Elektrobit

# EB tresos® AutoCore OS documentation

product release 6.1

Elektrobit Automotive GmbH
Am Wolfsmantel 46
91058 Erlangen, Germany
Phone: +49 9131 7701 0
Fax: +49 9131 7701 6333
Email: info.automotive@elektrobit.com

## Technical support

https://www.elektrobit.com/support

## Legal disclaimer

# Table of Contents

# Begin here

## 1. If you are upgrading from a previous version

- To find what is new in this EB tresos AutoCore OS version:

  See the document EB tresos AutoCore OS release notes, located in your EB tresos AutoCore OS <INSTALL_PATH>/doc directory.

- Regarding the known problems, fixed problems, incompatibilities to previous releases, limitations and restrictions that have been reported for the current EB tresos AutoCore OS version:

  See the EB tresos AutoCore known problems, which you may download from the download server *EB Command*. The link to EB Command as well as your user login and password were sent to you via email.

- To migrate from an older version to the current version of EB tresos AutoCore OS:

  See the EB tresos AutoCore OS release notes, located in your EB tresos AutoCore OS <INSTALL_PATH>/doc directory.

## 2. If you are using EB tresos AutoCore OS for the first time

- If you are a first time user of EB tresos AutoCore OS, you can get familiar with some of the concepts behind the AUTOSAR Os module at Section 3.2, “Background Information”.
- Section 3.5, “Application example” gives you an example of an EB tresos AutoCore OS project along with instructions you may follow for practice.

## 3. If you want to know how to work with EB tresos AutoCore OS

In Section 3.3.1, “Development workflow” you find a description of the typical workflow when you work with EB tresos AutoCore OS. Instructions for performing these single steps are available in the user's guide, arranged in the order of the workflow:

- Configuring the `Os` module (Section 3.3.2, “Configuring and using the OS objects”).

- Verifying and generating the `Os` (Section 3.3.3, “Generating the code of the `Os` module”).
- Using the makefiles (Section 3.3.4, “Using the makefiles”).
- Generating a linker script (Section 3.3.5, “Creating a linker script”).

# 4. If you are an advanced user

- If you want to optimize the code of your module, see Section 3.3.6.1, “Optimizing your `Os` module”.
- If you want to build the `AUTOSAR Os module` with your own build environment, see Section 3.3.6.2, “Compiling EB tresos AutoCore OS in custom build environments”.

# 5. If you need help or further information

- Receive technical support via email or on our support page http://automotive.elektrobit.com/support .
- Information on the typographical and style conventions used throughout this documentation is provided in Chapter 1, “About this documentation“.
- A description of certain key words or abbreviations is provided in Glossary.

# 1. About this documentation

## 1.1. Typography and style conventions

The signal word *WARNING* indicates information that is vital for the success of the configuration.

**WARNING**

**Source and kind of the problem**

What can happen to the software?

What are the consequences of the problem?

How does the user avoid the problem?

The signal word *NOTE* indicates important information on a subject.

**NOTE**

**Important information**

Gives important information on a subject

The signal word *TIP* provides helpful hints, tips and shortcuts.

**TIP**

**Helpful hints**

Gives helpful hints

Throughout the documentation, you find words and phrases that are displayed in **bold**, *italic*, or `monospaced` font.

To find out what these conventions mean, see the following table.

All default text is written in Arial Regular font.

| Font | Description | Example |
|---|---|---|
| Arial italics | Emphasizes new or important terms | The *basic building blocks* of a configuration are module configurations. |
| Arial boldface | GUI elements and keyboard keys | 1. In the **Project** drop-down list box, select Project_A. |

| Font | Description | Example |
|---|---|---|
| | | 2. Press the **Enter** key. |
| Monospaced font (Courier) | User input, code, and file directories | The module calls the `BswM_Dcm_RequestSessionMode()` function.<br><br>For the project name, enter `Project_Test`. |
| Square brackets [ ] | Denotes optional parameters; for command syntax with optional parameters | `insertBefore [<opt>]` |
| Curly brackets {} | Denotes mandatory parameters; for command syntax with mandatory parameters | `insertBefore {<file>}` |
| Ellipsis … | Indicates further parameters; for command syntax with multiple parameters | `insertBefore [<opt>…]` |
| A vertical bar \| | Indicates all available parameters; for command syntax in which you select one of the available parameters | `allowinvalidmarkup {on\|off}` |

# 2. Safe and correct use of EB tresos products

## 2.1. Intended usage of EB tresos products

EB tresos products are intended to be used in automotive projects based on AUTOSAR. For more information on the AUTOSAR consortium, see www.autosar.org.

## 2.2. Possible misuse of EB tresos products

- You may use the software only in accordance with the intended usage and as permitted in the applicable license terms and agreements. Elektrobit Automotive GmbH assumes no liability and cannot be held responsible for any use of software not in compliance with the applicable license terms and agreements.
- If you use the product in applications that are not defined by the AUTOSAR consortium, the product and its technology may not conform to the requirements of your application. Elektrobit Automotive GmbH is not liable for such misuse.
- Use of this product without taking appropriate risk-reduction measures throughout the entire development phase can result in unexpected behavior. Elektrobit Automotive GmbH is not liable for this misuse. To find out about risk-reduction measures, see the **EB tresos maintenance and support annex**. You have received this document together with your product quote.

## 2.3. Target group and required knowledge

- Basic software engineers
- Application developers
- Programming skills and experience in programming AUTOSAR-compliant ECUs

# 3. User's guide

## 3.1. Overview

This chapter provides you the information about EB tresos AutoCore OS:

- Section 3.2, "Background Information" provides an overview of the functionality and the features of EB tresos AutoCore OS.
- Section 3.3, "Configuring the Operating system" provides information on how to configure EB tresos AutoCore OS, generate the configuration files, advanced configuration options and how to compile and link the operating system image.
- Section 3.4, "Using atomic functions" provides information on the use of atomics functions.
- Section 3.5, "Application example" provides information about an application example and its workflow.

## 3.2. Background Information

EB tresos AutoCore OS is an implementation of the operating system (OS) module of the classic AUTOSAR platform for multi-core processors. The OS includes support for single core processors as a subset.

The Os module is a static OS; all system objects and their characteristics, including their core assignment, are fixed by the configuration. Objects cannot be created nor destroyed dynamically at run-time. Neither can their configured characteristics be changed at run-time.

This chapter provides information on the following topics:

- Section 3.2.1, "Basic features" provides the information about the basic features in EB tresos AutoCore OS.
- Section 3.2.2, "Starting and stopping the OS" provides the information about how to start and stop EB tresos AutoCore OS.
- Section 3.2.3, "Protection features" provides the information about the protection features in EB tresos AutoCore OS.
- Section 3.2.4, "Interrupt handling" provides the information about the general interrupt handling in EB tresos AutoCore OS.
- Section 3.2.5, "Atomic functions" provides the information about the EB-specific atomic functions in EB tresos AutoCore OS.
- Section 3.2.6, "The Os generator" provides the information about the Os generator which is used to generate the configuration of EB tresos AutoCore OS.

- Section 3.2.7, "The directory structure" provides the information about the directory structure of EB tresos AutoCore OS.

## 3.2.1. Basic features

From the point of view of the operating system, a user application consists of a set of tasks that run concurrently using real-time scheduling. When the application is distributed on multiple cores, the scheduling operates independently on every core.

In addition to the tasks, the application contains interrupt service routines (ISRs), counters, alarms, schedule tables, resources, and hook functions. All of these objects are referred to collectively as OS objects.

OS applications allow you to group OS objects together. An OS application is assigned to a core with a specific logical core ID, and all of its OS objects run on that core.

A task is activated when another task or ISR calls `ActivateTask()` or `ChainTask()`. Basic tasks (task; basic) run to completion and call `TerminateTask()`. Extended tasks (task; extended) usually remain permanently active and wait for notifications by means of `WaitEvent()`. events can be sent to extended tasks by other tasks and ISRs by means of `SetEvent()`. Tasks can be activated and events sent across cores as well as on the local core.

Counters are used to provide a basis for time-triggered scheduling. Alarms and schedule tables attached to the counters can activate tasks or send events either cyclically or after a programmed delay.

Each task has a configured priority. When a task is running, it can be preempted by a higher-priority task but not by a task of the same or lower priority. Tasks that have been activated run in descending order of priority. For tasks of equal priority, the order of execution is the same as the order of activation. The dispatcher selects the task with the highest priority to run.

Resources provide a mutual exclusion (mutex) mechanism. A task can acquire a resource by calling `GetResource()`. For the time that a task occupies a resource, the OS raises the priority of the task to the ceiling priority of the resource, thus preventing preemption by other tasks that also use the resource. When the task calls `ReleaseResource()`, the OS restores the task's previous priority and tasks are then scheduled accordingly.

The resource mechanism is called a priority ceiling protocol and is free from the risk of deadlock and unbounded priority inversion. A task can also prevent preemption by locking interrupts using one of the three APIs `SuspendOSInterrupts()`, `SuspendAllInterrupts()`, and `DisableAllInterrupts()`. A task must unlock interrupts using one of the three corresponding APIs `ResumeOSInterrupts()`, `ResumeAllInterrupts()`, and `EnableAllInterrupts()`.

Resources and interrupt locks only apply to tasks running on the same core. They have no effect on tasks running on other cores. spinlocks provide mutual exclusion that applies across two or more cores. Tasks and ISRs can call `GetSpinlock()`, `TryToGetSpinlock()`, and `ReleaseSpinlock()`. Unlike resources, spinlocks can cause priority inversion and deadlock if not configured and used correctly.

The operating system calls hook functions at various times; the OS calls `PreTaskHook()` (hook; pre-task) just before switching to a task and `PostTaskHook()` (hook; post-task) just after switching away from a task. `PreISRHook()` (hook; pre-ISR) and `PostISRHook()` (hook; post-ISR) are the equivalent callouts for ISRs. `ErrorHook()` (hook; error) is called whenever an application requests a service that cannot be granted. For example, trying to activate a task that is already active or trying to send an event to a task that isn't active.

Each OS application can have application-specific start-up, shutdown, and error hooks that are defined by the user at configuration time. The application-specific hooks (hook; application-specific) are called just before or just after their corresponding global hook.

## 3.2.2. Starting and stopping the OS

Start-up is the sequence that takes an ECU from reset to running the OS.

After a reset, the low-level initialization code calls `main()` on one of the microcontroller's cores. This core is called the master core. The other cores remain in the reset state or in a holding loop, depending on the hardware.

`main()` on the master core normally calls `StartCore()` for each core on which the application shall run. Eventually, all cores call `StartOS()`. When `StartOS()` is called for the first time on a core, it does not return to its caller. If `StartOS()` is called again e.g. from a task, it calls an error hook if configured. Any problem detected by the `StartOS()` will result in a shutdown of the OS.

After initializing the OS, `StartOS()` calls the hook function `StartupHook()`, and then starts normal task scheduling. The user application usually provides an initialization task that the OS activates automatically during `StartOS()`. You can also configure schedule tables and alarms to start automatically during `StartOS()`.

All the cores that you configure must call `StartOS()` before any core starts normal scheduling.

Shutdown is the sequence that stops the OS running. The user application can call `ShutdownOS()` to shut down a core. In this case, the OS stops the scheduling activities on the calling core, calls `ShutdownHook()` and then enters an endless loop.

The user application can call `ShutdownAllCores()` to shut down all cores.

## 3.2.3. Protection features

EB tresos AutoCore OS provides *service protection*, *memory protection* and *timing protection* features.

*Service protection* prevents objects in one OS application from calling APIs to change the state of OS objects belonging to other OS applications, unless you configure permission to do so. If a task or ISR violates service protection, the OS calls the error hook.

*Memory protection* permits you to place variables in memory regions so that they can only be modified by objects of the OS applications that you permit to modify them. There are also memory regions that can only be modified by a specified task or ISR. The OS protects the private stack of a task or ISR too.

Memory protection only applies to the tasks, ISRs, and application-specific hook functions belonging to OS applications that you configure as non-trusted OS application (OS application; non-trusted). The tasks, ISRs, and hook functions of a non-trusted OS application run in an unprivileged mode of the processor. Memory protection detects and prevents stack overflow.

The tasks, ISRs, and hook functions belonging to trusted OS applications (OS application; trusted) run in a privileged mode of the processor. The OS itself and all the global hook functions run in privileged mode and are implicitly trusted. The OS does not apply memory protection to trusted objects, and therefore they can modify variables without restriction.

stack monitoring is an additional memory protection mechanism that monitors the amount of stack used by tasks, ISRs, and the OS. Unlike true memory protection, stack monitoring only detects overflow after it has happened. It can also fail to detect some kinds of overflow. For example, a function can declare an array that extends beyond its own task's stack and consequently overwrite data of another task without overwriting the stack end marker. Since the stack end marker is not overwritten this error is not detected. Stack monitoring applies to both trusted objects as well as non-trusted objects.

*Timing protection* allows you to configure the maximum allowed execution time for a task, as well as the maximum occupation time for resources and interrupt locks. In addition, you can configure a maximum arrival rate for interrupts and task activations.

If the OS detects a violation of its memory or timing protection it calls `ProtectionHook()` (hook; protection). The return value from `ProtectionHook()` determines how the OS reacts to the violation; the reactions range from terminating the offending task or ISR to shutting down the entire core.

### 3.2.4. Interrupt handling

This section provides an overview of the general handling of interrupts in EB tresos AutoCore OS. Architecture-specific details are described in the related architecture notes document.

The EB tresos AutoCore OS *kernel* configures all IRQs and handles all interrupts. The handling of interrupts is done by setting the appropriate registers of the interrupt controller. The operating system does not modify any other peripheral-specific interrupt settings outside of the interrupt controller e.g. the CAN peripheral interrupt settings.

The OS supports both category 1 and category 2 interrupts. Category 1 interrupts must have a higher priority than category 2 interrupts in the configuration. *User-defined interrupts* can be automatically enabled at startup by setting the `OsEnable_On_Startup` attribute for the ISR to `TRUE`. This is done by using the enable mechanism of the interrupt controller.

All interrupt routines are written in the following format:

```
ISR(isr_name)
{
  /* handling of interrupt */
  /* clear the interrupt flags */
  return;
}
```

The function body is written like a normal C function. The function should take care of the interrupt cause within the corresponding peripheral module. Not doing so may cause the very same interrupt request to be issued continuously.

For *category 1 interrupts*, the operating system stores the current context on the kernel stack before calling the ISR.

Execution-time monitoring is not suspended during execution of category 1 ISRs, so time spent in these ISRs is counted as part of the time for the execution of the interrupted task or ISR. It is therefore not recommended to use category 1 interrupts in conjunction with execution-time monitoring.

*Category 2 interrupts* are handled by the kernel with a full wrapper function and may exit via the dispatcher. The OS provides the appropriate prologue and epilogue code to create an environment in which OS API functions may be called.

Category 2 ISRs may use resources. They are supported by disabling all ISRs with a lower priority. As a consequence, a task may block ISRs if it takes a resource that is shared with ISR handlers as well.

*Nested interrupts* are supported by the OS for both category 1 and category 2 interrupts. Nesting of interrupts will occur according to the priority that you configure with the architecture-specific `OS<TARGET>IrqLevel` parameter.

**NOTE**

**Handling of unknown interrupts**

If an interrupt is triggered that has no configured ISR, EB tresos AutoCore OS will shut down.

## 3.2.5. Atomic functions

The need for atomic functions is motivated by developments in microprocessor design and the trend towards more concurrent architectures. This means that multiple threads of execution may exist at any point in time.

These threads must be coordinated to achieve proper synchronization with the serial parts of a program. Furthermore, the concurrent parts of a program need to work together without friction. Other synchronization mechanisms provided by EB tresos AutoCore OS can result in a high performance penalty due to the context switches and the many instructions executed when they are used. In some cases, this performance overhead is not acceptable, and for such cases specialized instructions provided by the hardware can be used as an alternative. Although these specialized instructions provide less complex synchronization mechanisms, an efficient algorithm can avoid the performance penalty mentioned above.

Key aspects to consider in this context are the *atomicity* and the *consistency* of memory accesses.

The term atomicity relates to a certain trait of memory accesses. Atomic accesses are never interrupted by other concurrent threads, even when they access the same memory location at the same time. They can only be executed either completely or not at all. Therefore the result of an atomic access is as if there is no contention at all. This also extends to read-modify-write instructions which are susceptible to interferences from other concurrent threads. Such interference can occur because the instructions first have to read from memory, then process the data, and then write it back. This gives other threads the opportunity to interrupt these steps so that the final value in memory might not be as expected.

With guaranteed atomicity these issues cannot occur and concurrent threads can work on shared data without the need of more expensive synchronization mechanisms provided by EB tresos AutoCore OS. This statement would be true if there weren't further optimizations done in hardware and during compilation which may reorder memory accesses.

The term consistency in this context refers to the order in which concurrent threads perceive the write accesses of other threads to shared memory locations. This ordering may be affected by two different layers: hardware and compiler.

The microprocessor hardware may employ different kinds of buffering. to avoid updating the system memory each time a write instruction is executed. Furthermore, read instructions may be executed speculatively ahead of time if this seems beneficial. The combined effects of these mechanisms must be taken into account if precise control of memory accesses is required, for example, when peripherals are operated or for synchronization algorithms.

The compiler may reorder read and write instructions for optimization purposes.

The atomic functions provided by EB tresos AutoCore OS enforce sequential consistency. This means that, firstly, hardware empties all of its buffers, so that the effects of past write instructions become visible to other threads. Furthermore, no later read instructions are executed speculatively and no later write instructions are executed. This gives an atomic function the properties of a memory fence. Such memory fences prevent the reordering of memory accesses at the hardware level. Sequentially consistent memory fences prevent read and write instructions from being moved either way across them.

Secondly, sequential consistency imposes a total order on all concurrent accesses to a shared memory location. The consequence is that all concurrent threads agree upon one and only one order in which they observe concurrent accesses to shared memory locations of all accessing threads. This is an important property of the atomic functions, because the implementation of synchronization algorithms depend on it.

### 3.2.6. The Os generator

The configuration files for the Os module can be created using a normal programmer's text editor, but the recommended method is to use EB tresos Studio with the graphical configurator plug-in. Detailed information on EB tresos Studio is available in the EB tresos Studio documentation, located in your folder `$TRESOS_BASE\doc\2.0_EB_tresos_Studio`.

### 3.2.7. The directory structure

EB tresos AutoCore OS is intended to be used with the EB tresos Studio environment and uses a default directory structure. The structure can be modified by an experienced user if needed. The base directory is located at `$TRESOS_BASE\`, which is the directory into which you have installed EB tresos Studio.

`$TRESOS_BASE\bin`
: Contains executables needed for running EB tresos Studio.

`$TRESOS_BASE\demos\AutoCore_OS\os_demo_<target>_<version>`
: Contains some example code. The demo is a good starting point when learning how to use EB tresos AutoCore OS.

`$TRESOS_BASE\doc`
: Contains the user documentation in PDF format.

`$TRESOS_BASE\plugins\Make_<variant_string>`
: Contains the `Make` plugin for EB tresos Studio. A compatible `Make` plugin is necessary for using the standard customer build environment.

`$TRESOS_BASE\plugins\Platforms_<variant_string>`
: Contains the `Platforms` plugin for EB tresos Studio. A compatible `Platforms` plugin is necessary for using the standard customer build environment.

`$TRESOS_BASE\plugins\Os_<variant_string>`
: Contains the `OS` plugin and is referred to as `$OS_BASE` in the following descriptions.

`$TRESOS_BASE\plugins\Os_<variant_string>_<release_suffix>`
: Contains release specific parts of the `OS` plugin.

`$OS_BASE\data`
: Contains subdirectories which hold data files for the `Os` Generator.

`$OS_BASE\make`
: The OS-specific parts of the standard customer build environment can be found in this directory. This is used to build the examples.

`$OS_BASE\src`
: Some source files are always compiled by the user because they depend on the configuration. These source files can be found in this directory. Architecture-specific source files can be found in the architecture subdirectory of this directory.

`$OS_BASE\lib_src`
: If you have a source-code license (for example in order to build optimized kernel libraries), the library source files can be found in this directory. The files are further divided into kernel files, user files, error-table files and architecture-specific files in appropriately-named subdirectories.

`$OS_BASE\include`
: The include subdirectory contains header files which are either referenced by the generated kernel or board specific files.

There may be some additional directories under `$OS_BASE`, depending on the specific architecture.

EB tresos AutoCore OS does not support directories with names containing spaces. Always ensure that the installation directories of EB tresos AutoCore OS and supporting tools (including the compiler toolchain) do not have spaces in their names.

## 3.3. Configuring the Operating system

This chapter provides the information about configuring EB tresos AutoCore OS:

- Section 3.3.1, “Development workflow” provides the information about the workflow starting from configuration, build and linking.
- Section 3.3.2, “Configuring and using the OS objects” provides the information about configuring EB tresos AutoCore OS i.e., configuring the parameters of EB tresos AutoCore OS.
- Section 3.3.3, “Generating the code of the `Os` module” provides the information about the steps to generate the configuration files for EB tresos AutoCore OS.
- Section 3.3.4, “Using the makefiles” provides the information about using the makefiles provided in EB tresos AutoCore OS.
- Section 3.3.5, “Creating a linker script” provides the information about how to adapt the linker scripts according to the memory configuration.
- Section 3.3.6, “Advanced configuring” provides the information about using advanced EB tresos AutoCore OS configurations like source optimization, AUTOSAR customization and using custom build environment.

### 3.3.1. Development workflow

To develop a complete operating system using EB tresos AutoCore OS, the following workflow applies:

- Create a configuration file in XDM format for the generator.
- Use the Os generator to create the source and header files that configure the kernel.
- Create source that contain all the OS objects like tasks, ISRs etc that are configured in the configuration file.
- Compile all source files to relocatable object files.

- Create a linker script.
- Link the object files and libraries to produce the system binary image.

The order given above is only an illustration; some of the activities can be carried out in parallel.

## 3.3.2. Configuring and using the OS objects

To configure the OS with EB tresos Studio, you need to add the `Os` module to your EB tresos Studio project. Instructions for adding modules to your EB tresos Studio project are available in the EB tresos Studio documentation, chapter `EB tresos Studio user's guide/Editing module configurations`.

**TIP**

**Parameter descriptions**

User can find parameter descriptions here:

- Context sensitive help in the **Description** view on the lower right corner of EB tresos Studio
- In the list at Chapter 4, "References"

This chapter explains about the configuration of the following:

- Section 3.3.2.1, "Configuring and using general parameters" provides the information about configuring the general parameters of EB tresos AutoCore OS.
- Section 3.3.2.2, "Configuring the OS objects in EB tresos Studio and using the OS objects" provides the information about adding and configuring OS objects in EB tresos AutoCore OS.

### 3.3.2.1. Configuring and using general parameters

When you start configuring a new operating system, it is recommended to start with the configuration of the general parameters of the `Os` module. To configure the general parameters of the `Os` module:

1. Open the **OsOS** tab of the **OS** editor:
2. Configure the parameters to your needs. For information about the single parameters, see Section 4.1, "EB tresos AutoCore OS configuration language".

   Detailed information about the parameters of the `OsAutosarCustomization` container is available at Section 3.3.6.1.2, "Enabling kernel customizations".

### 3.3.2.2. Configuring the OS objects in EB tresos Studio and using the OS objects

This section describes about adding and configuring the below mentioned important OS objects:

- Section 3.3.2.2.1, "Configuring and using OS application" provides the information about adding and configuring applications in EB tresos AutoCore OS and adapting the user source for using the applications.
- Section 3.3.2.2.2, "Configuring and using OsTask" provides the information about adding and configuring tasks in EB tresos AutoCore OS and adapting the user source for using the task.
- Section 3.3.2.2.3, "Configuring and using OsIsr" provides the information about adding and configuring ISRs in EB tresos AutoCore OS and adapting the user source for using the ISRs.
- Section 3.3.2.2.4, "Configuring and using OsResource" provides the information about adding and configuring resource in EB tresos AutoCore OS and adapting the user source for using the resource.
- Section 3.3.2.2.5, "Configuring and using OsEvent" provides the information about adding and configuring events in EB tresos AutoCore OS and adapting the user source for using the events.
- Section 3.3.2.2.6, "Configuring and using OsAlarm" provides the information about adding and configuring alarms in EB tresos AutoCore OS and adapting the user source for using the alarms.
- Section 3.3.2.2.7, "Configuring and using OsSpinlock" provides the information about adding and configuring spinlock in EB tresos AutoCore OS and adapting the user source for using the spinlock.
- Section 3.3.2.2.8, "Configuring and using OsCounter" provides the information about adding and configuring counters in EB tresos AutoCore OS and adapting the user source for using the counters.
- Section 3.3.2.2.9, "Configuring and using OsScheduleTable" provides the information about adding and configuring schedule tables in EB tresos AutoCore OS and adapting the user source for using the schedule tables.

#### 3.3.2.2.1. Configuring and using OS application

All the OS objects that are configured has to be mapped to the physical core of the hardware via OS application.

If no user application is configured, all the OS objects will be mapped to default SYSTEM application.

Configuring OS application

Step 1
Open the **OsApplication** tab of the **OS**

Step 2
In the **OsApplication** tab add a new application with appropriate name.

Step 3
Open the newly added application and configure the following parameters in the general tab:

Step 3.1
**OsTrusted**, select either true or false.

Step 3.2
**OsRestartTask**, specify the task for the application to begin with in case of an application restart.

Step 3.3
**OsApplicationHooks**, enable or disable the application specific hooks along with the stack size for each hook if enabled.

Step 3.4
**OsApplicationCoreAssignment**, enter the core ID to which the application has to be mapped in the hardware. This depends on the hardware support.

Step 4
Map all the OS objects configured to the application in the respective tabs **OsAlarmRef**, **OsTaskRef**, **OsIsrRef** etc.

Step 5
You have to create a source file "Application_Name.c" to map all the private variables (initialized and non-initialized) and private constants that belong to the application using its own private section (helps in achieving memory protection).

**NOTE**

**Grouping and mapping of system objects**

EB tresos AutoCore OS offers OS applications which are a collection of different OS objects. Normally, OS applications are only available in specific scalability classes (SC). For a single-core OS, they are only available in scalability classes 3 and 4 (SC3/SC4). In a multi-core system, the scalability classes SC1 and SC2 allow the usage of OS applications, since the OS applications are used to map OS objects to cores. As a consequence, all OS objects must belong to an OS application. The only exception are spinlocks, which are used to synchronize access to shared data.

#### 3.3.2.2.2. Configuring and using OsTask

Configuring OsTask:

Step 1
Open the **OsTask** tab of the **Os**

Step 2
In the **OsTask** tab add a new task with appropriate name.

Step 3
Open the newly added task and configure the following parameters in the general tab:

Step 3.1
**OsTaskActivation**, enter the maximum number of activations allowed for the task.

Step 3.2
**OsTaskPriority**, enter the base priority of the task.

Step 3.3
**OsTaskType**, select the task type either BASIC or EXTENDED.

Step 3.4
**OsStacksize**, enter the size of the stack to be allocated for the task.

Step 3.5
**OsTaskTimingProtection**, enable and configure this container accordingly in case timing protection is required for the Task.

Step 3.6
**OsTaskAutostart**, either enable or disable this parameter.

Step 3.7
**OsTaskSchedule**, select the type of scheduling NON or FULL or MIXED.

Step 4
In the tab **OsTaskAccessingApplication** select applications from the list which will access this task.

Step 5
In the tab **OsTaskEventRef** select the events from the list that are used by this task.

Step 6
In the tab **OsTaskResourceRef** select the resources from the list that are used by this task.

Step 7
You have to declare the task created and add a task body. For information on how to declare a task refer DeclareTask and on how to add a task body refer TASKBody.

#### 3.3.2.2.3. Configuring and using OsIsr

Configuring OsIsr:

Step 1
Open the **OsIsr** tab of the **Os**

Step 2
In the **OsIsr** tab add a new ISR with appropriate name.

Step 3
Open the newly added ISR and configure the following parameters in the general tab:

Step 3.1
**OsIsrCategory**, select the category of the ISR from the list. Either CATEGORY1 or CATEGORY2.

Step 3.2
**Os<TARGET>Vector**, select the interrupt to which the ISR has to be mapped to from the list of available interrupts on the hardware. This parameter is hardware specific.

Step 3.3
**Os<TARGET>IrqLevel** , select the priority level of the interrupt. This parameter is hardware specific.

Step 3.4
**OsStacksize**, enter the size of the stack to be allocated for the ISR.

Step 3.5
**OsIsrTimingProtection**, enable and configure this container accordingly in case timing protection is required for the ISR.

Step 4
In the tab **OsIsrAccessingApplication** select applications from the list which will access this ISR.

Step 5
In the tab **OsIsrResourceRef** select the resources from the list that are used by this ISR.

Step 6
You have to add the required ISR body. For information on how to add an ISR body refer ISRBody.

#### 3.3.2.2.4. Configuring and using OsResource

Configuring OsResource:

Step 1
Open the **OsResource** tab of the **Os**

Step 2
In the **OsResource** tab add a new resource with appropriate name.

Step 3
Open the newly added resource and configure the following parameters in the general tab:

Step 3.1
**OsResourceProperty**, select the type of resource STANDARD, LINKED or INTERNAL.

Step 3.2
**OsResourceLinkedResourceRef**, if the resource type is linked select the resource to which it is linked.

Step 4
In the tab **OsResourceAccessingApplication** select applications from the list which will access this resource.

Step 5
You have to declare the resource created. For information on how to declare a resource refer DeclareResource.

#### 3.3.2.2.5. Configuring and using OsEvent

Configuring OsEvent:

Step 1
Open the **OsEvent** tab of the **Os**

Step 2
In the **OsEvent** tab add a new event with appropriate name.

Step 3
Open the newly added event and configure the following parameters in the general tab:

Step 3.1
**OsEventMask**, enter the mask value for the event.

Step 4
You have to declare the event created. For information on how to declare a event refer DeclareEvent.

#### 3.3.2.2.6. Configuring and using OsAlarm

Configuring OsAlarm:

Step 1
Open the **OsAlarm** tab of the **Os**

Step 2
In the **OsAlarm** tab add a new alarm with appropriate name.

Step 3
Open the newly added alarm and configure the following parameters in the general tab:

Step 3.1
**OsAlarmCounterRef**, select the counter from the list which is the reference for the alarm.

Step 3.2
**OsAlarmAction**, select the alarm action from one of the following: **OsAlarmActivateTask**, **OsAlarmCallback**, **OsAlarmIncrementCounter**, and **OsAlarmSetEvent**.

Step 3.3
In the case of **OsAlarmActivateTask** action select the task to be activated, in the case of **OsAlarmCallback** action provide the name of the callback function, in the case of **OsAlarmIncrementCounter** select the counter which has to be incremented and in the case of **OsAlarmSetEvent** select the event which has to be set.

Step 3.4
**OsAlarmAutostart**, either enable or disable. If enabled, set the tick value when the alarm expires for the first time using **OsAlarmAlarmTime**, select the type of autostart for the alarm using **OsAlarmAutostartType**, set the cycle time of a cyclic alarm using **OsAlarmCycleTime**, and finally select the time unit type for the alarm using **OsTimeUnit**.

Step 4
In the tab **OsAlarmAccessingApplication** select applications from the list which will access this alarm.

Step 5
You have to declare the alarm created. For information on how to declare an alarm refer DeclareAlarm.

#### 3.3.2.2.7. Configuring and using OsSpinlock

Configuring OsSpinlock:

Step 1
Open the **OsSpinlock** tab of the **Os**

Step 2
In the **OsSpinlock** tab add a new spinlock with appropriate name.

Step 3
Open the newly added spinlock and configure the following parameters in the general tab:

Step 3.1
**OsSpinlockSuccessor**, select the spinlock that has to succeed this spinlock when acquired.

Step 3.2
**OsSpinlockLockMethod**, select the lockmethod from the following: LOCK_ALL_INTERRUPTS, LOCK_CAT2_INTERRUPTS, LOCK_NOTHING, and LOCK_WITH_RESSCHEDULER.

Step 4
In the tab **OsSpinlockAccessingApplication** select applications from the list which will access this spinlock.

#### 3.3.2.2.8. Configuring and using OsCounter

Configuring OsCounter:

Step 1
Open the **OsCounter** tab of the **Os**

Step 2
In the **OsCounter** tab add a new counter with appropriate name.

Step 3
Open the newly added counter and configure the following parameters in the general tab:

Step 3.1
**OsCounterMaxAllowedValue**, enter the maximum allowed count value for the counter.

Step 3.2
**OsCounterMinCycle**, enter the minimum allowed tick value for an alarm linked to the counter to be used to perform an action.

Step 3.3
**OsCounterTicksPerBase**, enter the number of ticks per time base.

Step 3.4
**OsCounterType**, select the type of counter HARDWARE or SOFTWARE.

Step 3.5
**OsCounter<TARGET>Timer**, if the counter type is HARDWARE select the timer available in the hardware that is mapped to the counter. There can be only one hardware counter mapped to a timer.

Step 3.6
**Os<TARGET>IrqLevel** , select the priority level of the timer interrupt. Applicable only for HARDWARE counters.

Step 4
In the tab **OsCounterAccessingApplication** select applications from the list which will access this counter.

#### 3.3.2.2.9. Configuring and using OsScheduleTable

Configuring OsScheduleTable:

Step 1
Open the **OsScheduleTable** tab of the **Os**

Step 2
In the **OsScheduleTable** tab add a new schedule table with appropriate name.

Step 3
Open the newly added schedule table and configure the following parameters in the general tab:

Step 3.1
**OsScheduleTableDuration**, enter the duration of the schedule table.

Step 3.2
**OsScheduleTableRepeating**, either enable or disable.

Step 3.3
**OsScheduleTableCounterRef**, select the counter reference for the schedule table.

Step 3.4
**OsScheduleTableAutostart**,either enable or disable. If enabled, select the type of autostart for the schedule table using **OsScheduleTableAutostartType** and the starting offset value for the schedule table using **OsScheduleTableStartValue**.

Step 3.5
**OsScheduleTableSync**, if enabled select the precision threshold using **OsScheduleTblExplicitPrecision** and select the synchronization strategy using **OsScheduleTblSyncStrategy**.

Step 4
In the tab **OsSchTblAccessingApplication** select applications from the list which will access this schedule table.

Step 5

Step 5.1
In the tab **OsScheduleTableExpiryPoint** add the expiry points for the schedule table.

Step 5.2
In the newly added expiry point, in general tab specify the expiry point using **OsScheduleTblExpPointOffset** parameter. This expiry point offset must be within the range of **OsScheduleTableDuration**.

Step 5.3
In **OsScheduleTableEventSetting** add an entry and select the event to be set during this expiry point.

Step 5.4
In **OsScheduleTableTaskActivation** add an entry and select the task to be activated during this expiry point.

### 3.3.3. Generating the code of the `Os` module

After you have configured your `Os` module, you need to generate the code. You may either generate the code using the EB tresos Studio user interface or on the command line. If you generate the code with the EB tresos Studio interface, you can verify the code before generating it.

To generate or verify the code of your project with EB tresos Studio:

- select your project in the **Project Explorer** view of EB tresos Studio.
- To verify the configuration of your project, click on the **Verify** button in the toolbar of EB tresos Studio.
- To generate code for your project, click on the **Generate** button in the toolbar of EB tresos Studio.

Per default, the code is generated into the folder <INSTALL_PATH>\workspace\<project_name>\output.

For further information on generating code with EB tresos Studio, see the EB tresos Studio documentation.

### 3.3.4. Using the makefiles

The AUTOSAR standard build environment uses a set of two configuration makefiles for each module. Additionally, the shipment contains a set of compiler plugins for the supported toolchains and a set of plugins for the configuration environment EB tresos Studio. EB tresos AutoCore OS specific plugin files are located in `$OS_BASE\make`:

`Os_defs.mak`
: The `Os_defs.mak` file describes all files that need to be built, directories that must be created and where output files are placed. It defines all generic files that are part of the OS-libraries. Architecture- and derivative-specific files are included from this file. They define the extra files that are needed for each architecture and derivative.

`Os_rules.mak`
: The `Os_rules.mak` file contains rules concerning the OS, e.g. the **clean** rule. The generation rule is part of the EB tresos Studio plugin files.

In addition to these files, the following file is located in the project's `util` directory:

`$TARGET_$DERIVATIVE_$TOOLCHAIN_cfg.mak`
: The `$TARGET_$DERIVATIVE_$TOOLCHAIN_cfg.mak` file is part of the application and contains options for building the OS.

## 3.3.5. Creating a linker script

The linker script defines the placement in memory of all text and data sections, and defines symbols that are used by the kernel and start-up code to locate the sections for initialization and memory protection purposes.

If the system you are developing does not need to use memory protection, the standard linker script for the board and toolchain can be used. This linker script places all `.text` sections and `.rodata` sections together in ROM and all `.data` sections and `.bss` sections together in RAM. The `.data` and `.bss` sections are initialized at start-up by the board-specific start-up code.

For systems using memory protection it is necessary to create a custom linker script. The script can gather together the code and data sections that belong to applications and can define symbols that the kernel needs in order to find these sections. You may either generate a linker script using a supplied Perl program, or create the linker script by hand. This is described in the following sections. To optimize your linker script, you can have a look at this hints for tuning the linker script in Section 3.3.5.4, “Tuning the linker script for memory protection”

### 3.3.5.1. Generating the linker script with Perl

The EB tresos AutoCore build environment already includes rules to generate a linker script for projects using memory protection. These rules are automatically enabled as soon as the appropriate parameters for memory protection are configured. Linking, especially the mapping between protected OS objects and their memory regions, is based on the object file names.

To inform the linker script generator about your implementation, you need to map your object files to OS objects by providing the following variables in your `Makefile`:

`CC_FILES_TO_BUILD`
: Holds all files to build for your project. Here, the source files for your OS objects must be added.

`OBJS_`*xxx*
: Holds the names of all object files holding code or data for OS object *xxx*.

For example, assume you have configured a non-trusted OS application called `App`, consisting of two tasks named `FooTask` and `BarTask`. For each task, you have a corresponding C file containing its code and the task-private data. You want both tasks to share some private data; of the application. The C definition of the corresponding variables is in the file `Appdata.c`. In your `Makefile`, you would add the following section:

```
CC_FILES_TO_BUILD += $(PROJECT_ROOT)\source\Appdata.c \
                     $(PROJECT_ROOT)\source\FooTask.c \
                     $(PROJECT_ROOT)\source\BarTask.c

                     OBJS_App = Appdata.$(OBJ_FILE_SUFFIX)
                     OBJS_FooTask = FooTask.$(OBJ_FILE_SUFFIX)
                     OBJS_BarTask = BarTask.$(OBJ_FILE_SUFFIX)
```

Then you can call `make`, and the Perl linker script generator would be invoked. It's output is put in your project's output directory, usually at `$(PROJECT_ROOT)/output/generated`.

### 3.3.5.2. Creating a linker script by hand

This section assumes that you are familiar with your toolchain and have knowledge about writing linker scripts. It describes the linker symbols expected by EB tresos AutoCore OS. This description features commonly used symbols; see your architecture notes for hardware specific features.

In the following descriptions, a *start* address of a region specifies the first address of that region and an *end* address specifies the first address (greater than or equal to the *start* address) that does not belong to the region.

The kernel, and the associated start-up code provided with the kernel, expects the linker script to define the following symbols:

`__STARTDATA`
: Start of the global data section

`__ENDDATA`
: End of the global data section

`__STARTBSS`
: Start of the global bss section

`__ENDBSS`
: End of the global bss section

`__INITDATA`
: Start of the ROM image for the global data section

The above symbols are only used by the default start-up code in `board.c`. If you are not using the default start-up code, these symbols do not need to be defined. If you use the default start-up code, but nevertheless provide your own memory initialization code (or the compiler's start-up code), set these symbols to NULL (0). Or in case of the KEIL ARM toolchain, create empty execution regions with `0x0` as start address.

The following symbols are required, when OsOS/OsTrappingKernel is enabled:

`__GLBL_TEXT_START`
: Start of the program text section

`__GLBL_TEXT_END`
  End of the program text section

The above symbols are used by the kernel to set up the code protection registers for all non-trusted applications. All memory between the two symbols is executable and the rest is non-executable.

`__GLBL_RODATA_START`
  Start of the read-only data (constants, C strings etc.) in ROM

`__GLBL_RODATA_END`
  End of the read-only data

`__GLBL_DATA_START`
  Start of the variable data and bss

`__GLBL_DATA_END`
  End of the variable data and bss

The above symbols are used by the kernel to grant non-trusted applications read-only access to ROM data and data belonging to other applications. All memory between the two symbols in each pair is readable. On processors, such as TriCore, that have a limited number of regions, all memory between the lesser of the two START symbols and the greater of the two END symbols delimit a single read-only region for all non-trusted applications.

`__DATA_`*xxx*
  Start of the private data and bss belonging to the non-trusted application, task or ISR named *xxx*.

`__DEND_`*xxx*
  End of the private data and bss belonging to the non-trusted application, task or ISR named *xxx*.

The above symbols mark the private data belonging to the named object. The data belonging to an application is readable and writeable by all tasks and ISRs in that application. The data belonging to a task or ISR is readable and writeable only by that task or ISR. Other tasks, ISRs and applications gain read-only access through the global region. The linker must define these symbols for each application, and for each task and ISR. Setting these symbols to NULL (0) indicates that the named object has no private data.

**NOTE** **The kernel does not use these symbols for trusted applications**

The kernel does not use these symbols for trusted applications and for each task and ISR that belongs to a trusted application. For trusted applications, their tasks and ISRs, you should define these symbols to NULL (0).

`__IDAT_`*xxx*
  Start of the initialization data for the non-trusted application, task or ISR named *xxx*.

`__IEND_`*xxx*
  End of the initialization data for the non-trusted application, task or ISR named *xxx*.

These symbols are used by the code that initializes the private data areas during `StartOS()`. The ROM image from `__IDAT_`*xxx* to `__IEND_`*xxx* is copied to the addresses `__DATA_`*xxx* and so on. The RAM data region must therefore be bigger than, or equal in size to, the ROM image. Any remaining portion of the `__DATA_`*xxx* region is set to zero. It is therefore assumed that the linker places all `.data` sections belonging to the object into the area, followed by all `.bss` sections. Setting these symbols to NULL (0) has no special meaning. If the symbols are equal, no data will be copied, but the entire `__DATA_` area will be set to zero. If you wish to provide your own initialization code or use that provided by the compiler, it is therefore necessary to override the kernel's initialization of private data areas. This can be achieved by overriding the kernel's initialization function with an empty stub, i.e. by linking the following code:

```
void OS_InitApplicationData(void)
{
}
```

The Perl scripts provided with EB tresos AutoCore OS create a linker script with a default memory layout and define all these symbols accordingly. However, the default layout will not suit all systems. You can write your own layout program based on the scripts provided, or simply create a linker script manually.

### 3.3.5.3. Support for the KEIL ARM toolchain

To support the KEIL ARM toolchain with EB tresos AutoCore OS a slightly different approach was implemented. This subchapter describes how the needed symbols can be set using the `armlink` linker and what rules should be followed to satisfy EB tresos AutoCore OS.

Because it is not possible to set additional global symbols within the `armlink` linker script, the auto generated linker symbols for set execution regions must be used. Therefore EB tresos AutoCore OS relies on a corresponding naming of created execution regions inside of the linker script file.

If the provided start-up code is used, following names for the global data and bss section must be used:

`data_DATA`
: Contains all global data objects

`bss_DATA`
: Contains all global bss objects

The linker automatically creates the following symbols for the above mentioned execution regions that are used within the start-up code to copy the data and initialize the bss section:

`Image$$data_DATA$$Base`
: Start of the global data section

  also available as preprocessor define: `OS_TOOL_STARTDATA`

`Image$$data_DATA$$ZI$$Limit`
: End of the global data section

  also available as preprocessor define: `OS_TOOL_ENDDATA`

`Load$$data_DATA$$Base`
: Start of the ROM image for the global data section

  also available as preprocessor define: `OS_TOOL_INITDATA`

`Image$$bss_DATA$$Base`
: Start of the global bss section

  also available as preprocessor define: `OS_TOOL_STARTBSS`

`Image$$bss_DATA$$ZI$$Limit`
: End of the global bss section

  also available as preprocessor define: `OS_TOOL_ENDBSS`

For the OsOS/OsTrappingKernel support, the symbols mentioned in Section 3.3.5.2, “Creating a linker script by hand” must be provided. At the KEIL ARM toolchain this can be achieved by creating empty execution regions at appropriate spots. Since the symbols are only used to setup the memory protection and not to copy any data, this is totally sufficient.

Empty execution sections with the following names must be created to generate the expected symbols that are named below.

`__GLBL_TEXT_START`
: Empty execution section used to mark the start of the text section

`__GLBL_TEXT_END`
: Empty execution section used to mark the end of the text section

`__GLBL_RODATA_START`
: Empty execution section used to mark the start of the rodata section

`__GLBL_RODATA_END`
: Empty execution section used to mark the end of the rodata section

`__GLBL_DATA_START`
: Empty execution section used to mark the start of the variable data and bss sections

`__GLBL_DATA_END`
: Empty execution section used to mark the end of the variable data and bss sections

From the above specified execution sections the linker creates the following symbols, which are used in EB tresos AutoCore OS:

`Load$$__GLBL_TEXT_START$$Base`
: Start of the program text section

  also available as preprocessor define: `OS_TOOL_TEXT_START`

`Load$$__GLBL_TEXT_END$$Base`
: End of the program text section

also available as preprocessor define: `OS_TOOL_TEXT_END`

`Load$$__GLBL_RODATA_START$$Base`
: Start of the read-only data (constants, C strings etc.) in ROM

also available as preprocessor define: `OS_TOOL_RODATA_START`

`Load$$__GLBL_RODATA_END$$Base`
: End of the read-only data

also available as preprocessor define: `OS_TOOL_RODATA_END`

`Image$$__GLBL_RAM_START$$Base`
: Start of the variable data and bss

also available as preprocessor define: `OS_TOOL_RAM_START`

`Image$$__GLBL_RAM_END$$Base`
: End of the variable data and bss

also available as preprocessor define: `OS_TOOL_RAM_END`

The naming of execution sections for non-trusted applications, tasks, and ISRs must follow a specific structure. In case the data and bss regions are split in two sections for each of the provided functions, it does not mean that the sections can be located separately in the memory. The bss section must always follow the data section.

To achieve the right initialization of this section, the following name scheme must be used:

`data_`*xxx*
: The name of the data section must be prefixed by the word `data_` following the name of the application, task or ISR (*xxx*)

`bss_`*xxx*
: The name of the bss section must be prefixed by the word `bss_` following the name of the application, task or ISR (*xxx*)

From this sections the linker creates the following symbols which are used to copy the data and to initialize the bss section:

`Image$$data_`*xxx*`$$Base`
: Start of the private data and bss belonging to the non-trusted application, task or ISR named*xxx*.

`Image$$bss_`*xxx*`$$ZI$$Limit`
: End of the private data and bss belonging to the non-trusted application, task or ISR named *xxx*.

`Load$$data_`*xxx*`$$Base`
: Start of the initialization data for the non-trusted application, task or ISR named *xxx*.

`Load$$data_`*xxx*`$$ZI$$Limit`
: End of the initialization data for the non-trusted application, task or ISR named *xxx*.

**TIP**

**Load and execution address**

Take care of the difference between the load (prefix `Load`, normally the address at ROM) and execution address (prefix `Image`, normally the address at RAM) to achieve a successful copy process with EB tresos AutoCore OS.

### 3.3.5.4. Tuning the linker script for memory protection

The memory layout generated by the Perl program works and provides a near-optimal protection. However, this comes at the cost of potentially leaving large areas of memory unused and unusable.

In most real applications it is necessary to tune the linker script generation process or tune the linker script manually to provide the best possible protection, within the limitations of the processor or board. The following paragraphs give information on how to optimize the configuration of the OS and the linking process without seriously compromising the protection.

#### 3.3.5.4.1. Sharing data within an application

If it is not really necessary to protect tasks and ISRs within an application from each other, the private data for all the tasks and ISRs in an application can be placed in the same page as the application's own private data. This can be achieved by two methods:

- List all the files that belong to the tasks and ISRs as belonging to the application; or
- Modify the linker script and place all the task and ISR data sections in the same page as the application's data.

#### 3.3.5.4.2. Restricting the task and ISR stack sharing

If the Os generator shares stacks among all objects that do not preempt each other, the stacks need to be placed in their own pages. This is necessary to prevent a task or ISR from gaining write access to another application's data through its stack permissions.

If sharing is restricted to within applications using the configuration option, the task and ISR stacks for applications can be placed in the same page as the application's data. This reduces the effectiveness of the stack-overflow protection, but this can be mitigated by placing the stack at the bottom of the page. In any case, stack overflow and stack underflow can only affect the application and not the whole system.

#### 3.3.5.4.3. Placing the hook stacks all in the same page

The kernel allocates two stacks for the hook functions of non-trusted applications: one for start-up and shutdown hooks, and one for error hooks. On analysis, it can be clear that these can never be used simultaneously by 2 applications, so it is safe to place them both in the same page.

#### 3.3.5.4.4. Placing the kernel stack and data in the same page

The kernel stack is best placed at the bottom of the available RAM, so if a kernel stack overflow occurs, the processor can trap to an exception handler. The kernel's data can be placed in the same page as the kernel's stack. However, many linkers process their linker script serially, so the selection of data from remaining files must appear at the end of the script. Without knowing beforehand how much data is used outside non-trusted applications, it is impossible to reserve a number of pages for the data of trusted code, so the linker script places kernel stack and kernel data together at the upper end of the allocated memory.

When the characteristics of the system are better known, it should be possible to place the kernel stack and non-trusted data lower in memory.

#### 3.3.5.4.5. Taking care of the alignment of memory regions

If a memory region covers 4 or more minimum-size pages, i.e., if it is bigger than 12 kB, then the number of Translation Lookaside Buffer (TLB) entries required can change depending on the alignment of the region. A region of between 12 and 16 kB aligned on a 16 kB boundary only needs one TLB entry. If it is aligned on an odd 4 kB boundary or an odd 8 kB boundary it can require 4 TLB entries, so larger memory regions should be aligned with care.

### 3.3.5.5. Mapping the memory using `Memmap.h`

EB tresos AutoCore OS supports the standard AUTOSAR memory mapping mechanism provided by `MemMap.h`. Since placing the various sections of the OS to specific memory regions is a very crucial task on many architectures, i.e., for CPUs which have protection mechanisms or which use banked memory, the use of `MemMap.h` is disabled by default.

---

**NOTE**

**Mapping via `Memmap.h` usually is not necessary**

For normal use cases, mapping the memory using `Memmap.h` is not necessary. For example, in a protected system using a linker script like explained above, it is sufficient to map the memory based on the names of the object files.

---

To enable the `MemMap.h` support, compile EB tresos AutoCore OS with the macro `OS_MEMMAP` set to `1`.

## 3.3.6. Advanced configuring

### 3.3.6.1. Optimizing your `Os` module

EB tresos AutoCore OS is highly configurable. Even when using the standard library, all the standard functionality of the standard AUTOSAR Os, right up to scalability class 4, is available. The disadvantage of this approach is that the kernel is often much bigger and slower than it needs to be for a given ECU, even with all the techniques that are used in the kernel to avoid linking unnecessary functions and data into the final binary. As a countermeasure, it is possible to build customized libraries tailored to your configuration.

There are two ways to optimize your `Os` module:

- Section 3.3.6.1.1, “Optimizing the library”: make your kernel smaller and thus faster.
- Section 3.3.6.1.2, “Enabling kernel customizations”: activate optimization options in the configuration.

#### 3.3.6.1.1. Optimizing the library

EB tresos AutoCore OS is built as a library. This means that functions and data that are not referenced can normally not be linked into the final binary. The configurability is achieved by making extensive use of decisions based on external (ROM) constants, and function pointers that can be redirected to null (empty) functions. This method of construction means that we can minimize the size of the kernel in case the operating system is only compiled once and is used as a generic library, but it means that at each decision there is code for both the true and false cases even though the decision always follows the same branch for a given configuration.

For function pointers, the overhead of calling the null function and returning from it is still present. So you can eliminate all the unnecessary decisions and unneeded function calls from the final kernel. The way you achieve this is by building a customized library that is exactly tailored to a given configuration. The optimizations are determined from the OS configuration. Many of them come from the standard configuration, but there is a set of OS customization options that can result in a smaller, faster kernel with the possible loss of some AUTOSAR conformance.

If you want the kernel to be smaller or faster, you need an optimized build. Depending on the target processor and the configuration, an optimized kernel can be as small as around 30% of the size of a standard kernel. There can also be performance gains, although not as dramatic.

If compile time is a problem, rather use a standard library in the early stages of a project, when the configuration is undergoing change. If optimization is used extensively while the configuration is changing, lots of customized libraries can be generated. The OS's library directory can fill up with the different library versions and the corresponding object files, but these can be deleted if disk space becomes a problem.

#### 3.3.6.1.1.1. Building an optimized library with the EB tresos AutoCore OS build environment

To build an optimized library with the EB tresos AutoCore OS build environment:

- Open the **OsOS** tab of the **OS (Os)** editor.
- Check the **OsSourceOptimization** switch.
- Generate the project.

As a result, the make variable `OS_BUILD_OPTIMIZED_LIB_FROM_SOURCE` is set to `TRUE` in the file `Os_objects.make`. The C preprocessor macro `OS_USE_OPTIMIZATION_OPTIONS` is defined as `1` in the file `Os_libcfg.h`. This causes the definitions of the `OS_EXCLUDE_something` macros in the same file to be enabled. Both files are created by the Os generator in the output directory.

By setting `OS_BUILD_OPTIMIZED_LIB_FROM_SOURCE` to `TRUE`, the name of the object file directory and the kernel library get a library ID encoded into them. This ID identifies uniquely all the optimizations that affect the kernel library, so that if you change your configuration in a way that changes the optimization, a new library is automatically selected and, if necessary, compiled.

#### 3.3.6.1.1.2. Building an optimized library with a custom build environment

If you are using your own build environment, you only have to check the configuration option **OsSourceOptimization** as described in the previous section to compile the optimized library correctly. You do not need to define any preprocessor macro yourself.

Using a library ID for the library and object files is not mandatory, but if the same name is already used, you must delete the libraries and object files and rebuild the library whenever the configuration changes significantly. The generated header file `Os_libcfg.h` defines several macros, typically called `OS_EXCLUDE_something`, which tell the kernel that it can omit the code that performs the `something`. The macros are described in the following section.

#### 3.3.6.1.1.3. Kernel optimization parameters

The following is a list of all the optimization options recognized by the kernel.

**WARNING**

**Do not define these macros manually**

These optimizations are obtained directly from the OS configuration by the Os generator. The macros are defined automatically to get the most optimizations for your configuration. Do not define these macros manually. Manual definition may result in compile-time, link-time and run-time errors.

`OS_EXCLUDE_CALLINGCONTEXTCHECK`

This macro removes the explicit calling-context check from kernel API functions.

`OS_EXCLUDE_CAT2ISR`
: This macro excludes category 2 interrupt service routines (ISRs) from your module configuration. Some functions related to ISRs can be omitted.

`OS_EXCLUDE_ERRORHANDLING`
: This macro omits the complete error handling code. Error codes are returned to the caller. No application specific hook functions are called.

`OS_EXCLUDE_ERRORHOOK`
: This macro excludes the code that calls the error hook from the OS configuration.

`OS_EXCLUDE_ERRORHOOK_APP`
: This macro omits the code that calls the application's error hook. This means that the processor mode switch is omitted, too.

`OS_EXCLUDE_EVENTS`
: This macro omits all functions related to events, such as `WaitEvent`, `SetEvent`, etc. A few other optimizations in `ActivateTask` are also possible.

`OS_EXCLUDE_EXCEPTIONS`
: This macro omits the standard exception handling. Instead of the standard exception handling, a user-provided function is called.

`OS_EXCLUDE_EXTENDED`
: This macro omits all error checking that takes place in `EXTENDED` status.

`OS_EXCLUDE_EXTRA_CHECK`
: This macro omits all code that is related to extra run-time checks.

`OS_EXCLUDE_HWCOUNTER`
: This macro omits all the functionality for hardware counters: initialization, starting and stopping during alarm processing.

`OS_EXCLUDE_HW_FP`
: This macro omits floating-point context switching. It only has an effect on architectures which offer hardware floating point support.

`OS_EXCLUDE_INTSENABLEDCHECK`
: This macro omits checking whether interrupts are enabled. these checks are required by AUTOSAR for almost all API functions.

`OS_EXCLUDE_KILLABLE_APPEHOOK`
: This macro calls error hooks belonging to applications directly without saving the context. This means that these error hooks run as trusted code and cannot be terminated.

`OS_EXCLUDE_KILLABLE_APPSHOOK`
: This macro calls the start-up and shutdown hooks belonging to applications directly without saving context. This means that these hooks run as trusted code and cannot be terminated.

`OS_EXCLUDE_KILLABLE_ISR`
: This macro calls interrupt service routines (ISRs) directly without saving context. This means that the ISRs run as trusted code and cannot be terminated.

`OS_EXCLUDE_KILLALARM`
: This macro omits the function for terminating an alarm. This means an application with an active alarm cannot be properly terminated.

`OS_EXCLUDE_KILLISR`
: This macro omits the function for terminating an interrupt service routine (ISR). This means an ISR can never be terminated in response to a protection error.

`OS_EXCLUDE_KILLTASK`
: This macro omits the function for terminating a task. This means a task cannot be terminated in response to a protection error.

`OS_EXCLUDE_MULTIPLE_ACTIVATIONS`
: If there are no tasks with multiple activations, this macro omits the code to handle those in `ActivateTask` and `TerminateTask`.

`OS_EXCLUDE_PARAMETERACCESS`
: If the error hooks do not need to determine what API parameters cause the error, the code that passes the parameters through the error handling can be omitted. This affects all API functions.

`OS_EXCLUDE_POSTISRHOOK`
: This macro omits the code that calls the `PostISRHook`.

`OS_EXCLUDE_POSTTASKHOOK`
: This macro omits the code that calls the `PostTaskHook`.

`OS_EXCLUDE_PREISRHOOK`
: This macro omits the code that calls the `PreISRHook`.

`OS_EXCLUDE_PREEMPTION`
: This macro simplifies the possible context switch at the end of the interrupt handler.

`OS_EXCLUDE_PRETASKHOOK`
: This macro omits the code in the error handler that calls the `PreTaskHook`.

`OS_EXCLUDE_PROTECTION`
: This macro omits all code that is related to memory protection.

`OS_EXCLUDE_PROTECTIONHOOK`
: The code in the error handler that calls the `ProtectionHook` is omitted.

`OS_EXCLUDE_RATEMONITORS`
: This code omits all the arrival rate limiting code in `ActivateTask`, `SetEvent`, `WaitEvent` and in the handling of category 2 ISRs.

`OS_EXCLUDE_RESOURCEONISR`
: This macro omits the code that adjusts the interrupt levels in `GetResource` and `ReleaseResource`.

`OS_EXCLUDE_SHUTDOWNHOOK`
: This macro omits the code that calls the shutdown hook.

`OS_EXCLUDE_STACKCHECK`
: This macro omits all software stack checking.

`OS_EXCLUDE_STARTUPHOOK`
: This macro omits the code that calls the `StartupHook` in `StartOS`.

`OS_EXCLUDE_SWCOUNTER`
: This macro omits all code related to software counters (including the `IncrementCounter` API).

`OS_EXCLUDE_TIMINGPROTECTION`
: This macro omits all the code that implements execution-time protection (execution budget, resource and interrupt lock timing).

`OS_EXCLUDE_USERTASKRETURN`
: This macro omits the code that handles a return from a task's main function. This means that if a task fails to call `TerminateTask` and simply returns from its main function, the result is undefined but will most likely result in the task entering an endless loop, which will lock out equal and lower priority tasks.

`OS_EXCLUDE_AGGREGATELIMIT`
: This macro is no longer used. The aggregate execution-time limit was removed in AUTOSAR version 3.0.

#### 3.3.6.1.2. Enabling kernel customizations

Kernel customizations are further options that have been added by EB. These options must be explicitly enabled in the OS configuration and can provide further reductions in size and run-time. However, many of them result in a kernel whose behavior is not strictly AUTOSAR compliant, so these options must be used with care. In particular, extreme caution must be exercised if customized error handling is selected in a system with protection features enabled.

To use the kernel customizations:

- In EB tresos Studio, open your OS configuration in the **Os (OS)** editor.
- Open the **OsOS** tab.
- Enable the `OsAutosarCustomization` container.
- Configure the options inside the `OsAutosarCustomization` container as desired:

| **Parameter** | **Description** |
|---|---|
| `OsErrorHandling` | `AUTOSAR`<br>Select `AUTOSAR` to choose AUTOSAR-compliant error handling.<br>`FULL`<br>Select `FULL` to choose error handling in which errors in APIs that do not return `StatusType` are detected and handled. The `ErrorHook` is called |

| Parameter | Description |
|---|---|
| | and the default error action is performed, which could result in the calling Task, ISR or hook function being terminated. If the action is to return an error code, the API silently fails to do its job.<br><br>`MINIMAL`<br>Select `MINIMAL` to choose error handling in which API functions return an error code if an error is detected. In EB tresos AutoCore OS versions 4.1.-7 and upwards, the `ErrorHook()` is called if configured, and parameter access is also supported. In older EB tresos AutoCore OS versions the error hook is not called, so internal errors are detected but not reported. With `MINIMAL` error handling all errors are reported the same way regardless of type, and the `ProtectionHook` and application-specific `ErrorHooks` are not supported. This option is not suitable for use with scalability classes SC3 and SC4.<br><br>Note that the `MINIMAL` error handling will only be effective if `OsSourceOptimization` is enabled (see also Section 3.3.6.1.1, "Optimizing the library"). |
| `OsStrictServiceProtection` | `TRUE`<br>To set `OsStrictServiceProtection` to `TRUE`, select the check box. This is the default behavior.<br><br>`FALSE`<br>To set `OsStrictServiceProtection` to `FALSE`, clear the check box. If you set `OsStrictServiceProtection` to `FALSE`, the very strict calling checks required by AUTOSAR are disabled. However, the implicit checks that are necessary for correct functioning of the kernel, such as `TerminateTask` being called from a task, are still present, so this does not affect the safety of the kernel.<br><br>In the EB tresos AutoCore OS kernel, many APIs work correctly when they are called from a context that is forbidden by AUTOSAR. The functions `ActivateTask` and `SetEvent` work correctly when they are called from an alarm callback or from the `ErrorHook`. |
| `OsInterruptLockingChecks` | `MINIMAL`<br>Select `MINIMAL` to only check the interrupt lock status when it affects the kernel's operation. The interrupt lock status affects the kernel's operation e.-g. in the functions `GetResource` and `ReleaseResource`.<br><br>`EXTRACHECK`<br>Select `EXTRACHECK` to check the interrupt lock status in all API functions which may cause a task switch. |

<table>
<tr><th>Parameter</th><th>Description</th></tr>
<tr><td></td><td><code>AUTOSAR</code><br>Select <code>AUTOSAR</code> to be fully compliant with the AUTOSAR specification.<br><br>NOTE<br>EB tresos AutoCore OS tasks always start with interrupts enabled<br>In EB tresos AutoCore OS the interrupt lock status is considered to be part of the task's context. This means that each newly activated task starts with an interrupt enabled.</td></tr>
<tr><td><code>OsCallIsr</code></td><td><code>DIRECTLY</code><br>Select <code>DIRECTLY</code> to always run ISRs as supervisor with kernel protection boundaries. If you select <code>DIRECTLY</code> ISRs cannot be terminated: If a protection error occurs in an ISR, the only possible course of action is <code>SHUTDOWN</code>.<br><br><code>VIA_WRAPPER</code><br>Select <code>VIA_WRAPPER</code> to run ISRs inside an OS wrapper. In this case, the ISRs may run in user mode and can be terminated in case of an error.</td></tr>
<tr><td><code>OsCallAppErrorHook</code></td><td><code>DIRECTLY</code><br>Select <code>DIRECTLY</code> to always run application-specific error hooks as supervisor with kernel protection boundaries. If you select <code>DIRECTLY</code>, error hooks cannot be terminated: If a protection error occurs in a hook function, the only possible course of action is <code>SHUTDOWN</code>.<br><br><code>VIA_WRAPPER</code><br>Select <code>VIA_WRAPPER</code> to run error hooks inside an OS wrapper. In this case, the hook functions may run in user mode and can be terminated in case of an error.</td></tr>
<tr><td><code>OsCallAppStartupShutdownHook</code></td><td><code>DIRECTLY</code><br>Select <code>DIRECTLY</code> to always run application-specific start-up and shutdown hooks as supervisor with kernel protection boundaries. If you select <code>DIRECTLY</code>, application-specific start-up and shutdown hooks cannot be terminated: If a protection error occurs in a hook function, the only possible course of action is <code>SHUTDOWN</code>.<br><br><code>VIA_WRAPPER</code><br>Select <code>VIA_WRAPPER</code> to run start-up and shutdown hooks inside an OS wrapper. In this case, the hook functions may run in user mode and can be terminated in case of an error.</td></tr>
<tr><td><code>OsPermitSystemObjects</code></td><td><code>TRUE</code><br>To set <code>OsPermitSystemObjects</code> to <code>TRUE</code>, select the check box. If <code>OsPermitSystemObjects</code> is set to <code>TRUE</code>, and if your system contains OS applications, OS objects are permitted to belong to the system itself and not</td></tr>
</table>

| Parameter | Description |
|---|---|
| | to any particular OS application. Such objects can access other objects without restrictions. This feature is useful for objects such as schedule tables that control the scheduling of all applications in a system.<br>`FALSE`<br>To set `OsPermitSystemObjects` to `FALSE`, clear the check box. In this case, each OS object must belong to an OS application if OS applications are used. |
| `OsUserTaskReturn` | This option determines what happens when a task returns from its main function.<br>`KILL_TASK`<br>Select `KILL_TASK` to end the task after returning from its main function. `KILL_TASK` is AUTOSAR compliant but requires the full error handling and error action to be enabled for correct functionality.<br>`LOOP`<br>Select `LOOP` to make a task that returns from its main function try to terminate. If termination fails, the task tries to shutdown the OS. If shutting down the OS fails, the task enters an endless loop with the effect of locking out all tasks of equal or lower priority.<br>**NOTE** **This option has no effect when return-from-task is caught by a special exception handler**<br>On architectures such as TriCore on which return-from-task is caught using a special exception handler, this option has no effect. |

### 3.3.6.2. Compiling EB tresos AutoCore OS in custom build environments

This section provides instructions as to how EB tresos AutoCore OS can be compiled outside the user build environment provided by EB. You may derive all the information necessary from the makefiles provided by the demo application and the EB tresos Studio plugins it uses.

**WARNING** **Generation of non-executable or non-compilable code**

If you use another build environment than the one delivered with EB tresos AutoCore OS, your EB tresos AutoCore OS version is considered untested. This might lead to non-executable or non-compilable code.

To avoid non-executable or non-compilable code, do not use another build environment than the build environment integrated into EB tresos AutoCore OS.

#### 3.3.6.2.1. Determining the source files and include paths

The list of source files that is necessary to build EB tresos AutoCore OS is located in the OS plugin makefiles; these are files that end with `.mak` that are located in the `$TRESOS_BASE\plugins\Os_TS_T<a>D<b>M4I4R0_<release suffix>\make`[1] directory.

To find out the needed files, do the following:

- Provide an environment similar to the one in the demo application. Set the variables `PLUGINS_BASE`, `PROJECT_ROOT`, `PROJECT_OUTPUT_PATH` and `TOOLCHAIN` according to the makefiles in the demo application.
- Include the files `Os_defs.mak` and `Os_rules.mak` in that order.
- The makefiles define a set of variables which specify the libraries needed to build the OS and their needed source files:

  `LIBRARIES_TO_BUILD`
  : the names of the libraries needed to build the OS

  `<libname>_FILES`
  : for each library in `LIBRARIES_TO_BUILD`, a list of source files to build for that library

  `CC_INCLUDE_PATH`
  : all needed include directories to build C files

  `ASM_INCLUDE_PATH`
  : all needed include directories to build assembler files

- Use the variables set by the makefiles to determine the files to build. For example, you could use a makefile snippet like the following:

```
# list all needed source files in SOURCE_FILES
SOURCE_FILES := $(foreach lib,$(LIBRARIES_TO_BUILD),$($(lib)_FILES))
```

- Use the variables `CC_INLCUDE_PATH` and `ASM_INCLUDE_PATH` to determine all directories containing header files. Add these directories to your include path.

[1] The actual name of your installation directory depends on your OS variant, e.g. the target CPU and the AUTOSAR release. It may look like the following: `$TRESOS_BASE\plugins\Os_TS_T17D1M4I4R0_AS40\make`

> **NOTE**
>
> **Assembler files must be preprocessed**
>
> The assembler files provided by EB tresos AutoCore OS include C preprocessor macros. If your assembler does not include a C preprocessor, feed the assembler files to the C preprocessor before running the assembler.

#### 3.3.6.2.2. Determining the compiler options

##### 3.3.6.2.2.1. Options influencing the build process

The compiler options used to build the module are located in the EB tresos AutoCore OS quality statement. In general, only the set of options described there has been validated to work correctly.

##### 3.3.6.2.2.2. Options for defining preprocessor macros

EB tresos AutoCore OS relies on some configuration-dependent preprocessor definitions during compilation.

To find out the set of preprocessor definitions needed for your configuration, do the following:

- Provide an environment similar to the one in the demo application. Set the variables `PLUGINS_BASE`, `PROJECT_ROOT`, `PROJECT_OUTPUT_PATH` and `TOOLCHAIN` according to the makefiles in the demo application.
- Include the files `Os_defs.mak` and `Os_rules.mak` in that order.
- The makefiles define a set of variables which specify the libraries needed to build the OS and their needed source files:

  `PREPROCESSOR_DEFINES`
  : a list of identifiers to distinguish each define in the makefiles

  `<define>_KEY`
  : for each define in `PREPROCESSOR_DEFINES`, the key to use for the C preprocessor

  `<define>_VALUE`
  : for each define in `PREPROCESSOR_DEFINES`, the value to use for the C preprocessor

- Use the variables set by the makefiles to determine the compiler command line needed to set the corresponding defines. For example, if your compiler uses `-D<key>=<value>` to set the define `<key>` to `<value>`, you could use a makefile snippet like the following:

```
# get compiler command line snippet for preprocessor defines
```

```
PREPROC_OPTS := $(foreach d,$(PREPROCESSOR_DEFINES), -D$($(d)_KEY)=$($(d)_VALUE))
```

# 3.4. Using atomic functions

All atomic functions operate on objects with platform-specific types. The type `os_atomic_t` is used for atomic objects which are accessed by multiple threads concurrently. It is *opaque* and therefore you must access them only with the functions provided. Before you use an atomic object, you must initialize it. To do this, use the function `OS_AtomicInit()` and the macro `OS_ATOMIC_OBJECT_INITIALIZER`. You can use the former at run-time and the latter at program load time. Atomic objects with static storage duration are automatically initialized at program load time with the initial value zero.

The value of an atomic object has the type `os_atomic_value_t`. This type is not opaque and hence you may use it in C language expressions as any other basic numerical type. It has no atomicity and memory ordering guarantees associated with it and is meant to be accessed by only one thread at any point in time. The maximum value that you can store in an object of type `os_atomic_value_t` is given by the macro `OS_ATOMICS_VALUE_MAX`.

Furthermore, all the atomic functions (except `OS_AtomicInit()`) exhibit sequential consistency and preclude certain compiler optimizations which strive for moving read and write operations across them. Hence, one can think of this as an implicit call of `OS_AtomicThreadFence()` at the start and end of every atomic function.

**NOTE**

**Use of atomics functions is supported by the atomics module that is provided by EB tresos AutoCore Generic Base.**
You find further information and instructions for using the atomics module in the EB tresos AutoCore Generic Base documentation.

For a standalone delivery of EB tresos AutoCore OS the underlying atomics functions may be used directly. Refer to EB tresos AutoCore OS architecture notes for known architecture-specific restrictions on the use of atomic functions.

## 3.4.1. Constraints for exclusive areas using `EB_FAST_LOCK`

The following constraints apply when you guard exclusive areas with the type `EB_FAST_LOCK` in your EB tresos AutoCore Generic RTE configuration.

- You shall not call any OS API functions except atomic functions from within the exclusive area.
- The execution time budget monitoring is ineffective. Therefore, you are strongly advised to minimize the time spent inside an exclusive area.

- The time stamps returned by `OS_GetTimeStamp()` might become inaccurate. Therefore, you are strongly advised to minimize the time spent inside an exclusive area. Since you are not allowed to call `OS_GetTimeStamp()` from inside an exclusive area, this impact only becomes evident afterwards.

The following documents help you to evaluate the implications of using `EB_FAST_LOCK` for exclusive areas further when you face safety goals.

- EB tresos AutoCore OS safety application guide for ASIL-B applications
- EB tresos Safety OS user's guide
- EB tresos Safety OS safety manual

# 3.5. Application example

## 3.5.1. Overview

In this section, we give you an example on how to start a new project. The application examples are simple starting points and must not be used as a basis for a real ECU. Projects for real ECUs are much more complex and you need knowledge about several parts of the AUTOSAR standard.

The workflow of the `os_demo` application example is as follows:

1. Importing the application example as a project into EB tresos Studio.
2. Adapting your build environment.
3. Configuring the `Os` module.
4. Building the sample code of the application example.
5. Checking whether the sample code was built correctly.

The following chapter provides you with background information, as well as instructions for setting up and working with the `os_demo` application example:

- Section 3.5.2, “Background information” provides the information about the directory structure, prerequisites, and functional behavior of the application with a flow diagram.
- Section 3.5.3, “Setting up the application example” provides the information about importing the project and adapting the makefiles to build the project.
- Section 3.5.4, “Building the application example” provides the information about building the project which includes generating the code and building.
- Section 3.5.5, “Checking whether your code was built correctly” provides the information about verifying the build.

## 3.5.2. Background information

### 3.5.2.1. Prerequisites for running the os_demo application example

To run the os_demo application example, you need the following prerequisites:

- EB tresos Studio is installed on your PC.
- The EB tresos AutoCore Os module is in the plugin folder of your EB tresos Studio installation.
- The EB tresos AutoCore Make module is in the plugin folder of your EB tresos Studio installation. The Make module implements hardware-independent parts of the EB tresos AutoCore build environment.
- The EB tresos AutoCore Platforms module is in the plugin folder of your EB tresos Studio installation. The Platforms module implements hardware-specific parts of the EB tresos AutoCore build environment.
- A target device to which you may transfer the resulting code of the application example. Ideally, the target should support a debug connection to verify the results. An LED panel on the board is also useful.

For information on installing EB tresos Studio and EB tresos AutoCore modules, see the EB tresos installation guide.

### 3.5.2.2. File and directory structure

In your installed EB tresos AutoCore OS package, you find all application example-dependent files in the directory `$TRESOS_BASE\demos\AutoCore_OS\os_demo_<target>_<version>` During the setup of the application example, this directory will be copied into your workspace directory `$TRESOS_BASE\workspace`.

`$TRESOS_BASE` is the directory into which you have installed EB tresos Studio, e.g. `C:\EB\tresos`.

**NOTE**

The actual name of the directory depends on the target platform and OS version. It may look like the following: `$TRESOS_BASE\demos\AutoCore_OS\os_multicore_demo_TC277_6.0.54`

#### 3.5.2.2.1. Location of the makefiles

The makefiles of the application example os_demo are located in the directory util.

#### 3.5.2.2.2. Location of the configuration file

The configuration file Os.xdm of the application example os_demo is an XDM <file>, which is located in the directory config\Os.xdm.

#### 3.5.2.2.3. Location of the source files

The C source files used for the application example consist of:

- The main common application file `source\demo.c`, which contains the implementation of all tasks and ISR routines.
- A bundle of board-specific files implementing board-specific functions and macros located in the `source\boards` directory.

#### 3.5.2.2.4. Debug files and workspaces

Some EB tresos AutoCore OS plugins are delivered with additional files such as workspaces or projects for the specific toolchain environments or debugger script files. If such files are available for your plugin, they are located in the `source\boards` directory.

### 3.5.2.3. Functional behavior of the application example

The application example consists of:

- the ten tasks:
  - `InitTask (Priority 1)`
  - `Loop (Priority 2)`
  - `Cyclic (Priority 5)`
  - `Task_St1 (Priority 3)`
  - `Task_St2 (Priority 4)`
  - `Bits2Led (Priority 3)`
  - `IncrementBit0 (Priority 4)`
  - `IncrementBit1 (Priority 4)`
  - `IncrementBit2 (Priority 4)`
  - `IncrementBit3 (Priority 4)`
- the three alarms:
  - `AlarmActCyclic`
  - `SysCounterIncrementer`
  - `SysCounterIncrementer_App2`
- the resource `Res_LedsVar`
- two software counters:

- `SysCounter`
- `SysCounter_App2`

- one hardware counter: `HW_COUNTER`
- one schedule table: `St1`
- one Event: `WriteLEDs`
- six applications:
  - `App1`
  - `App2`
  - `App3`
  - `App4`
  - `App5`
  - `App6`

The OS objects mapped to the applications are as mentioned below,

- `OS objects mapped to App1:`
  - Alarm `AlarmActCyclic`
  - Counter `SysCounter`
  - Resource `Res_LedsVar`
  - Task `InitTask`
  - Task `Loop`
  - Task `Bits2Led`
- `OS objects mapped to App2:`
  - Alarm `SysCounterIncrementer`
  - Alarm `SysCounterIncrementer_App2`
  - Counter `HW_COUNTER`
  - Counter `SysCounter_App2`
  - Schedule Table `St1`
  - Task `TaskSt1`
  - Task `TaskSt2`
  - Task `Cyclic`
- `OS objects mapped to App3:`
  - Task `IncrementBit0`

- `OS objects mapped to App4:`
  - Task `IncrementBit1`
- `OS objects mapped to App5:`
  - Task `IncrementBit2`
- `OS objects mapped to App6:`
  - Task `IncrementBit3`

All the tasks except `Bits2Led` are basic tasks. The task `Bits2Led` is an extended task mapped to the event `WriteLEDs`.

The auto-started task `InitTask` activates the cyclic alarm `AlarmActCyclic`, starts schedule table `St1`, activates task `Bits2Led` and switches to `Loop` task. This task performs an endless loop, which continuously takes and releases the resource `Res_LedsVar`.

The auto-started alarm `SysCounterIncrementer` increments the software counter, which is linked with the alarm `AlarmActCyclic`, at each alarm event.

The extended task `Bits2Led` enters into an endless loop, which continuously waits for the event `WriteLEDs` and once the `WriteLEDs` event is set by using `led_counter` variable's value blinks the LED. This is based on the respective bits which are set.

At the appearance of an alarm event by the `AlarmActCyclic`, the task `Cyclic` is activated whose priority is higher than the priority of the task `Loop`. As soon as the task `Cyclic` is activated and the resource is no longer occupied, the task `Loop` will be interrupted and the task `Cyclic` runs.

The task `Cyclic` activates the task `IncrementBit0` periodically.

The task `IncrementBit0` toggles bit0 of the variable `led_counter`. If the bit0 of the variable `led_counter` is set, the task `IncrementBit0` sets the event `WriteLEDs`. Once the task `IncrementBit0` is terminated, the task `Bits2Led` resumes execution as the event `WriteLEDs` is set and blinks the LED accordingly. If the bit0 of the variable `led_counter` is not set, the task `IncrementBit0` activates the task `IncrementBit1`.

The task `IncrementBit1` toggles bit1 of the variable `led_counter`. If the bit1 of the variable `led_counter` is set, the task `IncrementBit1` sets the event `WriteLEDs`. Once the task `IncrementBit1` is terminated, the task `Bits2Led` resumes execution as the event `WriteLEDs` is set and blinks the LED accordingly. If the bit1 of the variable `led_counter` is not set, the task `IncrementBit1` activates the task `IncrementBit2`.

The task `IncrementBit2` toggles bit2 of the variable `led_counter`. If the bit1 of the variable `led_counter` is set, the task `IncrementBit2` sets the event `WriteLEDs`. Once the task `IncrementBit1` is terminated, the task `Bits2Led` resumes execution as the event `WriteLEDs` is set and blinks the LED accordingly. If the bit1 of the variable `led_counter` is not set, the task `IncrementBit2` activates the task `IncrementBit3`.

The task `IncrementBit3` toggles bit3 of the variable `led_counter` and sets the event `WriteLEDs`. Once the task `IncrementBit1` is terminated, the task `Bits2Led` resumes execution as the event `WriteLEDs` is set and blinks the LED accordingly.

The task `Cyclic` activating the task `IncrementBit0`, the task `IncrementBit0` activating the task `IncrementBit1`, the task `IncrementBit1` activating the task `IncrementBit2` and the task `IncrementBit2` activating the task `IncrementBit3` is a cycle. This cycle mentioned above toggles the bits starting from bit0 of the variable `led_counter` (using which the LEDs are blinked) to bit3 repeatedly. i.e., from 0000 to 1111 and wraps around to 0000 from 1111.

Time units of the counter are typically dimensioned so that the task `Cyclic` is activated once per second.

Parallel to the above described behavior, a schedule table is started. This schedule table has two expiry points, each with one task activation for the task `Task_St1` and for the task `Task_St2`. Both tasks are configured with a higher priority than the `Loop` task, thus this task is interrupted over and over.

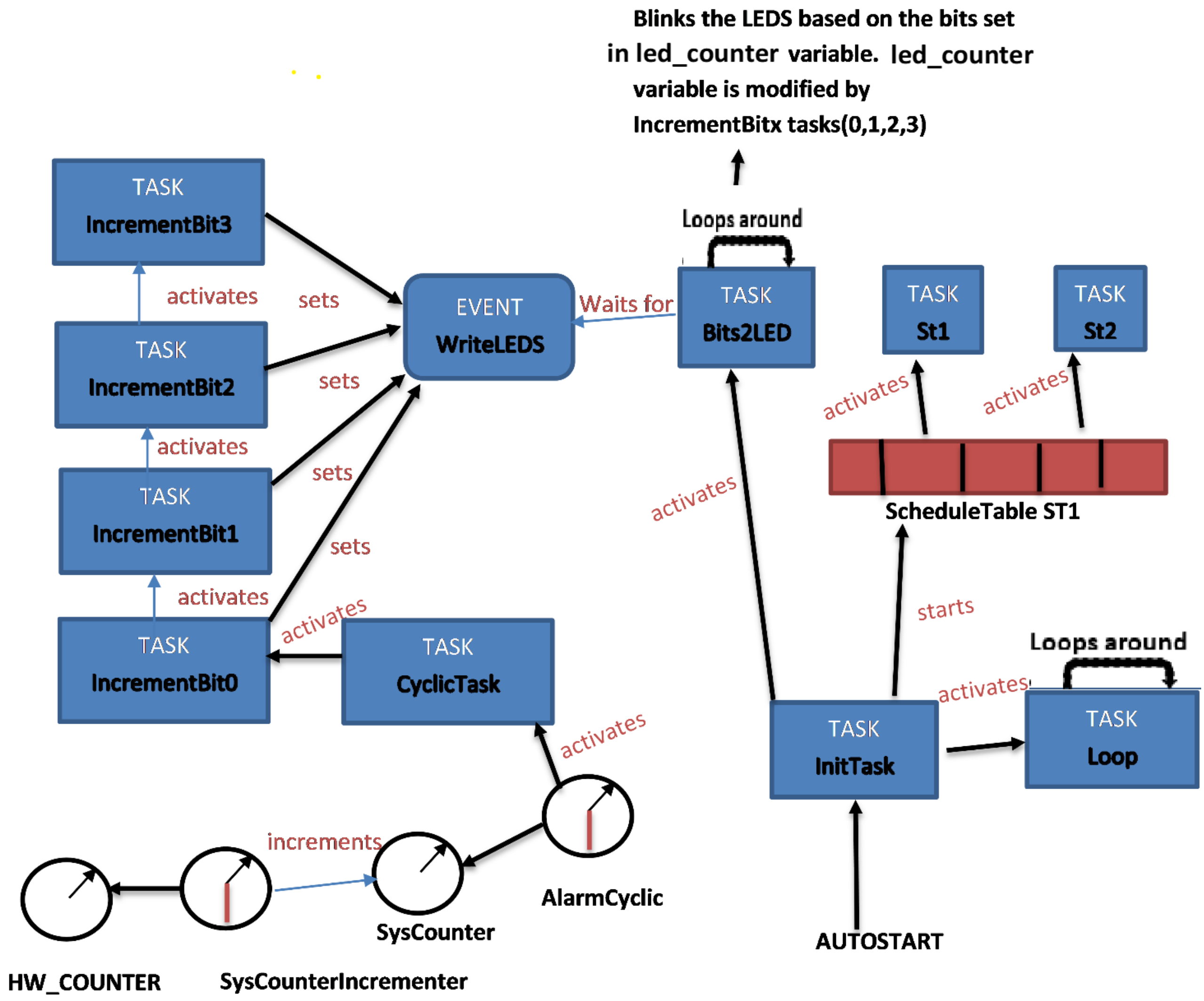


Figure 3.1. Interaction of OS objects

## 3.5.3. Setting up the application example

### 3.5.3.1. Importing the application example

The application examples are delivered as an EB tresos Studio project. You need to import the project into your EB tresos Studio workspace, e.g. at `$TRESOS_BASE\workspace`. `$TRESOS_BASE` is the directory into which you have installed EB tresos Studio, e.g. `C:\EB\tresos`.

Importing the application example into your workspace

Step 1
Locate `$TRESOS_BASE\bin\tresos_gui.exe`.

Step 2
Open `tresos_gui.exe`.

Step 3
In the **File** menu, select **Import**.

Step 4
Select **Existing Projects into Workspace**.

Step 5
Select **Next**.

Step 6
Select **Select root directory** and choose the folder `$TRESOS_BASE\demos\AutoCore_OS`.

Step 7

Step 7.1
Select **OK**.

Step 7.2
The project appears in the **Projects** window of the **Import** dialog.

Step 7.3
The actual name of the project depends on the target platform and the `OS` version. It may look like the following: `os_multicore_demo_TC277_6.0.54`.

Step 8

Step 8.1
Select **Copy projects into workspace**.

Step 8.2
The project is now copied to the default workspace at `$TRESOS_BASE\workspace`.

Step 9
Select **Finish**.

You are done.

In the **Project Explorer** view you now see the project:

- Open the recently created project.
- Open the configuration by selecting the gray chip symbol .

### 3.5.3.2. Adapting the build environment

**NOTE**

**For EB tresos WinCore, no compiler configuration is needed**

EB tresos WinCore is delivered with the MinGW development environment, which is installed automatically when you build the application example. The MinGW development environment also contains a compiler. Therefore, you only need to follow these instructions when you want to configure a different compiler.

Changing the settings or the path of the compiler

Step 1
Locate the project folder in the workspace directory, e.g. `$TRESOS_BASE\workspace\os_demo_$DEMO`. Place-holder `$DEMO` contains the version of your release and the name of the target hardware of this demo.

Step 2
Open the file `util\launch_cfg.bat` in a text editor.

Step 3
Set the environment variable `TRESOS_BASE` to the directory, into which you have installed EB tresos Studio. For example add "`SET TRESOS_BASE=C:\EB\tresos`".

Step 4
Use variable `PLUGINS_BASE` to specify the `plugins` folder you want to use. For example add "`SET PLUGINS_BASE=%TRESOS_BASE%\plugins`".

Step 5
Save file `util\launch_cfg.bat`.

Step 6
Open the file `util\<target>_<derivative>_<compiler>_cfg.mak` in a text editor.

Step 7
Change the variable `TOOLPATH_COMPILER` to the actual compiler installation path. For example: `TOOLPATH_COMPILER ?= C:\WindRiver\diab\5.5.1.0`.

Step 8
The variable `CC_OPT` in the same file contains the compiler options. If you need any specific settings, adjust `CC_OPT`.

### 3.5.3.3. Changing the compiler

If your architecture supports multiple compilers, you can change the compiler.

Changing the compiler

Step 1
Locate the project folder in the workspace directory, e.g. $TRESOS_BASE\workspace\os_demo.

Step 2
Open the file util\<target>_<derivative>_Makefile.mak in a text editor.

Step 3
Set the variable TOOLCHAIN to the name of the toolchain, e.g. TOOLCHAIN = dcc.

Step 4
Set the variable COMPILER to the compiler, e.g. COMPILER = PA_XPC5777M_dcc.

### 3.5.3.4. Changing the board settings

If you want to use a different board, you need to change the board settings.

The directory source\boards\<board name> in your project folder contains board-specific makefiles and source files.

Using a different board

Step 1
Locate the project folder in your workspace directory, e.g. $TRESOS_BASE\workspace\os_demo.

Step 2
Open the file util\<target>_<derivative>_Makefile.mak in a text editor.

Step 3
Change the variable BOARD.

The value of the variable has to be the same as the name of the board directory, e.g. BOARD = EvaXPC5777M.

### 3.5.3.5. Configuring the Os module

For possible adaptations of the Os module, see Section 3.3.2, “Configuring and using the OS objects”. For information on target-dependent adaptations of makefiles, refer to the EB tresos AutoCore documentation and to the EB tresos AutoCore OS architecture notes.

### 3.5.4. Building the application example

In order to build an EB tresos AutoCore project, you need to perform the following two steps in the order they are described:

1. Generate the project.
2. Build the project.

To build the application example, make sure that EB tresos Studio is not running anymore.

Building the application example

Step 1
In the `$TRESOS_BASE\workspace\os_demo_$DEMO\util` directory, double-click the file `launch.bat`. The first start of `launch.bat` takes some time. Place-holder `$DEMO` contains the version of your release and the name of the target hardware of this demo.

Step 2
Type **make generate** and hit the **Enter** key.

The project is being generated.

Step 3
Type **make** and hit the **Enter** key.

Your application example is being built.

If you work with EB tresos WinCore, the MinGW development environment is being installed with the build of the application example.

You find the resulting binary file in the `$TRESOS_BASE\workspace\os_demo_$DEMO\output\bin` directory.

### 3.5.5. Checking whether your code was built correctly

After the code of the application example was built, transfer it to your target and check whether the code was built correctly. There are two ways to check your code:

- Running the code on your target board.
- Using a debugger.

#### 3.5.5.1. Checking the sample code on the hardware board

For many targets the application example is customized in a way that it controls an LED panel on the board. With this LED panel you may control whether the tasks of the sample `OS` are activated correctly and how the resource is used:

- Four LEDs indicate the value of a count variable that is incremented in the `Cyclic` task.
- A fifth LED shows whether the resource `Res_CounterVar` is taken or not.

If the program is running on your target, counter-LEDs are incrementing each second and the resource LED is flashing.

---

**NOTE**

**The flashing rate depends on the CPU frequency**

The flashing rate of the LEDs inform you about the CPU frequency: The resource is taken and released using a delay loop. Thus, the faster the lights blink, the faster this delay loop is running.

---

If you use a target board the `Os` module does not support directly, you may have to adapt the board macros `LEDS_INIT` and `LEDS_SET` in the file `board.h` residing in the `source\boards` directory for your board.

For information on whether or not the `Os` module supports your board, see the respective architecture notes.

#### 3.5.5.2. Checking the sample code with a debugger

Load your sample code into the debugger to check its correctness.

---

**TIP**

**Use the make debug command to start the debugger**

Some target implementations support the **make debug** command that is useful for starting the debugger, setting up the debug environment, etc. To set up your debugger with this command, open the file `launch.bat` in the `util` directory, type **make debug**, and press **Enter**.

For information on whether your target supports this command, see the respective architecture notes.

---

To check if your code runs correctly:

1. Check if the `Cyclic` task runs by using a breakpoint. If yes, then check if the last 4 bits of the `counter` variable of this task are incremented each second. If they are, your code was built correctly.
2. If you use the variables `task_St1_counter` and `task_St2_counter`, you may check if the schedule table runs correctly: If the variables `task_St1_counter` and `task_St2_counter` are continuously incremented, and the value of `task_St1_counter` is always higher than the value of `task_St2_counter`, the schedule table works correctly.

Note that if the value of `task_St1_counter` is higher than `task_St2_counter` is calculated independent from the data range of the counter variables. In case of an overflow, there might be situations in which the value of `task_St1_counter` is smaller (i.e. zero or negative).

When the demo is running correctly, you are ready to begin adapting it to your needs.

# 4. References

## 4.1. EB tresos AutoCore OS configuration language

The EB tresos AutoCore OS generator supports the XML Data Model (XDM).

This chapter describes the configuration of standard objects, attributes and the architecture-independent extensions implemented by EB tresos AutoCore OS.

### 4.1.1. Configuration parameters

| **Containers included** | | |
|---|---|---|
| **Container name** | **Multiplicity** | **Description** |
| OsAlarm | 0..n | An **OsAlarm** may be used to asynchronously inform or activate a specific task. It is possible to start alarms automatically at system start-up depending on the application mode. |
| OsAppMode | 1..255 | **OsAppMode** objects are used to define which tasks, alarms, etc. will be started automatically when the kernel is first started. In a valid OS configuration the **CPU** must contain at least one **OsAppMode** object. An OsAppMode called OSDEFAULTAPPMODE must always be present for OSEK compatibility. |
| OsApplication | 0..n | An **OS** must be capable of supporting a collection of Os objects (tasks, interrupts, alarms, hooks for instance) that form a cohesive functional unit. This collection of objects is termed an OsApplication. All objects which belong to the same OS application have access to each other. Access means to allow to use these objects within API services. Access by other applications can be granted separately. |
| OsCounter | 0..n | A counter is the mechanism by which alarms are triggered. |
| OsEvent | 0..n | **OsEvent** objects are used to provide inter-task coordination. Events are represented by their event masks. |
| OsSpinlock | 0..n | An OsSpinlock object is used to co-ordinate concurrent access by TASKs/ISR2s on different cores to a shared resource. |

| Containers included | | |
|---|---|---|
| OsIsr | 0..n | **OsIsr** objects are used to represent interrupt service routines. All ISRs should be declared in the application code using the *ISR()* API. The attributes of the ISR object are defined in the following section |
| OsOS | 1..1 | The Os object defines the existence of, and configuration, for the OS kernel. In a valid OS configuration the **CPU** must contain exactly one Os object. |
| OsPeripheralArea | 0..65534 | Container to structure the configuration parameters of one peripheral area. This configuration parameter is not supported by AutoCore OS. |
| OsResource | 0..n | An **OsResource** object is used to co-ordinate the concurrent access by tasks and ISRs to a shared resource, e.g. the scheduler, any program sequence, memory or any hardware area. |
| OsScheduleTable | 0..n | An OsScheduleTable addresses the synchronization issue by providing an encapsulation of a statically defined set of alarms that cannot be modified at run-time. |
| OsTask | 0..n | **OsTask** objects are used to define which tasks are present in the system. The attributes of the **OsTask** object are defined in the following sections. |

| Parameters included | |
|---|---|
| **Parameter name** | **Multiplicity** |
| POST_BUILD_VARIANT_USED | 1..1 |
| IMPLEMENTATION_CONFIG_VARIANT | 1..1 |

| Parameter Name | POST_BUILD_VARIANT_USED |
|---|---|
| **Label** | Post Build Variant Used |
| **Description** | Indicates whether a module implementation has or plans to have (i.e., introduced at link or post-build time) new post-build variation points. |
| **Multiplicity** | 1..1 |
| **Type** | BOOLEAN |
| **Default value** | false |
| **Origin** | AUTOSAR_ECUC |

| Parameter Name | IMPLEMENTATION_CONFIG_VARIANT |
| --- | --- |
| Label | Config Variant |
| Multiplicity | 1..1 |
| Type | ENUMERATION |
| Default value | VariantPreCompile |
| Range | VariantPreCompile |

#### 4.1.1.1. OsAlarm

| Containers included | | |
| --- | --- | --- |
| **Container name** | **Multiplicity** | **Description** |
| OsAlarmAction | 1..1 | The **OsAlarmAction** attribute is a parametrized enum value specifying what shall happen when the alarm expires. The values are:<br>▶ ACTIVATETASK<br>▶ SETEVENT<br>▶ ALARMCALLBACK<br>▶ INCREMENTCOUNTER<br>The parameters are:<br>▶ TASK: The task that shall be activated or have an event set<br>▶ EVENT: The event that shall be set for the task<br>▶ ALARMCALLBACKNAME:the name of the alarm callback function to be called. The function should be declared using the *ALARMCALLBACK(name)* API. |
| OsAlarmAutostart | 0..1 | **OsAlarmAutostart** is a boolean attribute whose value specifies whether the alarm shall be started automatically when the kernel starts. If the value is **TRUE**, the **OsAlarmAppModeRef** sub-attribute specifies in which application modes the task shall be automatically started, and the sub-attributes **OsAlarmAlarmTime** and **OsAlarmCycleTime** specify the first and subsequent relative values of the counter at which the alarm shall expire. |

| Parameters included | |
| --- | --- |
| **Parameter name** | **Multiplicity** |

| Parameters included | |
|---|---|
| OsAlarmAccessingApplication | 0..n |
| OsAlarmCounterRef | 1..1 |

| Parameter Name | OsAlarmAccessingApplication |
|---|---|
| Description | Reference to applications which have an access to this object. The objects of referenced OsAplication can access and change the state of current OsAlarm by calling the system service APIs. For example, the current alarm can be started, stopped or inquired about its state by the objects of referenced OsApplication. |
| Multiplicity | 0..n |
| Type | REFERENCE |
| Origin | AUTOSAR_ECUC |

| Parameter Name | OsAlarmCounterRef |
|---|---|
| Description | The **OsAlarmCounterRef** attribute specifies the Counter with which the alarm is associated. Each alarm must be associated with exactly one Counter. |
| Multiplicity | 1..1 |
| Type | REFERENCE |
| Origin | AUTOSAR_ECUC |

### 4.1.1.2. OsAlarmAction

| Containers included | | |
|---|---|---|
| **Container name** | **Multiplicity** | **Description** |
| OsAlarmActivateTask | 1..1 | This container specifies the parameters to activate a task. |
| OsAlarmCallback | 1..1 | This container specifies the parameters to call a callback for alarm. |
| OsAlarmIncrementCounter | 1..1 | This container specifies the parameters to increment a counter. |
| OsAlarmSetEvent | 1..1 | This container specifies the parameters to set an event |

### 4.1.1.3. OsAlarmActivateTask

| Parameters included | |
|---|---|
| **Parameter name** | **Multiplicity** |
| OsAlarmActivateTaskRef | 1..1 |

| Parameter Name | OsAlarmActivateTaskRef |
|---|---|
| **Description** | Reference to the task that will be activated by that alarm. |
| **Multiplicity** | 1..1 |
| **Type** | REFERENCE |
| **Origin** | AUTOSAR_ECUC |

### 4.1.1.4. OsAlarmCallback

| Parameters included | |
|---|---|
| **Parameter name** | **Multiplicity** |
| OsAlarmCallbackName | 1..1 |

| Parameter Name | OsAlarmCallbackName |
|---|---|
| **Description** | Name of the function that is called when this alarm callback is triggered. |
| **Multiplicity** | 1..1 |
| **Type** | FUNCTION-NAME |
| **Origin** | AUTOSAR_ECUC |

### 4.1.1.5. OsAlarmIncrementCounter

| Parameters included | |
|---|---|
| **Parameter name** | **Multiplicity** |
| OsAlarmIncrementCounterRef | 1..1 |

| Parameter Name | OsAlarmIncrementCounterRef |
|---|---|
| **Description** | Reference to the counter that will be incremented by that alarm. |
| **Multiplicity** | 1..1 |

| | |
|---|---|
| **Type** | REFERENCE |
| **Origin** | AUTOSAR_ECUC |

### 4.1.1.6. OsAlarmSetEvent

| **Parameters included** | |
|---|---|
| **Parameter name** | **Multiplicity** |
| OsAlarmSetEventRef | 1..1 |
| OsAlarmSetEventTaskRef | 1..1 |

| **Parameter Name** | **OsAlarmSetEventRef** |
|---|---|
| **Description** | Reference to the event that will be set by that alarm. |
| **Multiplicity** | 1..1 |
| **Type** | REFERENCE |
| **Origin** | AUTOSAR_ECUC |

| **Parameter Name** | **OsAlarmSetEventTaskRef** |
|---|---|
| **Description** | Reference to the task that will be activated by that event. |
| **Multiplicity** | 1..1 |
| **Type** | REFERENCE |
| **Origin** | AUTOSAR_ECUC |

### 4.1.1.7. OsAlarmAutostart

| **Parameters included** | |
|---|---|
| **Parameter name** | **Multiplicity** |
| OsAlarmAlarmTime | 1..1 |
| OsAlarmAutostartType | 1..1 |
| OsAlarmCycleTime | 1..1 |
| OsAlarmAppModeRef | 1..n |
| OsTimeUnit | 0..1 |

| **Parameter Name** | **OsAlarmAlarmTime** |
|---|---|

| | |
|---|---|
| **Description** | The relative or absolute tick value when the alarm expires for the first time. Note that for an alarm which is RELATIVE the value must be bigger than 0. |
| **Multiplicity** | 1..1 |
| **Type** | INTEGER |
| **Default value** | 0 |
| **Range** | >=1<br><=4294967295 |
| **Origin** | AUTOSAR_ECUC |

| **Parameter Name** | **OsAlarmAutostartType** |
|---|---|
| **Description** | This specifies the type of autostart for the alarm. |
| **Multiplicity** | 1..1 |
| **Type** | ENUMERATION |
| **Default value** | RELATIVE |
| **Range** | ABSOLUTE<br>RELATIVE |
| **Origin** | AUTOSAR_ECUC |

| **Parameter Name** | **OsAlarmCycleTime** |
|---|---|
| **Description** | Cycle time of a cyclic alarm in ticks. If the value is 0 than the alarm is not cyclic. |
| **Multiplicity** | 1..1 |
| **Type** | INTEGER |
| **Default value** | 0 |
| **Range** | >=0<br><=4294967295 |
| **Origin** | AUTOSAR_ECUC |

| **Parameter Name** | **OsAlarmAppModeRef** |
|---|---|
| **Description** | Reference to the application modes for which the AUTOSTART shall be performed. |
| **Multiplicity** | 1..n |
| **Type** | REFERENCE |
| **Origin** | AUTOSAR_ECUC |

| **Parameter Name** | **OsTimeUnit** |
|---|---|

| | |
|---|---|
| **Description** | **OsTimeUnit** contains the time unit type used for this alarm. |
| **Multiplicity** | 0..1 |
| **Type** | ENUMERATION |
| **Default value** | TICKS |
| **Range** | NANOSECONDS<br>TICKS |
| **Origin** | Elektrobit Automotive GmbH |

#### 4.1.1.8. OsAppMode

#### 4.1.1.9. OsApplication

| **Containers included** | | |
|---|---|---|
| **Container name** | **Multiplicity** | **Description** |
| OsApplicationHooks | 1..1 | This container structures the OS-Application-specific hooks. |
| OsApplicationTrustedFunction | 0..n | The **OsApplicationTrustedFunction** attribute is a list of **BOOLEAN** attributes specifying trusted functions belonging to this application. If the value is **TRUE**, further sub-attributes specify the **NAME** of the trusted function. There are further implementation-specific sub-attributes. Trusted functions can be called by other applications using the *CallTrustedFunction* API. |

| **Parameters included** | |
|---|---|
| **Parameter name** | **Multiplicity** |
| OsTrusted | 1..1 |
| OsApplicationCoreRef | 1..1 |
| OsAppAlarmRef | 0..n |
| OsAppCounterRef | 0..n |
| OsAppIsrRef | 0..n |
| OsAppScheduleTableRef | 0..n |
| OsAppTaskRef | 0..n |
| OsRestartTask | 0..1 |

| Parameters included | |
|---|---|
| OsAppEcucPartitionRef | 0..1 |
| OsAppResourceRef | 0..n |
| OsApplicationCoreAssignment | 0..1 |

| Parameter Name | OsTrusted |
|---|---|
| Description | **OsTrusted** is a boolean attribute that specifies whether Tasks, ISRs etc. associated with the application are to run with the kernel's Privileged or Non-Privileged protection parameters. Privileged applications have access to more of the CPU's resources than non-privileged applications. When **OsTrusted** is **TRUE**, the **TRUSTED_FUNCTION** sub-attributes are available. |
| Multiplicity | 1..1 |
| Type | BOOLEAN |
| Default value | false |
| Origin | AUTOSAR_ECUC |

| Parameter Name | OsApplicationCoreRef |
|---|---|
| Description | Reference to the Core Definition in the Ecuc Module where the **CoreId** is defined. This reference is used to describe to which Core the OsApplication is bound. This configuration parameter is not supported by EB tresos AutoCore OS. Instead use **OsApplicationCoreAssignment** in OsApplication. |
| Multiplicity | 1..1 |
| Type | REFERENCE |
| Origin | AUTOSAR_ECUC |

| Parameter Name | OsAppAlarmRef |
|---|---|
| Description | Specifies the OsAlarms that belong to the OsApplication. |
| Multiplicity | 0..n |
| Type | REFERENCE |
| Origin | AUTOSAR_ECUC |

| Parameter Name | OsAppCounterRef |
|---|---|
| Description | References the OsCounters that belong to the OsApplication. |
| Multiplicity | 0..n |
| Type | REFERENCE |

| | |
|---|---|
| **Origin** | AUTOSAR_ECUC |

| **Parameter Name** | **OsAppIsrRef** |
|---|---|
| **Description** | References which OsIsrs belong to the OsApplication. |
| **Multiplicity** | 0..n |
| **Type** | REFERENCE |
| **Origin** | AUTOSAR_ECUC |

| **Parameter Name** | **OsAppScheduleTableRef** |
|---|---|
| **Description** | References the OsScheduleTables that belong to the OsApplication. |
| **Multiplicity** | 0..n |
| **Type** | REFERENCE |
| **Origin** | AUTOSAR_ECUC |

| **Parameter Name** | **OsAppTaskRef** |
|---|---|
| **Description** | References which OsTasks belong to the OsApplication. |
| **Multiplicity** | 0..n |
| **Type** | REFERENCE |
| **Origin** | AUTOSAR_ECUC |

| **Parameter Name** | **OsRestartTask** |
|---|---|
| **Description** | If **OsRestartTask** parameter is enabled, the value of **OsRestartTask** specifies which task shall be automatically activated when the application is terminated and restarted after a serious error. |
| **Multiplicity** | 0..1 |
| **Type** | REFERENCE |
| **Origin** | AUTOSAR_ECUC |

| **Parameter Name** | **OsAppEcucPartitionRef** |
|---|---|
| **Description** | Denotes which **EcucPartition** is implemented by this **OS application**. **This reference is not used by the Os generator.** |
| **Multiplicity** | 0..1 |
| **Type** | REFERENCE |
| **Origin** | AUTOSAR_ECUC |

| **Parameter Name** | **OsAppResourceRef** |
|---|---|

| | |
|---|---|
| **Description** | References the OsResources that belong to the OsApplication. |
| **Multiplicity** | 0..n |
| **Type** | REFERENCE |
| **Origin** | Elektrobit Automotive GmbH |

| **Parameter Name** | **OsApplicationCoreAssignment** |
|---|---|
| **Description** | ID of the core onto which the OsApplication is bound. |
| **Multiplicity** | 0..1 |
| **Type** | INTEGER |
| **Origin** | Elektrobit Automotive GmbH |

### 4.1.1.10. OsApplicationHooks

| **Parameters included** | |
|---|---|
| **Parameter name** | **Multiplicity** |
| OsAppErrorHook | 1..1 |
| OsAppShutdownHook | 1..1 |
| OsAppStartupHook | 1..1 |
| OsAppErrorHookStack | 0..1 |
| OsAppShutdownHookStack | 0..1 |
| OsAppStartupHookStack | 0..1 |

| **Parameter Name** | **OsAppErrorHook** |
|---|---|
| **Description** | **OsAppErrorHook** is a boolean attribute that specifies whether this application has a private error-hook function. If the value is **TRUE**, the kernel calls the user-supplied *void ErrorHook_<application-name>(StatusType errorcode)* instead of the global error hook whenever an error is detected in the application, unless the error was caused within an error hook. |
| **Multiplicity** | 1..1 |
| **Type** | BOOLEAN |
| **Default value** | false |
| **Origin** | AUTOSAR_ECUC |

| **Parameter Name** | **OsAppShutdownHook** |
|---|---|

| | |
|---|---|
| **Description** | **OsAppShutdownHook** is a boolean attribute that specifies whether this application has a private shutdown-hook function. If the value is **TRUE**, the kernel calls the user-supplied *void ShutdownHook_<applicationname> (StatusType errorcode)* when the system has been shutdown, before calling the global shutdown hook. The parameter is the value of the error code passed to *ShutdownOS()* Application-specific start-up hooks must always return because the order of calling is not defined. Any final action such as restarting the system should take place in the global shutdown hook. |
| **Multiplicity** | 1..1 |
| **Type** | BOOLEAN |
| **Default value** | false |
| **Origin** | AUTOSAR_ECUC |

| **Parameter Name** | **OsAppStartupHook** |
|---|---|
| **Description** | The **OsAppStartupHook** attribute specifies whether the application has a private startup-hook function. If the value is **TRUE**, the kernel calls the user-supplied *void StartupHook_<application-name>(void)* immediately before starting the scheduler, after calling the global start-up hook. |
| **Multiplicity** | 1..1 |
| **Type** | BOOLEAN |
| **Default value** | false |
| **Origin** | AUTOSAR_ECUC |

| **Parameter Name** | **OsAppErrorHookStack** |
|---|---|
| **Description** | **OsAppErrorHookStack** defines the stack size of the error hook in bytes. |
| **Multiplicity** | 0..1 |
| **Type** | INTEGER |
| **Range** | >=1<br><=2000000000 |
| **Origin** | Elektrobit Automotive GmbH |

| **Parameter Name** | **OsAppShutdownHookStack** |
|---|---|
| **Description** | **OsAppShutdownHookStack** defines the stack size of the shutdown hook in bytes. |
| **Multiplicity** | 0..1 |
| **Type** | INTEGER |

| | |
|---|---|
| **Range** | >=1<br><=2000000000 |
| **Origin** | Elektrobit Automotive GmbH |

| **Parameter Name** | **OsAppStartupHookStack** |
|---|---|
| **Description** | **OsAppStartupHookStack** defines the stack size of the start-up hook in bytes. |
| **Multiplicity** | 0..1 |
| **Type** | INTEGER |
| **Range** | >=1<br><=2000000000 |
| **Origin** | Elektrobit Automotive GmbH |

### 4.1.1.11. OsApplicationTrustedFunction

| **Parameters included** | |
|---|---|
| **Parameter name** | **Multiplicity** |
| OsTrustedFunctionName | 1..1 |
| OsTrustedFunctionStacksize | 0..1 |

| **Parameter Name** | **OsTrustedFunctionName** |
|---|---|
| **Description** | This is an identifier for a trusted function available to Os-Applications. It is generated as TRUSTED_Name. |
| **Multiplicity** | 1..1 |
| **Type** | FUNCTION-NAME |
| **Origin** | AUTOSAR_ECUC |

| **Parameter Name** | **OsTrustedFunctionStacksize** |
|---|---|
| **Description** | This attribute specifies the amount of stack required by the trusted function in bytes. **EB tresos AutoCore OS:** The kernel checks that the calling task or ISR has sufficient stack remaining before calling the trusted function. **It is vitally important that the stack size for trusted functions is set correctly. Too small a value means that the trusted function could overflow the task or the stack region of the ISR, and since it is trusted the overflow will not be caught by the memory protection mechanisms.** |

| | |
|---|---|
| **Multiplicity** | 0..1 |
| **Type** | INTEGER |
| **Range** | >=1<br><=2000000000 |
| **Origin** | Elektrobit Automotive GmbH |

### 4.1.1.12. OsCounter

| **Containers included** | | |
|---|---|---|
| **Container name** | **Multiplicity** | **Description** |
| OsDriver | 0..1 | This container contains the information how a software counter can be incremented automatically without specifying an alarm. This configuration is only valid if the parameter OsCounterType is set to **Software**. If the container is disabled, the OS manages the counter and increments it, if configured by the user, with an alarm. If the container is enabled the OS can use a hardware module to automatically increment the counter. For this, a hardware module has to be specified. |
| OsTimeConstant | 0..n | Allows the user to define constants which can be e.g. used to compare time values with timer tick values. A time value will be converted to a timer tick value during generation and can be accessed later on via its **OsConstName**. The conversion is done by rounding time values to the nearest fitting tick value. |

| **Parameters included** | |
|---|---|
| **Parameter name** | **Multiplicity** |
| OsCounterMaxAllowedValue | 1..1 |
| OsCounterMinCycle | 1..1 |
| OsCounterTicksPerBase | 1..1 |
| OsCounterType | 1..1 |
| OsSecondsPerTick | 0..1 |
| OsCounterAccessingApplication | 0..n |

| **Parameter Name** | **OsCounterMaxAllowedValue** |
|---|---|

| | |
|---|---|
| **Description** | Maximum possible allowed value of the system counter in ticks. When the counter reaches this value, the next advancement will cause it to restart from zero. |
| **Multiplicity** | 1..1 |
| **Type** | INTEGER |
| **Origin** | AUTOSAR_ECUC |

| **Parameter Name** | **OsCounterMinCycle** |
|---|---|
| **Description** | The MINCYCLE attribute specifies the minimum allowed number of counter ticks for a cyclic alarm linked to the counter. |
| **Multiplicity** | 1..1 |
| **Type** | INTEGER |
| **Origin** | AUTOSAR_ECUC |

| **Parameter Name** | **OsCounterTicksPerBase** |
|---|---|
| **Description** | **OsCounterTicksPerBase** is a UINT32 value that specifies how many ticks of the counter represent a known unit of counting. The value of this attribute is not used by the kernel, but is available for application purposes. |
| **Multiplicity** | 1..1 |
| **Type** | INTEGER |
| **Range** | >=1<br><=4294967295 |
| **Origin** | AUTOSAR_ECUC |

| **Parameter Name** | **OsCounterType** |
|---|---|
| **Description** | This parameter contains the natural type or unit of the counter. |
| **Multiplicity** | 1..1 |
| **Type** | ENUMERATION |
| **Range** | HARDWARE<br>SOFTWARE |
| **Origin** | AUTOSAR_ECUC |

| **Parameter Name** | **OsSecondsPerTick** |
|---|---|
| **Description** | Time of one hardware tick in seconds. |
| **Multiplicity** | 0..1 |

| | |
|---|---|
| **Type** | FLOAT |
| **Default value** | 0.1 |
| **Range** | >=0.0<br><=86400.0 |
| **Origin** | AUTOSAR_ECUC |

| **Parameter Name** | **OsCounterAccessingApplication** |
|---|---|
| **Description** | Reference to applications which have an access to this object. The objects of referenced OsAplication can access and change the state of current OsCounter by calling the system service APIs. For example the value of OsCounter can be read or incremented by the objects of referenced OsApplication. |
| **Multiplicity** | 0..n |
| **Type** | REFERENCE |
| **Origin** | AUTOSAR_ECUC |

### 4.1.1.13. OsDriver

| **Containers included** | | |
|---|---|---|
| **Container name** | **Multiplicity** | **Description** |
| OsHwIncrementer | 0..1 | **OsHwIncrementer** specifies a hardware module that automatically increments the software counter. Specify the period of the incrementer module in the **OsSecondsPerTick** parameter. |

### 4.1.1.14. OsHwIncrementer

| **Parameters included** | |
|---|---|
| **Parameter name** | **Multiplicity** |
| OsHwModule | 1..1 |
| OsIncrementerIrqLevel | 1..1 |

| **Parameter Name** | **OsHwModule** |
|---|---|
| **Description** | **OsHwModule** provides a list of supported hardware modules that can be used to increment a software counter. |
| **Multiplicity** | 1..1 |

| | |
|---|---|
| **Type** | ENUMERATION |
| **Origin** | Elektrobit Automotive GmbH |

| **Parameter Name** | **OsIncrementerIrqLevel** |
|---|---|
| **Multiplicity** | 1..1 |
| **Type** | INTEGER |
| **Origin** | Elektrobit Automotive GmbH |

### 4.1.1.15. OsTimeConstant

| **Parameters included** | |
|---|---|
| **Parameter name** | **Multiplicity** |
| OsTimeValue | 1..1 |
| OsConstName | 1..1 |

| **Parameter Name** | **OsTimeValue** |
|---|---|
| **Description** | This parameter contains the value of the constant in seconds. |
| **Multiplicity** | 1..1 |
| **Type** | FLOAT |
| **Range** | >=0.0<br><=86400.0 |
| **Origin** | AUTOSAR_ECUC |

| **Parameter Name** | **OsConstName** |
|---|---|
| **Description** | The name which is accessed by the application to get the OsTimeValue of the constant. |
| **Multiplicity** | 1..1 |
| **Type** | STRING |
| **Origin** | Elektrobit Automotive GmbH |

### 4.1.1.16. OsEvent

| **Parameters included** | |
|---|---|
| **Parameter name** | **Multiplicity** |

| Parameters included | |
|---|---|
| OsEventMask | 0..1 |

| Parameter Name | OsEventMask |
|---|---|
| Description | The **OsEventMask** attribute is a UINT64 attribute that specifies the set of bits to be associated with the event. The EB tresos AutoCore OS kernel supports up to 32 events per task, therefore the event mask must be restricted to the lower 32 bits of the word. |
| Multiplicity | 0..1 |
| Type | INTEGER |
| Range | >=1<br><=4294967295 |
| Origin | AUTOSAR_ECUC |

### 4.1.1.17. OsSpinlock

| Parameters included | |
|---|---|
| **Parameter name** | **Multiplicity** |
| OsSpinlockAccessingApplication | 1..n |
| OsSpinlockSuccessor | 0..1 |
| OsSpinlockLockMethod | 1..1 |

| Parameter Name | OsSpinlockAccessingApplication |
|---|---|
| Description | Reference to OsApplications that have an access to this object. Objects of the referenced OsApplication can acquire or release this OsSpinlock. |
| Multiplicity | 1..n |
| Type | REFERENCE |
| Origin | AUTOSAR_ECUC |

| Parameter Name | OsSpinlockSuccessor |
|---|---|
| Description | Reference to the next OsSpinlock object in the linked list. To check whether a spinlock can be occupied (in a nested way) without any danger of deadlock, a linked list of spinlocks can be defined. A spinlock can only be occupied in the order of the linked list. It is allowed to skip a spinlock. If no linked list is specified, spinlocks cannot be nested. |

| | |
|---|---|
| **Multiplicity** | 0..1 |
| **Type** | REFERENCE |
| **Origin** | AUTOSAR_ECUC |

| **Parameter Name** | **OsSpinlockLockMethod** |
|---|---|
| **Description** | OsSpinlockLockMethod is an enumerated type whose value is one of:<br>▶ LOCK_NOTHING<br>▶ LOCK_ALL_INTERRUPTS<br>▶ LOCK_CAT2_INTERRUPTS<br>▶ LOCK_WITH_RES_SCHEDULER<br><br>OsSpinlockLockMethod describes the lock method, which is additionally applied when a spinlock is taken. This method modifies priority and interrupt lock level of tasks, which hold this spinlock. If **LOCK_NOTHING** is chosen, taking the spinlock will not change the current task's priority or interrupt lock level. **LOCK_ALL_INTERRUPTS** will cause all interrupts to be disabled. **LOCK_CAT2_INTERRUPTS** will disable all category 2 ISRs while the spinlock is taken. **LOCK_WITH_RES_SCHEDULER** will prevent the task, holding this spinlock, from being preempted by another task. It is recommended to lock out all tasks and ISRs which could try to take a spinlock to prevent certain kinds of deadlocks. |
| **Multiplicity** | 1..1 |
| **Type** | ENUMERATION |
| **Default value** | LOCK_NOTHING |
| **Range** | LOCK_NOTHING<br>LOCK_ALL_INTERRUPTS<br>LOCK_CAT2_INTERRUPTS<br>LOCK_WITH_RES_SCHEDULER |
| **Origin** | AUTOSAR_ECUC |

### 4.1.1.18. OsIsr

| **Containers included** | | |
|---|---|---|
| **Container name** | **Multiplicity** | **Description** |
| OsIsrTimingProtection | 0..1 | **OsIsrTimingProtection** is a boolean attribute that specifies whether the kernel should apply timing protection to the ISR. |

| Containers included | | |
|---|---|---|
| | | When this attribute is **TRUE**, the sub-attributes **OsIsrExecutionBudget**, **OsIsrTimeFrame** and **OsIsrLockBudget** are available. They are described in the following sections. |

| Parameters included | |
|---|---|
| **Parameter name** | **Multiplicity** |
| OsIsrCategory | 1..1 |
| OsIsrPeriod | 0..1 |
| OsIsrResourceRef | 0..n |
| OsMeasure_Max_Runtime | 0..1 |
| OsEnable_On_Startup | 0..1 |
| OsStacksize | 1..1 |
| OsIsrAccessingApplication | 0..n |

| Parameter Name | OsIsrCategory |
|---|---|
| **Description** | **OsIsrCategory** is a UINT32 attribute that defines the IRS's Category. Only the values "CATEGORY_1" and "CATEGORY_2" are permitted. |
| **Multiplicity** | 1..1 |
| **Type** | ENUMERATION |
| **Range** | CATEGORY_1<br>CATEGORY_2 |
| **Origin** | AUTOSAR_ECUC |

| Parameter Name | OsIsrPeriod |
|---|---|
| **Description** | **OsIsrPeriod** specifies the period in seconds of a periodically-triggered ISR.<br><br>The value can be used by the RTE module so that you can map timing events to an ISR.<br><br>It is your responsibility to ensure that the hardware triggers the ISR at the correct frequency. The OS does not use and cannot verify the correctness of the value you configure.<br><br>If you do not provide a value for this parameter, you cannot map RTE timing events to the ISR. |

| | |
|---|---|
| **Multiplicity** | 0..1 |
| **Type** | FLOAT |
| **Range** | >=0.0<br><=86400.0 |
| **Origin** | AUTOSAR_ECUC |

| **Parameter Name** | **OsIsrResourceRef** |
|---|---|
| **Description** | This reference defines the resources accessed by this ISR. |
| **Multiplicity** | 0..n |
| **Type** | REFERENCE |
| **Origin** | AUTOSAR_ECUC |

| **Parameter Name** | **OsMeasure_Max_Runtime** |
|---|---|
| **Description** | **OsMeasure_Max_Runtime** is a boolean attribute that tells the kernel to record the longest-observed execution-time for this ISR. The value can be obtained by calling the function *OS_GetIsrMaxRuntime*. |
| **Multiplicity** | 0..1 |
| **Type** | BOOLEAN |
| **Default value** | false |
| **Origin** | Elektrobit Automotive GmbH |

| **Parameter Name** | **OsEnable_On_Startup** |
|---|---|
| **Description** | **OsEnable_On_Startup** is a boolean attribute that determines whether the kernel should automatically enable the interrupt source at start-up. If this attribute is set to **FALSE**, the application code is responsible for enabling this interrupt source using `OS_EnableInterruptSource()` when needed. |
| **Multiplicity** | 0..1 |
| **Type** | BOOLEAN |
| **Default value** | true |
| **Origin** | Elektrobit Automotive GmbH |

| **Parameter Name** | **OsStacksize** |
|---|---|
| **Description** | **OsStackSize** specifies the stack size of the ISR in bytes. |
| **Multiplicity** | 1..1 |
| **Type** | INTEGER |
| **Range** | >=0 |

| | |
|---|---|
| | <=2000000000 |
| **Origin** | Elektrobit Automotive GmbH |

| **Parameter Name** | **OsIsrAccessingApplication** |
|---|---|
| **Description** | Reference to OsApplications that have an access to this object. Objects of the referenced OsApplication can enable or disable this interrupt. |
| **Multiplicity** | 0..n |
| **Type** | REFERENCE |
| **Origin** | Elektrobit Automotive GmbH |

### 4.1.1.19. OsIsrTimingProtection

| **Containers included** | | |
|---|---|---|
| **Container name** | **Multiplicity** | **Description** |
| OsIsrResourceLock | 0..n | This container contains a list of times the interrupt uses resources. |

| **Parameters included** | |
|---|---|
| **Parameter name** | **Multiplicity** |
| OsIsrAllInterruptLockBudget | 0..1 |
| OsIsrExecutionBudget | 0..1 |
| OsIsrOsInterruptLockBudget | 0..1 |
| OsIsrTimeFrame | 0..1 |
| OsIsrCountLimit | 0..1 |

| **Parameter Name** | **OsIsrAllInterruptLockBudget** |
|---|---|
| **Description** | This parameter contains the maximum time for which the ISR is allowed to lock all interrupts (via SuspendAllInterrupts() or DisableAllInterrupts()) (in seconds). |
| **Multiplicity** | 0..1 |
| **Type** | FLOAT |
| **Range** | >=0.0<br><=86400.0 |
| **Origin** | AUTOSAR_ECUC |

| Parameter Name | OsIsrExecutionBudget |
|---|---|
| Description | **OsIsrExecutionBudget** specifies, in seconds, the maximum execution time permitted for the ISR, from call to return. If the ISR is interrupted by a higher priority category 2 ISR, the interruption does not count towards the execution time of the ISR. However, time spent in category 1 ISRs is counted in the time of the interrupted ISR. |
| Multiplicity | 0..1 |
| Type | FLOAT |
| Origin | AUTOSAR_ECUC |

| Parameter Name | OsIsrOsInterruptLockBudget |
|---|---|
| Description | This parameter contains the maximum time for which the ISR is allowed to lock all Category 2 interrupts (via SuspendOSInterrupts()) (in seconds). |
| Multiplicity | 0..1 |
| Type | FLOAT |
| Origin | AUTOSAR_ECUC |

| Parameter Name | OsIsrTimeFrame |
|---|---|
| Description | This parameter contains the minimum inter-arrival time between successive interrupts (in seconds). |
| Multiplicity | 0..1 |
| Type | FLOAT |
| Range | >=0.0<br><=86400.0 |
| Origin | AUTOSAR_ECUC |

| Parameter Name | OsIsrCountLimit |
|---|---|
| Description | **OsIsrCountLimit** specifies the number of allowed interrupt arrivals within the time frame specified by OsIsrTimeFrame. |
| Multiplicity | 0..1 |
| Type | INTEGER |
| Default value | 1 |
| Range | >=0<br><=65535 |
| Origin | Elektrobit Automotive GmbH |

### 4.1.1.20. OsIsrResourceLock

| Parameters included | |
|---|---|
| **Parameter name** | **Multiplicity** |
| OsIsrResourceLockBudget | 1..1 |
| OsIsrResourceLockResourceRef | 1..1 |

| Parameter Name | OsIsrResourceLockBudget |
|---|---|
| **Description** | This parameter contains the maximum time the interrupt is allowed to hold the given resource (in seconds). |
| **Multiplicity** | 1..1 |
| **Type** | FLOAT |
| **Range** | >=0.0<br><=86400.0 |
| **Origin** | AUTOSAR_ECUC |

| Parameter Name | OsIsrResourceLockResourceRef |
|---|---|
| **Description** | Reference to the resource the locking time is depending on |
| **Multiplicity** | 1..1 |
| **Type** | REFERENCE |
| **Origin** | AUTOSAR_ECUC |

### 4.1.1.21. OsOS

| Containers included | | |
|---|---|---|
| **Container name** | **Multiplicity** | **Description** |
| OsHooks | 1..1 | Container to structure all hooks belonging to the OS |
| OsAutosarCustomization | 0..1 | The OsAutosarCustomization container and its attrributes can be used to can be use to fine-tune the OS to gain size, performance or other benefits. **Warning: Use of non-default values for these options means that the OS is not fully conformant with the AUTOSAR specification.** |
| OsCoreConfig | 0..n | |

| Parameters included | |
|---|---|
| **Parameter name** | **Multiplicity** |
| OsScalabilityClass | 0..1 |
| OsNumberOfCores | 0..1 |
| OsStackMonitoring | 1..1 |
| OsStatus | 1..1 |
| OsUseGetServiceId | 1..1 |
| OsUseParameterAccess | 1..1 |
| OsUseResScheduler | 1..1 |
| OsCC | 0..1 |
| OsTrace | 1..1 |
| OsExtra_Run-time_Checks | 1..1 |
| OsStartupChecks | 1..1 |
| OsServiceTrace | 1..1 |
| OsSourceOptimization | 1..1 |
| OsStackOptimization | 1..1 |
| OsProtection | 1..1 |
| OsUseLastError | 1..1 |
| OsTracebuffer | 1..1 |
| OsSchedule | 0..1 |
| OsTrappingKernel | 0..1 |
| OsGenerateSWCD | 1..1 |
| OsUseLogicalCoreIDs | 1..1 |
| OsInitCoreId | 0..1 |
| OsMaxNumberOfCores | 1..1 |

| Parameter Name | OsScalabilityClass |
|---|---|
| **Description** | A scalability class for each System Object OS has to be selected. In order to customize the operating system to the needs of the user and to take full advantage of the processor features the operating system can be scaled according to the scalability classes. The value is one of:<br>▶ SC1<br>▶ SC2 |

| | |
|---|---|
| | - SC3<br>- SC4 |
| **Multiplicity** | 0..1 |
| **Type** | ENUMERATION |
| **Range** | SC1<br>SC2<br>SC3<br>SC4 |
| **Origin** | AUTOSAR_ECUC |

| **Parameter Name** | **OsNumberOfCores** |
|---|---|
| **Description** | Maximum number of cores that are controlled by EB tresos AutoCore OS. |
| **Multiplicity** | 0..1 |
| **Type** | INTEGER |
| **Default value** | 1 |
| **Origin** | AUTOSAR_ECUC |

| **Parameter Name** | **OsStackMonitoring** |
|---|---|
| **Description** | The **OsStackMonitoring** attribute is a BOOLEAN attribute that specifies whether the kernel should perform software stack monitoring at run-time. If it is set to **TRUE**, the stack monitoring is enabled. |
| **Multiplicity** | 1..1 |
| **Type** | BOOLEAN |
| **Default value** | false |
| **Origin** | AUTOSAR_ECUC |

| **Parameter Name** | **OsStatus** |
|---|---|
| **Description** | STATUS is an enumerated type whose value is one of:<br>- STANDARD<br>- EXTENDED<br><br>In OS there is no possibility of the system entering an undefined state because of an error in the application code. Errors are always checked for and reported. The **STATUS** setting determines how the kernel handles the error. In **STANDARD** mode OSEK/VDX does not specify how the kernel should react. In this mode, a typical OS reaction to a static error is to quarantine the offending task or |

| | |
|---|---|
| | application. In **EXTENDED** mode OSEK/VDX specifies that the system services should return certain error codes when an error is detected. |
| **Multiplicity** | 1..1 |
| **Type** | ENUMERATION |
| **Default value** | STANDARD |
| **Range** | EXTENDED<br>STANDARD |
| **Origin** | AUTOSAR_ECUC |

| **Parameter Name** | **OsUseGetServiceId** |
|---|---|
| **Description** | In the precompiled OS kernel the *OSErrorGetServiceID()* API is always available within the *ErrorHook()*. However, if you are compiling an optimized kernel from the source code, the **USEGETSERVICEID** attribute can be used to enable or disable the feature and could result in a smaller, faster kernel. |
| **Multiplicity** | 1..1 |
| **Type** | BOOLEAN |
| **Default value** | false |
| **Origin** | AUTOSAR_ECUC |

| **Parameter Name** | **OsUseParameterAccess** |
|---|---|
| **Description** | In the precompiled OS kernel the *OSError_x1_x2()* APIs are always available within the *ErrorHook()*. However, if you are compiling an optimized kernel from the source code, the **USEPARAMETERACCESS** attribute can be used to enable or disable the feature and could result in a smaller, faster kernel. |
| **Multiplicity** | 1..1 |
| **Type** | BOOLEAN |
| **Default value** | false |
| **Origin** | AUTOSAR_ECUC |

| **Parameter Name** | **OsUseResScheduler** |
|---|---|
| **Description** | **OsUseResScheduler** is a boolean attribute. If it is **TRUE**, the Generator creates a resource called **RES_SCHEDULER** whose resource ID is typically 0. Any task is eligible to take this resource. The ceiling priority of this resource is at least as high as the highest task priority. The OSEK/VDX API *RES_SCHEDULER* is defined in terms of this resource. |
| **Multiplicity** | 1..1 |

| | |
|---|---|
| **Type** | BOOLEAN |
| **Default value** | true |
| **Origin** | AUTOSAR_ECUC |

| **Parameter Name** | **OsCC** |
|---|---|
| **Description** | Choose automatic selection or one of the following conformance classes:<br>▶ BCC1<br>▶ BCC2<br>▶ ECC1<br>▶ ECC2<br>The precompiled OS kernel always supports an **ECC2** system, but the setting here is used to check that all the tasks satisfy the desired conformance class constraints. If an optimized kernel is compiled from the sources, a lower CC setting might result in a smaller, faster kernel. |
| **Multiplicity** | 0..1 |
| **Type** | ENUMERATION |
| **Range** | BCC1<br>BCC2<br>ECC1<br>ECC2 |
| **Origin** | Elektrobit Automotive GmbH |

| **Parameter Name** | **OsTrace** |
|---|---|
| **Description** | **OsTrace** is a boolean attribute. If it is **TRUE**, the macro OS_USE_TRACE will be passed via the Make environment to the OS library code, which enables the trace hooks for the Debug and Trace module. |
| **Multiplicity** | 1..1 |
| **Type** | BOOLEAN |
| **Default value** | false |
| **Origin** | Elektrobit Automotive GmbH |

| **Parameter Name** | **OsExtra_Runtime_Checks** |
|---|---|
| **Description** | **OsExtra_Runtime_Checks** is a boolean attribute. If it is **TRUE**, the kernel makes a range of extra checks at specific points while the system is running. This is helpful during the development phase for debugging purposes. |

| | |
|---|---|
| **Multiplicity** | 1..1 |
| **Type** | BOOLEAN |
| **Default value** | false |
| **Origin** | Elektrobit Automotive GmbH |

| **Parameter Name** | **OsStartupChecks** |
|---|---|
| **Description** | **OsStartupChecks** is a boolean attribute. If it is **TRUE**, the kernel makes a range of extra checks at system start-up. This is helpful during the development phase to detect possible configuration errors and to ensure a coherent system state after start-up. |
| **Multiplicity** | 1..1 |
| **Type** | BOOLEAN |
| **Default value** | false |
| **Origin** | Elektrobit Automotive GmbH |

| **Parameter Name** | **OsServiceTrace** |
|---|---|
| **Description** | Check this if you want to trace system calls via ORTI |
| **Multiplicity** | 1..1 |
| **Type** | BOOLEAN |
| **Default value** | false |
| **Origin** | Elektrobit Automotive GmbH |

| **Parameter Name** | **OsSourceOptimization** |
|---|---|
| **Description** | Check this if you want to build a library optimized according to the configuration. |
| **Multiplicity** | 1..1 |
| **Type** | BOOLEAN |
| **Default value** | false |
| **Origin** | Elektrobit Automotive GmbH |

| **Parameter Name** | **OsStackOptimization** |
|---|---|
| **Description** | **OsStackOptimization** is an enumerated attribute that controls how the Generator optimizes task stacks across tasks and applications. The values are:<br>▶ NO<br>▶ WITHIN_APPLICATIONS<br>▶ GLOBAL |

| | |
|---|---|
| | With NO optimization, each task gets its own stack area. This option uses the most RAM but is useful to determine how much stack each individual task really uses. Optimization WITHIN_APPLICATIONS allows tasks of the same application to share a stack when the tasks types and priorities permit. GLOBAL optimization allows tasks from different applications to share stacks. This option provides the most efficient RAM footprint, but might conflict with memory protection mechanisms on some architectures. |
| **Multiplicity** | 1..1 |
| **Type** | ENUMERATION |
| **Default value** | GLOBAL |
| **Range** | NO<br>WITHIN_APPLICATIONS<br>GLOBAL |
| **Origin** | Elektrobit Automotive GmbH |

| **Parameter Name** | **OsProtection** |
|---|---|
| **Description** | On microcontrollers that support memory protection the presence of non-trusted applications in the configuration will cause memory protection to be enabled. On some microcontrollers this can cause problems with debugger breakpoints in the flash memory. On such processors the **OsProtection** attribute permits you to disable the memory protection features without changing the trust status of the applications. The possible values of the **OsProtection** attribute are ON and OFF. Note that the use of this attribute does not affect the trust status of applications, nor does it affect the CPU mode in which the tasks run, so if a task performs an action that is not permitted in the user mode of the CPU, the protection system will still detect it. Setting the PROTECTION attribute to any value other than ON, invalidates any safety certification of the OS. The Generator produces a warning for this. |
| **Multiplicity** | 1..1 |
| **Type** | ENUMERATION |
| **Default value** | ON |
| **Range** | OFF<br>ON |
| **Origin** | Elektrobit Automotive GmbH |

| **Parameter Name** | **OsUseLastError** |
|---|---|
| **Description** | **OsUseLastError** is a boolean attribute. If it is **TRUE**, the last error is stored internally and can be accessed via ORTI. |

| | |
|---|---|
| **Multiplicity** | 1..1 |
| **Type** | BOOLEAN |
| **Default value** | false |
| **Origin** | Elektrobit Automotive GmbH |

| **Parameter Name** | **OsTracebuffer** |
|---|---|
| **Description** | **OsTracebuffer** defines the size of the trace buffer for tracing. A value of 0 disables tracing. |
| **Multiplicity** | 1..1 |
| **Type** | INTEGER |
| **Default value** | 0 |
| **Range** | >=0<br><=65535 |
| **Origin** | Elektrobit Automotive GmbH |

| **Parameter Name** | **OsSchedule** |
|---|---|
| **Description** | ▶ NON<br>▶ FULL<br>▶ MIXED<br><br>**NON** means that all Tasks must have their **OsTaskSchedule** attribute set to **NON**. **FULL** means that all Tasks must have their **OsTaskSchedule** attribute set to **FULL**. **MIXED** means that a mixture of Task scheduling types is permitted. The precompiled OS kernel always supports mixed scheduling, but the attribute allows the generator to check that all tasks satisfy the desired scheduling constraints. If an optimized kernel is compiled from the sources, a more restrictive **OsSchedule** setting might result in a smaller, faster kernel. |
| **Multiplicity** | 0..1 |
| **Type** | ENUMERATION |
| **Default value** | MIXED |
| **Range** | NON<br>FULL<br>MIXED |
| **Origin** | Elektrobit Automotive GmbH |

| **Parameter Name** | **OsTrappingKernel** |
|---|---|

| | |
|---|---|
| **Description** | **OsTrappingKernel** is an optional boolean attribute. If it is **TRUE**, the kernel is entered via a Systrap mechanism. This is necessary for memory protection. If it is **FALSE**, the kernel is entered via function calls. Memory protection is not available in this case. If the optional parameter is disabled, it will be automatically enabled if non-trusted applications are found. **Note:** This parameter is only available on architectures that allow a selection between Systrap and function calls. |
| **Multiplicity** | 0..1 |
| **Type** | BOOLEAN |
| **Origin** | Elektrobit Automotive GmbH |

| **Parameter Name** | **OsGenerateSWCD** |
|---|---|
| **Description** | **OsGenerateSWCD** is a boolean attribute. If it is enabled, the OS specific software component description (SWCD) files will be generated, exporting a subset of the OS API via RTE interfaces. **Note:** Enabling this parameter will result in bigger generation and compile times. You only need to enable it if you are using software components that access OS API via RTE calls, which is unlikely. |
| **Multiplicity** | 1..1 |
| **Type** | BOOLEAN |
| **Default value** | false |
| **Origin** | Elektrobit Automotive GmbH |

| **Parameter Name** | **OsUseLogicalCoreIDs** |
|---|---|
| **Description** | **Note:** Advanced logical core mapping is currently not supported. The default setting and behavior is as for disabled.<br><br>**OsUseLogicalCoreIDs** enables the Advanced Logical Core ID configuration feature.<br><br>If this value is disabled, all logical core IDs are equivalent to their physical counterparts.<br><br>If the feature is enabled, the logical core IDs must be configured within OsCoreConfig. |
| **Multiplicity** | 1..1 |
| **Type** | BOOLEAN |
| **Default value** | false |
| **Origin** | Elektrobit Automotive GmbH |

| **Parameter Name** | **OsInitCoreId** |
|---|---|
| **Description** | **OsInitCoreId** designates the processor core, which will control the OS start-up. |

| | |
|---|---|
| | If this value is disabled, the generator will choose it by itself. It depends on the target hardware and the current configuration, which one of the cores is chosen automatically. If you want a certain core to control the OS start-up, then enable OsInitCoreId.<br><br>Note: If ALCI feature is enabled, this value represents the logical core ID of the master core. |
| **Multiplicity** | 0..1 |
| **Type** | INTEGER |
| **Origin** | Elektrobit Automotive GmbH |

| **Parameter Name** | **OsMaxNumberOfCores** |
|---|---|
| **Description** | This is the number of cores provided by the hardware. |
| **Multiplicity** | 1..1 |
| **Type** | INTEGER |
| **Default value** | 1 |
| **Origin** | Elektrobit Automotive GmbH |

### 4.1.1.22. OsHooks

| **Parameters included** | |
|---|---|
| **Parameter name** | **Multiplicity** |
| OsErrorHook | 1..1 |
| OsPostTaskHook | 1..1 |
| OsPreTaskHook | 1..1 |
| OsProtectionHook | 0..1 |
| OsShutdownHook | 1..1 |
| OsStartupHook | 1..1 |
| OsPreISRHook | 1..1 |
| OsPostISRHook | 1..1 |

| **Parameter Name** | **OsErrorHook** |
|---|---|
| **Description** | **OsErrorHook** is a boolean attribute. If it is **TRUE**, the kernel calls the user-supplied *void ErrorHook(StatusType errorcode)* whenever an error is detected, unless the error was caused within **ErrorHook()**. |
| **Multiplicity** | 1..1 |

| | |
|---|---|
| **Type** | BOOLEAN |
| **Default value** | true |
| **Origin** | AUTOSAR_ECUC |

| **Parameter Name** | **OsPostTaskHook** |
|---|---|
| **Description** | **OsPostTaskHook** is a boolean attribute. If it is **TRUE**, the kernel calls the user-supplied *void PostTask-Hook(void)* when a task is about to leave the running state. |
| **Multiplicity** | 1..1 |
| **Type** | BOOLEAN |
| **Default value** | false |
| **Origin** | AUTOSAR_ECUC |

| **Parameter Name** | **OsPreTaskHook** |
|---|---|
| **Description** | **OsPreTaskHook** is a boolean attribute. If it is **TRUE**, the kernel calls the user-supplied *void PreTaskHook(void)* just before task execution resumes, but after the incoming task has entered the running state. |
| **Multiplicity** | 1..1 |
| **Type** | BOOLEAN |
| **Default value** | false |
| **Origin** | AUTOSAR_ECUC |

| **Parameter Name** | **OsProtectionHook** |
|---|---|
| **Description** | **OsProtectionHook** is a boolean attribute. If it is **TRUE**, the kernel calls the user-supplied *ProtectionReturn-Type ProtectionHook(StatusType errorcode)* whenever a protection violation is detected, unless the error was caused within *ProtectionHook()*. The return value of the *ProtectionHook()* function can be one of:<br>▶ PRO_TERMINATETASKISR<br>▶ PRO_TERMINATEAPPL<br>▶ PRO_TERMINATEAPPL_RESTART<br>▶ PRO_SHUTDOWN<br>▶ PRO_IGNORE |
| **Multiplicity** | 0..1 |
| **Type** | BOOLEAN |
| **Default value** | true |

| | |
|---|---|
| **Origin** | AUTOSAR_ECUC |

| **Parameter Name** | **OsShutdownHook** |
|---|---|
| **Description** | **OsShutdownHook** is a boolean attribute. If it is **TRUE**, the kernel calls the user-supplied *void ShutdownHook(StatusType errorcode)* when the system has been shutdown. The parameter is the value of the error code passed to *ShutdownOS()* |
| **Multiplicity** | 1..1 |
| **Type** | BOOLEAN |
| **Default value** | true |
| **Origin** | AUTOSAR_ECUC |

| **Parameter Name** | **OsStartupHook** |
|---|---|
| **Description** | **OsStartupHook** is a boolean attribute. If it is **TRUE**, the kernel calls the user-supplied *void StartupHook(void)* immediately before starting the scheduler. |
| **Multiplicity** | 1..1 |
| **Type** | BOOLEAN |
| **Default value** | false |
| **Origin** | AUTOSAR_ECUC |

| **Parameter Name** | **OsPreISRHook** |
|---|---|
| **Description** | **OsPreISRHook** is a boolean attribute. If it is **TRUE**, the kernel calls the user-supplied *void PreIsrHook(os_isrid_t isrid)* just before an ISR is called. The parameter is the ID of the ISR. For each ISR, the Generator defines a macro whose name is the name of the ISR and whose value is its ISR ID. |
| **Multiplicity** | 1..1 |
| **Type** | BOOLEAN |
| **Default value** | false |
| **Origin** | Elektrobit Automotive GmbH |

| **Parameter Name** | **OsPostISRHook** |
|---|---|
| **Description** | **OsPostISRHook** is a boolean attribute. If it is **TRUE**, the kernel calls the user-supplied *void PostIsrHook(os_isrid_t isrid)* just after an ISR returns. The parameter is the ID of the ISR. For each ISR, the Generator defines a macro whose name is the name of the ISR and whose value is its ISR ID. |
| **Multiplicity** | 1..1 |
| **Type** | BOOLEAN |

| | |
|---|---|
| **Default value** | false |
| **Origin** | Elektrobit Automotive GmbH |

### 4.1.1.23. OsAutosarCustomization

| **Parameters included** | |
|---|---|
| **Parameter name** | **Multiplicity** |
| OsExceptionHandling | 1..1 |
| OsErrorHandling | 1..1 |
| OsStrictServiceProtection | 1..1 |
| OsCat1DirectCall | 1..1 |
| OsInterruptLockingChecks | 1..1 |
| OsCallIsr | 1..1 |
| OsCallAppErrorHook | 1..1 |
| OsCallAppStartupShutdownHook | 1..1 |
| OsPermitSystemObjects | 1..1 |
| OsUserTaskReturn | 1..1 |

| **Parameter Name** | **OsExceptionHandling** |
|---|---|
| **Description** | This parameter can be used to disable the execption handling. If set to **FALSE**, a minimal exception vector table is used, which can be adapted if necessary. The exact behaviour is architecture-dependent. On some architectures this option may have no effect because the OS relies on some exceptions to perform its duties. Setting the option to **FALSE** will remove the ability of the OS to detect and correctly react to protection errors. |
| **Multiplicity** | 1..1 |
| **Type** | BOOLEAN |
| **Default value** | true |
| **Origin** | Elektrobit Automotive GmbH |

| **Parameter Name** | **OsErrorHandling** |
|---|---|
| **Description** | This parameter can be used to restrict the amount of error handling that is performed by the OS. The permitted values are **MINIMAL** , **AUTOSAR** , and **FULL.** |

| | |
|---|---|
| | If you choose **MINIMAL**, the error handler *OS_Error()* is never called, and the default error code is returned to the user. Choosing this option means that error and protection hooks cannot be called and the correct action after an error (such as terminating a task) does not take place. If you choose **AUTOSAR**, the error handler *OS_Error()* will be called for all errors except those that occur in System Services that do not return **StatusType**. This is the Autosar-conformant option. If you choose **FULL**, the error handler *OS_Error()* will be called for all errors, including those that occur in System Services that do not return **StatusType**. It will also call the error hook for those errors. |
| **Multiplicity** | 1..1 |
| **Type** | ENUMERATION |
| **Default value** | AUTOSAR |
| **Range** | MINIMAL<br>AUTOSAR<br>FULL |
| **Origin** | Elektrobit Automotive GmbH |

| **Parameter Name** | **OsStrictServiceProtection** |
|---|---|
| **Description** | Setting this option to **FALSE** disables most of the calling-context checks in the System Services. The OS will then only check the calling context when it is necessary for the correct functioning of the OS. |
| **Multiplicity** | 1..1 |
| **Type** | BOOLEAN |
| **Default value** | true |
| **Origin** | Elektrobit Automotive GmbH |

| **Parameter Name** | **OsCat1DirectCall** |
|---|---|
| **Description** | This parameter selects whether a category 1 ISR is called directly or via the operating system's category 1 interrupt handler.<br><br>When this option is disabled, the operating system's category 1 handler is used as the entry in the interrupt vector table. This handler performs a context save, switches to the kernel stack (if applicable, depending on the architecture) and sets internal context data for operating system for service protection. This setting is conformant with the AUTOSAR specification.<br><br>If you enable this option, the configured ISR is entered directly into the interrupt vector table. This allows a fast entry into the ISR, but AUTOSAR service protection fails. Furthermore, use of operating system APIs is not supported, be- |

| | |
|---|---|
| | cause the APIs do not know that they have been called from a category 1 ISR and may not function correctly. This applies even to the interrupt locking APIs (SuspendAllInterrupts/ResumeAllInterrupts etc.).<br><br>Please note that on several architectures the ISR routine needs to be prefixed with an __interrupt keyword (check compiler documentation for further details) which saves the context prior to entering the ISR. You can pass the keyword to the ISR prototype by defining the macro OS_INTERRUPT_KEYWORD prior to including Os.h. For example, if the toolchain uses the keyword **__interrupt**, use the following code:<br><br>`#define OS_INTERRUPT_KEYWORD __interrupt`<br>`#include "Os.h"`<br>`__interrupt ISR(foo)`<br>`{`<br>`  (...)`<br>`}` |
| **Multiplicity** | 1..1 |
| **Type** | BOOLEAN |
| **Default value** | false |
| **Origin** | Elektrobit Automotive GmbH |

| **Parameter Name** | **OsInterruptLockingChecks** |
|---|---|
| **Description** | ▶ **MINIMAL**: Select **MINIMAL** to only check the interrupt lock status when it affects the kernel's operation. The interrupt lock status affects the kernel's operation e.g. in the functions `GetResource` and `ReleaseResource`.<br>▶ **EXTRACHECK**: Select **EXTRACHECK** to check the interrupt lock status in all API functions which may cause a task switch.<br>▶ **AUTOSAR**: Select **AUTOSAR** to be fully compliant with the AUTOSAR specification.<br><br>**Tasks always start with interrupts enabled** The interrupt lock status is considered to be part of the task's context. This means that each newly activated task starts with interrupts enabled. |
| **Multiplicity** | 1..1 |
| **Type** | ENUMERATION |
| **Default value** | AUTOSAR |
| **Range** | MINIMAL |

| | |
|---|---|
| | EXTRACHECK<br>AUTOSAR |
| **Origin** | Elektrobit Automotive GmbH |

| **Parameter Name** | **OsCallIsr** |
|---|---|
| **Description** | Setting this option to **DIRECTLY** causes the OS to call all category 2 ISRs directly rather than via a wrapper function. This means that all ISRs (even non-trusted) run with the permissions of the OS, and ISRs cannot be terminated if they cause a protection fault. |
| **Multiplicity** | 1..1 |
| **Type** | ENUMERATION |
| **Default value** | VIA_WRAPPER |
| **Range** | DIRECTLY<br>VIA_WRAPPER |
| **Origin** | Elektrobit Automotive GmbH |

| **Parameter Name** | **OsCallAppErrorHook** |
|---|---|
| **Description** | Setting this option to DIRECTLY causes the OS to call all application error hooks directly rather than via a wrapper function. This means that all error hooks (even those belonging to non-trusted applications) run with the permissions of the OS, and the error hooks cannot be terminated if they cause a protection fault. **The global ErrorHook and ProtectionHook functions are always called directly.** |
| **Multiplicity** | 1..1 |
| **Type** | ENUMERATION |
| **Default value** | VIA_WRAPPER |
| **Range** | DIRECTLY<br>VIA_WRAPPER |
| **Origin** | Elektrobit Automotive GmbH |

| **Parameter Name** | **OsCallAppStartupShutdownHook** |
|---|---|
| **Description** | Setting this option to **DIRECTLY** causes the OS to call all application start-up and shutdown hooks directly rather than via a wrapper function. This means that all these hooks (even those belonging to non-trusted applications) run with the permissions of the OS, and the hooks cannot be terminated if they cause a protection fault. **The global StartupHook and ShutdownHook are always called directly.** |

| | |
|---|---|
| **Multiplicity** | 1..1 |
| **Type** | ENUMERATION |
| **Default value** | VIA_WRAPPER |
| **Range** | DIRECTLY<br>VIA_WRAPPER |
| **Origin** | Elektrobit Automotive GmbH |

| **Parameter Name** | **OsPermitSystemObjects** |
|---|---|
| **Description** | Setting this option to **TRUE** inhibits the check that, if an OS application exists, all Tasks and ISRs must belong to an OS application. Objects that do not belong to an application are known as **system objects** and are always trusted. |
| **Multiplicity** | 1..1 |
| **Type** | BOOLEAN |
| **Default value** | false |
| **Origin** | Elektrobit Automotive GmbH |

| **Parameter Name** | **OsUserTaskReturn** |
|---|---|
| **Description** | The *OS_MissingTerminateTask()* function is entered if a task returns from its main function without calling *TerminateTask()* or *ChainTask()*. This optimisation option controls how *OS_MissingTerminateTask()* handles the error.<br><br>Setting this option to LOOP configures *OS_MissingTerminateTask()* to be a simple endless loop. If you know that none of your tasks can ever return from its main function you can select this option to save code space. However, if a task ever returns without calling *TerminateTask()*, or *TerminateTask()* returns unexpectedly (for example if the task still occupies a resource or has disabled interrupts) it will remain in an endless loop.<br><br>Setting this option to KILL_TASK configures *OS_MissingTerminateTask()* to enter the kernel and execute *OS_KernTaskReturn()*. *OS_KernTaskReturn()* will either call the error handler to terminate the task or, if error handling is disabled, terminate the task itself. If this fails for any reason, *OS_MissingTerminateTask()* will try to shutdown the OS. If this fails, too, there is still an endless loop to prevent the task from executing undefined code. |
| **Multiplicity** | 1..1 |
| **Type** | ENUMERATION |
| **Default value** | KILL_TASK |
| **Range** | KILL_TASK |

| | |
|---|---|
| | LOOP |
| **Origin** | Elektrobit Automotive GmbH |

### 4.1.1.24. OsCoreConfig

| **Parameters included** | |
|---|---|
| **Parameter name** | **Multiplicity** |
| OsCoreId | 1..1 |
| OsLogicalCoreId | 1..1 |

| **Parameter Name** | **OsCoreId** |
|---|---|
| **Description** | **OsCoreId** physical core index based on the CPU core ID.<br><br>Values range from 0 to OsMaxNumberOfCores-1.<br><br>The logical core value OsLogicalCoreId is mapped to the value of the physical core index shown in OsCoreId. |
| **Multiplicity** | 1..1 |
| **Type** | INTEGER |
| **Default value** | 0 |
| **Origin** | Elektrobit Automotive GmbH |

| **Parameter Name** | **OsLogicalCoreId** |
|---|---|
| **Description** | **OsLogicalCoreId** manually changes the logical core ID for the physical core with index OsCoreId.<br><br>To change this configuration item, OsUseLogicalCoreIDs has to be enabled within OsOS.<br><br>Potential values range from -1 (default value) up to OsMaxNumberOfCores-1.<br><br>The default logical core value of '-1' means that you choose to use the physical core index shown in OsCoreId for the value of the logical core ID.<br><br>The logical core values that you choose for the whole configuration should be zero-based, consecutive and unique. |
| **Multiplicity** | 1..1 |
| **Type** | INTEGER |
| **Default value** | -1 |

| | |
|---|---|
| **Origin** | Elektrobit Automotive GmbH |

#### 4.1.1.25. OsPeripheralArea

#### 4.1.1.26. OsResource

| **Parameters included** | |
|---|---|
| **Parameter name** | **Multiplicity** |
| OsResourceProperty | 1..1 |
| OsResourceAccessin-gApplication | 0..n |
| OsResourceLinke-dResourceRef | 0..1 |

| **Parameter Name** | **OsResourceProperty** |
|---|---|
| **Description** | **RESOURCEPROPERTY** is an enumerated attribute that whose values are: <dl> <dt>STANDARD</dt> <dd>a normal resource that can be expicitly taken and released by application code</dd> <dt>LINKED</dt> <dd>a resource that is linked to another resource of type **STANDARD** or **LINKED**. The sub-attribute **LINKEDRESOURCE** specifies the resource to which it is linked.</dd> <dt>INTERNAL</dt> <dd>a resource that cannot be expicitly taken and released by application code. The resource is automatically given to a task whenever the task enters the running state.</dd> </dl> |
| **Multiplicity** | 1..1 |
| **Type** | ENUMERATION |
| **Default value** | STANDARD |
| **Range** | INTERNAL<br>LINKED<br>STANDARD |
| **Origin** | AUTOSAR_ECUC |

| **Parameter Name** | **OsResourceAccessingApplication** |
|---|---|
| **Description** | Reference to OsApplications that have an access to this object. Objects of the referenced OsApplication can acquire or release this OsResource. |

| | |
|---|---|
| **Multiplicity** | 0..n |
| **Type** | REFERENCE |
| **Origin** | AUTOSAR_ECUC |

| **Parameter Name** | **OsResourceLinkedResourceRef** |
|---|---|
| **Description** | The link to the resource. Must be valid if OsResourceProperty is LINKED. If OsResourceProperty is not LINKED the value is ignored. |
| **Multiplicity** | 0..1 |
| **Type** | REFERENCE |
| **Origin** | AUTOSAR_ECUC |

### 4.1.1.27. OsScheduleTable

| **Containers included** | | |
|---|---|---|
| **Container name** | **Multiplicity** | **Description** |
| OsScheduleTableAutostart | 0..1 | **OsScheduleTableAutostart** is a boolean attribute whose value specifies whether the alarm shall be started automatically when the kernel starts. If the value is **TRUE**, the **OsAppmode** sub-attribute specifies in which application modes the task shall be automatically started, and the sub-attribute OsScheduleTableOffset specifies the time at which the first event of the schedule shall take place. The OsScheduleTableOffset is specified in ticks or nanoseconds depending on the **UNIT** attribute of the schedule table. |
| OsScheduleTableExpiryPoint | 1..n | The point on a Schedule Table at which the OS activates tasks and/or sets events |
| OsScheduleTableSync | 0..1 | This parameter specifies the synchronization parameters of the schedule table. |

| **Parameters included** | |
|---|---|
| **Parameter name** | **Multiplicity** |
| OsScheduleTableDuration | 1..1 |
| OsScheduleTableRepeating | 1..1 |
| OsSchTblAccessingApplication | 0..n |

| **Parameters included** | |
|---|---|
| OsScheduleTableCounterRef | 1..1 |
| OsTimeUnit | 0..1 |

| **Parameter Name** | **OsScheduleTableDuration** |
|---|---|
| **Description** | The **OsScheduleTableDuration** attribute specifies the length of time for which the schedule table runs, from start to finish. For periodic schedule tables, it is the period. The **OsScheduleTableDuration** sub-attribute is specified in nanoseconds or ticks depending on the UNIT attribute of the schedule table. |
| **Multiplicity** | 1..1 |
| **Type** | INTEGER |
| **Default value** | 0 |
| **Origin** | AUTOSAR_ECUC |

| **Parameter Name** | **OsScheduleTableRepeating** |
|---|---|
| **Description** | The **OsScheduleTableRepeating** attribute specifies whether the schedule table is periodic. <dl> <dt>TRUE</dt> <dd>periodic schedule tables repeat indefinitely until explicitly stopped</dd> <dt>FALSE</dt> <dd>the schedule table processing stops when the final expiry point is processed</dd> </dl> |
| **Multiplicity** | 1..1 |
| **Type** | BOOLEAN |
| **Default value** | false |
| **Origin** | AUTOSAR_ECUC |

| **Parameter Name** | **OsSchTblAccessingApplication** |
|---|---|
| **Description** | Reference to OsApplications that have an access to this object. Objects of the referenced OsApplication can start, stop, synchronize or enquire about the status of this OsScheduleTable. |
| **Multiplicity** | 0..n |
| **Type** | REFERENCE |
| **Origin** | AUTOSAR_ECUC |

| **Parameter Name** | **OsScheduleTableCounterRef** |
|---|---|
| **Description** | This parameter contains a reference to the counter which drives the schedule table.Each Schedule Table must be associated with exactly one Counter. |
| **Multiplicity** | 1..1 |

| | |
|---|---|
| **Type** | REFERENCE |
| **Origin** | AUTOSAR_ECUC |

| **Parameter Name** | **OsTimeUnit** |
|---|---|
| **Description** | **OsTimeUnit** contains the time unit type used for this schedule table. |
| **Multiplicity** | 0..1 |
| **Type** | ENUMERATION |
| **Default value** | TICKS |
| **Range** | NANOSECONDS<br>TICKS |
| **Origin** | Elektrobit Automotive GmbH |

### 4.1.1.28. OsScheduleTableAutostart

| **Parameters included** | |
|---|---|
| **Parameter name** | **Multiplicity** |
| OsScheduleTableAutostartType | 1..1 |
| OsScheduleTableAppModeRef | 1..n |
| OsScheduleTableStartValue | 1..1 |

| **Parameter Name** | **OsScheduleTableAutostartType** |
|---|---|
| **Description** | This specifies the type of the autostart for the schedule table. |
| **Multiplicity** | 1..1 |
| **Type** | ENUMERATION |
| **Default value** | RELATIVE |
| **Range** | ABSOLUTE<br>RELATIVE<br>SYNCHRON |
| **Origin** | AUTOSAR_ECUC |

| **Parameter Name** | **OsScheduleTableAppModeRef** |
|---|---|

| | |
|---|---|
| **Description** | Reference in which application modes the schedule table should be started during start-up |
| **Multiplicity** | 1..n |
| **Type** | REFERENCE |
| **Origin** | AUTOSAR_ECUC |

| **Parameter Name** | **OsScheduleTableStartValue** |
|---|---|
| **Description** | Value depending on **OsScheduleTableAutostartType**:<br>▶ ABSOLUTE: Absolute autostart tick value when the schedule table starts.<br>▶ RELATIVE: Relative offset in ticks when the schedule table starts. |
| **Multiplicity** | 1..1 |
| **Type** | INTEGER |
| **Default value** | 0 |
| **Range** | >=0<br><=4294967295 |
| **Origin** | AUTOSAR_ECUC |

### 4.1.1.29. OsScheduleTableExpiryPoint

| **Containers included** | | |
|---|---|---|
| **Container name** | **Multiplicity** | **Description** |
| OsScheduleTableEventSetting | 0..n | Event that is triggered by that schedule table. |
| OsScheduleTableTaskActivation | 0..n | Task that is triggered by that schedule table. |
| OsScheduleTblAdjustableExpPoint | 0..1 | Adjustable expiry point |

| **Parameters included** | |
|---|---|
| **Parameter name** | **Multiplicity** |
| OsScheduleTblExpPointOffset | 1..1 |

| **Parameter Name** | **OsScheduleTblExpPointOffset** |
|---|---|

| | |
|---|---|
| **Description** | The offset from zero (in ticks) at which the expiry point is to be processed. |
| **Multiplicity** | 1..1 |
| **Type** | INTEGER |
| **Origin** | AUTOSAR_ECUC |

### 4.1.1.30. OsScheduleTableEventSetting

| **Parameters included** | |
|---|---|
| **Parameter name** | **Multiplicity** |
| OsScheduleTableSetEventRef | 1..1 |
| OsScheduleTableSetEventTaskRef | 1..1 |

| **Parameter Name** | **OsScheduleTableSetEventRef** |
|---|---|
| **Description** | Reference to event that will be set by action |
| **Multiplicity** | 1..1 |
| **Type** | REFERENCE |
| **Origin** | AUTOSAR_ECUC |

| **Parameter Name** | **OsScheduleTableSetEventTaskRef** |
|---|---|
| **Multiplicity** | 1..1 |
| **Type** | REFERENCE |
| **Origin** | AUTOSAR_ECUC |

### 4.1.1.31. OsScheduleTableTaskActivation

| **Parameters included** | |
|---|---|
| **Parameter name** | **Multiplicity** |
| OsScheduleTableActivateTaskRef | 1..1 |

| **Parameter Name** | **OsScheduleTableActivateTaskRef** |
|---|---|
| **Description** | Reference to task that will be activated by action |

| | |
|---|---|
| **Multiplicity** | 1..1 |
| **Type** | REFERENCE |
| **Origin** | AUTOSAR_ECUC |

### 4.1.1.32. OsScheduleTblAdjustableExpPoint

| **Parameters included** | |
|---|---|
| **Parameter name** | **Multiplicity** |
| OsScheduleTable-MaxLengthen | 1..1 |
| OsScheduleTable-MaxShorten | 1..1 |

| **Parameter Name** | **OsScheduleTableMaxLengthen** |
|---|---|
| **Description** | The maximum positive adjustment that can be made to the expiry point offset specified in nanoseconds or ticks depending on the UNIT attribute of the schedule table. |
| **Multiplicity** | 1..1 |
| **Type** | INTEGER |
| **Default value** | 0 |
| **Range** | >=0<br><=4294967295 |
| **Origin** | AUTOSAR_ECUC |

| **Parameter Name** | **OsScheduleTableMaxShorten** |
|---|---|
| **Description** | The maximum negative adjustment that can be made to the expiry point offset specified in nanoseconds or ticks depending on the UNIT attribute of the schedule table. |
| **Multiplicity** | 1..1 |
| **Type** | INTEGER |
| **Default value** | 0 |
| **Range** | >=0<br><=4294967295 |
| **Origin** | AUTOSAR_ECUC |

### 4.1.1.33. OsScheduleTableSync

| Parameters included | |
|---|---|
| **Parameter name** | **Multiplicity** |
| OsScheduleTblExplicit-Precision | 0..1 |
| OsScheduleTblSyncStrat-egy | 1..1 |

| Parameter Name | OsScheduleTblExplicitPrecision |
|---|---|
| **Description** | **OsScheduleTblExplicitPrecision** defines the deviation threshold for considering a schedule table to be "synchronous". This parameter is only needed if explicit synchronisation is used. |
| **Multiplicity** | 0..1 |
| **Type** | INTEGER |
| **Range** | >=0<br><=4294967295 |
| **Origin** | AUTOSAR_ECUC |

| Parameter Name | OsScheduleTblSyncStrategy |
|---|---|
| **Description** | The OS provides support for synchronisation in two ways: explicit and implicit.<br>▶ **EXPLICIT**: The schedule table is driven by an OS counter but processing needs to be synchronized with a different counter which is not an OS counter object. The API function **SyncScheduleTable()** provides the synchronization count to the schedule table. Expiry points with **OsScheduleTblAdjustableExpPoint** configuration are used to adjust the schedule table to the synchronization count.<br>▶ **IMPLICIT**: The counter driving the schedule table is the counter with which synchronisation is required.<br>▶ **NONE**: No support for synchronisation. (default) |
| **Multiplicity** | 1..1 |
| **Type** | ENUMERATION |
| **Default value** | NONE |
| **Range** | EXPLICIT<br>IMPLICIT<br>NONE |

| Origin | AUTOSAR ECUC |
|---|---|

### 4.1.1.34. OsTask

| Containers included | | |
|---|---|---|
| **Container name** | **Multiplicity** | **Description** |
| OsTaskAutostart | 0..1 | **OsTaskAutostart** is a boolean attribute whose value specifies whether the task shall be started automatically when the kernel starts. If the value is **TRUE**, the **OsTaskAppModeRef** sub-attribute specifies in which application modes the task shall be automatically started. |
| OsTaskTimingProtection | 0..1 | OsTaskTimingProtection is a boolean attribute that specifies whether the kernel should apply timing protection to the task. When this attribute is **TRUE**, the sub-attributes EXECUTIONBUDGET, TIMEFRAME and LOCKINGTIME are available. |

| Parameters included | |
|---|---|
| **Parameter name** | **Multiplicity** |
| OsTaskActivation | 1..1 |
| OsTaskPriority | 1..1 |
| OsTaskPeriod | 0..1 |
| OsTaskSchedule | 1..1 |
| OsTaskAccessingApplication | 0..n |
| OsTaskEventRef | 0..n |
| OsTaskResourceRef | 0..n |
| OsMeasure_Max_Runtime | 0..1 |
| OsTaskUse_Hw_Fp | 0..1 |
| OsTaskCallScheduler | 0..1 |
| OsTaskType | 0..1 |
| OsStacksize | 1..1 |

| Parameter Name | OsTaskActivation |
|---|---|
| Description | **ACTIVATION** is a UINT32 attribute whose value defines the maximum number of activations that a task can have at any one time. |

| | |
|---|---|
| **Multiplicity** | 1..1 |
| **Type** | INTEGER |
| **Default value** | 1 |
| **Range** | >=1<br><=255 |
| **Origin** | AUTOSAR_ECUC |

| **Parameter Name** | **OsTaskPriority** |
|---|---|
| **Description** | **OsTaskPriority** is a UINT32 attribute whose value defines the relative base priority of the task. The lowest priority is zero; larger values correspond to higher priorities. The values given for the **OsTaskPriority** attribute only specify a relative ordering. The actual values configured for the kernel by the Generator can be different from those specified. |
| **Multiplicity** | 1..1 |
| **Type** | INTEGER |
| **Range** | >=0<br><=2147483647 |
| **Origin** | AUTOSAR_ECUC |

| **Parameter Name** | **OsTaskPeriod** |
|---|---|
| **Description** | **OsTaskPeriod** specifies the period in seconds of a periodically-activated task.<br><br>The value can be used by the RTE module so that you can map timing events to a task whose time scheduling is not generated by RTE.<br><br>It is your responsibility to ensure that the task's activations take place at the correct frequency. The OS does not use and cannot verify the correctness of the value you configure.<br><br>If you do not provide a value for this parameter, you might not be able to map RTE timing events to the task. |
| **Multiplicity** | 0..1 |
| **Type** | FLOAT |
| **Range** | >=0.0<br><=86400.0 |
| **Origin** | AUTOSAR_ECUC |

| **Parameter Name** | **OsTaskSchedule** |
|---|---|

| | |
|---|---|
| **Description** | OsTaskSchedule is an enumerated type whose value is one of:<br>▶ NON<br>▶ FULL<br>**FULL** specifies that the task is preemptable. **NON** specifies that the task is not preemptable. |
| **Multiplicity** | 1..1 |
| **Type** | ENUMERATION |
| **Default value** | FULL |
| **Range** | FULL<br>NON |
| **Origin** | AUTOSAR_ECUC |

| **Parameter Name** | **OsTaskAccessingApplication** |
|---|---|
| **Description** | Reference to applications which have an access to this object. Objects of the referenced OsAplication can change the state of current Task by calling the system service APIs. For example this task can be activated or an event can be set for it by objects of the referenced OsApplication. |
| **Multiplicity** | 0..n |
| **Type** | REFERENCE |
| **Origin** | AUTOSAR_ECUC |

| **Parameter Name** | **OsTaskEventRef** |
|---|---|
| **Description** | This reference defines the list of events the extended task may react on. |
| **Multiplicity** | 0..n |
| **Type** | REFERENCE |
| **Origin** | AUTOSAR_ECUC |

| **Parameter Name** | **OsTaskResourceRef** |
|---|---|
| **Description** | This reference defines a list of resources accessed by this task. |
| **Multiplicity** | 0..n |
| **Type** | REFERENCE |
| **Origin** | AUTOSAR_ECUC |

| **Parameter Name** | **OsMeasure_Max_Runtime** |
|---|---|

| | |
|---|---|
| **Description** | **OsMeasure_Max_Runtime** is a boolean attribute that tells the kernel to record the longest-observed executiontime for this task. The value can be obtained by calling the function *OS_GetTaskMaxRuntime*. |
| **Multiplicity** | 0..1 |
| **Type** | BOOLEAN |
| **Default value** | false |
| **Origin** | Elektrobit Automotive GmbH |

| **Parameter Name** | **OsTaskUse_Hw_Fp** |
|---|---|
| **Description** | **OsTaskUse_Hw_Fp** is a boolean attribute that tells the kernel whether to provide a full floating-point environment for the task. The implementation of floating-point environments is architecture-dependent. See the Architecture Supplement. |
| **Multiplicity** | 0..1 |
| **Type** | BOOLEAN |
| **Origin** | Elektrobit Automotive GmbH |

| **Parameter Name** | **OsTaskCallScheduler** |
|---|---|
| **Description** | The **OsTaskCallScheduler** attribute informs the generator whether the task calls the *Schedule()* service. If **OsTaskCallScheduler** is set to **NO**, the generator assumes that *Schedule()* is never called by the task. If it is set to **YES** or to **DONTKNOW**, the generator assumes that *Schedule()* may be called. This information is used to determine which tasks are able to preempt each other. |
| **Multiplicity** | 0..1 |
| **Type** | ENUMERATION |
| **Range** | DONTKNOW<br>YES<br>NO |
| **Origin** | Elektrobit Automotive GmbH |

| **Parameter Name** | **OsTaskType** |
|---|---|
| **Description** | **OsTaskType** is an enumerated type whose value is one of:<br>- BASIC<br>- EXTENDED<br><br>BASIC specifies that the task is a basic task. EXTENDED specifies that the task is an extended task. |

| | |
|---|---|
| **Multiplicity** | 0..1 |
| **Type** | ENUMERATION |
| **Range** | BASIC<br>EXTENDED |
| **Origin** | Elektrobit Automotive GmbH |

| **Parameter Name** | **OsStacksize** |
|---|---|
| **Description** | **OsStacksize** specifies the stack size of the task in bytes. Note that the generator adds an overhead for saving the task context on the stack during task switches, depending on the task and OS configuration. |
| **Multiplicity** | 1..1 |
| **Type** | INTEGER |
| **Range** | >=0<br><=2000000000 |
| **Origin** | Elektrobit Automotive GmbH |

### 4.1.1.35. OsTaskAutostart

| **Parameters included** | |
|---|---|
| **Parameter name** | **Multiplicity** |
| OsTaskAppModeRef | 1..n |

| **Parameter Name** | **OsTaskAppModeRef** |
|---|---|
| **Description** | Reference to application modes in which that task is activated on start-up of the OS |
| **Multiplicity** | 1..n |
| **Type** | REFERENCE |
| **Origin** | AUTOSAR_ECUC |

### 4.1.1.36. OsTaskTimingProtection

| **Containers included** | | |
|---|---|---|
| **Container name** | **Multiplicity** | **Description** |

| Containers included | | |
|---|---|---|
| OsTaskResource-Lock | 0..n | This parameter contains the worst case time between getting and releasing a given resource (in seconds). |

| Parameters included | |
|---|---|
| **Parameter name** | **Multiplicity** |
| OsTaskAllInterruptLock-Budget | 0..1 |
| OsTaskExecutionBudget | 0..1 |
| OsTaskOsInterruptLock-Budget | 0..1 |
| OsTaskTimeFrame | 0..1 |
| OsTaskCountLimit | 0..1 |

| Parameter Name | OsTaskAllInterruptLockBudget |
|---|---|
| **Description** | This parameter contains the maximum time for which the task is allowed to lock all interrupts (via SuspendAllInterrupts() or DisableAllInterrupts()) (in seconds). |
| **Multiplicity** | 0..1 |
| **Type** | FLOAT |
| **Range** | >=0.0<br><=86400.0 |
| **Origin** | AUTOSAR_ECUC |

| Parameter Name | OsTaskExecutionBudget |
|---|---|
| **Description** | **OsTaskExecutionBudget** specifies, in seconds, the maximum execution time permitted for the task, from activation to termination. If the task is interrupted by a higher priority task or a category 2 ISR, the interruption does not count towards the task's execution time. However, time spent in category 1 ISRs is counted in the time of the interrupted task. An extended task's execution timer is stopped when it enters the WAITING state, and is restarted from the beginning when the event occurs. Waiting for an event that is already pending also restarts the execution timer from the beginning. |
| **Multiplicity** | 0..1 |
| **Type** | FLOAT |
| **Origin** | AUTOSAR_ECUC |

| Parameter Name | OsTaskOsInterruptLockBudget |
|---|---|

| | |
|---|---|
| **Description** | This parameter contains the maximum time for which the task is allowed to lock all Category 2 interrupts (via SuspendOSInterrupts()) (in seconds). |
| **Multiplicity** | 0..1 |
| **Type** | FLOAT |
| **Origin** | AUTOSAR_ECUC |

| **Parameter Name** | **OsTaskTimeFrame** |
|---|---|
| **Description** | The minimum inter-arrival time between activations and/or releases of a task (in seconds). |
| **Multiplicity** | 0..1 |
| **Type** | FLOAT |
| **Range** | >=0.0<br><=86400.0 |
| **Origin** | AUTOSAR_ECUC |

| **Parameter Name** | **OsTaskCountLimit** |
|---|---|
| **Description** | **OsTaskCountLimit** specifies the number of allowed task arrivals within the time frame specified by OsTaskTimeFrame. |
| **Multiplicity** | 0..1 |
| **Type** | INTEGER |
| **Default value** | 1 |
| **Range** | >=0<br><=65535 |
| **Origin** | Elektrobit Automotive GmbH |

### 4.1.1.37. OsTaskResourceLock

| **Parameters included** | |
|---|---|
| **Parameter name** | **Multiplicity** |
| OsTaskResourceLock-Budget | 1..1 |
| OsTaskResourceLockRe-sourceRef | 1..1 |

| Parameter Name | OsTaskResourceLockBudget |
|---|---|
| Description | This parameter contains the maximum time the task is allowed to lock the resource (in seconds) |
| Multiplicity | 1..1 |
| Type | FLOAT |
| Range | >=0.0 |
| | <=86400.0 |
| Origin | AUTOSAR_ECUC |

| Parameter Name | OsTaskResourceLockResourceRef |
|---|---|
| Description | Reference to the resource used by the task |
| Multiplicity | 1..1 |
| Type | REFERENCE |
| Origin | AUTOSAR_ECUC |

## 4.2. OSEK/AUTOSAR reference

### 4.2.1. General description

The OSEK/AUTOSAR API is implemented in terms of the underlying EB tresos AutoCore OS API through an interface layer that is implemented as a set of macros and library functions.

In most cases, the OSEK API function `XxxYyy()` is implemented by calling the EB tresos AutoCore OS user-library function `OS_UserXxxYyy()`. When minor differences in the API occur, these are translated either directly in the macro or indirectly using a library function called `OS_XxxYyy()`.

OSEK/AUTOSAR API data types are implemented in terms of the underlying EB tresos AutoCore OS data types using macros. In some cases the range of values returned by the underlying API is larger than the OSEK/AUTOSAR standard allows. In such cases the extended values are translated by a library function.

The interface layer therefore behaves exactly like a standard OSEK/AUTOSAR implementation. The underlying EB tresos AutoCore OS API are required only if the extended features need to be accessed, or in the unlikely event that the address of an API function needs to be used.

The OSEK/AUTOSAR API can be used by including the header file `Os.h` in your programs.

## 4.2.2. OSEK/AUTOSAR data types

This section describes the OSEK/AUTOSAR data types.

| Datatype | Description |
|---|---|
| `AccessType` | A scalar type that holds information about how a specific memory region can be accessed. |
| `AlarmBaseType` | A structure holding the characteristics of the counter associated with an alarm. The fields of the structure include the following:<br>`maxallowedvalue`<br>The maximum count before the counter rolls over.<br>`ticksperbase`<br>The number of ticks required to reach a counter-specific unit.<br>`mincycle`<br>The minimum number of ticks required for a cyclic alarm (extended mode only). |
| `AlarmBaseRefType` | A pointer type that holds the address of a variable of type `AlarmBaseType`. |
| `AlarmType` | A scalar type for an alarm. |
| `AppModeType` | A scalar type that specifies the mode of an OS application at start-up. |
| `ApplicationType` | A scalar type for an OS application. |
| `CounterType` | A scalar type for a counter. |
| `EventMaskType` | A scalar type for an event. |
| `EventMaskRefType` | A pointer type that holds the address of a variable of type `EventMaskType`. |
| `IdleModeType` | A scalar type for the idle mode behavior. |
| `ISRType` | A scalar type for an ISR. |
| `MemoryStartAddressType` | A pointer type that holds the address of any location in memory. |
| `MemorySizeType` | A scalar type for the size of a memory region. |
| `ObjectAccessType` | A scalar type that holds information on whether an object can be accessed by an OS application. |
| `ObjectTypeType` | A scalar type for a type of an object. |
| `PhysicalTimeType` | A scalar type for a physical time in nanoseconds, microseconds, milliseconds, or seconds. |
| `ProtectionReturnType` | A scalar type which controls further actions of the OS on return from the `ProtectionHook()`. |

| Datatype | Description |
|---|---|
| `ResourceType` | A scalar type for a resource. |
| `RestartType` | A scalar type that specifies whether or not an OS application should be restarted after it is terminated. |
| `ScheduleTableType` | A scalar type for a schedule table. |
| `ScheduleTableStatusType` | A scalar type that describes the status of a schedule table. |
| `ScheduleTableStatusRefType` | A pointer type that holds the address of a variable of type `ScheduleTableStatusType`. |
| `StatusType` | A scalar type for the status returned by API calls. The status can either be `E_OK` if the API was executed successfully, or one of the error codes listed in Section 4.4.2, "List of OSEK/AUTOSAR error codes". |
| `TaskType` | A scalar type for an index of the task entry in the task table. |
| `TaskRefType` | A pointer type that holds the address of a variable of type `TaskType`. |
| `TaskStateType` | A scalar type for the status of a task. |
| `TaskStateRefType` | A pointer type that holds the address of a variable of type `TaskStateType`. |
| `TickType` | A scalar type for the counter value in ticks. |
| `TickRefType` | A pointer type that holds the address of a variable of type `TickType`. |
| `TrustedFunctionIndexType` | A scalar type for a trusted function. |
| `TrustedFunctionParameterRefType` | A pointer type that holds the address of the data passed to a trusted function. |

Table 4.1. OSEK/AUTOSAR Data Types

## 4.2.3. OSEK/AUTOSAR constants

This section describes the OSEK/AUTOSAR constants.

| Type | Identifier | Description |
|---|---|---|
| Macro definitions for alarm base values for the system counter. | `OSMAXALLOWEDVALUE` | The maximum tick count before the counter rolls over. |
| | `OSTICKSPERBASE` | The number of system counter ticks required to reach a specific unit |
| | `OSMINCYCLE` | The minimum number of ticks required for a cyclic alarm. |
| | `OSTICKDURATION` | The duration of a system counter tick in nanoseconds. |

| Type | Identifier | Description |
|---|---|---|
| Macro definitions for alarm base values of other counters, where `x` is the name of the counter. | `OSMAXALLOWEDVALUE_x` | The maximum tick count before counter x rolls over. |
| | `OSTICKSPERBASE_x` | The number of ticks required to reach a specific unit. |
| | `OSMINCYCLE_x` | The minimum allowed number of ticks required for a cyclic alarm of counter x. |
| `ApplicationType` | `INVALID_OSAPPLICATION` | The ID of an invalid application |
| `AppModeType` | `OSDEFAULTAPPMODE` | The default application mode |
| `IdleModeType`. Note that each architecture might specify specific idle modes. | `IDLE_NO_HALT` | The core does not perform any specific actions during idle time. |
| `ISRType` | `INVALID_ISR` | The ID of an invalid ISR |
| `ObjectAccessType`. | `ACCESS` | The object can be accessed. |
| | `NO_ACCESS` | The object can not be accessed |
| `ObjectTypeType` | `OBJECT_TASK` | The object is a task. |
| | `OBJECT_ISR` | The object is an ISR. |
| | `OBJECT_ALARM` | The object is an alarm. |
| | `OBJECT_RESOURCE` | The object is a resource. |
| | `OBJECT_COUNTER` | The object is a counter. |
| | `OBJECT_SCHEDULETABLE` | The object is a schedule table. |
| `RestartType`. It is a parameter to `TerminateApplication()` | `RESTART` | The application should be restarted. |
| | `NO_RESTART` | The application must not be restarted. |
| `ResourceType` | `RES_SCHEDULER` | The scheduler resource |
| `ProtectionReturnType`. It is the return value from `ProtectionHook()` | `PRO_KILLTASKISR` | The offending task or ISR is to be terminated. |
| | `PRO_KILLAPPL` | The offending application is to be terminated. |
| | `PRO_KILLAPPL_RESTART` | The offending application is to be terminated and then restarted. |
| | `PRO_SHUTDOWN` | The entire system is to be shut down. |
| `ScheduleTableStatusType` | `SCHEDULETABLE_NOT_STARTED` | The schedule table is not running. |
| | `SCHEDULETABLE_RUNNING` | The schedule table is running but is not currently synchronized with global time. |

| Type | Identifier | Description |
|---|---|---|
| | SCHEDULETABLE_RUNNING_AND_SYNCHRONOUS | The schedule table is running and is synchronized with global time. |
| | SCHEDULETABLE_NEXT | The schedule table is waiting for the end of a running schedule table. |
| | SCHEDULETABLE_WAITING | The schedule table is waiting for global time. |
| TaskStateType | RUNNING | Task is in the running state |
| | WAITING | Task is in the waiting state |
| | READY | Task is in the ready state. |
| | SUSPENDED | Task is in the suspended state. |
| TaskType | INVALID_TASK | The ID of an invalid task |

Table 4.2. OSEK/AUTOSAR constants

## 4.2.4. OSEK/AUTOSAR API

This section describes the OSEK/AUTOSAR API and hook functions.

## Name

ActivateTask — Activate a task

## Synopsis

```
#include <Os.h>
StatusType ActivateTask(TaskType t);
```

## Description

`ActivateTask()` activates a task. If the specified task is currently in the *suspended* state, its new state will be *ready*. If the task is already *ready* or *running* the activation will be recorded and performed after the task terminates, if permitted.

## Service identification

`OS_SID_ActivateTask`.

## Return value

A return value of `E_OK` indicates a successful completion of the function. Any other return value indicates an error code as described below.

| ErrorId(OS_ERROR_XX) | AUTOSAR error code | Description |
|---|---|---|
| ApplicationNotAccessible | E_OS_ACCESS | The application to which the task belongs was terminated and has not yet restarted. |
| CoreIsDown | E_OS_CORE | The core on which the task resides has been shutdown. |
| WrongContext | E_OS_CALLLEVEL | The system service was called from a context that is not permitted. |
| InterruptDisabled | E_OS_INTDISABLE | The system service was called with interrupts disabled. |
| InvalidTaskId | E_OS_ID | The specified task ID is invalid. |
| Quarantined | E_OS_DENIED | The specified task has been quarantined and will not be activated. |
| MaxActivations | E_OS_LIMIT | The specified task has exceeded its activation limit. |
| RateLimitExceeded | E_OS_RATEPROT | The specified task has exceeded its activation rate limit. |
| Permission | E_OS_ACCESS | Permission has not been granted for the caller to access the referenced task. |

## Name

ActivateTaskAsyn — Activate a task asynchronously

## Synopsis

```
#include <Os.h>
void ActivateTaskAsyn(TaskType t);
```

## Description

`ActivateTaskAsyn()` is similar to `ActivateTask()` and can be used to activate the task. If the specified task is in the callers core, the behavior is exactly same as `ActivateTask()`. But when the task to be activated is on the *remote cores*, *activation request is placed on the* remote core and no return value will be captured. The core on which task activation happens would take care of the activation and in case of errors, error hooks (if configured) will be called.

## Service identification

`OS_SID_ActivateTaskAsyn`.

## Return value

Returns nothing. In case of any errors, ErrorHook will be called (if ErrorHook is configured and "OsErrorHandling" parameter is configured as FULL) with any one of the error code described below.

| **ErrorId(OS_ERROR_XX)** | **AUTOSAR error code** | **Description** |
|---|---|---|
| ApplicationNotAccessible | E_OS_ACCESS | The application to which the task belongs was terminated and has not yet restarted. |
| WrongContext | E_OS_CALLLEVEL | The system service was called from a context that is not permitted. |
| InterruptDisabled | E_OS_INTDISABLE | The system service was called with interrupts disabled. |
| InvalidTaskId | E_OS_ID | The specified task ID is invalid. |
| Quarantined | E_OS_DENIED | The specified task has been quarantined and will not be activated. |
| MaxActivations | E_OS_LIMIT | The specified task has exceeded its activation limit. |
| RateLimitExceeded | E_OS_RATEPROT | The specified task has exceeded its activation rate limit. |
| Permission | E_OS_ACCESS | Permission has not been granted for the caller to access the referenced task. |

## Name

AdvanceCounter — Increment the given counter

## Synopsis

```
#include <Os.h>
StatusType AdvanceCounter(CounterIdType CounterName);
```

## Description

`AdvanceCounter()` increments the given counter by 1. Any alarm that expires as a result of this will cause the appropriate alarm action to take place. If the action is an alarm callback, the callback function runs in the context of the caller of `AdvanceCounter()`.

This service is called only from the task level and not from the interrupt level. For incrementing counters within an interrupt, see IAdvanceCounter.

In AUTOSAR Os, `AdvanceCounter()` and `IAdvanceCounter()` are identical, but failure to observe the above distinction may result in non-portable code.

## Return value

A return value of `E_OK` indicates a successful completion of the function.

A return value of `E_OS_ID` indicates that the alarm ID is wrong.

## Name

ALARMCALLBACK — Define an alarm Callback function

## Synopsis

**`ALARMCALLBACK`**`(alarmcallbackname);`

## Description

The `ALARMCALLBACK()` macro defines an OS function to implement the alarm callback whose name is given in the `alarmcallbackname` parameter. The code you wish to execute when the alarm expires is placed in the body of the function.

The alarm callback function is executed in the context of the kernel, so it may be necessary to increase the size of the kernel stack to ensure that a stack overflow does not occur. This can be achieved by increasing the stack size of an ISR or adding a dummy ISR.

The `ALARMCALLBACK()` macro can only be used at the outer level of a C source file.

## Service identification

Not Applicable.

## Return value

Returns Nothing.

## Name

AllowAccess — Grant access to the calling application

## Synopsis

```
#include <Os.h>
StatusType AllowAccess(void);
```

## Description

`AllowAccess()` sets the state of application of the calling task or ISR to ACCESSIBLE, provided it is in the RESTARTING state.

`AllowAccess()` may only be called from a task or an ISR.

## Service identification

`OS_SID_AllowAccess`.

## Return value

A return value of `E_OK` indicates a successful completion of the function. Any other return value indicates an error code as described below.

| **ErrorId(OS_ERROR_XX)** | **AUTOSAR error code** | **Description** |
|---|---|---|
| ApplicationNotRestarting | E_OS_STATE | AllowAccess was called from an application that was not restarting. |
| InterruptDisabled | E_OS_INTDISABLE | The system service was called with interrupts disabled. |
| WrongContext | E_OS_CALLLEVEL | The system service was called from a context that is not permitted. |

## Name

CallTrustedFunction — Call a trusted function

## Synopsis

```
#include <Os.h>
StatusType CallTrustedFunction(TrustedFunctionIndexType fid, void *parms);
```

## Description

`CallTrustedFunction()` calls the referenced trusted function with the parameter supplied, provided that the caller is in a permitted context and has permission to make the call.

It is recommended to make trusted functions as short as possible, doing only those jobs such as accessing peripheral devices that can only be done with full privileges. It is not recommended to call OSEK or AUTOSAR system services from a trusted function.

However, if it is absolutely necessary to use system services from a trusted function, you must note the following restrictions and differences in semantic behaviour:

A trusted function is called in a kernel environment, which means that all system calls that it makes will return immediately to the caller; any resulting task switch will not happen until the trusted function returns, thus affecting the calling task but not the trusted function.

If a trusted function has been called from an ISR (category 2) context, the system services that it can call are restricted accordingly. Calling a system service that is not permitted will result in an error code being returned to the trusted function. In normal status mode it is possible that the calling application could have been terminated.

## Service identification

`OS_SID_CallTrustedFunction`.

## Return value

A return value of `E_OK` indicates a successful completion of the function. Any other return value indicates an error code as described below.

| **ErrorId(OS_ERROR_XX)** | **AUTOSAR error code** | **Description** |
|---|---|---|
| ApplicationNotAccessible | E_OS_ACCESS | The application to which the trusted function belongs was terminated and has not yet restarted. |
| InvalidFunctionId | E_OS_TFID | The specified trusted function does not exist. |
| InterruptDisabled | E_OS_INTDISABLE | The system service was called with interrupts disabled. |
| WrongContext | E_OS_CALLLEVEL | The system service was called from a context that is not permitted. |
| Permission | E_OS_ACCESS | Permission has not been granted for the caller to call the referenced trusted function. |
| StackError | E_OS_STACKPROT | The call could result in the trusted function using stack outside the caller's stack boundary. |

| ErrorId(OS_ERROR_XX) | AUTOSAR error code | Description |
|---|---|---|
| CallTrustedFunction-Crosscore | E_OS_ACCESS | If the target trusted function is part of an OS application on another core |

## Name

CancelAlarm — Cancel an alarm

## Synopsis

```
#include <Os.h>
StatusType CancelAlarm(AlarmType a);
```

## Description

`CancelAlarm()` removes the specified alarm from its counter's alarm list. The alarm must have been previously started with `SetRelAlarm()`, `SetAbsAlarm()` or by autostart.

## Service identification

`OS_SID_CancelAlarm`.

## Return value

A return value of `E_OK` indicates a successful completion of the function. Any other return value indicates an error code as described below.

| **ErrorId(OS_ERROR_XX)** | **AUTOSAR error code** | **Description** |
|---|---|---|
| ApplicationNotAccessible | E_OS_ACCESS | The application to which the alarm belongs was terminated and has not yet restarted. |
| CoreIsDown | E_OS_CORE | The core on which the alarm resides has been shutdown. |
| WrongContext | E_OS_CALLLEVEL | The system service was called from a context that is not permitted. |
| InterruptDisabled | E_OS_INTDISABLE | The system service was called with interrupts disabled. |
| InvalidAlarmId | E_OS_ID | The specified alarm ID is invalid. |
| AlarmNotInUse | E_OS_NOFUNC | The specified alarm is not currently in use. |
| Permission | E_OS_ACCESS | Permission has not been granted for the caller to access the referenced alarm. |

## Name

ChainScheduleTable — Chain a schedule table

## Synopsis

```
#include <Os.h>
StatusType ChainScheduleTable(ScheduleTableType sc, ScheduleTableType sn);
```

## Description

`ChainScheduleTable()` chains the schedule table `sn` to start after the current round of the table `sc` ends. Chaining is only permitted if the table to be chained is stopped and if the current table is running and does not already have a chained table.

The timing is arranged such that the first action point of the chained table occurs at its proper offset after the end of the period of the *current* table. If the *current* table is not periodic, the first action point takes place at its offset from the last action point of the *current* table. The AUTOSAR specification is silent on the latter case.

Note: The chaining takes place at the last action point of the *current* table. This means that if `NextSchedul-eTable()` (or `ChainScheduleTable()`) is called after this (for example, in the last schedule task) the running table will process one more complete round before the chaining takes place. If the *current* table is not periodic it may already have stopped and the call to `NextScheduleTable()` will fail with `OS_E_STATE`.

## Service identification

`OS_SID_ChainScheduleTable`.

## Return value

A return value of `E_OK` indicates a successful completion of the function. Any other return value indicates an error code as described below.

| **ErrorId(OS_ERROR_XX)** | **AUTOSAR error code** | **Description** |
|---|---|---|
| ApplicationNotAccessible | E_OS_ACCESS | The application to which the schedule table belongs was terminated and has not yet restarted. |
| CoreIsDown | E_OS_CORE | The core on which the schedule table resides has been shutdown. |
| WrongContext | E_OS_CALLLEVEL | The system service was called from a context that is not permitted. |
| InterruptDisabled | E_OS_INTDISABLE | The system service was called with interrupts dis-abled. |
| InvalidScheduleId | E_OS_ID | One or both of the referenced schedule tables does not exist. |
| Permission | E_OS_ACCESS | Permission has not been granted for the caller to ac-cess one or both of the referenced schedule tables. |
| DifferentCounters | E_OS_ID | The referenced *current* and *next* schedule tables are driven by different counters. |
| NotRunning | E_OS_NOFUNC | The referenced *current* schedule table is not running. |

| **ErrorId(OS_ERROR_XX)** | **AUTOSAR error code** | **Description** |
|---|---|---|
| NotStopped | E_OS_STATE | The referenced *next* schedule table is not in the STOPPED state. |

## Name

ChainTask — Terminate the current task and activate another

## Synopsis

```
#include <Os.h>
StatusType ChainTask(TaskType t);
```

## Description

`ChainTask()` causes the termination of the calling task and activates the task specified by the *t* parameter.

The task to be activated can be the same as the calling task. In this case, the chaining does not result in the maximum number of activations being exceeded. This means that a task with only 1 activation can chain itself.

The calling task must release all resources before calling `ChainTask()`.

## Service identification

`OS_SID_ChainTask`.

## Return value

A return value of `E_OK` indicates a successful completion of the function. Any other return value indicates an error code as described below.

| **ErrorId(OS_ERROR_XX)** | **AUTOSAR error code** | **Description** |
|---|---|---|
| CoreIsDown | E_OS_CORE | The core on which the task resides has been shut-down. |
| WrongContext | E_OS_CALLLEVEL | The system service was called from a context that is not permitted. |
| InterruptDisabled | E_OS_INTDISABLE | The system service was called with interrupts dis-abled. |
| HoldsLock | E_OS_SPINLOCK | The terminating task still occupies one or more spin-locks. |
| HoldsResource | E_OS_RESOURCE | The terminating task still occupies one or more re-sources. |
| InvalidTaskId | E_OS_ID | The specified task ID is invalid. |
| Quarantined | E_OS_DENIED | The specified task has been quarantined and cannot be activated. |
| MaxActivations | E_OS_LIMIT | The specified task has exceeded its activation limit. |
| RateLimitExceeded | E_OS_RATEPROT | The specified task has exceeded its activation rate limit. |
| Permission | E_OS_ACCESS | Permission has not been granted for the caller to ac-cess the referenced task. |
| ApplicationNotAccessible | E_OS_ACCESS | The application to which the task belongs was termi-nated and has not yet restarted. |

## Name

CheckIsrMemoryAccess — Return memory access permissions for an ISR

## Synopsis

```
#include <Os.h>
AccessType CheckIsrMemoryAccess(ISRType i, void *ptr, MemorySizeType len);
```

## Description

`CheckIsrMemoryAccess()` returns information about the access rights of the ISR over the specified memory region. The return value contains a bitwise *OR* of the return values listed below to indicate that the memory region is readable, writeable, executable, and located in the stack.

If the ISR is trusted, it has read, write, and execute permission over the entire memory and the stack bit indicates that the region lies entirely within the global interrupt stack.

The macros `OSMEMORY_IS_READABLE()`, `OSMEMORY_IS_WRITEABLE()`, `OSMEMORY_IS_EXECUTABLE()` and `OSMEMORY_IS_STACKSPACE()` can be used to examine the return value.

## Service identification

`OS_SID_CheckIsrMemoryAccess`.

## Return value

Returns the information about the access rights of the ISR. In case of any errors, ErrorHook will be called (if ErrorHook is configured and *OsErrorHandling* parameter is configured as FULL) with any one of the error code described below.

| ErrorId(OS_ERROR_XX) | AUTOSAR error code | Description |
|---|---|---|
| WrongContext | E_OS_CALLLEVEL | The system service was called from a context that is not permitted. |
| InterruptDisabled | E_OS_INTDISABLE | The system service was called with interrupts disabled. |
| InvalidIsrId | E_OS_ID | The referenced ISR does not exist. |
| InvalidMemoryRegion | E_OS_VALUE | The specified memory region is invalid. It is either of zero length or it extends beyond the processor's addressing limits. |

## Name

CheckObjectAccess — Indicate whether an application has access to an object

## Synopsis

```
#include <Os.h>
ObjectAccessType CheckObjectAccess(ApplicationType a, ObjectTypeType typ, os_-
objectid_t id);
```

## Description

`CheckObjectAccess()` checks if the referenced application has access permission to the specified object. The application's permission mask is checked against the permission bits of the object.

The function returns true (OS_TRUE) if access is granted and false (OS_FALSE) if access is denied. If either the application or the object does not exist the error handler is called and the return value is false.

## Service identification

`OS_SID_CheckObjectAccess`.

## Return value

Returns TRUE if access is granted, else returns FALSE. In case of any errors, ErrorHook will be called (if ErrorHook is configured and *OsErrorHandling* parameter is configured as FULL) with any one of the error code described below.

| **ErrorId(OS_ERROR_XX)** | **AUTOSAR error code** | **Description** |
|---|---|---|
| WrongContext | E_OS_CALLLEVEL | The system service was called from a context that is not permitted. |
| InterruptDisabled | E_OS_INTDISABLE | The system service was called with interrupts disabled. |
| InvalidObjectId | E_OS_ID | The referenced object does not exist. |
| InvalidObjectType | E_OS_VALUE | The object type is invalid. |
| InvalidApplicationId | E_OS_ID | The referenced application does not exist. |

## Name

CheckObjectOwnership — Return the ID of the application that owns the object

## Synopsis

```
#include <Os.h>
StatusType CheckObjectOwnership(ObjectTypeType typ, os_objectid_t id);
```

## Description

`CheckObjectOwnership()` returns the ID of the application that owns the object specified by `typ` and `id`. Permitted object types are:

- OS_OBJ_APPLICATION
- OS_OBJ_TASK
- OS_OBJ_ISR
- OS_OBJ_RESOURCE
- OS_OBJ_COUNTER
- OS_OBJ_ALARM
- OS_OBJ_SCHEDULETABLE

If no owner application can be found, the return value is OS_NULLAPP. The error handler is called if the `typ` parameter is an unknown or unhandled object type, or if the specified object does not exist.

## Service identification

`OS_SID_CheckObjectOwnership`.

## Return value

Returns the identifier of the owning application. In case of any errors, ErrorHook will be called (if ErrorHook is configured and *OsErrorHandling* parameter is configured as FULL) with any one of the error code described below.

| **ErrorId(OS_ERROR_XX)** | **AUTOSAR error code** | **Description** |
|---|---|---|
| WrongContext | E_OS_CALLLEVEL | The system service was called from a context that is not permitted. |
| InterruptDisabled | E_OS_INTDISABLE | The system service was called with interrupts disabled. |
| InvalidObjectType | E_OS_VALUE | The specified object type is unknown. |
| InvalidObjectId | E_OS_ID | The referenced object does not exist. |

## Name

CheckTaskMemoryAccess — Return memory access permissions for a task

## Synopsis

```
#include <Os.h>
AccessType CheckTaskMemoryAccess(TaskType t, void *ptr, MemorySizeType len);
```

## Description

`CheckTaskMemoryAccess()` returns the access permissions (read/write/execute) for the referenced task for the specified memory region. In addition, the return value indicates whether the memory is in the task's stack. The stack is only considered to be accessible when the task is active.

The return value is a logical OR of the bit fields given below.

The macros `OSMEMORY_IS_READABLE()`, `OSMEMORY_IS_WRITEABLE()`, `OSMEMORY_IS_EXECUTABLE()` and `OSMEMORY_IS_STACKSPACE()` can be used to examine the return value.

## Service identification

`OS_SID_CheckTaskMemoryAccess`.

## Return value

Returns the information about the access rights of the task's. In case of any errors, ErrorHook will be called (if ErrorHook is configured and *OsErrorHandling* parameter is configured as FULL) with any one of the error codes described below.

| ErrorId(OS_ERROR_XX) | AUTOSAR error code | Description |
|---|---|---|
| WrongContext | E_OS_CALLLEVEL | The system service was called from a context that is not permitted. |
| InterruptDisabled | E_OS_INTDISABLE | The system service was called with interrupts disabled. |
| InvalidTaskId | E_OS_ID | The referenced task does not exist. |
| InvalidMemoryRegion | E_OS_VALUE | The specified memory region is invalid. It is either of zero length or it extends beyond the processor's addressing limits. |

## Name

ClearEvent — Clear one or more events

## Synopsis

```
#include <Os.h>
StatusType ClearEvent(EventMaskType e);
```

## Description

`ClearEvent()` clears all the specified events from the current task's pending events. Multiple events can be combined using the bitwise-OR ('`|`') operator.

`ClearEvent()` may only be called from a task.

## Service identification

`OS_SID_ClearEvent`.

## Return value

A return value of `E_OK` indicates a successful completion of the function. Any other return value indicates an error code as described below.

| **ErrorId(OS_ERROR_XX)** | **AUTOSAR error code** | **Description** |
| --- | --- | --- |
| WrongContext | E_OS_CALLLEVEL | The system service was called from a context that is not permitted. |
| InterruptDisabled | E_OS_INTDISABLE | The system service was called with interrupts disabled. |
| TaskNotExtended | E_OS_ACCESS | The specified task is not an extended task. Only extended tasks are permitted to wait for events. |

## Name

ClearPendingInterrupt — Clear the pending Interrupt Source

## Synopsis

```
#include <Os.h>
StatusType ClearPendingInterrupt(ISRType isrId);
```

## Description

`ClearPendingInterrupt()` clears the pending interrupt source given in the parameter `isrId`.

## Service identification

`OS_SID_ClearPendingInterrupt`.

## Return value

A return value of `E_OK` indicates a successful completion of the function. Any other return value indicates an error code as described below.

| **ErrorId(OS_ERROR_XX)** | **AUTOSAR error code** | **Description** |
|---|---|---|
| ApplicationNotAccessible | E_OS_ACCESS | The application to which the ISR belongs was terminated and has not yet restarted. |
| WrongContext | E_OS_CALLLEVEL | The system service was called from a context that is not permitted. |
| CoreIsDown | E_OS_CORE | The core on which the ISR resides has been shutdown. |
| InvalidIsrId | E_OS_ID | The referenced ISR does not exist. |
| Permission | E_OS_ACCESS | The caller has insufficient permissions to access the referenced Interrupt source. |

## Name

ControlIdle — Set idle mode for a core

## Synopsis

```
#include <Os.h>
StatusType ControlIdle(CoreIdType c, IdleModeType mode);
```

## Description

`ControlIdle()` sets the idle mode given in `mode` for the specified core.

## Service identification

`OS_SID_ControlIdle`.

## Return value

A return value of `E_OK` indicates a successful completion of the function. Any other return value indicates an error code as described below.

| ErrorId(OS_ERROR_XX) | AUTOSAR error code | Description |
|---|---|---|
| CoreIsDown | E_OS_CORE | The core whose idle mode shall be set has been shutdown. |
| WrongContext | E_OS_CALLLEVEL | The system service was called from a context that is not permitted. |
| InvalidCoreId | E_OS_ID | The core ID is invalid. |
| InvalidIdleMode | E_OS_ID | The idle mode is invalid. |

## Name

DeclareAlarm — Declare an alarm

## Synopsis

**`DeclareAlarm`**`(AlarmName);`

## Description

`DeclareAlarm()` is a macro that is used to declare the specified alarm.

The macro can be used wherever an *extern* declaration can be used. The best place is at the external level of the source file.

## Name

DeclareEvent — Declare an evemt

## Synopsis

**`DeclareEvent`**`(EventName);`

## Description

`DeclareEvent()` is a macro that is used to declare the specified Event.

The macro can be used wherever an *extern* declaration can be used. The best place is at the external level of the source file.

## Name

DeclareResource — Declare a resource

## Synopsis

**`DeclareResource`**`(ResourceName);`

## Description

`DeclareResource()` is a macro that is used to declare the specified resource.

The macro can be used wherever an *extern* declaration can be used. The best place is at the external level of the source file.

## Name

DeclareTask — Declare a task

## Synopsis

```
DeclareTask(TaskName);
```

## Description

`DeclareTask()` is a macro that is used to declare the specified task.

The macro can be used wherever an *extern* declaration can be used. The best place is at the external level of the source file.

## Name

DisableInterruptSource — Disable the Specified Interrupt Source

## Synopsis

```
#include <Os.h>
StatusType DisableInterruptSource(ISRType isrId);
```

## Description

`DisableInterruptSource()` disables the interrupt source given in the parameter `isrId`.

## Service identification

`OS_SID_DisableInterruptSource`.

## Return value

A return value of `E_OK` indicates a successful completion of the function. Any other return value indicates an error code as described below.

| ErrorId(OS_ERROR_XX) | AUTOSAR error code | Description |
|---|---|---|
| ApplicationNotAccessible | E_OS_ACCESS | The application to which the ISR belongs was terminated and has not yet restarted. |
| WrongContext | E_OS_CALLLEVEL | The system service was called from a context that is not permitted. |
| CoreIsDown | E_OS_CORE | The core on which the ISR resides has been shutdown. |
| ISRAlreadyDisabled | E_OS_NOFUNC | The interrupt source has been already disabled. |
| InvalidIsrId | E_OS_ID | The referenced ISR does not exist. |
| Permission | E_OS_ACCESS | The caller has insufficient permissions to access the referenced Interrupt source. |

## Name

EnableInterruptSource — Enable the Specified Interrupt Source

## Synopsis

```
#include <Os.h>
StatusType EnableInterruptSource(os_isrid_t isrId, ObjectAccessType flag);
```

## Description

`EnableInterruptSource()` enables the interrupt source given in the parameter `isrId` and based on the parameter `flag`, clears the pending interrupt.

## Service identification

`OS_SID_EnableInterruptSource`.

## Return value

A return value of `E_OK` indicates a successful completion of the function. Any other return value indicates an error code as described below.

| **ErrorId(OS_ERROR_XX)** | **AUTOSAR error code** | **Description** |
|---|---|---|
| ApplicationNotAccessible | E_OS_ACCESS | The application to which the ISR belongs was terminated and has not yet restarted. |
| WrongContext | E_OS_CALLLEVEL | The system service was called from a context that is not permitted. |
| CoreIsDown | E_OS_CORE | The core on which the ISR resides has been shutdown. |
| InvalidIsrId | E_OS_ID | The referenced ISR does not exist. |
| ISRAlreadyEnabled | E_OS_NOFUNC | The interrupt source has been already disabled. |
| Permission | E_OS_ACCESS | The caller has insufficient permissions to access the referenced Interrupt source. |

## Name

ErrorHook — A hook function to obtain error information.

## Synopsis

```
#include <Os.h>
void ErrorHook(StatusType Error);
```

## Description

If so configured, the kernel calls the user-supplied `ErrorHook()` function whenever an error occurs. This typically happens when a system service would return a status code other than `E_OK`, but system services that do not return a status code can also cause the `ErrorHook()` to be called. In addition, the `ErrorHook()` can be called when the kernel detects an internal error.

The `ErrorHook()` function is called in the context of the kernel with category 2 interrupts disabled.

## Return value

Returns Nothing.

## Name

ErrorHook_App — An application specific hook routine for error situations

## Synopsis

```
#include <Os.h>
void ErrorHook_App(StatusType Error);
```

## Description

When an error occurs AND an application-specific `ErrorHook()` is configured for the faulty OS application, the operating system shall call that application-specific error hook `ErrorHook_App()` after the system specific ErrorHook is called (if configured).

The `ErrorHook_App()` function is called in the context of the kernel with category 2 interrupts disabled.

## Return value

Returns Nothing.

## Name

GetActiveApplicationMode — Get the current application mode

## Synopsis

```
#include <Os.h>
AppModeType GetActiveApplicationMode(void);
```

## Description

`GetActiveApplicationMode()` returns the application mode that was given to `StartOS()` when the system started.

## Service identification

`OS_SID_GetActiveApplicationMode`.

## Return value

Returns the active application mode. In case of any errors, ErrorHook will be called (if ErrorHook is configured and *OsErrorHandling* parameter is configured as FULL) with any one of the error code described below.

| **ErrorId(OS_ERROR_XX)** | **AUTOSAR error code** | **Description** |
|---|---|---|
| WrongContext | E_OS_CALLLEVEL | The system service was called from a context that is not permitted. |
| InterruptDisabled | E_OS_INTDISABLE | The system service was called with interrupts disabled. |

## Name

GetAlarm — Get the time remaining on the alarm

## Synopsis

```
#include <Os.h>
StatusType GetAlarm(AlarmType a, TickRefType *out);
```

## Description

`GetAlarm()` calculates the time remaining before the specified alarm expires and places the result in the designated `out` variable and returns `E_OK`. If the alarm is not in use or another error is detected, the `out` variable remains unchanged.

If `GetAlarm()` is called from an ISR, it is possible that the alarm is about to expire in a lower-priority ISR. In this case `GetAlarm()` places zero in the `out` parameter and returns E_OK.

## Service identification

`OS_SID_GetAlarm`.

## Return value

A return value of `E_OK` indicates a successful completion of the function. Any other return value indicates an error code as described below.

| **ErrorId(OS_ERROR_XX)** | **AUTOSAR error code** | **Description** |
|---|---|---|
| CoreIsDown | E_OS_CORE | The core on which the alarm resides has been shut-down. |
| WrongContext | E_OS_CALLLEVEL | The system service was called from a context that is not permitted. |
| InterruptDisabled | E_OS_INTDISABLE | The system service was called with interrupts dis-abled. |
| WriteProtect | E_OS_ADDRESS | The application has attempted to write to a memory area where writing is not permitted. |
| InvalidAlarmId | E_OS_ID | The specified alarm ID is invalid. |
| Permission | E_OS_ACCESS | Permission has not been granted for the caller to ac-cess the referenced alarm. |
| InvalidAlarmState | E_OS_PANIC | The specified alarm is in an invalid state. This is an internal kernel error. Please notify your vendor. |
| AlarmNotInUse | E_OS_NOFUNC | The specified alarm is not currently in use. |

## Name

GetAlarmBase — Get alarm configuration

## Synopsis

```
#include <Os.h>
StatusType GetAlarmBase(AlarmType a, AlarmBaseType *out);
```

## Description

`GetAlarmBase()` places the configured parameters `maxallowedvalue`, `mincycle` and `ticksperbase` of the counter which is attached to the alarm into the specified `out` variable and returns `E_OK`. If an error occurs, the `out` variable remains unchanged.

## Service identification

`OS_SID_GetAlarmBase`.

## Return value

A return value of `E_OK` indicates a successful completion of the function. Any other return value indicates an error code as described below.

| **ErrorId(OS_ERROR_XX)** | **AUTOSAR error code** | **Description** |
|---|---|---|
| ApplicationNotAccessible | E_OS_ACCESS | The application to which the alarm belongs was terminated and has not yet restarted. |
| WrongContext | E_OS_CALLLEVEL | The system service was called from a context that is not permitted. |
| InterruptDisabled | E_OS_INTDISABLE | The system service was called with interrupts disabled. |
| WriteProtect | E_OS_ADDRESS | The application has attempted to write to a memory area where writing is not permitted. |
| InvalidAlarmId | E_OS_ID | The specified alarm ID is invalid. |

## Name

GetApplicationID — Get the current application ID

## Synopsis

```
#include <Os.h>
ApplicationType GetApplicationID(void);
```

## Description

`GetApplicationID()` returns the ID of the current application. If no category 2 ISR or task is running, or if the current ISR or task does not belong to an application, `OS_NULLAPP` is returned instead.

## Service identification

`OS_SID_GetApplicationId`.

## Return value

See the description above for the return value. In case of any errors, ErrorHook will be called (if ErrorHook is configured and *OsErrorHandling* parameter is configured as FULL) with any one of the error code described below.

| **ErrorId(OS_ERROR_XX)** | **AUTOSAR error code** | **Description** |
|---|---|---|
| WrongContext | E_OS_CALLLEVEL | The system service was called from a context that is not permitted. |
| InterruptDisabled | E_OS_INTDISABLE | The system service was called with interrupts disabled. |

## Name

GetApplicationState — Get state of an application

## Synopsis

```
#include <Os.h>
StatusType GetApplicationState(ApplicationType t, ApplicationStateType *out);
```

## Description

`GetApplicationState()` writes the current state of the specified application to the location specified in the `out` parameter.

## Service identification

`OS_SID_GetApplicationState`.

## Return value

Returns the state of the current application. In case of any errors, ErrorHook will be called (if ErrorHook is configured and *OsErrorHandling* parameter is configured as FULL) with any one of the error code described below.

| **ErrorId(OS_ERROR_XX)** | **AUTOSAR error code** | **Description** |
|---|---|---|
| InterruptDisabled | E_OS_INTDISABLE | The system service was called with interrupts disabled. |
| WrongContext | E_OS_CALLLEVEL | The system service was called from a context that is not permitted. |
| CoreIsDown | E_OS_CORE | The core on which the alarm task resides has been shutdown. |
| InvalidApplicationId | E_OS_ID | The referenced application does not exist. |
| WriteProtect | E_OS_ADDRESS | The application has attempted to write to a memory area where writing is not permitted. |

## Name

GetCoreID — Get the ID of the caller's core

## Synopsis

```
#include <Os.h>
CoreIdType GetCoreID(void);
```

## Description

`GetCoreID()` returns the unique number identifier of the core where the caller is executing.

## Return value

See the description above.

## Name

GetCounterValue — Get the current value of the counter

## Synopsis

```
#include <Os.h>
StatusType GetCounterValue(os_counterid_t c, TickRefType *out);
```

## Description

`GetCounterValue()` places the current value of the specified counter in the designated `out` variable.

If the counter does not exist or another error is detected, the `out` variable remains unchanged.

If this system service is called from an ISR of higher priority than the counter's own ISR, the count value might occasionally be less than expected, but this will reflect the state of the alarms in the counter's queue.

## Service identification

`OS_SID_GetCounterValue`.

## Return value

A return value of `E_OK` indicates a successful completion of the function. Any other return value indicates an error code as described below.

| ErrorId(OS_ERROR_XX) | AUTOSAR error code | Description |
|---|---|---|
| ApplicationNotAccessible | E_OS_ACCESS | The application to which the counter belongs was terminated and has not yet restarted. |
| CoreIsDown | E_OS_CORE | The core on which the counter resides has been shutdown. |
| WrongContext | E_OS_CALLLEVEL | The system service was called from a context that is not permitted. |
| InterruptDisabled | E_OS_INTDISABLE | The system service was called with interrupts disabled. |
| WriteProtect | E_OS_ADDRESS | The application has attempted to write to a memory area where writing is not permitted. |
| InvalidCounterId | E_OS_ID | The specified counter ID is invalid. |

## Name

GetCurrentApplicationID — Get the current application

## Synopsis

```
#include <Os.h>
ApplicationType GetCurrentApplicationID(void);
```

## Description

`GetCurrentApplicationID()` returns the ID of the current application. If no category 2 ISR or task is running, or if the current ISR or task does not belong to an application, `OS_NULLAPP` is returned instead. When called from the trusted function, the API returns the application ID of the caller and not the application ID to which the trusted function belongs to.

## Service identification

`OS_SID_GetCurrentApplicationId`.

## Return value

See the description above for the return value. In case of any errors, ErrorHook will be called (if ErrorHook is configured and *OsErrorHandling* parameter is configured as FULL) with any one of the error code described below.

| ErrorId(OS_ERROR_XX) | AUTOSAR error code | Description |
|---|---|---|
| WrongContext | E_OS_CALLLEVEL | The system service was called from a context that is not permitted. |
| InterruptDisabled | E_OS_INTDISABLE | The system service was called with interrupts disabled. |

## Name

GetElapsedCounterValue — Get the number of elapsed ticks

## Synopsis

```
#include <Os.h>
StatusType GetElapsedCounterValue(os_counterid_t c, TickRefType *last, Tick-
RefType *out);
```

## Description

`GetElapsedCounterValue()` places the number of ticks of the specified counter that have elapsed since the counter had the value in the designated `last` variable into the designated `out` variable. The current value of the counter is placed in the designated `last` variable.

If the counter does not exist or another error is detected, the `last` and `out` variables remain unchanged.

If this system service is called from an ISR of higher priority than the counter's own ISR, there might be expired alarms still in the queue that have not been processed.

Note: There is no way to calculate the number of elapsed ticks and get a new counter value simultaneously. This is specified by AUTOSAR.

## Service identification

`OS_SID_GetElapsedCounterValue`.

## Return value

A return value of `E_OK` indicates a successful completion of the function. Any other return value indicates an error code as described below.

| **ErrorId(OS_ERROR_XX)** | **AUTOSAR error code** | **Description** |
|---|---|---|
| ApplicationNotAccessible | E_OS_ACCESS | The application to which the counter belongs was terminated and has not yet restarted. |
| CoreIsDown | E_OS_CORE | The core on which the counter resides has been shutdown. |
| WrongContext | E_OS_CALLLEVEL | The system service was called from a context that is not permitted. |
| InterruptDisabled | E_OS_INTDISABLE | The system service was called with interrupts disabled. |
| WriteProtect | E_OS_ADDRESS | The application has attempted to write to a memory area where writing is not permitted. |
| InvalidCounterId | E_OS_ID | The specified counter ID is invalid. |
| ParameterOutOfRange | E_OS_VALUE | The PreviousValue parameter is out of range. It must not be greater than the MAXALLOWEDVALUE of the counter. |

## Name

GetEvent — Get the pending events for a task

## Synopsis

```
#include <Os.h>
StatusType GetEvent(TaskType t, EventMaskType *ep);
```

## Description

`GetEvent()` places the mask of pending events for the specified task into the location given by `ep` and returns `E_OK`. The task must be an extended task. If an error is detected, the location given by `ep` remains unchanged.

## Service identification

`OS_SID_GetEvent`.

## Return value

A return value of `E_OK` indicates a successful completion of the function. Any other return value indicates an error code as described below.

| ErrorId(OS_ERROR_XX) | AUTOSAR error code | Description |
|---|---|---|
| CoreIsDown | E_OS_CORE | The core on which the task resides has been shut-down. |
| WrongContext | E_OS_CALLLEVEL | The system service was called from a context that is not permitted. |
| InterruptDisabled | E_OS_INTDISABLE | The system service was called with interrupts dis-abled. |
| InvalidTaskId | E_OS_ID | The specified task ID is invalid. |
| TaskNotExtended | E_OS_ACCESS | The specified task is not an extended task. Only ex-tended tasks are permitted to wait for events. |
| TaskSuspended | E_OS_STATE | The specified task is currently suspended or quaran-tined. |
| WriteProtect | E_OS_ADDRESS | The application has attempted to write to a memory area where writing is not permitted. |

## Name

GetISRID — Return the ID of the current ISR

## Synopsis

```
#include <Os.h>
ISRType GetISRID(void);
```

## Description

If `GetISRID()` is called from an ISR of category category 2, or from an ErrorHook or ProtectionHook caused by an ISR of category 2, it returns the ID of the ISR. Otherwise it returns OS_NULLISR.

If the more relaxed (but not Autosar-conformant) calling context checks are configured, the ISR ID is also returned when called from a category 1 ISR or from an alarm callback function.

## Service identification

`OS_SID_GetIsrId`.

## Return value

Returns the identifier of the current ISR. In case of any errors, ErrorHook will be called (if ErrorHook is configured and *OsErrorHandling* parameter is configured as FULL) with any one of the error code described below.

| **ErrorId(OS_ERROR_XX)** | **AUTOSAR error code** | **Description** |
|---|---|---|
| WrongContext | E_OS_CALLLEVEL | The system service was called from a context that is not permitted. |
| InterruptDisabled | E_OS_INTDISABLE | The system service was called with interrupts disabled. |

## Name

GetResource — Enter a critical section by acquiring a resource

## Synopsis

```
#include <Os.h>
StatusType GetResource(ResourceType r);
```

## Description

`GetResource()` allows the calling task to enter a critical section of code associated with the specified resource `r`. Other tasks that use the same resource must wait until this task releases the resource again. The resource is released when the acquiring task calls `ReleaseResource()`.

Resources that are associated with ISRs will also cause the associated ISR to be blocked. This may result in other ISRs being blocked too. The exact behavior is architecture-dependent.

A task may not call `GetResource()` for a resource that it already holds.

When multiple resources are acquired they must be released in reverse order.

A task that occupies a resource must not call `TerminateTask()`, `ChainTask()` or `WaitEvent()`.

`GetResource()` may be used in tasks. On some architectures `GetResource()` can be called from Category 2 ISRs as well.

## Service identification

`OS_SID_GetResource`.

## Return value

A return value of `E_OK` indicates a successful completion of the function. Any other return value indicates an error code as described below.

| ErrorId(OS_ERROR_XX) | AUTOSAR error code | Description |
|---|---|---|
| WrongContext | E_OS_CALLLEVEL | The system service was called from a context that is not permitted. |
| InterruptDisabled | E_OS_INTDISABLE | The system service was called with interrupts disabled. |
| InvalidResourceId | E_OS_ID | The specified resource ID is invalid. |
| ResourceInUse | E_OS_ACCESS | The specified resource is in use. |
| ResourcePriorityError | E_OS_ACCESS | The specified resource has a lower ceiling priority than the base priority of the calling task. The probable cause is that the task does not declare the resource. |
| Permission | E_OS_ACCESS | Permission has not been granted for the caller to access the referenced resource. |

## Name

GetScheduleTableStatus — Get a schedule table's status

## Synopsis

```
#include <Os.h>
StatusType GetScheduleTableStatus(ScheduleTableType s, os_uint8_t *out);
```

## Description

`GetScheduleTableStatus()` writes the current status of the schedule table to the specified location.

## Service identification

`OS_SID_GetScheduleTableStatus`.

## Return value

A return value of `E_OK` indicates a successful completion of the function. Any other return value indicates an error code as described below.

| ErrorId(OS_ERROR_XX) | AUTOSAR error code | Description |
|---|---|---|
| ApplicationNotAccessible | E_OS_ACCESS | The application to which the schedule table belongs was terminated and has not yet restarted. |
| CoreIsDown | E_OS_CORE | The core on which the schedule table resides has been shutdown. |
| WrongContext | E_OS_CALLLEVEL | The system service was called from a context that is not permitted. |
| InterruptDisabled | E_OS_INTDISABLE | The system service was called with interrupts disabled. |
| InvalidScheduleId | E_OS_ID | The referenced schedule table does not exist. |
| WriteProtect | E_OS_ADDRESS | The application has attempted to write to a memory area where writing is not permitted. |

## Name

GetTaskID — Get the ID of the current task

## Synopsis

```
#include <Os.h>
StatusType GetTaskID(TaskRefType out);
```

## Description

`GetTaskID()` writes the ID of the current task to the user-specified location given by the `out` parameter and returns `E_OK`. If no task is currently running, `OS_NULLTASK` is written to `out` instead.

## Service identification

`OS_SID_GetTaskId`.

## Return value

A return value of `E_OK` indicates a successful completion of the function. Any other return value indicates an error code as described below.

| **ErrorId(OS_ERROR_XX)** | **AUTOSAR error code** | **Description** |
|---|---|---|
| WrongContext | E_OS_CALLLEVEL | The system service was called from a context that is not permitted. |
| InterruptDisabled | E_OS_INTDISABLE | The system service was called with interrupts disabled. |
| WriteProtect | E_OS_ADDRESS | The application has attempted to write to a memory area where writing is not permitted. |

## Name

GetTaskState — Get the state of a task

## Synopsis

```
#include <Os.h>
StatusType GetTaskState(TaskType t, TaskStateType *out);
```

## Description

`GetTaskState()` writes the current state of the specified task to the location given in the `out` parameter and returns `E_OK`.
If an error code is returned, the referenced location given by `out` is not overwritten.

## Service identification

`OS_SID_GetTaskState`.

## Return value

A return value of `E_OK` indicates a successful completion of the function. Any other return value indicates an error code as described below.

| **ErrorId(OS_ERROR_XX)** | **AUTOSAR error code** | **Description** |
|---|---|---|
| ApplicationNotAccessible | E_OS_ACCESS | The application to which the task belongs was terminated and has not yet restarted. |
| CoreIsDown | E_OS_CORE | The core on which the task resides has been shutdown. |
| WrongContext | E_OS_CALLLEVEL | The system service was called from a context that is not permitted. |
| InterruptDisabled | E_OS_INTDISABLE | The system service was called with interrupts disabled. |
| WriteProtect | E_OS_ADDRESS | The application has attempted to write to a memory area where writing is not permitted. |
| InvalidTaskId | E_OS_ID | The specified task ID is invalid. |

## Name

IAdvanceCounter — Increment the given counter at interrupt level.

## Synopsis

```
#include <Os.h>
StatusType IAdvanceCounter(CounterIDType CounterName);
```

## Description

`IAdvanceCounter()` increments the given counter by 1. Any alarm that expires as a result of this will cause the appropriate alarm action to take place. If the action is an alarm callback, the callback function runs in the context of the caller of `IAdvanceCounter()`.

This service is called only from interrupt level and not from task level. For incrementing counters within a task, see `AdvanceCounter()`.

In AUTOSAR Os, `AdvanceCounter()` and `IAdvanceCounter()` are identical, but failure to observe the above distinction may result in non-portable code.

## Return value

A return value of `E_OK` indicates a successful completion of the function.

A return value of `E_OS_ID` indicates that the alarm ID is wrong.

## Name

IncrementCounter — Increment a counter

## Synopsis

```
#include <Os.h>
StatusType IncrementCounter(os_counterid_t c);
```

## Description

`IncrementCounter()` increments a counter. If any alarm attached to the counter expires as a result, the configured action for that alarm is performed. The alarm action always runs in the context of the kernel.

## Service identification

`OS_SID_IncrementCounter`.

## Return value

A return value of `E_OK` indicates a successful completion of the function. Any other return value indicates an error code as described below.

| **ErrorId(OS_ERROR_XX)** | **AUTOSAR error code** | **Description** |
|---|---|---|
| ApplicationNotAccessible | E_OS_ACCESS | The application to which the counter belongs was terminated and has not yet restarted. |
| CoreIsDown | E_OS_CORE | The core on which the counter resides has been shutdown. |
| WrongContext | E_OS_CALLLEVEL | The system service was called from a context that is not permitted. |
| InterruptDisabled | E_OS_INTDISABLE | The system service was called with interrupts disabled. |
| InvalidCounterId | E_OS_ID | The specified counter ID is invalid. |
| Permission | E_OS_ACCESS | Permission has not been granted for the caller to access the referenced counter. |
| CounterIsHw | E_OS_ID | The referenced counter is a hardware counter and cannot be advanced by software. |

## Name

ISR — Define a category 2 ISR function

## Synopsis

**ISR**(isrname);

## Description

The `ISR()` macro defines a function to implement the body of the category 2 ISR whose name is given in the `isrname` parameter. The code you wish to execute when the ISR runs is placed in the body of the function.

The `ISR()` macro can only be used at the outer level of a C source file.

## Return value

Returns Nothing.

## Code example

```
ISR(isrname)
{
  /* handling of interrupt */
  /* clear the interrupt flags */
  return;
}
```

## Name

ISR1 — Define a category 1 ISR function

## Synopsis

```
#include <Os.h>
ISR1(isrname);
```

## Description

The `ISR1()` macro defines a function to implement the body of the category 1 ISR whose name is given in the `isrname` parameter. The code you wish to execute when the ISR runs is placed in the body of the function.

The `ISR1()` macro can only be used at the outer level of a C source file.

## Return value

Returns nothing.

## Code example

```
ISR1(isrname)
{
  /* handling of interrupt */
  /* clear the interrupt flags */
  return;
}
```

## Name

NextScheduleTable — Start a schedule table at the end of another

## Synopsis

```
#include <Os.h>
StatusType NextScheduleTable(ScheduleTableType ScheduleTableID_From, ScheduleTableType ScheduleTableID_To);
```

## Description

`NextScheduleTable()` chains `ScheduleTableID_To` to `ScheduleTableID_From` so that when `ScheduleTableID_From` comes to the end of its list of actions, `ScheduleTableID_To` will replace it. The timing is arranged so that the first action point of `ScheduleTableID_To` occurs at its specified offset from the full end of `ScheduleTableID_From`'s period.

See the notes given for `ChainScheduleTable()` to understand the limitations of this system service.

## Service identification

`OS_SID_NextScheduleTable`.

## Return value

A return value of `E_OK` indicates a successful completion of the function. Any other return value indicates an error code as described below.

| **ErrorId(OS_ERROR_XX)** | **AUTOSAR error code** | **Description** |
|---|---|---|
| ApplicationNotAccessible | E_OS_ACCESS | The application to which the schedule table belongs was terminated and has not yet restarted. |
| CoreIsDown | E_OS_CORE | The core on which the schedule table resides has been shutdown. |
| WrongContext | E_OS_CALLLEVEL | The system service was called from a context that is not permitted. |
| InterruptDisabled | E_OS_INTDISABLE | The system service was called with interrupts disabled. |
| InvalidScheduleId | E_OS_ID | One or both of the referenced schedule tables does not exist. |
| Permission | E_OS_ACCESS | Permission has not been granted for the caller to access one or both of the referenced schedule tables. |
| DifferentCounters | E_OS_ID | The referenced *current* and *next* schedule tables are driven by different counters. |
| NotRunning | E_OS_NOFUNC | The referenced *current* schedule table is not running. |
| NotStopped | E_OS_STATE | The referenced *next* schedule table is not in the STOPPED state. |

## Name

OSMEMORY_IS_EXECUTABLE — Test access rights for execute permission.

## Synopsis

```
#include <Os.h>
int OSMEMORY_IS_EXECUTABLE(AccessType a);
```

## Description

`OSMEMORY_IS_EXECUTABLE()` returns TRUE if the parameter indicates that the execute permission is granted.

## Return value

Returns TRUE if the permission is granted else FALSE.

## Name

OSMEMORY_IS_READABLE — Test access rights for read permission.

## Synopsis

```
#include <Os.h>
int OSMEMORY_IS_READABLE(AccessType a);
```

## Description

`OSMEMORY_IS_READABLE()` returns TRUE if the parameter indicates that the read permission is granted.

## Return value

Returns TRUE if the permission is granted else FALSE.

## Name

OSMEMORY_IS_STACKSPACE — Test access rights for stack space indication.

## Synopsis

```
#include <Os.h>
int OSMEMORY_IS_STACKSPACE(AccessType a);
```

## Description

`OSMEMORY_IS_STACKSPACE()` returns TRUE if the parameter indicates that the memory is in the stack space.

## Return value

Returns TRUE if the permission is granted else FALSE.

## Name

OSMEMORY_IS_WRITEABLE — Test access rights for write permission.

## Synopsis

```
#include <Os.h>
int OSMEMORY_IS_WRITEABLE(AccessType a);
```

## Description

`OSMEMORY_IS_WRITEABLE()` returns TRUE if the parameter indicates that the write permission is granted.

## Return value

Returns TRUE if the permission is granted else FALSE.

## Name

PostISRHook — A hook routine for notifying ISR termination.

## Synopsis

```
#include <Os.h>
void PostISRHook(ISRType isrid);
```

## Description

If so configured, the kernel calls the user-supplied `PostISRHook()` function whenever a category 2 ISR has finished its execution. The ID of the executed ISR is given as parameter `isrid`.

The `PostISRHook()` function is called in the context of the kernel with category 2 interrupts disabled.

## Return value

Returns nothing.

## Name

PostTaskHook — A hook routine for notifying task termination.

## Synopsis

```
#include <Os.h>
void PostTaskHook(void);
```

## Description

If so configured, the kernel calls the user-supplied `PostTaskHook()` function whenever a task leaves the `RUNNING` state. This happens when `TerminateTask()` or `ChainTask()` is called, when `WaitEvent()` is called and results in a transfer to the waiting state or when a task is preempted by a higher-priority task.

When called from the `PostTaskHook()` function, `GetTaskID()` returns the ID of the outgoing task and `GetTaskState()` for the outgoing task returns `RUNNING`.

The `PostTaskHook()` function is called in the context of the kernel with category 2 interrupts disabled.

## Return value

Returns nothing.

## Name

PreISRHook — A hook routine for notifying ISR start

## Synopsis

```
#include <Os.h>
void PreISRHook(ISRType isrid);
```

## Description

If so configured, the kernel calls the user-supplied `PreISRHook()` function whenever a category 2 ISR starts being executed. The ID of the started ISR is given as parameter `isrid`.

The `PreISRHook()` function is called in the context of the kernel with category 2 interrupts disabled.

## Return value

Returns nothing.

## Name

PreTaskHook — A hook routine for notifying task start

## Synopsis

```
#include <Os.h>
void PreTaskHook(void);
```

## Description

If so configured, the kernel calls the user-supplied `PreTaskHook()` function whenever a task enters the `RUNNING` state. This happens when the task first starts, when it returns from `WaitEvent()` after having been in the `WAITING` state and when it regains the CPU after having been pre-empted.

When called from the `PreTaskHook()` function, `GetTaskID()` returns the ID of the incoming task and `GetTaskState()` for the incoming task returns `RUNNING`.

The `PreTaskHook()` function is called in the context of the kernel with category 2 interrupts disabled.

## Return value

Returns nothing.

## Name

ProtectionHook — A hook routine for serious error situations

## Synopsis

```
#include <Os.h>
ProtectionReturnType ProtectionHook(StatusType Fatalerror);
```

## Description

The protection hook is always called if a serious error occurs. E.g. exceeding the worst case execution time or violating against the memory protection. Depending on the return value the OS will either kill the task/category 2 ISR which causes the problem, kill the OS application to which the task/category 2 ISR belongs (optional with restart) or shutdown the system.

The `ProtectionHook()` function is called in the context of the kernel with category 2 interrupts disabled.

## Return value

Returns PRO_KILLTASKISR, PRO_KILLAPPL, PRO_KILLAPPL_RESTART or PRO_SHUTDOWN based on the ACTION return value for the respective protection error.

PRO_KILLTASKISR: Kills the task or category 2 ISR which causes the problem.

PRO_KILLAPPL: Kills the application (all application belonging objects).

PRO_KILLAPPL_RESTART: Kills the application which causes the problem and restarts it (using the restart task).

PRO_SHUTDOWN: Shutdown the OS.

## Name

ReleaseResource — Leave a critical section by releasing a resource

## Synopsis

```
#include <Os.h>
StatusType ReleaseResource(ResourceType r);
```

## Description

`ReleaseResource()` signals that the calling task has left a critical section of code associated with the specified resource `r`. Other tasks that use the same resource are now permitted to run.

A task must release resources in the reverse order to which they were taken.

Each call to `GetResource()` must be matched by a correctly-nested call to `ReleaseResource()`.

`ReleaseResource()` may be used in tasks. On some architectures `ReleaseResource()` can be called from Category 2 ISRs as well.

## Service identification

`OS_SID_ReleaseResource`.

## Return value

A return value of `E_OK` indicates a successful completion of the function. Any other return value indicates an error code as described below.

| ErrorId(OS_ERROR_XX) | AUTOSAR error code | Description |
|---|---|---|
| InterruptDisabled | E_OS_INTDISABLE | The system service was called with interrupts disabled. |
| InvalidResourceId | E_OS_ID | The specified resource ID is invalid. |
| ResourceNestingError | E_OS_NOFUNC | The specified resource has not been taken by the task, or another resource needs to be released first. Resources must be released in the reverse order to which they were taken. |
| WrongContext | E_OS_CALLLEVEL | The system service was called from a context that is not permitted. |

## Name

ReleaseSpinlock — Releases an already occupied spinlock variable

## Synopsis

```
#include <Os.h>
StatusType ReleaseSpinlock(SpinlockIdType lockId);
```

## Description

`ReleaseSpinlock()` releases the spinlock given by `lockId`. A task or an ISR must release all the spinlocks which were acquired either using `GetSpinLock()` or `TryGetSpinlock()`, before terminating a task or returning from an ISR.

## Service identification

`OS_SID_ReleaseSpinlock`.

## Return value

A return value of `E_OK` indicates a successful completion of the function. Any other return value indicates an error code as described below.

| ErrorId(OS_ERROR_XX) | AUTOSAR error code | Description |
|---|---|---|
| WrongContext | E_OS_CALLLEVEL | The system service was called from a context that is not permitted. |
| InvalidSpinlockId | E_OS_ID | The specified spinlock ID is invalid. |
| Permission | E_OS_ACCESS | Permission has not been granted for the caller to access the referenced spinlock. |
| SpinlockNotOccupied | E_OS_STATE | An attempt has been made to release a spinlock that is not held by the caller. |
| InvalidSpinlockNesting | E_OS_NOFUNC | An attempt has been made to release a spinlock that is not the most recent spinlock acquired by the caller. |
| HoldsResource | E_OS_RESOURCE | An attempt has been made to release a spinlock while a resource is held by the caller that has been acquired after the spinlock. Spinlocks and resources can only be acquired and released in strict *Last In, First Out* order. |

## Name

ResumeInterrupts — Resume interrupts up to a given level

## Synopsis

```
#include <Os.h>
void ResumeInterrupts(os_intlocktype_t locktype);
```

## Description

`ResumeInterrupts()` restores the interrupt level of the processor or interrupt controller to the level it had before the corresponding call to `SuspendInterrupts()`. It is used to implement the `ResumeOSInterrupts()`, `ResumeAllInterrupts()` and `DisableAllInterrupts()` system services by calling it with the `locktype` parameter equal to `OS_LOCKTYPE_OS`, `OS_LOCKTYPE_ALL` and `OS_LOCKTYPE_NONEST`, respectively.

`EnableAllInterrupts()` restores the interrupt locking to the state that it was in before the most recent call to `DisableAllInterrupts()`. `DisableAllInterrupts()` must have been called previously in the execution thread.

`ResumeAllInterrupts()` restores the interrupt lock status to the state it was in before the corresponding `SuspendAllInterrupts()` service was called.

`ResumeOSInterrupts()` restores the interrupt lock status to the state it was in before the corresponding `SuspendOSInterrupts()` service was called. The interrupt level is only truly manipulated on the outermost of the nested calls.

If `ResumeOSInterrupts()` is called from a permitted context other than a task or category 2 ISR it is a no-operation, or if it is called within a code section that is controlled by `ResumeAllInterrupts()` or `DisableAllInterrupts()`, it is treated as a no-operation since interrupts are already blocked at a higher level.

Interrupt lock timing is implemented for tasks and ISRs; timing state that was saved by the corresponding `SuspendInterrupts()` is restored.

## Service identification

`OS_SID_ResumeInterrupts`.

## Return value

Returns nothing. In case of any errors, ErrorHook will be called (if ErrorHook is configured and "OsErrorHandling" parameter is configured as FULL) with any one of the error code described below.

| **ErrorId(OS_ERROR_XX)** | **AUTOSAR error code** | **Description** |
|---|---|---|
| WrongContext | E_OS_CALLLEVEL | The system service was called from a context that is not permitted. |
| NestingUnderflow | E_OS_NOFUNC | The calls to SuspendOSInterrupts/ResumeOSInterrupts are not correctly nested. |

## Name

Schedule — Voluntarily yield the CPU

## Synopsis

```
#include <Os.h>
StatusType Schedule(void);
```

## Description

`Schedule()` allows the calling task to yield the CPU voluntarily. Active tasks whose running priorities are lower than the running priority of the current task but higher than its configured priority are allowed to run. `Schedule()` returns when there are no more such tasks.

Tasks get a higher running priority than their base priority when they are preemptive or have an internal resource allocated to them.

A task that holds a standard resource is not permitted to call `Schedule()` since this would interfere with the resource's ceiling priority.

## Service identification

`OS_SID_Schedule`.

## Return value

A return value of `E_OK` indicates a successful completion of the function. Any other return value indicates an error code as described below.

| ErrorId(OS_ERROR_XX) | AUTOSAR error code | Description |
|---|---|---|
| WrongContext | E_OS_CALLLEVEL | The system service was called from a context that is not permitted. |
| InterruptDisabled | E_OS_INTDISABLE | The system service was called with interrupts disabled. |
| HoldsLock | E_OS_SPINLOCK | The calling task still occupies one or more spinlocks. |
| HoldsResource | E_OS_RESOURCE | The calling task still occupies one or more resources. |

## Name

SetAbsAlarm — Set an alarm at an absolute counter value

## Synopsis

```
#include <Os.h>
StatusType SetAbsAlarm(AlarmType a, TickRefType start, TickRefType cyc);
```

## Description

`SetAbsAlarm()` sets the specified alarm to expire the next time that its counter reaches the `start` value. When the counter reaches that value, the action associated with the alarm (activate a task, set an event etc) will take place.

If the `cyc` parameter is non-zero, the alarm will be reset on expiry to occur again after a further `cyc` ticks of the counter have occurred. This will be repeated indefinitely unless `CancelAlarm()` is called.

The values of `start` and `cyc` must lie within the permitted range configured for the counter.

The specified alarm must not already be in use.

If the counter is about to reach the `start` value, the alarm could expire before `SetAbsAlarm()` returns.

If the counter has already reached the specified `start` value, the alarm will not expire until the counter wraps around and reaches the value again. Depending on the configuration of the counter, this could be a very long time.

## Service identification

`OS_SID_SetAbsAlarm`.

## Return value

A return value of `E_OK` indicates a successful completion of the function. Any other return value indicates an error code as described below.

| ErrorId(OS_ERROR_XX) | AUTOSAR error code | Description |
|---|---|---|
| ApplicationNotAccessible | E_OS_ACCESS | The application to which the alarm belongs was terminated and has not yet restarted. |
| CoreIsDown | E_OS_CORE | The core on which the alarm resides has been shutdown. |
| WrongContext | E_OS_CALLLEVEL | The system service was called from a context that is not permitted. |
| InterruptDisabled | E_OS_INTDISABLE | The system service was called with interrupts disabled. |
| InvalidAlarmId | E_OS_ID | The specified alarm ID is invalid. |
| ParameterOutOfRange | E_OS_VALUE | One or both of the specified increment and cycle parameters is out of range. |
| Quarantined | E_OS_DENIED | The specified alarm has been quarantined and will not be activated. |

| ErrorId(OS_ERROR_XX) | AUTOSAR error code | Description |
|---|---|---|
| AlarmInUse | E_OS_STATE | The specified alarm is already in use. |
| Permission | E_OS_ACCESS | Permission has not been granted for the caller to access the referenced alarm. |

## Name

SetEvent — Set one or more events for a task

## Synopsis

```
#include <Os.h>
StatusType SetEvent(TaskType t, EventMaskType evt);
```

## Description

`SetEvent()` sets the events given in `evt` for the specified task given in `t`. If the task is in the `WAITING` state for one or more events, it will be reawakened i.e., `READY` state and queued for execution.

The task must be an extended task.

Multiple events can be combined by using the bitwise-OR ('`|`') operator.

## Service identification

`OS_SID_SetEvent`.

## Return value

A return value of `E_OK` indicates a successful completion of the function. Any other return value indicates an error code as described below.

| **ErrorId(OS_ERROR_XX)** | **AUTOSAR error code** | **Description** |
|---|---|---|
| ApplicationNotAccessible | E_OS_ACCESS | The application to which the event belongs was terminated and has not yet restarted. |
| CoreIsDown | E_OS_CORE | The core on which the task resides has been shutdown. |
| WrongContext | E_OS_CALLLEVEL | The system service was called from a context that is not permitted. |
| InterruptDisabled | E_OS_INTDISABLE | The system service was called with interrupts disabled. |
| InvalidTaskId | E_OS_ID | The specified task ID is invalid. |
| TaskSuspended | E_OS_STATE | The specified task is currently suspended or quarantined. |
| TaskNotExtended | E_OS_ACCESS | The specified task is not an extended task. Only extended tasks are permitted to wait for events. |
| Permission | E_OS_ACCESS | Permission has not been granted for the caller to access the referenced task. |
| RateLimitExceeded | E_OS_RATEPROT | The specified task has exceeded its activation rate limit. |

## Name

SetEventAsyn — Set one or more events for a task asynchronously

## Synopsis

```
#include <Os.h>
void SetEventAsyn(TaskType t, EventMaskType evt);
```

## Description

`SetEventAsyn()` sets the events given in `evt` for the specified task given in `t`. If the task is waiting for one or more of the events, it will be reawakened and queued for execution. The call of `SetEventAsyn()` does not return any errors if the calling core and remote core are different. Error hook (if configured) will be called on the remote core in case of the errors.
The task must be an extended task.

## Service identification

`OS_SID_SetEventAsyn`.

## Return value

Returns nothing. In case of any errors, ErrorHook will be called (if ErrorHook is configured and "OsErrorHandling" parameter is configured as FULL) with any one of the error code described below.

| **ErrorId(OS_ERROR_XX)** | **AUTOSAR error code** | **Description** |
|---|---|---|
| ApplicationNotAccessible | E_OS_ACCESS | The application to which the event belongs was terminated and has not yet restarted. |
| WrongContext | E_OS_CALLLEVEL | The system service was called from a context that is not permitted. |
| InterruptDisabled | E_OS_INTDISABLE | The system service was called with interrupts disabled. |
| InvalidTaskId | E_OS_ID | The specified task ID is invalid. |
| TaskSuspended | E_OS_STATE | The specified task is currently suspended or quarantined. |
| TaskNotExtended | E_OS_ACCESS | The specified task is not an extended task. Only extended tasks are permitted to wait for events. |
| Permission | E_OS_ACCESS | Permission has not been granted for the caller to access the referenced task. |
| RateLimitExceeded | E_OS_RATEPROT | The specified task has exceeded its activation rate limit. |

## Name

SetRelAlarm — Set an alarm at a relative counter value

## Synopsis

```
#include <Os.h>
StatusType SetRelAlarm(AlarmType a, TickRefType inc, TickRefType cyc);
```

## Description

`SetRelAlarm()` sets the specified alarm to expire after `inc` ticks of its associated counter. When the counter reaches that value, the action associated with the alarm (activate a task, set an event etc) will take place.

If the `cyc` parameter is non-zero, the alarm will be reset on expiry to occur again after a further `cyc` ticks of the counter have occurred. This will be repeated indefinitely unless `CancelAlarm()` is called.

The values of `inc` and `cyc` must lie within the permitted range configured for the counter.

The specified alarm must not already be in use.

If the `inc` value is very small, the alarm could expire before `SetRelAlarm()` returns.

## Service identification

`OS_SID_SetRelAlarm`.

## Return value

A return value of `E_OK` indicates a successful completion of the function. Any other return value indicates an error code as described below.

| ErrorId(OS_ERROR_XX) | AUTOSAR error code | Description |
|---|---|---|
| ApplicationNotAccessible | E_OS_ACCESS | The application to which the alarm belongs was terminated and has not yet restarted. |
| CoreIsDown | E_OS_CORE | The core on which the alarm resides has been shutdown. |
| WrongContext | E_OS_CALLLEVEL | The system service was called from a context that is not permitted. |
| InterruptDisabled | E_OS_INTDISABLE | The system service was called with interrupts disabled. |
| InvalidAlarmId | E_OS_ID | The specified alarm ID is invalid. |
| IncrementZero | E_OS_VALUE | The value of the increment parameter is zero. This is not permitted by AUTOSAR. |
| ParameterOutOfRange | E_OS_VALUE | One or both of the specified increment and cycle parameters is out of range. |
| Quarantined | E_OS_DENIED | The specified alarm has been quarantined and will not be activated. |
| AlarmInUse | E_OS_STATE | The specified alarm is already in use. |

| ErrorId(OS_ERROR_XX) | AUTOSAR error code | Description |
|---|---|---|
| Permission | E_OS_ACCESS | Permission has not been granted for the caller to access the referenced alarm. |

## Name

SetScheduleTableAsync — Sets a schedule table's state to *asynchronous*

## Synopsis

```
#include <Os.h>
StatusType SetScheduleTableAsync(void);
```

## Description

`SetScheduleTableAsync()` sets a schedule table to the *asynchronous* state. The schedule table will remain asynchronous indefinitely and will continue to run governed only by local time. Any remaining synchronization steps from a previous invocation of `SyncScheduleTable()` will be dropped. A subsequent call to `SyncScheduleTable()` can resynchronize the schedule table.

`SetScheduleTableAsync()` is intended to inform the kernel that contact with the global time provider has been lost.

## Service identification

`OS_SID_SetScheduleTableAsync`.

## Return value

A return value of `E_OK` indicates a successful completion of the function. Any other return value indicates an error code as described below.

| **ErrorId(OS_ERROR_XX)** | **AUTOSAR error code** | **Description** |
|---|---|---|
| ApplicationNotAccessible | E_OS_ACCESS | The application to which the schedule table belongs was terminated and has not yet restarted. |
| CoreIsDown | E_OS_CORE | The core on which the schedule table resides has been shutdown. |
| WrongContext | E_OS_CALLLEVEL | The system service was called from a context that is not permitted. |
| InterruptDisabled | E_OS_INTDISABLE | The system service was called with interrupts disabled. |
| InvalidScheduleId | E_OS_ID | The referenced schedule table does not exist. |
| NotRunning | E_OS_STATE | The referenced current schedule table is not running. |
| Permission | E_OS_ACCESS | Permission has not been granted for the caller to access the referenced schedule table. |
| NotSyncable | E_OS_ID | The schedule table cannot be explicitly synchronised. |

## Name

ShutdownHook — A hook routine for notifying system shut-down

## Synopsis

```
#include <Os.h>
void ShutdownHook(StatusType Error);
```

## Description

If configured, the kernel calls the user-supplied `ShutdownHook()` function when the system shuts down.

The `ShutdownHook()` function is called in the context of the kernel with all interrupts disabled.

## Return value

Returns nothing.

## Name

ShutdownHook_App — An application specific hook for the shutdown

## Synopsis

```
#include <Os.h>
void ShutdownHook_App(StatusType Error);
```

## Description

This application-specific hook is called by the kernel with the access rights of the associated OS application on shutdown of the OS and before the system-specific `ShutdownHook()`.

The `ShutdownHook_App()` function is called in the context of the kernel with all interrupts disabled.

## Return value

Returns nothing.

## Name

ShutdownOS — Shutdown the OS kernel

## Synopsis

```
#include <Os.h>
void ShutdownOS(StatusType code);
```

## Description

`ShutdownOS()` shuts down the OS kernel. Interrupts are disabled and the scheduler is stopped. If the shutdown hook is configured, it is called with the `code` as the parameter.
If and when the shutdown hook returns, the kernel waits until the CPU is powered down or reset.

## Service identification

`OS_SID_ShutdownOs`.

## Return value

Returns nothing. In case of any errors, ErrorHook will be called (if ErrorHook is configured and "OsErrorHandling" parameter is configured as FULL) with any one of the error code described below.

| ErrorId(OS_ERROR_XX) | AUTOSAR error code | Description |
|---|---|---|
| WrongContext | E_OS_CALLLEVEL | The system service was called from a context that is not permitted. |
| InterruptDisabled | E_OS_INTDISABLE | The system service was called with interrupts disabled. |
| NotTrusted | E_OS_NOFUNC | `ShutdownOS()` is not permitted from a non-trusted application. |

## Name

StartCore — Start the given core

## Synopsis

```
#include <Os.h>
void StartCore(CoreIdType core, StatusRefType status);
```

## Description

`StartCore()` starts the given `core` and returns the result via the `status`. Valid calls can only be executed before `StartOS()` on the caller's core. E_OK is returned if the core was successfully started, E_OS_ID indicates a wrong core ID, E_OS_STATE indicates that the core was already activated. E_OS_ACCESS is returned if the service is called after `StartOS()`. The errors are returned via the `status` parameter.

## Return value

Returns Nothing.

## Name

StartOS — Start the OS

## Synopsis

```
#include <Os.h>
void StartOS(AppModeType mode);
```

## Description

`StartOS()` starts the OS. The `mode` parameter determines the set of tasks and alarms that should be started automatically.

After the kernel data structures have been initialized, the start-up hook is called, if it has been configured.

`StartOS()` never returns. If the OS has already been started or the `mode` parameter is invalid or `StartOS()` is called with interrupts disabled, the function could call `ErrorHook`, depending on how the error handler is defined to handle the error. If any problem is detected during the call, a panic is triggered resulting in a shutdown of the OS.

`StartOS()` can only be called once, from outside the OS. It is typically called from the system's `main()` function.

## Service identification

`OS_SID_StartOs`.

## Return value

Returns nothing. In case of any errors, ErrorHook will be called (if ErrorHook is configured and *OsErrorHandling* parameter is configured as FULL) with any one of the error code described below.

| **ErrorId(OS_ERROR_XX)** | **AUTOSAR error code** | **Description** |
|---|---|---|
| WrongContext | E_OS_CALLLEVEL | The system service was called from a context that is not permitted. This probably means that the OS has already been started. |
| InterruptDisabled | E_OS_INTDISABLE | The system service was called with interrupts disabled. |
| InvalidStartMode | E_OS_ID | The specified start-up (application) mode is invalid. |

## Name

StartScheduleTableAbs — Start a schedule table with absolute offset value.

## Synopsis

```
#include <Os.h>
StatusType StartScheduleTableAbs(ScheduleTableType s, TickRefType offset, ObjectAccessType rel);
```

## Description

`StartScheduleTableAbs()` starts a schedule table such that the first expiry point occurs when the underlying counter reaches the absolute `offset`value.

## Service identification

`OS_SID_StartScheduleTableAbs`.

## Return value

A return value of `E_OK` indicates a successful completion of the function. Any other return value indicates an error code as described below.

| **ErrorId(OS_ERROR_XX)** | **AUTOSAR error code** | **Description** |
|---|---|---|
| ApplicationNotAccessible | E_OS_ACCESS | The application to which the schedule table belongs was terminated and has not yet restarted. |
| CoreIsDown | E_OS_CORE | The core on which the task resides has been shutdown. |
| WrongContext | E_OS_CALLLEVEL | The system service was called from a context that is not permitted. |
| InterruptDisabled | E_OS_INTDISABLE | The system service was called with interrupts disabled. |
| ScheduleTableNotIdle | E_OS_STATE | The schedule table is already started. |
| AlarmInUse | E_OS_STATE | The schedule table's alarm is already in use. This indicates an internal error. Please notify your vendor. |
| InvalidScheduleId | E_OS_ID | The referenced schedule tables does not exist. |
| ParameterOutOfRange | E_OS_VALUE | The specified offset parameter is out of range. It is more than the MAXALLOWEDVALUE of the underlying counter. |
| Permission | E_OS_ACCESS | Permission has not been granted for the caller to access the referenced schedule table. |
| ImplicitSyncStartRel | E_OS_ID | A schedule table configured with IMPLICIT synchronization strategy cannot be started at a relative counter value. StartScheduleTableAbs() must be used! |

## Name

StartScheduleTableRel — Start a schedule table with relative offset value.

## Synopsis

```
#include <Os.h>
StatusType StartScheduleTableRel(ScheduleTableType s, TickRefType offset, ObjectAccessType rel);
```

## Description

`StartScheduleTableRel()` starts a schedule table such that the first expiry point occurs when the underlying counter reaches the `offset` ticks from current ticks.

## Service identification

`OS_SID_StartScheduleTableRel`.

## Return value

A return value of `E_OK` indicates a successful completion of the function. Any other return value indicates an error code as described below.

| **ErrorId(OS_ERROR_XX)** | **AUTOSAR error code** | **Description** |
|---|---|---|
| ApplicationNotAccessible | E_OS_ACCESS | The application to which the schedule table belongs was terminated and has not yet restarted. |
| CoreIsDown | E_OS_CORE | The core on which the task resides has been shutdown. |
| WrongContext | E_OS_CALLLEVEL | The system service was called from a context that is not permitted. |
| InterruptDisabled | E_OS_INTDISABLE | The system service was called with interrupts disabled. |
| ScheduleTableNotIdle | E_OS_STATE | The schedule table is already started. |
| AlarmInUse | E_OS_STATE | The schedule table's alarm is already in use. This indicates an internal error. Please notify your vendor. |
| InvalidScheduleId | E_OS_ID | The referenced schedule tables does not exist. |
| ParameterOutOfRange | E_OS_VALUE | The specified offset parameter is out of range. Either it is more than the MAXALLOWEDVALUE of the underlying counter, or it is zero. |
| Permission | E_OS_ACCESS | Permission has not been granted for the caller to access the referenced schedule tables. |
| ImplicitSyncStartRel | E_OS_ID | A schedule table configured with IMPLICIT synchronization strategy cannot be started at a relative counter value. StartScheduleTableAbs() must be used! |

## Name

StartScheduleTableSynchron — Start a schedule table synchronously

## Synopsis

```
#include <Os.h>
StatusType StartScheduleTableSynchron(ScheduleTableType s);
```

## Description

`StartScheduleTableSynchron()` places a schedule table into the `WAITING` state so that it will start synchronously when global time becomes available. The GlobalTime parameter is not used in the synchronization calculation.

## Service identification

`OS_SID_StartScheduleTableSynchron`.

## Return value

A return value of `E_OK` indicates a successful completion of the function. Any other return value indicates an error code as described below.

| **ErrorId(OS_ERROR_XX)** | **AUTOSAR error code** | **Description** |
|---|---|---|
| ApplicationNotAccessible | E_OS_ACCESS | The application to which the schedule table belongs was terminated and has not yet restarted. |
| CoreIsDown | E_OS_CORE | The core on which the schedule table resides has been shutdown. |
| WrongContext | E_OS_CALLLEVEL | The system service was called from a context that is not permitted. |
| InterruptDisabled | E_OS_INTDISABLE | The system service was called with interrupts disabled. |
| ScheduleTableNotIdle | E_OS_STATE | The schedule table is already started. |
| AlarmInUse | E_OS_STATE | The schedule table's alarm is already in use. This indicates an internal error. Please notify your vendor. |
| InvalidScheduleId | E_OS_ID | The referenced schedule tables does not exist. |
| Permission | E_OS_ACCESS | Permission has not been granted for the caller to access the referenced schedule table. |
| NotSyncable | E_OS_ID | The schedule table is not synchronisable. This is because its synchronization parameters have not been configured. Perhaps the schedule table is attached to a software counter. |

## Name

StartupHook — A hook routine for notifying system start

## Synopsis

```
#include <Os.h>
void StartupHook(void);
```

## Description

If configured, the kernel calls the user-supplied `StartupHook()` function when the system starts. It is called after all internal structures etc. have been initialized, but before the scheduler starts running. The `StartupHook()` function can be used to initialize hardware that cannot be initialized before calling `StartOS()`.

Note: On some architectures it is necessary to perform some hardware initialization before calling `StartOS()`. See the EB tresos AutoCore OS architecture notes for your CPU.

The `StartupHook()` function is called in the context of the kernel with category 2 interrupts disabled.

## Return value

Returns Nothing.

## Name

StartupHook_App — An application specific hook routine for system start-up

## Synopsis

```
#include <Os.h>
void StartupHook_App(void);
```

## Description

This application-specific hook is called by the kernel with the access rights of the associated OS application on start-up of the OS but after the system-specific `StartupHook()`.

The `StartupHook()` function is called in the context of the kernel with category 2 interrupts disabled.

## Return value

Returns nothing.

### Name

StopScheduleTable — Stop a schedule table

### Synopsis

```
#include <Os.h>
StatusType StopScheduleTable(ScheduleTableType s);
```

### Description

`StopScheduleTable()` stops a schedule table immediately. If another schedule table has been chained behind the specified schedule table, that chained table is also placed in the STOPPED state. If the specified schedule table is itself in the CHAINED state, the chaining link is broken.

### Service identification

`OS_SID_StopScheduleTable`.

### Return value

A return value of `E_OK` indicates a successful completion of the function. Any other return value indicates an error code as described below.

| ErrorId(OS_ERROR_XX) | AUTOSAR error code | Description |
|---|---|---|
| ApplicationNotAccessible | E_OS_ACCESS | The application to which the schedule table belongs was terminated and has not yet restarted. |
| CoreIsDown | E_OS_CORE | The core on which the schedule table resides has been shutdown. |
| WrongContext | E_OS_CALLLEVEL | The system service was called from a context that is not permitted. |
| InterruptDisabled | E_OS_INTDISABLE | The system service was called with interrupts disabled. |
| InvalidScheduleId | E_OS_ID | The referenced schedule table does not exist. |
| NotRunning | E_OS_NOFUNC | The referenced schedule table is not running. |
| Permission | E_OS_ACCESS | Permission has not been granted for the caller to access the referenced schedule table. |

## Name

SuspendInterrupts — Suspend interrupts up to a given level

## Synopsis

```
#include <Os.h>
void SuspendInterrupts(os_intlocktype_t locktype);
```

## Description

`SuspendInterrupts()` raises the interrupt level of the processor or interrupt controller to a level that depends on the `locktype` parameter. It is used to implement the `SuspendOSInterrupts()`, `SuspendAllInterrupts()` and `DisableAllInterrupts()` system services by calling it with the `locktype` parameter equal to `OS_LOCKTYPE_OS`, `OS_LOCKTYPE_ALL` and `OS_LOCKTYPE_NONEST` respectively.

`DisableAllInterrupts()` disables all category 1 and 2 interrupts. How this is achieved depends on the architecture, but it is not guaranteed that interrupts that are unknown to the kernel will be disabled.

`DisableAllInterrupts()` can be nested inside `SuspendOSInterrupts()`/`ResumeOSInterrupts()` pairs, but not inside `SuspendAllInterrupts()`/`ResumeAllInterrupts()` or further `DisableAllInterrupts()`/`EnableAllInterrupts()` pairs. Moreover, `DisableAllInterrupts()` prevents the caller from being preempted by another task. To achieve this `DisableAllInterrupts()` may delay cross-core kernel communication.

`SuspendAllInterrupts()` disables all category 1 and 2 interrupts and saves the previous state. Nested calls to this system service are permitted. The interrupt level is only truly manipulated on the outermost of the nested calls. Moreover, `SuspendAllInterrupts()` prevents the caller from being preempted by another task. To achieve this `SuspendAllInterrupts()` may delay cross-core kernel communication.

`SuspendOSInterrupts()` disables category 2 interrupts and saves the previous state. Nested calls to this system service are permitted. The interrupt level is only truly manipulated on the outermost of the nested calls. Moreover, `SuspendOSInterrupts()` prevents the caller from being preempted by another task. To achieve this `SuspendOSInterrupts()` may delay cross-core kernel communication.

If `SuspendOSInterrupts()` is called from a permitted context other than a task or category 2 ISR it is a no-operation, or if it is called within a code section that is controlled by `SuspendAllInterrupts()` or `DisableAllInterrupts()`, it is treated as a no-operation since interrupts are already blocked at a higher level.

interrupt lock timing is implemented for tasks and ISRs; the current context's *OS Interrupts Lock Time* is used for `SuspendOSInterrupts()` and *All Interrupts Lock Time* is used for the other two system services. If timing is already active, its state is saved before activating the interrupt lock timing.

WARNING: If `SuspendOSInterrupts()` is called for the first time within a code section protected by `SuspendAllInterrupts()` or `DisableAllInterrupts()`, the *OS* interrupt lock timing is not activated. The checker should always ensure that if the *OS interrupt lock Time* is activated for an OS object, the *All interrupt lock time* is also activated and is less than or equal to the *OS interrupt lock time*

## Service identification

`OS_SID_SuspendInterrupts`.

## Return value

Returns nothing. In case of any errors, ErrorHook will be called (if ErrorHook is configured and "OsErrorHandling" parameter is configured as FULL) with any one of the error code described below.

| ErrorId(OS_ERROR_XX) | AUTOSAR error code | Description |
|---|---|---|
| WrongContext | E_OS_CALLLEVEL | The system service was called from a context that is not permitted. |
| NestingOverflow | E_OS_NOFUNC | Too many nested calls to SuspendOSInterrupts. A possible cause is that the calls to SuspendOSInterrupts/ResumeOSInterrupts are not correctly nested. |

## Name

SyncScheduleTable — Synchronise a schedule table to global time

## Synopsis

```
#include <Os.h>
StatusType SyncScheduleTable(ScheduleTableType s, TickRefType globalTime);
```

## Description

`SyncScheduleTable()` sets up the synchronization variables of the schedule table such that the period will be adjusted at the next and subsequent end of round interrupts, subject to the configured maximum increase and maximum decrease values, until the time discrepancy is zero. When performing the adjustment, the adjustment direction is chosen to minimize the number of rounds taken to perform the synchronization.

The local time needed for the calculations is itself calculated from the time-to-next-interrupt and the offset of the next expiry point. This means that processing delays in the schedule table mechanisms, including this function, cannot be eliminated.

## Service identification

`OS_SID_SyncScheduleTable`.

## Return value

A return value of `E_OK` indicates a successful completion of the function. Any other return value indicates an error code as described below.

| ErrorId(OS_ERROR_XX) | AUTOSAR error code | Description |
|---|---|---|
| ApplicationNotAccessible | E_OS_ACCESS | The application to which the schedule table belongs was terminated and has not yet restarted. |
| CoreIsDown | E_OS_CORE | The core on which the schedule table resides has been shutdown. |
| WrongContext | E_OS_CALLLEVEL | The system service was called from a context that is not permitted. |
| InterruptDisabled | E_OS_INTDISABLE | The system service was called with interrupts disabled. |
| InvalidScheduleId | E_OS_ID | The referenced schedule table does not exist. |
| NotRunning | E_OS_STATE | The referenced current schedule table is not running or waiting for global time. |
| NotSyncable | E_OS_ID | The schedule table is not synchronisable. This is because its synchronization parameters have not been configured. Perhaps the schedule table is attached to a software counter. |
| Permission | E_OS_ACCESS | Permission has not been granted for the caller to access the referenced schedule table. |

| ErrorId(OS_ERROR_XX) | AUTOSAR error code | Description |
|---|---|---|
| ParameterOutOfRange | E_OS_VALUE | The the specified global time is not within the period of the schedule table. |

## Name

TASK — Define a Task function

## Synopsis

```
#include <Os.h>
TASK(taskname);
```

## Description

The `TASK()` macro defines a function to implement the body of the task whose name is give in the `taskname` parameter. The code you wish to execute when the task runs is placed in the body of the function.

The `TASK()` macro can only be used at the outer level of a C source file.

## Return value

Returns nothing.

## Code example

```
TASK(taskname)
{
  /* code for task */
  return;
}
```

## Name

TerminateApplication — Terminate the current application

## Synopsis

```
#include <Os.h>
StatusType TerminateApplication(ApplicationType aid, os_restart_t RestartOption);
```

## Description

`TerminateApplication()` terminates the application mentioned in `aid`. All tasks are terminated, all interrupts are disabled and pending interrupts cleared. All counters, alarms, and schedule tables are stopped and all resources are freed for the assigned application object. If the `RestartOption` parameter is `RESTART`, the application is restarted by activating its restart task if it has one. If the `RestartOption` parameter is `NO_RESTART`, the application remains terminated and cannot be restarted.

## Service identification

`OS_SID_TerminateApplication`.

## Return value

A return value of `E_OK` indicates a successful completion of the function. Any other return value indicates an error code as described below.

| ErrorId(OS_ERROR_XX) | AUTOSAR error code | Description |
|---|---|---|
| ApplicationNotAccessible | E_OS_STATE | The specified application has been terminated without restart. |
| CoreIsDown | E_OS_CORE | The core on which the application resides has been shutdown. |
| Permission | E_OS_ACCESS | Permission has not been granted for the caller to access the referenced application. |
| WrongContext | E_OS_CALLLEVEL | The system service was called from a context that is not permitted. |
| InterruptDisabled | E_OS_INTDISABLE | The system service was called with interrupts disabled. |
| InvalidApplicationId | E_OS_ID | The application could not be determined. |
| InvalidRestartOption | E_OS_VALUE | The restart option is neither RESTART nor NO_RESTART. |

## Name

TerminateTask — Terminate the current task

## Synopsis

```
#include <Os.h>
StatusType TerminateTask(void);
```

## Description

`TerminateTask()` terminates the current task. The calling task is transferred from the *RUNNING* state to the *SUSPENDED* state. The calling task must have released all resources and resumed all suspended interrupts before calling `TerminateTask()`.

The function does not normally return unless an error is detected.

Note: All resources occupied by the task must be released before calling `TerminateTask()`. If not, `TerminateTask()` will return E_OS_RESOURCE error.

`TerminateTask()` may only be called from a task.

## Service identification

`OS_SID_TerminateTask`.

## Return value

A return value of `E_OK` indicates a successful completion of the function. Any other return value indicates an error code as described below.

| ErrorId(OS_ERROR_XX) | AUTOSAR error code | Description |
|---|---|---|
| WrongContext | E_OS_CALLLEVEL | The system service was called from a context that is not permitted. |
| InterruptDisabled | E_OS_INTDISABLE | The system service was called with interrupts disabled. |
| HoldsLock | E_OS_SPINLOCK | The terminating task still occupies one or more spinlocks. |
| HoldsResource | E_OS_RESOURCE | The terminating task still occupies one or more resources. |

## Name

TryToGetSpinlock — Try to occupy a spinlock variable.

## Synopsis

```
#include <Os.h>
StatusType TryToGetSpinlock(SpinlockIdType lockId, TryToGetSpinlockType *out);
```

## Description

`TryToGetSpinlock()` tries to occupy the spinlock `lockId`. If the lock is acquired then OS_TRUE will be returned via `*out` else OS_FALSE will be returned via `*out`.

## Service identification

`OS_SID_TryToGetSpinlock`.

## Return value

A return value of `E_OK` indicates a successful completion of the function. Any other return value indicates an error code as described below.

| ErrorId(OS_ERROR_XX) | AUTOSAR error code | Description |
|---|---|---|
| WrongContext | E_OS_CALLLEVEL | The system service was called from a context that is not permitted. |
| WriteProtect | E_OS_ADDRESS | The application has attempted to write to a memory area where writing is not permitted. |
| InvalidSpinlockId | E_OS_ID | The specified spinlock ID is invalid. |
| Permission | E_OS_ACCESS | Permission has not been granted for the caller to access the referenced spinlock. |
| InvalidSpinlockNesting | E_OS_NESTING_DEADLOCK | An attempt has been made to acquire a spinlock while still holding another spinlock or, if spinlock nesting is enabled, to acquire a spinlock that is not a successor to the spinlock that is already held. |
| SpinlockAlreadyHeld | E_OS_STATE | An attempt has been made to acquire a spinlock that is already held by the caller. |
| SpinlockInterferenceDeadlock | E_OS_INTERFERENCE_DEADLOCK | An attempt has been made to acquire a spinlock that is already held by another task or ISR on the same core. |

## Name

WaitEvent — Wait for one of the set of events

## Synopsis

```
#include <Os.h>
StatusType WaitEvent(EventMaskType e);
```

## Description

`WaitEvent()` causes the calling task to wait until one or more events specified in the `e` parameter occurs. If an event is already pending, the function returns immediately. Otherwise, the task enters the `WAITING` state until one of the events occurs.

Calling `WaitEvent()` with an empty set of events is considered to be an error and handled accordingly.

A task in the `WAITING` state moves to the `READY` state and becomes eligible to run when one or more events for which it is waiting gets set. When the task resumes execution, it does so on the statement after the call to `WaitEvent()`.

`WaitEvent()` may only be called from an extended task.

## Service identification

`OS_SID_WaitEvent`.

## Return value

A return value of `E_OK` indicates a successful completion of the function. Any other return value indicates an error code as described below.

| **ErrorId(OS_ERROR_XX)** | **AUTOSAR error code** | **Description** |
|---|---|---|
| InterruptDisabled | E_OS_INTDISABLE | The system service was called with interrupts disabled. |
| NoEvents | E_OS_VALUE | The task has called WaitEvent but has specified no events to wait for. |
| TaskNotExtended | E_OS_ACCESS | The calling task is not an extended task. Only extended tasks are permitted to wait for events. |
| WrongContext | E_OS_CALLLEVEL | The system service was called from a context that is not permitted. |
| HoldsLock | E_OS_SPINLOCK | The terminating task still occupies one or more spinlocks. |
| HoldsResource | E_OS_RESOURCE | The terminating task still occupies one or more resources. |
| RateLimitExceeded | E_OS_RATEPROT | The calling task has exceeded its configured rate limit when waiting for an event that was already pending. |

## 4.2.5. Permitted calling context

Table 4.3, “Allowed calling context for OS service calls” shows which OS API function is callable from which context. In the table below, Y indicates that the call is allowed in this context.

| Service | Task | Cat1ISR | Cat2ISR | ErrorHook | PreTaskHook | PostTaskHook | StartupHook | ShutdownHook | AlarmCallback | ProtectionHook | PreISRHook | PostISRHook | TrustedFunction |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| `ActivateTask` | Y | | Y | | | | | | | | | | Y |
| `ActivateTaskAsyn` | Y | | Y | | | | | | | | | | Y |
| `TerminateTask` | Y | | | | | | | | | | | | |
| `ChainTask` | Y | | | | | | | | | | | | |
| `Schedule` | Y | | | | | | | | | | | | |
| `GetTaskID` | Y | | Y | Y | Y | Y | | | | Y | | | Y |
| `GetTaskState` | Y | | Y | Y | Y | Y | | | | | | | Y |
| `DisableAllInterrupts` | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | | | Y |
| `EnableAllInterrupts` | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | | | Y |
| `SuspendAllInterrupts` | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | | | Y |
| `ResumeAllInterrupts` | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | | | Y |
| `SuspendOSInterrupts` | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | | | Y |
| `ResumeOSInterrupts` | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y | | | Y |
| `GetResource` | Y | | Y | | | | | | | | | | Y |
| `ReleaseResource` | Y | | Y | | | | | | | | | | Y |
| `SetEvent` | Y | | Y | | | | | | | | | | Y |
| `SetEventAsyn` | Y | | Y | | | | | | | | | | Y |
| `ClearEvent` | Y | | | | | | | | | | | | |
| `GetEvent` | Y | | Y | Y | Y | Y | | | | | | | Y |
| `WaitEvent` | Y | | | | | | | | | | | | |
| `GetAlarmBase` | Y | | Y | Y | Y | Y | | | | | | | Y |
| `GetAlarm` | Y | | Y | Y | Y | Y | | | | | | | Y |
| `SetRelAlarm` | Y | | Y | | | | | | | | | | Y |
| `SetAbsAlarm` | Y | | Y | | | | | | | | | | Y |
| `CancelAlarm` | Y | | Y | | | | | | | | | | Y |

| Service | Task | Cat1ISR | Cat2ISR | ErrorHook | PreTaskHook | PostTaskHook | StartupHook | ShutdownHook | AlarmCallback | ProtectionHook | PreISRHook | PostISRHook | TrustedFunction |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| `GetActiveApplicationMode` | Y | | Y | Y | Y | Y | Y | Y | | | | | Y |
| `StartOS` | | | | | | | | | | | | | |
| `ShutdownOS` | Y | | Y | Y | | | Y | | | | | | Y |
| `GetApplicationID` | Y | | Y | Y | Y | Y | Y | Y | | Y | | | Y |
| `GetISRID` | Y | | Y | Y | | | | | | Y | | | Y |
| `CallTrustedFunction` | Y | | Y | | | | | | | | | | Y |
| `CheckISRMemoryAccess` | Y | | Y | Y | | | | | | Y | | | Y |
| `CheckTaskMemoryAccess` | Y | | Y | Y | | | | | | Y | | | Y |
| `CheckObjectAccess` | Y | | Y | Y | | | | | | Y | | | Y |
| `CheckObjectOwnership` | Y | | Y | Y | | | | | | Y | | | Y |
| `StartScheduleTableRel` | Y | | Y | | | | | | | | | | Y |
| `StartScheduleTableAbs` | Y | | Y | | | | | | | | | | Y |
| `StopScheduleTable` | Y | | Y | | | | | | | | | | Y |
| `NextScheduleTable` | Y | | Y | | | | | | | | | | Y |
| `ChainScheduleTable` | Y | | Y | | | | | | | | | | Y |
| `StartScheduleTableSyn-chron` | Y | | Y | | | | | | | | | | Y |
| `SyncScheduleTable` | Y | | Y | | | | | | | | | | |
| `GetScheduleTableStatus` | Y | | Y | | | | | | | | | | Y |
| `SetScheduleTableAsync` | Y | | Y | | | | | | | | | | Y |
| `IncrementCounter` | Y | | Y | | | | | | | | | | Y |
| `AdvanceCounter` | Y | | Y | | | | | | | | | | Y |
| `GetCounterValue` | Y | | Y | | | | | | | | | | Y |
| `GetElapsedCounterValue` | Y | | Y | | | | | | | | | | Y |
| `GetElapsedValue` | Y | | Y | | | | | | | | | | Y |
| `TerminateApplication` | Y | | Y | Y[a] | | | | | | | | | Y |
| `AllowAccess` | Y | | Y | | | | | | | | | | Y |
| `GetApplicationState` | Y | | Y | Y | Y | Y | Y | Y | | Y | | | Y |

| Service | Task | Cat1ISR | Cat2ISR | ErrorHook | PreTaskHook | PostTaskHook | StartupHook | ShutdownHook | AlarmCallback | ProtectionHook | PreISRHook | PostISRHook | TrustedFunction |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| `ControlIdle` | Y | | Y | | | | | | | | | | Y |
| `GetCurrentApplicationID` | Y | | Y | Y | Y | Y | Y | Y | | Y | | | Y |
| `DisableInterruptSource` | Y | | Y | | | | | | | | | | Y |
| `EnableInterruptSource` | Y | | Y | | | | | | | | | | Y |
| `ClearPendingInterrupt` | Y | | Y | | | | | | | | | | Y |

[a]Calling `TerminateApplication()` is only allowed in application-specific error hooks

Table 4.3. Allowed calling context for OS service calls

## 4.3. EB-specific API

This section describes the EB-specific API functions. Note that for EB tresos AutoCore Generic, atomic functions are supported by the atomics module in EB tresos AutoCore Generic Base. The atomic functions included here are intended for use with a standalone EB tresos AutoCore OS only.

## Name

OS_ATOMIC_OBJECT_INITIALIZER — Initializes an atomic object.

## Synopsis

**`OS_ATOMIC_OBJECT_INITIALIZER`**`(initialValue)`

## Description

The macro `OS_ATOMIC_OBJECT_INITIALIZER()` expands to an initializer for an atomic object of type `os_atomic_t` and initializes it with the given initial value.

## Name

OS_AtomicClearFlag — Atomically clears a flag in the atomic object.

## Synopsis

`void OS_AtomicClearFlag (os_atomic_t volatile *object, os_atomic_value_t flagSelectionMask)`

## Description

`OS_AtomicClearFlag()` atomically clears the flag selected by `flagSelectionMask` in the atomic object at `object`. The selection mask may have only one bit set.

## Return value

Returns nothing.

## Name

OS_AtomicCompareExchange — Atomically compares and exchanges values.

## Synopsis

`os_boolean_t **OS_AtomicCompareExchange**(os_atomic_t volatile *object, os_atomic_value_t *expected, os_atomic_value_t newValue)`

## Description

`OS_AtomicCompareExchange()` atomically exchanges the value of the atomic object at `object` with the value `newValue`, if and only if, its value is equal to the value at `expected`.

## Return value

Returns the value `OS_TRUE`, if the atomic object at `object` was changed and `OS_FALSE` otherwise. In the latter case, the memory pointed to by `expected` is updated to contain the value of the atomic object at `object`, that it had at the point in time, when this function was called.

## Name

OS_AtomicExchange — Atomically exchanges values.

## Synopsis

`os_atomic_value_t **OS_AtomicExchange** (os_atomic_t volatile *object, os_atomic_value_t newValue)`

## Description

`OS_AtomicExchange()` atomically exchanges the value of the atomic object at `object` with the value `newValue`.

## Return value

Returns the value of the atomic object at `object` before the exchange.

## Name

OS_AtomicFetchAdd — Atomically adds the given value to the atomic object.

## Synopsis

`os_atomic_value_t **OS_AtomicFetchAdd** (os_atomic_t volatile *object, os_atomic_value_t operand)`

## Description

`OS_AtomicFetchAdd()` atomically adds the value `operand` to the atomic object at `object` and updates it with the result.

## Return value

Returns the value of the atomic object at `object` before the operation.

## Name

OS_AtomicFetchAnd — Atomically ANDs the given value with the atomic object.

## Synopsis

`os_atomic_value_t **OS_AtomicFetchAnd** (os_atomic_t volatile *object, os_atomic_value_t operand)`

## Description

`OS_AtomicFetchAnd()` atomically performs the boolean AND operation with the value `operand` and the value of the atomic object at `object` and updates it with the result.

## Return value

Returns the value of the atomic object at `object` before the operation.

## Name

OS_AtomicFetchOr — Atomically ORs the given value with the atomic object.

## Synopsis

`os_atomic_value_t` **`OS_AtomicFetchOr`** `(os_atomic_t volatile *object, os_atomic_value_t operand)`

## Description

`OS_AtomicFetchOr()` atomically performs the boolean OR operation with the value `operand` and the value of the atomic object at `object` and updates it with the result.

## Return value

Returns the value of the atomic object at `object` before the operation.

## Name

OS_AtomicFetchSub — Atomically subtracts the given value from the atomic object.

## Synopsis

`os_atomic_value_t **OS_AtomicFetchSub** (os_atomic_t volatile *object, os_atomic_value_t operand)`

## Description

`OS_AtomicFetchSub()` atomically subtracts the value `operand` from the atomic object at `object` and updates it with the result.

## Return value

Returns the value of the atomic object at `object` before the operation.

## Name

OS_AtomicFetchXor — Atomically XORs the given value with the atomic object.

## Synopsis

`os_atomic_value_t **OS_AtomicFetchXor** (os_atomic_t volatile *object, os_atomic_value_t operand)`

## Description

`OS_AtomicFetchXor()` atomically performs the boolean XOR operation with the value `operand` and the value of the atomic object at `object` and updates it with the result.

## Return value

Returns the value of the atomic object at `object` before the operation.

## Name

OS_AtomicInit — Initializes an atomic object.

## Synopsis

```
void OS_AtomicInit (os_atomic_t volatile *object, os_atomic_value_t initialValue)
```

## Description

`OS_AtomicInit()` initializes the atomic object at `object` with the given initial value.

## Return value

Returns nothing.

## Name

OS_AtomicLoad — Loads from the given memory location atomically.

## Synopsis

`os_atomic_value_t **OS_AtomicLoad** (os_atomic_t const volatile *object)`

## Description

`OS_AtomicLoad()` atomically loads the value of the atomic object at `object`.

## Return value

Returns the value of the atomic object at `object`.

## Name

OS_AtomicStore — Stores the given value atomically.

## Synopsis

```
void OS_AtomicStore (os_atomic_t volatile *object, os_atomic_value_t newValue)
```

## Description

`OS_AtomicStore()` atomically stores the value `newValue` into the atomic object at `object`.

## Return value

Returns nothing.

## Name

OS_AtomicTestAndSetFlag — Atomically sets a flag in the atomic object.

## Synopsis

`os_boolean_t **OS_AtomicTestAndSetFlag**(os_atomic_t volatile *object, os_atomic_value_t flagSelectionMask)`

## Description

`OS_AtomicTestAndSetFlag()` atomically sets the flag selected by `flagSelectionMask` in the atomic object at `object`. The selection mask may have only one bit set.

## Return value

Returns the state of the selected flag before the operation.

## Name

OS_AtomicThreadFence — A sequential-consistent memory fence.

## Synopsis

`void OS_AtomicThreadFence(void)`

## Description

`OS_AtomicThreadFence()` inserts a sequential-consistent memory fence into the program, where it is called. It prevents read and write instructions from being reordered across it either way. This restriction applies to both the hardware and compiler level. When it returns, all past memory accesses are finished with system-wide visibility.

## Return value

Returns nothing.

## Name

OS_DiffTime32 — Calculates the 32-bit length of an interval between two times

## Synopsis

```
#include <Os.h>
os_uint32_t OS_DiffTime32(const os_timestamp_t*newTime, const os_timestamp_t*oldTime);
```

## Description

`OS_DiffTime32()` calculates the difference (`newTime` - `oldTime`) (i.e. the duration of the interval that starts at `oldTime` and ends at `newTime`). The result is returned as 32-bit number. If the time difference is too large to be represented in 32 bits, the function returns the maximum value that can be represented (`0xffffffff`).

## Return value

See the description above

## Name

OSErrorGetServiceId — Get the identifier of the system service that detected an error.

## Synopsis

```
#include <Os.h>
OSServiceIdType OSErrorGetServiceId(void);
```

## Description

`OSErrorGetServiceId()` returns the identifier of the system service that caused the `ErrorHook()` function to be called.

The possible return values are `OSServiceId_xx`, where `xx` is the name of a system service.

`OSErrorGetServiceId()` only returns valid information when called from an `ErrorHook()` function (including the Autosar application-specific error hooks and protection hook). The return value is undefined if `OSErrorGetServiceId()` is called from elsewhere.

If an optimized kernel is built from the source files, `OSErrorGetServiceId()` is only available when the `USEGETSERVICEID` attribute of the `OS` object is set to `TRUE`.

## Return value

Returns the service ID of the API for which the error has occurred.

## Name

OSError_x1_x2 — Get the value of a parameter to a service.

## Synopsis

```
#include <Os.h>
ParameterType OSError_x1_x2(void);
```

## Description

`OSError_x1_x2()` is a collection of macros that return the parameters passed to the system service that caused the `ErrorHook()` to be called. `x1` is the name of the system service and `x2` is the name of the parameter. The return type of the macro is the same as the type of the parameter.

`OSError_x1_x2()` only returns valid information when called from an `ErrorHook()` function (including the Autosar application-specific error hooks and protection hook). The return value is undefined if `OSError_x1_x2()` is called from elsewhere.

If an optimized kernel is build from the source files, `OSError_x1_x2()` is only available when the `USEPARAMETERACCESS` attribute of the `OS` object is set to `TRUE`.

## Return value

See the description above.

## Name

OS_GetCurrentStackArea — Get current stack boundaries

## Synopsis

```
#include <Os.h>
void OS_GetCurrentStackArea(void **begin, void **end);
```

## Description

`OS_GetCurrentStackArea()` places the base and limit addresses of the stack of the currently-executing object into the two referenced variables. For a Task, this is simply the stack area as allocated by the Os generator. For ISRs, if the ISR has a private stack, this is returned. Otherwise the entire kernel stack area is returned. This does not imply that the whole area is accessible by the caller.

## Return value

Returns nothing.

## Name

OS_GetErrorInfo — Get error status information

## Synopsis

```
#include <Os.h>
const os_errorstatus_t*OS_GetErrorInfo(void);
```

## Description

`OS_GetErrorInfo()` returns a pointer to the error information status structure of the current core. The information in this structure is valid during the `ErrorHook()` and `ProtectionHook()`. It will be overwritten, once the next error occurs.

## Return value

See the description above.

## Name

OS_GetIsrMaxRuntime — Get longest observed run-time of an ISR

## Synopsis

```
#include <Os.h>
StatusType OS_GetIsrMaxRuntime(ISRType t, TickRefType *out);
```

## Description

`OS_GetIsrMaxRuntime()` places the longest observed execution time of the specified ISR into the variable referenced by 'out'. If the ISR ID is invalid or the ISR does not have execution-time measurement enabled (attribute MEASURE_MAX_RUNTIME), `OS_GetIsrMaxRuntime()` returns `OS_E_ID`.

Available from all trusted tasks, ISRs and hook functions. On some architectures, it might be possible to call this function from non-trusted contexts as well.

## Return value

A return value of `E_OK` indicates a successful completion of the function.

A return value of `E_OS_ID` indicates that the ISR ID is invalid.

## Name

OS_GetStackInfo — Get information about a stack

## Synopsis

```
#include <Os.h>
StatusType OS_GetStackInfo(os_taskorisr_t id, os_stackinfo_t *out);
```

## Description

`OS_GetStackInfo()` places information about a task or ISR stack into the memory location given by the `out` parameter.
You must use `OS_TaskToTOI(task_id)` to specify a task ID, for example as follows:

```
OS_GetStackInfo(OS_TaskToTOI(task_id), os_stackinfo_t *out);
```

If the task ID given in `id` is OS_NULLTASK, information about the current task is returned. If there is no current task, the `out` location is not modified and OS_E_NOFUNC is returned, but the error handler is not called.

You must use `OS_IsrToTOI(isr_id)` to specify an ISR ID, for example as follows:

```
OS_GetStackInfo(OS_IsrToTOI(isr_id), os_stackinfo_t *out);
```

If the ISR ID given in `id` is OS_NULLISR, information about the current ISR is returned. If there is no current ISR, information about the global kernel stack is returned. Depending on the architecture and on the calling mechanism of ISRs, the kernel stack may get shared for ISRs, or private ISR stacks may get used. If private ISR stacks are used - which is quite the exception - it is not advisable to estimate free ISR stack using OS_-NULLISR outside of an ISR.

As a special case, if the `id` parameter is OS_TOI_CURRENTCONTEXT, the information about the caller's context is returned. In this case the SP is always OS_NULL.

The `stackPointer` field of the `out` parameter is not updated if the request is for the current task. Therefore the caller has to preset the current SP value in the `stackPointer` field before calling `OS_GetStackInfo()`.

The fields `isrStackBase` and `isrStackLen` of the `out` parameter only apply to ISRs. When a task queries the stack information of an ISR, then these fields have the values NULL and 0 respectively.

`OS_GetStackInfo()` can be called from tasks and category 2 ISRs.

## Service identification

`OS_SID_GetStackInfo`.

## Return value

A return value of `E_OK` indicates a successful completion of the function. Any other return value indicates an error code as described below.

| ErrorId(OS_ERROR_XX) | AUTOSAR error code | Description |
|---|---|---|
| WriteProtect | E_OS_ADDRESS | The application has attempted to write to a memory area where writing is not permitted. |
| InvalidTaskId | E_OS_ID | The specified task ID is invalid. |
| InvalidIsrId | E_OS_ID | The specified ISR ID is invalid. |

| ErrorId(OS_ERROR_XX) | AUTOSAR error code | Description |
|---|---|---|
| WrongContext | E_OS_CALLLEVEL | The system service was called from a context that is not permitted. |

## Name

OS_GetTaskMaxRuntime — Get longest observed run-time of a task

## Synopsis

```
#include <Os.h>
StatusType OS_GetTaskMaxRuntime(TaskType t, TickRefType *out);
```

## Description

`OS_GetTaskMaxRuntime()` places the longest observed execution time of the specified task into the variable referenced by 'out'. If the task ID is invalid or the task does not have execution-time measurement enabled (attribute MEASURE_MAX_RUNTIME), `OS_GetTaskMaxRuntime()` returns `OS_E_ID`.

Available from all trusted tasks, ISRs and hook functions. On some architectures, it might be possible to call this function from non-trusted contexts as well.

## Return value

A return value of `E_OK` indicates a successful completion of the function.

A return value of `E_OS_ID` indicates that the task ID is invalid.

## Name

OS_GetTimeStamp — Puts a timestamp value into the indicated location

## Synopsis

```
#include <Os.h>
void OS_GetTimeStamp(os_timestamp_t *out);
```

## Description

`OS_GetTimeStamp()` stores the current timestamp value into the indicated "out" location. A timestamp is a counter that can never overflow during the expected up-time of the processor.

## Return value

Returns nothing.

## Name

OS_GetUnusedIsrStack — Get the amount of interrupt stack that remains unused

## Synopsis

```
#include <Os.h>
MemorySizeType OS_GetUnusedIsrStack(void);
```

## Description

`OS_GetUnusedIsrStack()` returns the amount of interrupt stack that has not been overwritten. At start-up, all stacks are filled with a fill pattern. The amount of interrupt stack that still contains the fill pattern is counted.

`OS_GetUnusedIsrStack()` can be used from Tasks and ISRs.

## Return value

Returns the amount of stack that is not yet used by the ISR.

## Name

OS_GetUnusedTaskStack — Get the amount of stack the task has not used

## Synopsis

```
#include <Os.h>
MemorySizeType OS_GetUnusedTaskStack(TaskType t);
```

## Description

`OS_GetUnusedTaskStack()` returns the amount of stack that has not been overwritten by the given task. At start-up, all stacks are filled with a fill pattern. The amount of stack that still contains the fill pattern is counted. If two or more tasks are sharing the same stack, it is not known which of the tasks has written to the stack. For this function to return 100% reliable values, the stack-sharing feature in the Generator should be turned off.

`OS_GetUnusedTaskStack()` can be used from Tasks and ISRs.

## Return value

Returns the amount of stack that is not yet used by the task.

## Name

OS_GetUsedIsrStack — Get the amount of interrupt stack that has been used

## Synopsis

```
#include <Os.h>
MemorySizeType OS_GetUsedIsrStack(void);
```

## Description

`OS_GetUsedIsrStack()` returns the amount of interrupt stack that has been overwritten. At start-up, all stacks are filled with a fill pattern. The amount of interrupt stack that still contains the fill pattern is counted and subtracted from the total amount.

`OS_GetUsedIsrStack()` can be used from Tasks and ISRs.

## Return value

Returns the amount of overwritten interrupt stack.

## Name

OS_GetUsedTaskStack — Get the amount of stack the task has used

## Synopsis

```
#include <Os.h>
MemorySizeType OS_GetUsedTaskStack(TaskType t);
```

## Description

`OS_GetUsedTaskStack()` returns the amount of stack that has been overwritten by the given task. At start-up, all stacks are filled with a fill pattern. The amount of stack that still contains the fill pattern is counted and subtracted from the total amount.
If two or more tasks are sharing the same stack, it is not known which of the tasks has written to the stack. For this function to return 100% reliable values, the stack-sharing feature in the Generator should be turned off.

`OS_GetUsedTaskStack()` can be used from Tasks and ISRs.

## Return value

Returns the amount of overwritten task stack.

## Name

OS_IsScheduleNecessary — Determine whether a call to Schedule() is necessary

## Synopsis

```
#include <Os.h>
ObjectAccessType OS_IsScheduleNecessary(void);
```

## Description

`OS_IsScheduleNecessary()` returns TRUE (non-zero) if there is no current task or another task in the task queue with a higher configured priority than the current task. Otherwise it returns FALSE.

`OS_IsScheduleNecessary()` should only be called from a task. If it is called from another context and there is a current task, it will return information about that task. If there is no current task it will return true, Schedule() gets called and reports context error. `OS_IsScheduleNecessary()` can only be called from tasks that have read access to kernel variables. On most systems this will be true, but in SC3 and SC4 memory protection might prevent access if so configured and will detect a memory protection error in the calling task.

## Return value

See the description above.

## Name

OS_IsScheduleWorthwhile — Determine whether a call to Schedule() is worthwhile

## Synopsis

```
#include <Os.h>
ObjectAccessType OS_IsScheduleWorthwhile(void);
```

## Description

`OS_IsScheduleWorthwhile()` returns TRUE (non-zero) if there is no current task or another task in the task queue (other than the current task). Otherwise it returns FALSE.

`OS_IsScheduleWorthwhile()` is faster than `OS_IsScheduleNecessary()`, but can return TRUE even if `Schedule()` will have no effect. However, it might result in a performance improvement in some circumstances, especially when called from a background task that is of the lowest priority.

`OS_IsScheduleWorthwhile()` should only be called from a task. If it is called from another context and there is a current task, it will return information about that task. If there is no current task it will return true, `Schedule()` gets called and reports context error. `OS_IsScheduleWorthwhile()` can only be called from tasks that have read access to kernel variables. On most systems this will be true, but in SC3 and SC4 memory protection might prevent access if so configured and will detect a memory protection error in the calling task.

## Return value

See the description above.

## Name

OS_ScheduleIfNecessary — Call Schedule() if necessary

## Synopsis

```
#include <Os.h>
StatusType OS_ScheduleIfNecessary(void);
```

## Description

`OS_ScheduleIfNecessary()` calls `OS_IsScheduleNecessary()` and if it returns TRUE, calls `Schedule()` and returns the result. Otherwise E_OS_OK is returned.

`OS_ScheduleIfNecessary()` should only be called from a task. The conditions and restrictions for `OS_IsScheduleNecessary()` apply here as well.

## Return value

See the description above.

## Name

OS_ScheduleIfWorthwhile — Call Schedule() if worthwhile

## Synopsis

```
#include <Os.h>
StatusType OS_ScheduleIfWorthwhile(void);
```

## Description

`OS_ScheduleIfWorthwhile()` calls `OS_IsScheduleWorthwhile()` and if it returns TRUE, calls `Schedule()` and returns the result. Otherwise E_OS_OK is returned.

`OS_ScheduleIfWorthwhile()` should only be called from a task. The conditions and restrictions for `OS_IsScheduleWorthwhile()` apply here as well.

## Return value

See the description above.

## Name

OS_SimTimerAdvance — Advances a simulated timer by a given value

## Synopsis

```
#include <Os.h>
StatusType OS_SimTimerAdvance(os_unsigned_t tmr, TickRefType incr);
```

## Description

`OS_SimTimerAdvance()` increments a simulated timer by the given value. It checks for each channel whether the timer is pending or passed the value programmed in its compare register. If the channel is enabled, it calls the respective associated ISR, otherwise the channel is set to pending.

## Return value

Returns `E_OK` if the function is executed successfully.

Returns `E_OS_ID` if the timer index is out of range.

Returns `E_OS_VALUE` if the increment value is greater than the mask value of the timer.

## Name

OS_SimTimerSetup — Set up a simulated timer channel

## Synopsis

```
#include <Os.h>
StatusType OS_SimTimerSetup(os_unsigned_t tmr, os_unsigned_t chan, ISRType isrId);
```

## Description

`OS_SimTimerSetup()` sets up a simulated timer channel by clearing its compare and control registers and setting its interrupts ID.

## Return value

A return value of `E_OK` indicates a successful completion of the function.

A return value of `E_OS_ID` indicates either timer index or the compare register value is out of range.

A return value of `E_OS_VALUE` indicates ISR ID is invalid.

## Name

OS_StackCheck — Check current stack use

## Synopsis

```
#include <Os.h>
MemorySizeType OS_StackCheck(void);
```

## Description

`OS_StackCheck()` checks the stack use in the current context. If there is or has been a stack overflow, `OS_StackCheck()` returns +1. If there is a stack underflow, `OS_StackCheck()` returns -1. Otherwise `OS_StackCheck()` returns 0. This function can be used in all tasks and ISRs

## Return value

See the description above.

## Name

OS_TimeGetHi — Returns high word of a timestamp value

## Synopsis

```
#include <Os.h>
os_uint32_t OS_TimeGetHi(os_timestamp_t t);
```

## Description

`OS_TimeGetHi()` returns the high word of a given timestamp value.

## Return value

See the description above.

## Name

OS_TimeGetLo — Returns low word of a timestamp value

## Synopsis

```
#include <Os.h>
os_uint32_t OS_TimeGetLo(os_timestamp_t t);
```

## Description

`OS_TimeGetLo()` returns the low word of a given timestamp value.

## Return value

See the description above.

## Name

OS_TimeSub64 — Returns high word of a timestamp value

## Synopsis

```
#include <Os.h>
void OS_TimeSub64(os_timestamp_t *diffTime, const os_timestamp_t *newTime, const
os_timestamp_t *oldTime);
```

## Description

`OS_TimeSub64()` calculates the difference (`newTime` - `oldTime`) (i.e. the duration of the interval that starts at `oldTime` and ends at `newTime`). The two input values are variables provided by the caller whose addresses are passed as parameters. The result is placed into the variable whose address is specified by the `diffTime` parameter. The caller must have permission to modify this variable.

## Return value

Returns the difference between newTime and oldTime.

# 4.4. Kernel error codes

This section describes the information provided in an error case scenario and the related OSEK/AUTOSAR error codes.

## 4.4.1. Error information

An error information C-structure `os_errorstatus_s` is filled with detailed information about the error by the EB tresos AutoCore OS error handler. The information in the structure is valid during the error-hook functions and protection-hook functions. Outside these functions its content is not defined.

The function `OS_GetErrorInfo()` returns the error information for the core on which it is called. This error information structure `os_errorstatus_s` contains the following fields:

```
struct os_errorstatus_s
            {
                os_paramtype_t parameter[OS_MAXPARAM];
                os_result_t result;
                os_erroraction_t action;
                os_uint8_t calledFrom;
                os_serviceid_t serviceId;
                os_error_t errorCondition;
            }
```

- `parameter` is an array of three parameters that contain useful information related to the point of failure.

  When the error is caused by an API function, the parameters that are passed to the function when it was called are placed into the `parameter` array, in the order given by the API interface. So the first parameter in the interface will be placed in `parameter[0]`, the second in `parameter[1]` and so on. If an API function has fewer than 3 input parameters, then the unused elements of the `parameter` array are undefined. The user can determine how many elements have valid information by checking the API interface specification.

  When the error is caused by a protection fault, then `parameter` contains hardware-specifc information. See the EB tresos AutoCore OS architecture notes for more details on hardware specific error handling.

- `result` contains the same value as the error code that was passed as a parameter to the error-hook function or protection-hook function. Its value will be returned to the caller if such an `action` is chosen and the affected system service returns a status code. See Section 4.4.2, “List of OSEK/AUTOSAR error codes”.

  If `result` is modified by an error-hook function the new value will be returned instead of the default. Modifying `result` in the protection hook has no effect.

- `action` indicates the action that will be taken when the hook function returns. If an error-hook function modifies the content of `action`, the new action will be taken instead of the default. Use this with caution!

Note: modifying `action` in the protection hook has no effect because the return value of the protection hook determines the action.

From the values defined in `Os_error.h` the following may be used for `action`:

| Value | Identifier | Description |
|---|---|---|
| **0** | `OS_ACTION_IGNORE` | Ignore the error and return `OS_E_OK` |
| **1** | `OS_ACTION_RETURN` | Only return `result` to caller |
| **2** | `OS_ACTION_KILL` | Kill the task or ISR that caused the error |
| **3** | `OS_ACTION_QUARANTINE` | Quarantine the task or ISR that caused the error |
| **4** | `OS_ACTION_QUARANTINEAPP` | Quarantine the application that caused the error |
| **5** | `OS_ACTION_RESTART` | Kill and restart the application that caused the error |
| **6** | `OS_ACTION_SHUTDOWN` | Shut down the OS |

Table 4.4. Possible values for action

- `calledFrom` indicates the context in which the error occurred.

The possible values are defined in `Os_kernel_task.h` and listed in the following table:

| Value | Identifier | Description |
|---|---|---|
| **0** | `OS_INBOOT` | Error occurred while the system was starting up |
| **1** | `OS_INTASK` | Error occurred while executing a task |
| **2** | `OS_INCAT1` | Error occurred while executing a Cat-1 ISR |
| **3** | `OS_INCAT2` | Error occurred while executing a Cat-2 ISR |
| **4** | `OS_INACB` | Error occurred while executing an alarm callback |
| **5** | `OS_INSHUTDOWN` | Error occurred while the system was shutting down |
| **6** | `OS_ININTERNAL` | Error occurred while executing an internal kernel function |
| **7** | `OS_INSTARTUPHOOK` | Error occurred while executing a start-up hook |
| **8** | `OS_INSHUTDOWNHOOK` | Error occurred while executing a shutdown hook |
| **9** | `OS_INERRORHOOK` | Error occurred while executing an error hook |
| **10** | `OS_INPRETASKHOOK` | Error occurred while executing a pre-task hook |
| **11** | `OS_INPOSTTASKHOOK` | Error occurred while executing a post-task hook |
| **12** | `OS_INPREISRHOOK` | Error occurred while executing a pre-ISR hook |
| **13** | `OS_INPOSTISRHOOK` | Error occurred while executing a post-ISR hook |
| **14** | `OS_INPROTECTIONHOOK` | Error occurred while executing a protection hook |

Table 4.5. Possible values for calledFrom

- `serviceId` indicates the system service in which the error was detected. This is one of the `OS_SID_-xxx` constants defined in `Os_error.h`. See Section 4.4.3, "List of service identifiers".
- `errorCondition` indicates the exact error condition. Its value is one of the `OS_ERROR_xxx` constants defined in `Os_error.h` . See Section 4.4.4, "List of error identifiers" .

## 4.4.2. List of OSEK/AUTOSAR error codes

The OSEK/AUTOSAR error codes are returned to the caller by various OS services and are passed as a parameter to the `ErrorHook()`, `ProtectionHook()` and `ShutdownHook()` functions. The OSEK/AUTOSAR error code is also stored in the structure returned by `OS_GetErrorInfo()` in the field `result`. These error macros are defined in `os_api.h`.

| Value | Identifier |
|---|---|
| **0** | E_OK |
| **1** | E_OS_ACCESS |
| **2** | E_OS_CALLEVEL |
| **3** | E_OS_ID |
| **4** | E_OS_LIMIT |
| **5** | E_OS_NOFUNC |
| **6** | E_OS_RESOURCE |
| **7** | E_OS_STATE |
| **8** | E_OS_VALUE |
| **9** | E_OS_STACKFAULT |
| **10** | E_OS_PROTECTION_MEMORY |
| **11** | E_OS_PROTECTION_TIME |
| **12** | E_OS_PROTECTION_LOCKED |
| **13** | E_OS_PROTECTION_ARRIVAL |
| **14** | E_OS_PROTECTION_EXCEPTION |
| **15** | E_OS_ILLEGAL_ADDRESS |
| **16** | E_OS_DISABLEDINT |
| **17** | E_OS_MISSINGEND |
| **18** | E_OS_SERVICEID |
| **23** | E_OS_CORE |
| **24** | E_OS_NESTING_DEADLOCK |
| **25** | E_OS_INTERFERENCE_DEADLOCK |

| Value | Identifier |
|---|---|
| **26** | E_OS_SPINLOCK |

Table 4.6. List of OSEK/AUTOSAR error codes

### 4.4.3. List of service identifiers

The service identifier specifies which kernel function reported the error. It is stored in the structure returned by `OS_GetErrorInfo()` in the field `serviceId`. These macros are defined in `Os_error.h`.

| Value | Identifier |
|---|---|
| **0** | OS_SID_GetApplicationId |
| **1** | OS_SID_GetIsrId |
| **2** | OS_SID_CallTrustedFunction |
| **3** | OS_SID_CheckIsrMemoryAccess |
| **4** | OS_SID_CheckTaskMemoryAccess |
| **5** | OS_SID_CheckObjectAccess |
| **6** | OS_SID_CheckObjectOwnership |
| **7** | OS_SID_StartScheduleTableRel |
| **8** | OS_SID_StartScheduleTableAbs |
| **9** | OS_SID_StopScheduleTable |
| **10** | OS_SID_ChainScheduleTable |
| **11** | OS_SID_StartScheduleTableSynchron |
| **12** | OS_SID_SyncScheduleTable |
| **13** | OS_SID_SetScheduleTableAsync |
| **14** | OS_SID_GetScheduleTableStatus |
| **15** | OS_SID_IncrementCounter |
| **16** | OS_SID_GetCounterValue |
| **17** | OS_SID_GetElapsedCounterValue |
| **18** | OS_SID_TerminateApplication |
| **19** | OS_SID_AllowAccess |
| **20** | OS_SID_GetApplicationState |
| **21** | OS_SID_UnknownSyscall |
| **22** | OS_SID_ActivateTask |

| Value | Identifier |
|---|---|
| **23** | OS_SID_TerminateTask |
| **24** | OS_SID_ChainTask |
| **25** | OS_SID_Schedule |
| **26** | OS_SID_GetTaskId |
| **27** | OS_SID_GetTaskState |
| **28** | OS_SID_SuspendInterrupts |
| **29** | OS_SID_ResumeInterrupts |
| **30** | OS_SID_GetResource |
| **31** | OS_SID_ReleaseResource |
| **32** | OS_SID_SetEvent |
| **33** | OS_SID_ClearEvent |
| **34** | OS_SID_GetEvent |
| **35** | OS_SID_WaitEvent |
| **36** | OS_SID_GetAlarmBase |
| **37** | OS_SID_GetAlarm |
| **38** | OS_SID_SetRelAlarm |
| **39** | OS_SID_SetAbsAlarm |
| **40** | OS_SID_CancelAlarm |
| **41** | OS_SID_GetActiveApplicationMode |
| **42** | OS_SID_StartOs |
| **43** | OS_SID_ShutdownOs |
| **44** | OS_SID_GetStackInfo |
| **45** | OS_SID_DisableInterruptSource |
| **46** | OS_SID_EnableInterruptSource |
| **47** | OS_SID_TryToGetSpinlock |
| **48** | OS_SID_ReleaseSpinlock |
| **49** | OS_SID_ShutdownAllCores |
| **50** | OS_SID_ActivateTaskAsyn |
| **51** | OS_SID_SetEventAsyn |
| **52** | OS_SID_ClearPendingInterrupt |
| **53** | OS_SID_ControlIdle |

| Value | Identifier |
|---|---|
| **54** | OS_SID_GetCurrentApplicationId |
| **55** | OS_SID_Dispatch |
| **56** | OS_SID_TrapHandler |
| **57** | OS_SID_IsrHandler |
| **58** | OS_SID_RunSchedule |
| **59** | OS_SID_KillAlarm |
| **60** | OS_SID_TaskReturn |
| **61** | OS_SID_HookHandler |
| **62** | OS_SID_ArchTrapHandler |
| **63** | OS_SID_MemoryManagement |

Table 4.7. List of service identifiers

### 4.4.4. List of error identifiers

The error identifier specifies exactly what the error is. It is stored in the structure returned by `OS_GetErrorInfo()` in the field `errorCondition`. These macros are defined in `Os_error.h`.

| Value | Identifier |
|---|---|
| **0** | OS_ERROR_NoError |
| **0** | OS_ERROR_UnknownError |
| **1** | OS_ERROR_UnknownSystemCall |
| **2** | OS_ERROR_InvalidTaskId |
| **3** | OS_ERROR_InvalidTaskState |
| **4** | OS_ERROR_Quarantined |
| **5** | OS_ERROR_MaxActivations |
| **6** | OS_ERROR_WriteProtect |
| **7** | OS_ERROR_ReadProtect |
| **8** | OS_ERROR_ExecuteProtect |
| **9** | OS_ERROR_InvalidAlarmId |
| **10** | OS_ERROR_InvalidAlarmState |
| **11** | OS_ERROR_AlarmNotInUse |
| **12** | OS_ERROR_WrongContext |

| Value | Identifier |
| --- | --- |
| **13** | OS_ERROR_HoldsResource |
| **14** | OS_ERROR_NoEvents |
| **15** | OS_ERROR_TaskNotExtended |
| **16** | OS_ERROR_TaskNotInQueue |
| **17** | OS_ERROR_InvalidCounterId |
| **18** | OS_ERROR_CorruptAlarmList |
| **19** | OS_ERROR_ParameterOutOfRange |
| **20** | OS_ERROR_AlarmInUse |
| **21** | OS_ERROR_AlreadyStarted |
| **22** | OS_ERROR_InvalidStartMode |
| **23** | OS_ERROR_AlarmNotInQueue |
| **24** | OS_ERROR_InvalidResourceId |
| **25** | OS_ERROR_ResourceInUse |
| **26** | OS_ERROR_ResourcePriorityError |
| **27** | OS_ERROR_ResourceNestingError |
| **28** | OS_ERROR_TaskSuspended |
| **29** | OS_ERROR_NestingUnderflow |
| **30** | OS_ERROR_NestingOverflow |
| **31** | OS_ERROR_NonfatalException |
| **32** | OS_ERROR_FatalException |
| **33** | OS_ERROR_UnhandledNmi |
| **34** | OS_ERROR_TaskTimeBudgetExceeded |
| **35** | OS_ERROR_IsrTimeBudgetExceeded |
| **36** | OS_ERROR_UnknownTimeBudgetExceeded |
| **37** | OS_ERROR_Permission |
| **38** | OS_ERROR_ImplicitSyncStartRel |
| **39** | OS_ERROR_CounterIsHw |
| **40** | OS_ERROR_InvalidScheduleId |
| **41** | OS_ERROR_NotRunning |
| **42** | OS_ERROR_NotStopped |
| **43** | OS_ERROR_AlreadyChained |

| Value | Identifier |
|---|---|
| **44** | OS_ERROR_InvalidObjectType |
| **45** | OS_ERROR_InvalidObjectId |
| **46** | OS_ERROR_InvalidApplicationId |
| **47** | OS_ERROR_InvalidIsrId |
| **48** | OS_ERROR_InvalidMemoryRegion |
| **49** | OS_ERROR_NotChained |
| **50** | OS_ERROR_InvalidFunctionId |
| **51** | OS_ERROR_NotSyncable |
| **52** | OS_ERROR_NotImplemented |
| **53** | OS_ERROR_StackError |
| **54** | OS_ERROR_RateLimitExceeded |
| **55** | OS_ERROR_InterruptDisabled |
| **56** | OS_ERROR_ReturnFromTask |
| **57** | OS_ERROR_InsufficientStack |
| **58** | OS_ERROR_WatchdogTimeout |
| **59** | OS_ERROR_PllLockLost |
| **60** | OS_ERROR_ArithmeticTrap |
| **61** | OS_ERROR_MemoryProtection |
| **62** | OS_ERROR_NotTrusted |
| **63** | OS_ERROR_TaskResLockTimeExceeded |
| **64** | OS_ERROR_IsrResLockTimeExceeded |
| **65** | OS_ERROR_TaskIntLockTimeExceeded |
| **66** | OS_ERROR_IsrIntLockTimeExceeded |
| **67** | OS_ERROR_IncrementZero |
| **68** | OS_ERROR_DifferentCounters |
| **69** | OS_ERROR_ScheduleTableNotIdle |
| **70** | OS_ERROR_InvalidRestartOption |
| **71** | OS_ERROR_TaskAggregateTimeExceeded |
| **72** | OS_ERROR_IncorrectKernelNesting |
| **73** | OS_ERROR_KernelStackOverflow |
| **74** | OS_ERROR_TaskStackOverflow |

| Value | Identifier |
|---|---|
| **75** | OS_ERROR_IntEException |
| **76** | OS_ERROR_ExceptionInKernel |
| **77** | OS_ERROR_SysReq |
| **78** | OS_ERROR_StackOverflow |
| **79** | OS_ERROR_StackUnderflow |
| **80** | OS_ERROR_SoftBreak |
| **81** | OS_ERROR_UndefinedOpcode |
| **82** | OS_ERROR_AccessError |
| **83** | OS_ERROR_ProtectionFault |
| **84** | OS_ERROR_IllegalOperandAccess |
| **85** | OS_ERROR_UnknownException |
| **86** | OS_ERROR_UndefinedInstruction |
| **87** | OS_ERROR_Overflow |
| **88** | OS_ERROR_BrkInstruction |
| **89** | OS_ERROR_WdgTimer |
| **90** | OS_ERROR_NMI |
| **91** | OS_ERROR_RegisterBank |
| **92** | OS_ERROR_DebugInterface |
| **93** | OS_ERROR_InsufficientPageMaps |
| **94** | OS_ERROR_InsufficientHeap |
| **95** | OS_ERROR_TLB_multiple_hit |
| **96** | OS_ERROR_Userbreak |
| **97** | OS_ERROR_InstructionAddressError |
| **98** | OS_ERROR_InstructionTlbMiss |
| **99** | OS_ERROR_TlbProtectionViolation |
| **100** | OS_ERROR_GeneralIllegalInstruction |
| **101** | OS_ERROR_SlotIllegalInstruction |
| **102** | OS_ERROR_GeneralFPUDisable |
| **103** | OS_ERROR_SlotFPUDisable |
| **104** | OS_ERROR_DataAddressErrorRead |
| **105** | OS_ERROR_DataAddressErrorWrite |

| Value | Identifier |
|---|---|
| **106** | OS_ERROR_DataTlbMissRead |
| **107** | OS_ERROR_DataTlbMissWrite |
| **108** | OS_ERROR_DataTlbReadProtViolation |
| **109** | OS_ERROR_DataTlbWriteProtViolation |
| **110** | OS_ERROR_FpuException |
| **111** | OS_ERROR_InitialPageWrite |
| **112** | OS_ERROR_UnconditionalTrap |
| **113** | OS_ERROR_PrefetchAbort |
| **114** | OS_ERROR_DataAbort |
| **115** | OS_ERROR_IllegalSupervisorCall |
| **116** | OS_ERROR_IllegalInterrupt |
| **117** | OS_ERROR_NonMaskableInterrupt |
| **118** | OS_ERROR_HardFault |
| **119** | OS_ERROR_MemoryManagement |
| **120** | OS_ERROR_BusFault |
| **121** | OS_ERROR_UsageFault |
| **126** | OS_ERROR_SupervisorCall |
| **127** | OS_ERROR_DebugMonitor |
| **129** | OS_ERROR_PendingSupervisorCall |
| **130** | OS_ERROR_SystemTick |
| **131** | OS_ERROR_OscillatorFailureTrap |
| **132** | OS_ERROR_StackErrorTrap |
| **133** | OS_ERROR_AddressErrorTrap |
| **134** | OS_ERROR_MathErrorTrap |
| **135** | OS_ERROR_DMACErrorTrap |
| **136** | OS_ERROR_GenericHardTrap |
| **137** | OS_ERROR_GenericSoftTrap |
| **138** | OS_ERROR_UnknownTrap |
| **139** | OS_ERROR_SysErr |
| **140** | OS_ERROR_HVTrap |
| **141** | OS_ERROR_FETrap |

| Value | Identifier |
| --- | --- |
| **142** | OS_ERROR_Trap |
| **143** | OS_ERROR_ReservedInstruction |
| **144** | OS_ERROR_CoprocessorUnusable |
| **145** | OS_ERROR_PrivilegedInstruction |
| **146** | OS_ERROR_MisalignedAccess |
| **147** | OS_ERROR_FEINT |
| **148** | OS_ERROR_InvalidSpinlockId |
| **149** | OS_ERROR_InvalidSpinlockNesting |
| **150** | OS_ERROR_SpinlockAlreadyHeld |
| **151** | OS_ERROR_SpinlockInterferenceDeadlock |
| **152** | OS_ERROR_CoreIsDown |
| **153** | OS_ERROR_InvalidCoreId |
| **154** | OS_ERROR_ApplicationNotAccessible |
| **155** | OS_ERROR_ApplicationNotRestarting |
| **156** | OS_ERROR_HoldsLock |
| **157** | OS_ERROR_SpinlockNotOccupied |
| **158** | OS_ERROR_CallTrustedFunctionCrosscore |
| **159** | OS_ERROR_MemoryError |
| **160** | OS_ERROR_InstructionError |
| **161** | OS_ERROR_EV_MachineCheck |
| **162** | OS_ERROR_EV_TLBMissI |
| **163** | OS_ERROR_EV_TLBMissD |
| **164** | OS_ERROR_EV_ProtV |
| **165** | OS_ERROR_EV_PrivilegeV |
| **166** | OS_ERROR_EV_SWI |
| **167** | OS_ERROR_EV_Trap |
| **168** | OS_ERROR_EV_Extension |
| **169** | OS_ERROR_EV_DivZero |
| **170** | OS_ERROR_EV_DCError |
| **171** | OS_ERROR_EV_Misaligned |
| **172** | OS_ERROR_EV_VecUnit |

| Value | Identifier |
|---|---|
| **173** | OS_ERROR_ISRAlreadyDisabled |
| **174** | OS_ERROR_ISRAlreadyEnabled |
| **175** | OS_ERROR_InvalidIdleMode |

Table 4.8. List of error identifiers

# Glossary

accessing_application
: An OS application from where an OS service is called or an OS object is accessed. For example, activating a task, setting an event etc.

alarm
: An alarm triggers an action when its associated counter reaches a specified value. Optionally, the action can be triggered at regular intervals. You can use an alarm to activate a task, send an event to a task, increment another counter or call a configured function (alarm callback function).

atomic functions
: A function that performs an operation on a memory location atomically.

atomicity
: Atomicity is a feature of memory accesses. Atomic accesses are never interrupted by other concurrent accesses to the same memory location. They either execute completely or not at all. Therefore, the result of an atomic access is as if there was no contention at all.

base priority
: The base priority is the configured priority of a task or ISR.

buffering
: The act of storing data temporarily in the buffer is called buffering.

buffers
: A buffer is an area in memory that is used to store data temporarily.

ceiling priority
: The ceiling priority is the priority of the mutex or resource which will be acquired by the task or ISR.

consistency
: When memory accesses are always performed the same way, they are consistent. This especially refers to the order in which they reach memory as seen by the various observers of that memory.

counter
: A counter counts up to a predefined value and then starts again at 0. A counter is usually used to drive alarms and schedule tables.

counter; hardware
: A counter whose value is derived from a hardware unit. Hardware counters typically increment at high frequency.

counter; software
: A counter whose value is controlled by software. The `IncrementCounter()` API advances the counter by 1.

critical section
: A critical section is a part of the executing code for which execution shall not be interrupted by other task or ISR, to avoid corruption of data for example. For example, a memory region which is accessed concurrently by two tasks may interpret a wrong value if not protected by a critical section.

deadlock
: In its simplest form, deadlock occurs when a task waits for a mutex that is held by another task and the other task is simultaneously waiting for a mutex

| | |
|---|---|
| | that is held by the first task. If the waiting is performed by the task in a loop, the deadlock may be caused because another task that cannot run holds a mutex. This type of deadlock occurs in multicore systems. |
| dispatcher | The dispatcher function selects the highest priority task from the queue of tasks that are ready to run. |
| event | Events are an inter-task communication mechanism. Extended tasks can wait for events. While *waiting*, the task is not ready to run and consumes no CPU time. When an event is sent to a *waiting* task, the task becomes *ready*. When the task eventually runs, it can react to the events that it has received. |
| end of round | An end of round is a condition in a schedule table that occurs when all the expiry points of the schedule table are reached. |
| exception | An exception is a mechanism by which the hardware reports an error. The OS handles these exceptions and in most cases calls the protection hook. |
| expiry point | An expiry point is a point on a schedule table at which one or more actions are performed by the OS. The following actions are possible:<br>▶ Activate a task.<br>▶ Send an event to a task. |
| hook function | The OS can be configured to call system-wide hook functions to allow user-defined actions within the internal processing of the OS. Examples of hook functions are: the protection hook for exception handling and the error hook for handling errors such as invalid parameters or wrongly called APIs.<br>Hook functions have the following characteristics:<br>▶ They are called by the operating system.<br>▶ They run with the access rights of the operating system.<br>▶ They can run on all cores, potentially simultaneously.<br>▶ They have a higher priority than all tasks.<br>▶ They are implemented by the user with user-defined functionality.<br>▶ They have a standardized interface, but not standardized functionality. |
| hook; application-specific | Each OS application can optionally provide its own hook functions for start-up, shutdown and error. These application-specific hook functions have similar characteristics to the system-wide hook functions, except that they run with the access rights of the OS application. |
| hook; error | An error hook is a hook function that the OS calls when it detects an error in an API call. There is a global error hook (`ErrorHook()`). |

| | |
|---|---|
| hook; post-ISR | The post-ISR hook is a hook function that the OS calls just after an ISR function returns. This is intended for application-specific tracing. |
| hook; pre-ISR | The pre-ISR hook is a hook function that the OS calls just before calling an ISR function. This is intended for application-specific tracing. |
| hook; post-task | The post-task hook is a hook function that the OS calls just after a context switch from a task. This is intended for application-specific tracing. |
| hook; pre-task | The pre-task hook is a hook function that the OS calls just before a context switch to a task. This is intended for application-specific tracing. |
| hook; protection | The protection hook is a hook function that is called if a protection violation occurs; for example if the memory protection or timing protection is violated. The value returned by the protection hook determines the subsequent action of the OS.<br><br>The following actions are possible:<br><br>► Terminate the task or ISR.<br>► Terminate the OS application to which the task belongs.<br>► Restart the OS application to which the task or ISR belongs.<br>► Shut down the core. |
| hook; shutdown | A hook function that the OS calls at shutdown. |
| hook; start-up | A hook function that the OS calls at start-up, after all initialization is completed but before task scheduling commences. The start-up hook starts simultaneously on all cores. |
| interrupt | An interrupt is a mechanism that allows external hardware to notify the processor of something that needs urgent attention. |
| interrupt level | An interrupt level is a hardware/microcontroller architecture specific value which is used to group interrupts to a certain priority level (depends on hardware), using which you can configure to allow or disable interrupts of the defined group. |
| interrupt lock | An interrupt lock is used to prevent interrupts of certain groups in occurring (see interrupt level) which is used for critical section implementation. |
| internal resource | An internal resource is the resource which is exclusively used by the kernel for an internal process. |
| ISR | An interrupt service routine (ISR) is a configured function that the operating system calls when it receives an interrupt. |

| | |
|---|---|
| logical core ID | Logical cores IDs provide the software implementation with an abstract view of the physical cores available on the CPU. The logical core IDs are zero based and consecutive with the range 0 to number of cores -1 whereas the physical cores might not be so. The logical core IDs are intended to provide a hardware independent method for indexing arrays in the software implementation. By default, these logical core IDs are mapped to their physical counterparts internally in the OS. The logical core ID is the value returned by the `GetCoreID()` API. |
| memory fence | An instruction that imposes a certain order on memory accesses. A sequentially consistent memory fence prevents read and write memory accesses from being moved either way across it. |
| multitasking | A multitasking environment allows a system to be constructed as a set of independent tasks, each with its own thread of execution. Multitasking creates the appearance of many threads running concurrently. However, the kernel only interleaves their execution on the basis of a scheduling algorithm. |
| mutual exclusion (mutex) | A synchronization mechanism by which two tasks co-operatively avoid concurrent access to the same physical component. |
| OS application | An OS application is a container for OS objects. |
| OS application; non-trusted | An OS application is called a non-trusted OS application when its tasks or ISRs are not allowed to run in privileged mode i.e., have restrictions on access to memory regions that are not configured to the tasks or ISRs of the application. |
| OS application; trusted | An OS application is called a trusted OS application when its tasks or ISRs run in privileged mode i.e., have no memory access restriction. |
| OS object | An OS object is a data structure that is managed by the Os. OS objects include tasks, ISRs, counters, alarms, and schedule tables. |
| permitted context | A permitted context is the context from which any given OS service (APIs) can be called. For example, tasks, ISRs, Hook functions etc. |
| priority ceiling protocol | A priority ceiling protocol avoids unbounded priority inversion or deadlock. The priority of the task that occupies a mutex/resource is raised (ceiled) to the priority of the resource (the priority of the mutex/resource is either equal or greater than that of the task with highest priority which occupies that mutex/resource.) Unbounded priority inversion happens when a high priority task waits for a mutex/resource which is occupied by a lower priority task and that lower priority task is interrupted by a mid priority task which is not related to the mutex/resource. |
| priority inversion | Priority inversion occurs when a high-priority task waits for a mutex that is held by a lower-priority task. Uncontrolled priority inversion occurs when the |

| | |
|---|---|
| | lower priority task is preempted by one or more unrelated tasks of intermediate priority that do not use the mutex. |
| private data | Private data is the global memory region that is configured to a particular OS application or task or ISR. |
| read-modify-write | Read-modify-write is a characteristic of memory accesses that reads a memory location, modifies the obtained value, and then writes that new value back to the same memory location. |
| reorder | Memory accesses may be reordered by the compiler or hardware so that the actual order in which they reach memory is different from the order in which they appear in the program. The goal of reordering is to hide latencies of memory accesses. |
| resource | A resource is an OS object that provides mutual exclusion. In a multitasking environment, tasks typically share access to a number of physical system components such as data structures, hardware units etc. Resource objects allow tasks to coordinate access to these shared components to prevent data corruption or hardware contention.<br><br>In AutoCore OS, resources use a priority ceiling protocol that is deadlock free and prevents unbounded priority inversion. |
| remote core | The core for which the action of an OS service is intended. For example, an API call from core A to activate a task that is configured for and executes on core B. In this case, core B is the remote core. |
| run priority | The run priority is the priority attained by a task or ISR during run time. For example, a task or ISR acquires the priority of a resource during run time when the task or ISR occupies that resource. |
| scheduler | The scheduler is a kernel function which enqueues a task for execution and calls the dispatcher based on the task state. |
| schedule table | A schedule table is a predefined time schedule that is configured by the user. A schedule table:<br><br>► has a configured duration<br>► has a set of expiry points at configured intervals<br>► can be configured to repeat indefinitely |
| schedule table states | A schedule table can be in one of the following states:<br><br>► *Waiting*: A schedule table will be in *Waiting* state when it is in synchronization with an external timer. |

- *Chained*: A schedule table will be in *Chained* state when it is about to start next, after the running schedule table.
- *Running*: A schedule table will be in *Running* state when the schedule table is enqueued for processing the expiry actions/ executing the current expiry action.
- *Stopped*: A schedule table will be in *Stopped* state when all the expiry actions are completed for that schedule table and the schedule table is not cyclic.

sequential consistency

A model that characterizes a concurrent execution of a program in which all memory accesses of all concurrent threads adhere to a global total order. Furthermore, the memory accesses of a single thread happen in the same order as defined in the program; neither the compiler nor the hardware reorders memory accesses.

shutdown

The operating system shuts down when the application requests it to do so or when the OS detects a serious internal fault. When the operating system is shut down, tasks are no longer executed and interrupts are no longer accepted.

spinlock

A spinlock is an OS object similar to a resource that provides mutual exclusion of physical components between two tasks which are configured on two different cores.

stack monitoring

Stack monitoring allows the OS to detect certain types of stack overflow and to report a protection fault. In addition to the monitoring, APIs are provided to determine the deepest extent to which a stack is used. This information can be useful when estimating the amount of stack that is needed.

start-up

When power is first supplied to the hardware, or after a reset, the processor executes software that performs low-level initialization of the hardware. After the low-level initialization is complete, the operating system starts and performs its own initialization. The whole procedure is known as start-up.

static OS

In a static OS, the list of all OS objects and the possible configuration is fixed.

OS objects cannot be created dynamically. This means that the entire layout of the system, including every single object, must be determined before the system is built.

task

A task is an OS object that controls the execution of code. Every task is assigned a priority. Tasks that are ready to run are scheduled according to their priorities. Tasks of equal priority are scheduled in the order in which they become ready.

During its run-time, a task's priority can increase and decrease, but can never decrease below its statically configured priority. This static priority cannot be changed.

Tasks are activated by a call to `ActivateTask()` or by an alarm or schedule table. Tasks are terminated by calling `TerminateTask()`. The `ChainTask()` API is essentially a combination of `ActivateTask()` and `TerminateTask()`.

task; basic

A basic task does not use blocking synchronization to coordinate its activity with other tasks. These tasks do not support events and cannot go into the *Waiting* state. Once running, basic tasks keep running until one of the following occurs:

- The task terminates itself (`TerminateTask()`).
- A higher priority task is scheduled.
- An interrupt causes the processor to execute an interrupt service routine (ISR).
- The tasks application is terminated.

task; extended

An extended task may use blocking synchronization to coordinate its activity with other tasks. You can assign events to these tasks which can therefore go into the *Waiting* state. In AutoCore OS, the only system service that can block a task is `WaitEvent()`. This service blocks the task unless one more of the specified events is already set.

task; restart

The restart task of an OS application is a task that is configured to be activated when the OS application is terminated with the restart option.

The purpose of a restart task is to recover or re-initialize the application's state after a protection fault or other forced termination.

task context

The context of a task is the set of processor registers that the OS permits a task to use exclusively for its own purposes. The processor usually has only one set of these registers. The OS saves the registers for one task and loads the registers for another task when switching the tasks. This *context switch* creates an illusion of exclusivity. It is performed by the dispatcher.

The task's context may contain:

- a thread of execution, i.e. the task's program counter
- a stack for local variables and function calls
- the CPU's general-purpose registers

- floating-point registers (optional in some cases)
- kernel control structures

task states

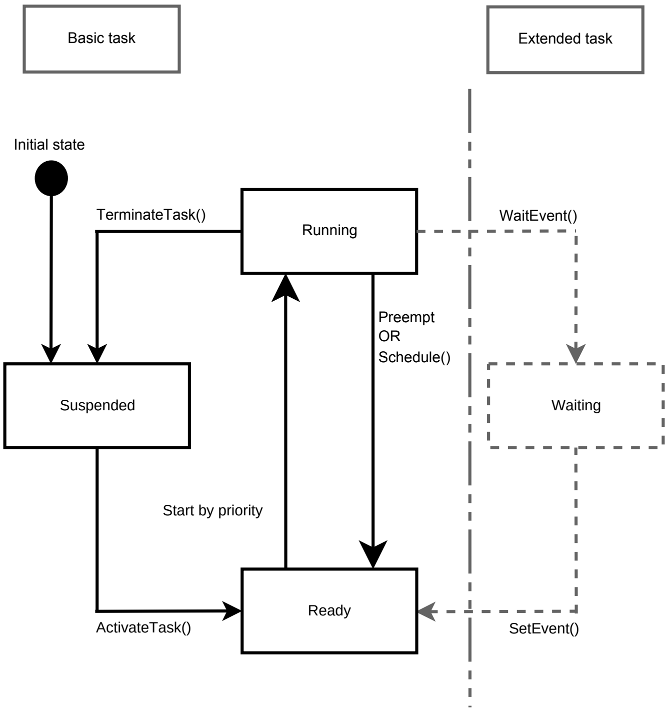


Figure 2. Task states and transitions

A task has the following states:

- *Running*: In the state *running*, a task is assigned to the CPU and the task's program code is executed on the core to which the task is assigned. At most one task per core can be in the *running* state at any time.
- *Ready*: In the state *ready*, a task is waiting for its turn to use the CPU. The dispatcher moves a *ready* task to *running* when there are no more higher-priority tasks that can execute.
- *Waiting*: This state is only used in extended tasks. The task is waiting for one or more events. A task in the *waiting* state does not consume any CPU time. The task becomes *ready* when it receives an event for which it is waiting.
- *Suspended*: A task in the state *suspended* is inactive and consumes no CPU time. This is the initial state of a task. Tasks in the *suspended* state

become *ready* when activated by means of `ActivateTask()` from another task or ISR or by an alarm or schedule table.

task transitions
: Task transitions are depicted in Figure 2, "Task states and transitions". In addition, `TerminateApplication()` changes all tasks belonging to the specified OS application to the *suspended* state, regardless of their state when the API is used. Optionally, the OS activates a configured restart task for the OS application. This API is not depicted on the state transition diagram.

total order
: A total order is a relation that is defined for any two elements of a set. In this context, the set contains the points in time at which the memory accesses reach memory. Thus, all memory accesses are ordered and you always know which accesses happen before or after a memory access.

trusted function
: A trusted function is an OS object that can be used by a non-trusted OS application to access memory regions that are protected. For example, accessing a driver.

user application
: A user application is defined and written by the user. Like any other application, it consists of set of tasks/ISRs which implements the real time functionality of a system.