as follows:

Manual version	Date	Software version	Reason for change
2.00	2020-01-31	2.06	New template
1.60	2018-06-08	2.03	Corrected text on OAL_EVENT_FLAG configuration option.
1.50	2018-02-23	2.03	Corrected *oal_event_timeout_t* description on *Event Definitions* page.
1.40	2017-06-28	2.03	New *Change History* format.
1.30	2016-02-24	2.02	Added *Change History*, new functions.
1.20	2014-10-23	2.80	Added *Feature Check*.
1.10	2014-09-01	2.01	Added *System Design* section.
1.00	2014-03-12	2.01	First online version.

2. Using the Base Package with your RTOS

There are three options as described below.

1. HCC has undertaken the porting to your RTOS

In this case you will have the following packages:

- The **oal_base** package described in this manual. This defines the standard functions that must be provided by the RTOS (the standard RTOS abstraction to be used by all HCC middleware).
- A package specific to your RTOS. This calls the real RTOS functions needed. Unzip the files from this package into the **oal/os** folder in the source tree. These files will automatically call the correct functions.

2. You are porting to your RTOS yourself

In this case, you will have the following packages:

- The **oal_base** package described in this manual. This defines the standard functions that must be provided by the OAL (the standard RTOS abstraction to be used by all HCC middleware).
- The **oal_os_template** package. The template provides a set of empty functions. You have to insert code into the various functions, as described in the _HCC OAL Template User Guide_, and place the resulting files in a folder, ideally the **3rd Party** folder. The abstraction will call the correct function in your finished package.

3. There is no RTOS

In this case you will have just the **oal_os_nos** package, which is described in its own manual.

3. Abstraction Example

This section uses one function, **oal_event_create()**, as an example of how abstraction works. All HCC modules that require event type functionality will call this OAL function to create the event.

The definition in the OAL, described in [Application Programming Interface](#) is:

```
int oal_event_create ( oal_event_t * p_event )
```

The implementation of this function is RTOS-specific and the individual implementations are contained in the RTOS-specific packages.

In the HCC port to FreeRTOS™, this becomes:

```
oal_ret_t oal_event_create ( oal_event_t * p_event )
{
    int  rc = OAL_SUCCESS;
    *p_event = xQueueCreate( 1, sizeof( oal_event_flags_t ) );
    if ( *p_event == 0 )
    {
        rc = OAL_ERR_RESOURCE;
    }
    return rc;
}
```

In the HCC port to CMX™, this becomes:

```
oal_ret_t oal_event_create ( oal_event_t * p_event )
{
    *p_event = 0;
    return OAL_SUCCESS;
}
```

Every other RTOS which the function is ported to may have different code, taking account of the way that the particular RTOS operates. As long as the RTOS package files are placed in the correct locations, the correct code will be called when it is required.

4. Source File List

This section describes all the source code files included in the system. These files follow the HCC Embedded standard source tree system, described in the *HCC Source Tree Guide*. All references to file pathnames refer to locations within this standard source tree, not within the package you initially receive.

> **Note:** Do not modify any files except the configuration file.

Configuration File

The file **src/config/config_oal.h** contains all the configurable parameters of the OAL. For details of these options, see Configuration Options, which also covers the RTOS-specific configuration files.

Source Files

These files are in the directory **src/oal**. **These files should only be modified by HCC**.

File	Description
oal_common.h	Common header file.
oal_event.h	Event header file.
oal_isr.h	ISR header file.
oal_mutex.h	Mutex header file.
oal_task.c	Task source file.
oal_task.h	Task header file.

Version File

The file **src/version/ver_oal.h** contains the version number of this module. This version number is checked by all modules that use this module to ensure system consistency over upgrades.

API Files

> **Note:** These files are not part of this base package, but every RTOS-specific package has a set of files in its **src/oal/os** folder. These files, which all have the prefix **oalp_**, contain the RTOS port of the OAL functions and types.

5. Configuration Options

Set the base configuration options in the file **src/config/config_oal.h**. Different packages may require different components so, in order to avoid including needless elements, every package has its own configuration file, **src/config/config_oal_os.h**.

5.1. config_oal.h

The base OAL configuration options are set in the **src/config/config_oal.h** configuration file. This section lists the available options, which define which components of the OAL are included. All of these have the default value of 1.

> **Note:** Some HCC modules contain conditional compilation options that use these configuration options.

OAL_TASK_SUPPORTED

By default the OAL, as implemented, supports the **oal_task_xxx()** functions. Set this to 0 to disable support for these functions.

OAL_TASK_GET_ID_SUPPORTED

By default the OAL, as implemented, supports the **oal_task_get_id()** function. Set this to 0 to disable support for this function.

OAL_TASK_SLEEP_SUPPORTED

By default the OAL, as implemented, supports the **oal_task_sleep()** function. Set this to 0 to disable support for this function.

OAL_MUTEX_SUPPORTED

By default the OAL, as implemented, supports the **oal_mutex_xxx()** functions. Set this to 0 to disable support for these functions.

OAL_EVENT_SUPPORTED

By default the OAL, as implemented, supports the **oal_event_xxx()** functions. Set this to 0 to disable support for these functions.

OAL_ISR_SUPPORTED

By default the OAL, as implemented, supports the **oal_isr_xxx()** functions. Set this to 0 to disable support for these functions.

5.2. RTOS Integration

This section describes two files within an OAL package for a specific OS that control how the OS is used Each OS abstraction must have a version of these two files.

oal/os/oalp_defs.h

All RTOS ports must have an **oal/os/oalp_defs.h** file with the following definitions:

> **Note:** This section describes what must be provided by the OS abstraction of a specific OS. If this is already provided then you just need to include that file.

OAL_TASK_POLL_MODE

Enable this to have tasks polled. Only set this when there is no OS. The default is 0.

OAL_PREEMPTIVE

Enable this for a preemptive system. The default is 1.

OAL_STATIC_TASK_STACK

Enable this if the stack of a task needs to be allocated statically. The default is zero.

OAL_INTERRUPT_ENABLE

Enable this to allow interrupts. The default is 1; only set this to 0 if there is no OS.

If there is no OS this option is in the file **config_oal.h** since the user can decide whether it is required.

OAL_USE_PLATFORM_ISR

Enable this to use platform ISR routines. The default is 1.

OAL_TICK_RATE

The tick rate in ms. The default is 10. If this varies depending on the port or platform, or if there is no variable in the RTOS to obtain this value, then it must be in **config_oal.h**.

config_oal_os.h

All RTOS ports must have a **config_oal_os.h** file with the following definitions:

OAL_HIGHEST_PRIORITY, OAL_HIGH_PRIORITY, OAL_NORMAL_PRIORITY, OAL_LOW_PRIORITY, OAL_LOWEST_PRIORITY

Lower numbers mean a higher priority. By default these are respectively 5, 10, 15, 20, and 25.

OAL_EVENT_FLAG

OAL_EVENT_FLAG's usage depends on the type of event system an RTOS uses. There are two types:

- Event groups are supported independently of everything else in the system. In this case OAL_EVENT_FLAG does not matter.
- Each event group is directly controlled by a specific task. In this case all HCC stack internal events use the OAL_EVENT_FLAG as the event flag to set on the tasks event group. None of the tasks invoking HCC API calls should use OAL_EVENT_FLAG for signalling an event.

The default is 0x100.

OAL_TASK_COUNT

The maximum number of tasks. The default is 8.

This option is required if an OS has any parameter that needs to be predefined in order to create a task. In this case this parameter can define the number of required entities.

OAL_ISR_COUNT

The maximum number of interrupt sources. The default is 4.

This is optional and its presence depends on the OS. For example, if there is a need for a wrapper function for every ISR, these have to be pre-written somewhere so this option would have to be defined.

6. System Design

This section gives basic information on events, ISRs, mutexes and tasks.

6.1. Events

Within the RTOS's files:

- **oal/os/oalp_event.c** contains the port of the functions.
- **oal/os/oalp_event.h** defines *oal_event _t*, the event type.

Event Groups and Flags

The OAL expects event groups and each event group has event flags (event bits).

> **Note:** There are many types of RTOS and these have different mechanisms and terminology for events. For example there are some where "event" as a term is not even known, and we translate this using group semaphores, messages and so on as appropriate to replicate our desired behaviour for events.

The *oal_event_t* structure defines the type of the event. There are two types of RTOS:

- In one RTOS type the event has a specific type and the task waits for a flag in this event.
- In the other the RTOS sends an event to the specific task (without the need for a global event). Here the type of *oal_event_t* is not important but you should define it for compatibility reasons. Here **oal_event_create()** probably needs to do nothing.

The purpose of the OAL_EVENT_FLAG varies as follows:

- If an RTOS waits for a flag in a statically allocated event, then the value of OAL_EVENT_FLAG is irrelevant.
- If a task needs to wait for an event flag in its dedicated event group, then only flags below OAL_EVENT_FLAG can be used for internal purposes. In this case there are two possibilities:
    - User task -> HCC function -> event wait(OAL_EVENT_FLAG) -> return to user function.
    - HCC task -> waits for an internal event -> event received -> HCC function -> event_wait(OAL_EVENT_FLAG) -> return to HCC task.

6.2. Interrupt Service Routines - ISRs

Within the RTOS's files:

- **oal/os/oalp_isr.c** contains the port of the functions.
- **oal/os/oalp_isr.h** defines *oal_isr_id_t*, the ISR identifier.

ISR Functions

Depending on the RTOS, ISR functions may be platform-specific or RTOS-specific. The platform determines whether **oal_xxx()** functions are used, or whether these are mapped to **psp_xxx()** functions. If it allows you to create an ISR invoking RTOS calls, OAL_USE_PLATFORM_ISR must be set to 0.

All ISR functions must be defined as follows.

In the .c file:

```
OAL_ISR_FN(my_isr)
    {
        OAL_ISR_PRE;
        ...
        my code
        ...
        OAL_ISR_POST;
    }
```

In the **.h** file:

```
OAL_ISR_FN(my_isr);
```

Optional Files

There are two optional files, **psp/target/isr/psp_isr.h** and **psp/target/isr/psp_isr.c**.

These are used as follows:

- If OAL_USE_PLATFORM_ISR is set, this should contain everything defined in **oalp_isr.h** and **oalp_isr.c**. In this case, note that all oal_/OAL_ prefixes should become psp_/PSP_.
- If OAL_USE_PLATFORM_ISR is not set, the modules can still be present and can provide platform-specific information to the **oal_isr** module. For example, if the vector number and priority can only be obtained with some additional steps based on *oal_isr_dsc_t->id* then helper functions can be placed here.

6.3. Mutexes

Within the RTOS's files:

- **oal/os/oalp_mutex.c** contains the port of the functions.
- **oal/os/oalp_mutex.h** defines *oal_mutex_t*, the mutex type.

6.4. Tasks

Within the RTOS's files:

- **oal/os/oalp_task.c** contains the port of the functions.
- **oal/os/oalp_task.h** defines *oal_task_id_t*, the task ID type.

Creating Tasks

Where possible, create a task from the module which uses it.

In some cases tasks cannot be dynamically created, in which case refer to the manual for information on the required task. However all functions in this section should be present for compatibility reasons.

All tasks must be defined as follows.

In the **.c** file:

```
OAL_TASK_FN(my_task)
    {
        int my_var;
        OAL_TASK_PRE;
    }
```

In the **.h** file:

```
OAL_TASK_FN(my_task);
```

7. Application Programming Interface

This section documents the Application Programming Interface (API). It includes all the functions that are available to an application program.

7.1. Event Functions

This section describes the following event functions:

Function	Description
oal_event_create()	Creates/initializes an OAL event.
oal_event_delete()	Deletes an OAL event.
oal_event_get()	Waits for an event.
oal_event_set()	Sets an event from a non-interrupt/interrupt.
oal_event_set_int()	Sets an OAL event from an interrupt.

oal_event_create

Use this function to create/initialize an OAL event.

Format

```
oal_ret_t oal_event_create ( oal_event_t * p_event )
```

Arguments

Argument	Description	Type
p_event	A pointer to the event.	oal_event_t *

Return Values

Return value	Description
OAL_SUCCESS	Successful execution.
Else	See Error Codes.

oal_event_delete

Use this function to delete an OAL event.

Format

```
oal_ret_t oal_event_delete ( oal_event_t * p_event )
```

Arguments

Argument	Description	Type
p_event	A pointer to the event.	oal_event_t *

Return Values

Return value	Description
OAL_SUCCESS	Successful execution.
Else	See Error Codes.

oal_event_get

Use this function to wait for an event.

Format

```
oal_ret_t oal_event_get (
    oal_event_t *         p_event,
    oal_event_flags_t     wflags,
    oal_event_flags_t *   sflags,
    oal_event_timeout_t   timeout )
```

Arguments

Argument	Description	Type
p_event	A pointer to the event.	oal_event_t *
wflags	Flag(s) to wait for (multiple flags are allowed).	oal_event_flags_t
sflags	Flags set if the event is obtained.	oal_event_flags_t *
timeout	Time to wait for an event (in ms).	oal_event_timeout_t

Return Values

Return value	Description
OAL_SUCCESS	Successful execution.
Else	See Error Codes.

oal_event_set

Use this function to set an event from a non-interrupt/interrupt.

Format

```
oal_ret_t oal_event_set (
    oal_event_t *       p_event,
    oal_event_flags_t   flags,
    oal_task_id_t       task_id )
```

Arguments

Argument	Description	Type
p_event	A pointer to the event.	oal_event_t *
flags	The event flags to set.	oal_event_flags_t
task_id	The destination task ID (this may not be required).	oal_task_id_t

Return Values

Return value	Description
OAL_SUCCESS	Successful execution.
Else	See Error Codes.

oal_event_set_int

Use this function to set an OAL event from an interrupt.

Format

```
oal_ret_t oal_event_set_int (
    oal_event_t *        p_event,
    oal_event_flags_t    flags,
    oal_task_id_t        task_id )
```

Arguments

Argument	Description	Type
p_event	A pointer to the event.	oal_event_t *
flags	The event flags to set.	oal_event_flags_t
task_id	The task ID (this may not be required).	oal_task_id_t

Return Values

Return value	Description
OAL_SUCCESS	Successful execution.
Else	See Error Codes.

7.2. ISR Functions

> **Note:** If OAL_USE_PLATFORM_ISR is enabled, the **oal_xxx()** functions shown here are mapped to **psp_xxx()** functions. See Interrupt Service Routines - ISRs.

This section describes the following Interrupt Service Routine (ISR) functions:

Function	Description
oal_int_enable()	Enables global interrupts.
oal_int_disable()	Disables global interrupts.
oal_isr_install()	Installs the ISR module.
oal_isr_enable()	Enables an ISR.
oal_isr_delete()	Deletes an ISR.
oal_isr_disable()	Disables an ISR.

oal_int_enable

Use this function to enable global interrupts.

Format

```
oal_ret_t oal_int_enable ( void )
```

Arguments

Argument
None.

Return Values

Return value	Description
OAL_SUCCESS	Successful execution.
Else	See Error Codes.

oal_int_disable

Use this function to disable global interrupts.

Format

```
oal_ret_t oal_int_disable ( void )
```

Arguments

Argument
None.

Return Values

Return value	Description
OAL_SUCCESS	Successful execution.
Else	See Error Codes.

oal_isr_install

Use this function to install the ISR module.

Format

```
oal_ret_t oal_isr_install (
    const oal_isr_dsc_t *   isr_dsc,
    oal_isr_id_t *          isr_id )
```

Arguments

Argument	Description	Type
isr_dsc	The ISR descriptor.	oal_isr_dsc_t *
isr_id	Returns the ISR ID.	oal_isr_id_t *

Return Values

Return value	Description
OAL_SUCCESS	Successful execution.
Else	See Error Codes.

oal_isr_enable

Use this function to enable an ISR.

Format

```
oal_ret_t oal_isr_enable ( oal_isr_id_t isr_id )
```

Arguments

Argument	Description	Type
isr_id	The ISR ID.	oal_isr_id_t

Return Values

Return value	Description
OAL_SUCCESS	Successful execution.
Else	See Error Codes.

oal_isr_delete

Use this function to delete an ISR.

Format

```
oal_ret_t oal_isr_delete ( oal_isr_id_t isr_id )
```

Arguments

Argument	Description	Type
isr_id	The ISR ID.	oal_isr_id_t

Return Values

Return value	Description
OAL_SUCCESS	Successful execution.
Else	See Error Codes.

oal_isr_disable

Use this function to disable an ISR.

Format

```
oal_ret_t oal_isr_disable ( oal_isr_id_t isr_id )
```

Arguments

Argument	Description	Type
isr_id	The ISR ID.	oal_isr_id_t

Return Values

Return value	Description
OAL_SUCCESS	Successful execution.
Else	See Error Codes.

7.3. Mutex Functions

This section describes the following mutex functions:

Function	Description
oal_mutex_create()	Creates a mutex.
oal_mutex_delete()	Deletes a mutex.
oal_mutex_get()	Gets a mutex.
oal_mutex_put()	Releases a mutex.

oal_mutex_create

Use this function to create a mutex.

Format

```
oal_ret_t oal_mutex_create ( oal_mutex_t * p_mutex )
```

Arguments

Argument	Description	Type
p_mutex	A pointer to the mutex.	oal_mutex_t *

Return Values

Return value	Description
OAL_SUCCESS	Successful execution.
Else	See Error Codes.

oal_mutex_delete

Use this function to delete a mutex.

Format

```
oal_ret_t oal_mutex_delete ( oal_mutex_t * p_mutex )
```

Arguments

Argument	Description	Type
p_mutex	A pointer to the mutex.	oal_mutex_t *

Return Values

Return value	Description
OAL_SUCCESS	Successful execution.
Else	See Error Codes.

oal_mutex_get

Use this function to get a mutex.

Format

```
oal_ret_t oal_mutex_get ( oal_mutex_t * p_mutex )
```

Arguments

Argument	Description	Type
p_mutex	A pointer to the mutex.	oal_mutex_t *

Return Values

Return value	Description
OAL_SUCCESS	Successful execution.
Else	See Error Codes.

oal_mutex_put

Use this function to release a mutex.

Format

```
oal_ret_t oal_mutex_put ( oal_mutex_t * p_mutex )
```

Arguments

Argument	Description	Type
p_mutex	A pointer to the mutex.	oal_mutex_t *

Return Values

Return value	Description
OAL_SUCCESS	Successful execution.
Else	See Error Codes.

7.4. Task Functions

You can create multiple instances by using the **oal_task_m_*()** functions.

This section describes the following event functions:

Function	Description
oal__task_m_dsc_init()	Initializes a task descriptor.
oal_task_m_create()	Creates an instance of an OAL task.
oal_task_m_delete()	Deletes a task created by using oal_task_m_create().
oal_task_m_get_index()	Gets the index of a task created by using oal_task_m_create().
oal_task_create()	Creates an OAL task.
oal_task_delete()	Deletes a caller task.
oal_task_get_id()	Gets the ID of the currently active task.
oal_task_poll()	Polls all the registered tasks.
oal_task_sleep()	Suspends the caller task.
oal_task_yield()	Yields the current task.

oal_task_m_dsc_init

Use this function to initialize a task descriptor.

> **Note:** You must call this function before using the other **oal_task_m_*** functions.

Format

```
oal_ret_t oal_task_m_dsc_init ( const oal_task_m_dsc_t * task_dsc )
```

Arguments

Argument	Description	Type
task_dsc	The task descriptor.	oal_task_m_dsc_t *

Return Values

Return value	Description
OAL_SUCCESS	Successful execution.
Else	See Error Codes.

oal_task_m_create

Use this function to create an instance of an OAL task.

> **Note:**
> - You must create the task's descriptor with the macro OAL_TASK_M_DSC and initialize it with **oal_task_m_dsc_init()** prior to using this function.
> - Delete the created task by using **oal_task_m_delete()**, not **oal_task_delete()**.

Format

```
oal_ret_t oal_task_m_create (
    oal_task_t *              p_task,
    const oal_task_dsc_t *    task_dsc,
    oal_task_index_t          task_index,
    oal_task_id_t *           task_id )
```

Arguments

Argument	Description	Type
p_task	On return, a pointer to the task.	oal_task_t *
task_dsc	The task descriptor.	oal_task_m_dsc_t *
task_index	The index of the task to create. The valid range is 0 .. (task_dsc->task_count - 1). Within the task the **oal_task_m_get_index()** function can be used to get the *task_index*.	oal_task_index_t
task_id	The created task ID.	oal_task_id_t *

Return Values

Return value	Description
OAL_SUCCESS	Successful execution.
Else	See Error Codes.

oal_task_m_delete

Use this function to delete a task created by using **oal_task_m_create()**.

Format

```
oal_ret_t oal_task_m_delete (
    oal_task_t *            p_task,
    const oal_task_m_dsc_t *   task_dsc,
    oal_task_id_t            task_id )
```

Arguments

Argument	Description	Type
p_task	A pointer to the task.	oal_task_t *
task_dsc	The task descriptor.	oal_task_m_dsc_t *
task_id	The task ID.	oal_task_id_t

Return Values

Return value	Description
OAL_SUCCESS	Successful execution.
Else	See Error Codes.

oal_task_m_get_index

Use this function to get the index of a task created by using **oal_task_m_create()**.

Format

```
oal_task_index_t oal_task_m_get_index (
    const oal_task_m_dsc_t *   task_dsc,
    oal_task_id_t              task_id )
```

Arguments

Argument	Description	Type
task_dsc	The task descriptor.	oal_task_m_dsc_t *
task_id	The created task ID.	oal_task_id_t *

Return Values

Return value	Description
The index of the task, if found	Successful operation.
TASK_INDEX_INVALID	The task's index is invalid.

oal_task_create

Use this function to create an OAL task.

When tasks cannot be created dynamically, this function searches for the task described by *task_dsc* and sets *task_id*.

Format

```
oal_ret_t oal_task_create (
    oal_task_t *            p_task,
    const oal_task_dsc_t *  task_dsc,
    oal_task_id_t *         task_id )
```

Arguments

Argument	Description	Type
p_task	A pointer to the task.	oal_task_t *
task_dsc	The task descriptor.	oal_task_m_dsc_t *
task_id	The created task ID.	oal_task_id_t *

Return Values

Return value	Description
OAL_SUCCESS	Successful execution.
Else	See Error Codes.

oal_task_delete

Use this function to delete a caller task.

Format

```
oal_ret_t oal_task_delete ( oal_task_t * p_task )
```

Arguments

Argument	Description	Type
p_task	A pointer to the task.	oal_task_t *

Return Values

Return value	Description
OAL_SUCCESS	Successful execution.
Else	See Error Codes.

oal_task_get_id

Use this function to get the ID of the currently active task.

Format

```
oal_task_id_t oal_task_get_id ( void )
```

Arguments

Argument
None.

Return Values

Return value	Description
Task ID	The ID of the currently active task. This can have any value as long as each value returned uniquely identifies a task.
Else	See Error Codes.

oal_task_poll

Use this function to poll all the registered tasks.

> **Note:**
> - This function is present only if a non-OS port is used.
> - Do not call this function from within a task as recursion may result unless measures are taken to ensure it does not occur. If a specific task needs to be scheduled then call the task directly. In general the cleanest solution is only to call this function from a single location outside the context of any task.

Format

```
void oal_task_poll (void)
```

Arguments

None.

Return Values

Return value	Description
OAL_SUCCESS	Successful execution.
Else	See Error Codes.

oal_task_sleep

Use this function to suspend the caller task.

This suspends the running task for ms milliseconds.

Format

```
void oal_task_sleep ( uint32_t ms )
```

Arguments

Argument	Description	Type
ms	The number of milliseconds to sleep for.	uint32_t

Return Values

Return value	Description
OAL_SUCCESS	Successful execution.
Else	See Error Codes.

oal_task_yield

Use this function to yield the current task.

Format

```
void oal_task_yield ( void )
```

Arguments

Argument
None.

Return Values

Return value	Description
OAL_SUCCESS	Successful execution.
Else	See Error Codes.

7.5. Error Codes

The table below shows all the error codes that are returned by calls to OAL functions.

Return code	Value	Description
OAL_SUCCESS	0	Successful execution.
OAL_ERR_RESOURCE	1	A resource error caused this operation to fail.
OAL_ERR_TIMEOUT	2	The requested operation timed out before it could complete.
OAL_ERROR	3	An undefined error occurred in the function.

7.6. Types and Definitions

This section describes the main definitions that are used by the module.

Event Definitions

oal_event_t

The definition of the type is provided by the RTOS. Refer to the file **os/oalp_event.h** for the RTOS.

Element	Description	Type
oal_event_t	The event type.	uint32_t

oal_event_flags_t

The definition of the type is provided by the RTOS. Refer to the file **os/oalp_event.h** for the RTOS.

Element	Description	Type
oal_event_flags_t	The type of event flag.	uint16_t

oal_event_timeout_t

The definition of the type is provided by the RTOS. Refer to the file **os/oalp_event.h** for the RTOS.

Element	Description	Type
oal_event_timeout_t	The type of timeout used when waiting for an event. OAL_WAIT_FOREVER is defined as (*oal_event_timeout_t*) - 1; use this to set an endless wait for an event.	uint16_t

ISR Definitions

oal_isr_id_t

The ISR identifier is used to refer to the created ISR.

Element	Description	Type
id	The ISR identifier.	uint32_t

oal_isr_dsc_t

The *oal_isr_dsc_t* structure takes this form:

Element	Description	Type
id	The ISR ID. This can be set to anything and used by **oal_isr_install()** or **psp_isr_install()**. It can be a vector number, an identifier, and so on.	uint32_t
pri	The priority of the ISR. (This is not always required.)	uint32_t
fn	The ISR function.	oal_isr_fn_t

ISR Function and Code

These are defined as follows:

Element	Description	Type
OAL_ISR_FN	Definition of the ISR function.	void fn (void)
OAL_ISR_PRE	Code to execute at the beginning of the ISR.	
OAL_ISR_POST	Code to execute at the end of the ISR.	

oal_mutex_t

The definition of the type is provided by the RTOS. Refer to the file **os/oalp_mutex.h** for the RTOS.

Element	Description	Type
oal_mutex_t	The mutex type.	uint32_t

Task Definitions

oal_task_id_t

The definition of the type is provided by the RTOS; refer to its **os/oalp_task.h** file.

Element	Type	Description
oal_task_id_t	uint32_t	The task ID type.

oal_task_dsc_t

The *oal_task_dsc_t* structure takes this form:

Element	Type	Description
name	const char_t *	The task name.
entry	oal_task_fn_t	The entry point.
priority	uint32_t	The priority of the task.
stack_size	uint32_t	The stack required by the task.
stack_ptr	uint32_t *	A pointer to the stack. This is needed only if the OS needs to have it statically allocated. That is, if OAL_STATIC_TASK_STACK.

oal_task_m_dsc_t

The *oal_task_m_dsc_t* structure is used by the multitask functions. It takes this form:

Element	Type	Description
name	const char_t *	The task name.
entry	oal_task_fn_t	The entry point.
priority	uint32_t	The priority of the task.
stack_size	uint32_t	The stack required by the task.
stack_ptr	uint32_t *	A pointer to the stack. This is needed only if the OS needs to have it statically allocated. (That is, if OAL_STATIC_TASK_STACK).
task_count	uint32_t	The number of tasks.
p_ids	oal_task_id_t *	The maximum number of task IDs.
pb_ids_used	uint8_t *	The number of task IDs in use.

OAL_TASK_FN

This defines the function type for the specific RTOS; refer to its **os/oalp_task.h** file.

```
#define OAL_TASK_FN(fn)        void (fn) (void)
```

OAL_TASK_PRE

This must be written to all tasks as the first instruction after possible variable declarations. If for example there is a parameter for the task then OAL_TASK_PRE must be defined to *(void)param* to avoid possible warnings. For example:

```
#define OAL_TASK_FN(fn)     void (fn) (unsigned long param)
#define OAL_TASK_PRE        (void)param
```

8. Version

Version 2.00

For use with OS Abstraction Layer versions 2.03 and above