ETAS

RTA-OS V6.1.1

Reference Guide
Status: Released

DRIVING | EMBEDDED EXCELLENCE

Copyright

Document: 10766-RG-6.1.1 EN-01-2021

Contents

1 Introduction

RTA-OS is a statically configurable, preemptive, real-time operating system (RTOS) for use in high-performance, resource-constrained applications. RTA-OS is a full implementation of the open-standard AUTOSAR R3.x and AUTOSAR R4.x OS specifications and includes functionality that is fully compliant and independently certified to Version 2.2.3 of the OSEK/VDX OS Standard.

This guide contains the complete technical reference for RTA-OS. The content is arranged into two parts:

- Part 1 deals with the OS kernel, describing the API, types, macros, etc. that are supported by RTA-OS and common to all target hardware.
- Part 2 deals with the PC-based tooling provided with RTA-OS. The command line interfaces, input and output file formats etc. that are common to all target hardware are described.

It is assumed you are already familiar with the *Getting Started Guide* and the *User Guide*.

For each supported target there is also an *Target/Compiler Port Guide* which provides auxiliary details for port-specific OS features.

1.1 About You

You are a trained embedded systems developer who wants to build real-time applications using a preemptive operating system. You should have knowledge of the C programming language, including the compilation, assembling and linking of C code for embedded applications with your chosen toolchain. Elementary knowledge about your target microcontroller, such as the start address, memory layout, location of peripherals and so on, is essential.

You should also be familiar with common use of the Microsoft Windows operating system, including installing software, selecting menu items, clicking buttons, navigating files and folders.

1.2 Document Conventions

The following conventions are used in this guide:

Choose **File > Open**.	Menu options appear in **bold, blue** characters.
Click **OK**.	Button labels appear in **bold** characters
Press <Enter>.	Key commands are enclosed in angle brackets.
The “Open file” dialog box appears	GUI element names, for example window titles, fields, etc. are enclosed in double quotes.

`Activate(Task1)`	Program code, header file names, C type names, C functions and API call names all appear in a `monospaced typeface`.
See Section 1.2.	Internal document hyperlinks are shown in blue letters.
	Functionality in RTA-OS that might not be portable to other implementations of AUTOSAR OS is marked with the RTA-OS icon.
	Important instructions that you must follow carefully to ensure RTA-OS works as expected are marked with a caution sign.

2 RTA-OS API calls

2.1 Guide to Descriptions

All API calls have the following structure:

Syntax

```
/* C function prototype for the API call */
ReturnValue NameOfAPICall(Parameter Type, ...)
```

Parameters

A list of parameters for each API call and their mode:

in The parameter is passed in to the call

out The parameter is passed out of the API call by passing a reference (pointer) to the parameter into the call.

inout The parameter is passed into the call and then (updated) and passed out.

Return Values

Where API calls return a `StatusType` the values of the type returned and an indication of the reason for the error/warning are listed. The build column indicates whether the value is returned for both standard and extended status builds or for extended status build only.

Description

A detailed description of the behavior of the API call.

Portability

The RTA-OS API includes six classes of API calls:

OSEK calls are those specified by the OSEK OS standard (now standardized in ISO 17356). OSEK OS calls are portable to other implementations of OSEK OS and are portable to other implementations of AUTOSAR OS.

R3.x calls are those specified by the AUTOSAR R3.x standards. AUTOSAR OS calls are portable to other implementations of AUTOSAR R3.x. The calls are portable to OSEK OS only if the call is also an OSEK OS call.

R4.x calls are those specified by the AUTOSAR R4.0, AUTOSAR R4.1, AUTOSAR R4.2, AUTOSAR R4.3, AUTOSAR R4.4 and AUTOSAR R4.5 (R19-11) standards. AUTOSAR OS calls are portable to other implementations of AUTOSAR R4.x. The calls are portable to OSEK OS only if the call is also an OSEK OS call.

MultiCore calls are those specified by the AUTOSAR R4.0 multicore OS standards. These calls are portable to other implementations of AUTOSAR R4.x. They are only available when more than one core has been configured and the target variant supports more than one core.

RTA-TRACE calls are provided by RTA-OS for controlling the RTA-TRACE run-time profiling tool. These calls are only available when RTA-TRACE support has been configured.

RTA-OS calls include all those from the other classes plus calls that provide extensions AUTOSAR OS functionality. These calls are unique to RTA-OS and are not portable to other implementations.

Example Code

```
A C code listing showing how to use the API calls
```

Calling Environment

The valid calling environment for the API call. A ✓ indicates that a call can be made in the indicated context. A ✗ indicates that the call cannot be made in the indicated context.

See Also

A list of related API calls.

2.2 ActivateTask

Activate a task.

Syntax

```
StatusType ActivateTask(
    TaskType TaskID
)
```

Parameters

Parameter	Mode	Description
TaskID	in	TaskType The task to activate.

Return Values

The call returns values of type StatusType.

Value	Build	Description
E_OK	all	No error.
E_OS_LIMIT	all	The requested activation would exceed the maximum number of queued activations specified by configuration. The requested activation is ignored.
E_OS_SYS_XCORE_QFULL	all	Only when OS option 'Asynchronous TASK activation' is active: the queue allocated for sending cross-core event notifications is full. The activation is ignored.
E_OS_CORE	all	The task belongs to a core that is stopped (in Shutdown).
E_OS_ID	extended	TaskID is not a valid TaskType.
E_OS_ACCESS	extended	TaskID is not accessible from the calling OS-Application.
E_OS_CALLEVEL	extended	Called from an invalid context (only when Service Protection is configured).
E_OS_DISABLEDINT	extended	Called while interrupts are disabled (only when Service Protection is configured).

Description

If TaskID is in the suspended state then it is transferred into the ready state.

If TaskID is in either the ready or the running state and the total number of queued activations is less than the task activation limit then the requested activation is queued.

Rescheduling behavior depends on the caller:

- if the caller is a non-preemptive task the rescheduling does not occur until the caller terminates or makes a Schedule() call.

- if the caller is a preemptive task and TaskID is higher priority then rescheduling will take place immediately.

- if the caller is a Category 2 ISR then rescheduling will not occur until after the Category 2 ISR terminates.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

Example

```
TASK(MyTask){
  ...
  ActivateTask(YourTask);
  ...
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

See Also

ActivateTaskAsyn
ChainTask
DeclareTask
GetTaskID
GetTaskState
TerminateTask

2.3 ActivateTaskAsyn

Activate a task asynchronously.

Syntax

```
void ActivateTaskAsyn(
    TaskType TaskID
)
```

Parameters

Parameter	Mode	Description
TaskID	in	TaskType The task to activate.

Description

This is a version of ActivateTask where cross-core task activation is performed by signalling an activation request to the appropriate core. The API does not block while waiting for the activation to occur.

This API does not return any error status values but the errors that can be raised by ActivateTask are still passed to ErrorHook.

The errors that are reported on the calling core are E_OS_CORE, E_OS_ID, E_OS_ACCESS, E_OS_CALLEVEL and E_OS_DISABLEDINT.

The errors that are reported on the core of the activated task are E_OS_LIMIT and E_OS_SYS_XCORE_QFULL.

The OS option 'Support ActivateTaskAsyn and SetEventAsyn' has to be set to TRUE used to cause this API to be generated because it adds a small amount of extra runtime overhead. Without this, ActivateTaskAsyn may be mapped to be the same as ActivateTask.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

Example

```
TASK(MyTask){
  ...
  ActivateTaskAsyn(YourTask);
  ...
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

See Also

ActivateTask
ChainTask
DeclareTask
GetTaskID
GetTaskState
TerminateTask

2.4 AllowAccess

Change the state of the current OS-Application from APPLICATION_RESTARTING to APPLICATION_ACCESSIBLE.

Syntax

```
StatusType AllowAccess(void)
```

Return Values

The call returns values of type StatusType.

Value	Build	Description
E_OK	all	No error.
E_OS_STATE	all	Application state is not APPLICATION_RESTARTING .
E_OS_CALLEVEL	extended	Called from an invalid context (only when Service Protection is configured).

Description

The call changes the state of the calling OS-Application to APPLICATION_ACCESSIBLE, provided that the current state was APPLICATION_RESTARTING.

This is normally used from a restart task to re-enable access to a terminated OS-Application.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✓	✓	✗

Example

```
TASK(AppRestarter) {
  AllowAccess();
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

See Also

GetApplicationState
TerminateApplication

2.5 CallAndProtectFunction

Call a time-limited OS-Application function.

Syntax

```
StatusType CallAndProtectFunction(
    TrustedFunctionIndexType FunctionIndex,
    TrustedFunctionParameterRefType FunctionParams,
    Os_TimeLimitType TimeLimit
)
```

Parameters

Parameter	Mode	Description
FunctionIndex	in	TrustedFunctionIndexType The index of the function to be called. This is the same as the name declared for the function.
FunctionParams	in	TrustedFunctionParameterRefType A pointer to the parameters of the function. Can be NULL.
TimeLimit	in	Os_TimeLimitType The maximum number of ticks of the stopwatch that the function is allowed to run for. If this value is less than 1 then no limit is applied.

Return Values

The call returns values of type StatusType.

Value	Build	Description
E_OK	all	No error.
E_OS_SERVICEID	all	FunctionIndex is not valid.
E_OS_PROTECTION_TIME	all	Function timed out AND the ProtectionHook returned PRO_TERMINATETASKISR.
E_OS_PROTECTION_LOCKED	all	Function locked a resource or interrupt for too long AND the ProtectionHook returned PRO_TERMINATETASKISR.
E_OS_PROTECTION_MEMORY	all	Function experienced a memory protection violation AND the ProtectionHook returned PRO_TERMINATETASKISR.
E_OS_PROTECTION_EXCEPTION	all	Function experienced an unexpected exception AND the ProtectionHook returned PRO_TERMINATETASKISR.
E_OS_CALLEVEL	extended	Called from an invalid context (only when Service Protection is configured).
E_OS_DISABLEDINT	extended	Called while interrupts are disabled (only when Service Protection is configured).
E_OS_ACCESS	extended	The function is part of an OS-Application on a different core

Description

This is exactly the same as CallTrustedFunction, but with the addition of a Timing Protection execution limit and the ability to recover from a memory protection violation.

If the function execution time reaches the specified limit, then ProtectionHook gets called with Reason 'E_OS_PROTECTION_TIME'. Within ProtectionHook you can choose to shutdown or kill the OS-Application. Alternatively if you return PRO_TERMINATETASKISR then you merely terminate the remaining execution of the function and CallAndProtectFunction will return with a status of E_OS_PROTECTION_TIME.

Similarly resource and interrupt lock violations can cause termination with 'E_OS_PROTECTION_LOCKED', and memory violations can cause termination with 'E_OS_PROTECTION_MEMORY' or 'E_OS_PROTECTION_EXCEPTION'.

CallAndProtectFunction can only be used if you configure the OS option 'Function Protection'.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
TASK(MyTask){
  struct func3_params {
    uint32 val1;
    uint32 val2;
  } data = {1U, 2U};
  ...
  CallAndProtectFunction(Func3, (TrustedFunctionParameterRefType)&data,
      (0.001 * OSSWTICKSPERSECOND)); /* Limit 1ms */
  ...
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

See Also

CallTrustedFunction
Os_Cbk_GetSetProtection
Os_Cbk_SetMemoryAccess
Os_TimeLimitType
ProtectionHook

2.6 CallTrustedFunction

Call an OS-Application function.

Syntax

```
StatusType CallTrustedFunction(
    TrustedFunctionIndexType FunctionIndex,
    TrustedFunctionParameterRefType FunctionParams
)
```

Parameters

Parameter	Mode	Description
FunctionIndex	in	TrustedFunctionIndexType The index of the function to be called. This is the same as the name declared for the function.
FunctionParams	in	TrustedFunctionParameterRefType A pointer to the parameters of the function. Can be NULL.

Return Values

The call returns values of type StatusType.

Value	Build	Description
E_OK	all	No error.
E_OS_SERVICEID	all	FunctionIndex is not valid.
E_OS_CALLEVEL	extended	Called from an invalid context (only when Service Protection is configured).
E_OS_DISABLEDINT	extended	Called while interrupts are disabled (only when Service Protection is configured).
E_OS_ACCESS	extended	The function is part of an OS-Application on a different core

Description

Call a function provided by an OS-application. The service is typically used to allow a non-trusted OS-Application to call a function provided by a trusted OS-Application.

RTA-OS also allows you to call a function provided by an access-restricted OS-Application (untrusted or trusted-with-protection). This means that you can have a trusted task that calls a set of functions in an access-restricted application.

When a function is called, it always runs with the access permissions of the OS-Application to which it belongs.

NOTE

For AUTOSAR version 4.0.3, the following requirement exists:

'Reaction to timing protection can be defined to terminate the OSApplication. If a task is inside CallTrustedFunction() and task rescheduling takes place within the same OSApplication, the newly running higher priority task may cause timing protection and terminate the OSApplication, thus indirectly aborting the trusted function. To avoid this, the scheduling of other Tasks which belong to the same OS-Application as the caller needs to be restricted, as well as the availability of interrupts of the same OS-Application.'

RTA-OS does not implement this behavior, because of the impact on performance at run-time.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

Example

```
TASK(MyTask) {
  struct func3_params {
    uint32 val1;
    uint32 val2;
  } data = {1U, 2U};
  ...
  CallTrustedFunction(Func1, (TrustedFunctionParameterRefType)0U);
  CallTrustedFunction(Func2, (TrustedFunctionParameterRefType)&value);
  CallTrustedFunction(Func3, (TrustedFunctionParameterRefType)&data);
  ...
}
...
void TRUSTED_Func3(TrustedFunctionIndexType FunctionIndex,
    TrustedFunctionParameterRefType FunctionParams) {
    struct func3_params *params = (struct func3_params *)FunctionParams;
    if (params->val2 < 100U) {
        ...
    }
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

See Also

CallAndProtectFunction
Os_Cbk_GetSetProtection
Os_Cbk_SetMemoryAccess

2.7 CancelAlarm

Cancel an alarm.

Syntax

```
StatusType CancelAlarm(
    AlarmType AlarmID
)
```

Parameters

Parameter	Mode	Description
AlarmID	in	AlarmType Name of the alarm to cancel.

Return Values

The call returns values of type StatusType.

Value	Build	Description
E_OK	all	No error.
E_OS_NOFUNC	all	AlarmID is not running.
E_OS_ID	extended	AlarmID is not a valid alarm.
E_OS_ACCESS	extended	AlarmID is not accessible from the calling OS-Application.
E_OS_CALLEVEL	extended	Called from an invalid context (only when Service Protection is configured).
E_OS_DISABLEDINT	extended	Called while interrupts are disabled (only when Service Protection is configured).

Description

This call cancels (stops) the specified alarm.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

Example

```
TASK(MyExtendedTask){
  ...
  CancelAlarm(TimeOutAlarm);
  ...
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

See Also

CancelAlarm
DeclareAlarm
GetAlarm
GetAlarmBase
SetRelAlarm

2.8 ChainTask

Terminate the calling task and activate another task

Syntax

```
StatusType ChainTask(
    TaskType TaskID
)
```

Parameters

Parameter	Mode	Description
TaskID	in	TaskType The task to be activated

Return Values

The call returns values of type StatusType.

Value	Build	Description
E_OS_LIMIT	all	The requested activation would exceed the maximum number of queued activations specified by configuration. The requested activation is ignored.
E_OS_CORE	all	The task belongs to a core that is stopped (in Shutdown).
E_OS_ID	extended	TaskID is not a valid TaskType.
E_OS_ACCESS	extended	TaskID is not accessible from the calling OS-Application.
E_OS_RESOURCE	extended	Calling task still holds resources.
E_OS_CALLEVEL	extended	Called at interrupt level.
E_OS_CALLEVEL	extended	Called from an invalid context (only when Service Protection is configured).
E_OS_DISABLEDINT	extended	Called while interrupts are disabled (only when Service Protection is configured).

Description

This service causes the termination of the calling task followed by the activation of TaskID. A successful call of ChainTask() does not return to the calling context.

Internal resources held by the calling task are released automatically.

Standard or linked resources held by the calling task are also released automatically and this is reported as an error condition in extended status.

A task can chain itself without affecting the queued activation count.

The ChainTask() call always causes re-scheduling. However, note that TaskID may not run immediately - there may be higher priority tasks in the ready queue that will run

in preference, for example tasks with a higher priority that share an internal resource with TaskID.

If the 'Fast Terminate' is enabled in Optimizations for RTA-OS then ChainTask() must only be called from the task entry function and the return status should not be checked (ErrorHook, when configured, will be called if there is an error). This optimization saves memory and execution time.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

Example

```
TASK(MyTask){
  ...
  ChainTask(YourTask);
  /* Any code here will not execute if the call is successful */
  ...
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✗	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

See Also

ActivateTask
DeclareTask
GetTaskID
GetTaskState
TerminateTask

2.9 CheckISRMemoryAccess

Check if a memory region is read/write/execute/stack accessible by a specified ISR.

Syntax

```
AccessType CheckISRMemoryAccess(
    ISRType ISRID,
    MemoryStartAddressType Address,
    MemorySizeType Size
)
```

Parameters

Parameter	Mode	Description
ISRID	in	ISRType The ISR for which the memory access is being checked.
Address	in	MemoryStartAddressType A pointer to the start address of the memory area (the base).
Size	in	MemorySizeType The size of the memory area in bytes (the bound).

Return Values

The call returns values of type AccessType.

Description

If ISRID represents a valid ISR, then CheckISRMemoryAccess() determines whether the inclusive range of memory addresses from Address to (Address+Size) is:

- readable by ISRID

- writeable by ISRID

- executable by ISRID

- represents stack space

If a memory access condition is not valid for the whole specified memory area, then CheckISRMemoryAccess() reports no access for the type. That is, if any address in the range is not writeable, CheckTaskMemoryAccess() reports the range is not writeable.

A call to this service results in the OS calling Os_Cbk_CheckMemoryAccess().

The result of the call is encoded in an AccessType that can be decoded using the OS-MEMORY_IS_* macros.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

Example

```
ISR(MyISR){
  if (OSMEMORY_IS_WRITEABLE(CheckISRMemoryAccess(MyISR, &datum,
      sizeof(datum)))) {
    datum = ...
  }
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✗	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✓		
		ProtectionHook	✓		

See Also

CheckTaskMemoryAccess
Os_Cbk_CheckMemoryAccess

2.10 CheckObjectAccess

Determine whether the OS-Application can access an OS Object.

Syntax

```
ObjectAccessType CheckObjectAccess(
    ApplicationType ApplID,
    ObjectTypeType ObjectType,
    Os_AnyType Object
)
```

Parameters

Parameter	Mode	Description
ApplID	in	ApplicationType The OS-Application identifier for which access is to be checked.
ObjectType	in	ObjectTypeType The type of object (OBJECT_TASK, OBJECT_ISR, OBJECT_ALARM, OBJECT_RESOURCE, OBJECT_COUNTER or OBJECT_SCHEDULETABLE).
Object	in	Os_AnyType The object identifier for which access is to be checked.

Return Values

The call returns values of type ObjectAccessType.

Value	Build	Description
NO_ACCESS	all	The OS-Application does not have access to the object, or it is an invalid Object and/or ObjectType
ACCESS	all	The OS-Application has access to the object

Description

The call returns ACCESS only if AppID can access the specified OS Object. NO_ACCESS is returned otherwise.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

Example

```
if (CheckObjectAccess(GetApplicationID(), OBJECT_TASK, Task1) == ACCESS) {
  ActivateTask(Task1);
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✗	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✓		
		ProtectionHook	✓		

See Also

CheckObjectOwnership

2.11 CheckObjectOwnership

Get the OS-Application that owns the Object.

Syntax

```
ApplicationType CheckObjectOwnership(
    ObjectTypeType ObjectType,
    Os_AnyType Object
)
```

Parameters

Parameter	Mode	Description
ObjectType	in	ObjectTypeType The type of object (OBJECT_TASK, OBJECT_ISR, OBJECT_ALARM, OBJECT_RESOURCE, OBJECT_COUNTER or OBJECT_SCHEDULETABLE).
Object	in	Os_AnyType The object whose ownership is to be checked.

Return Values

The call returns values of type ApplicationType.

Value	Build	Description
INVALID_OSAPPLICATION	all	Invalid Object and/or ObjectType

Description

The call returns the identifier of the OS-Application that owns the Object, or INVALID_OSAPPLICATION if the ObjectType and Object do not match an object that is owned by an OS-Application.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

Example

```
ApplicationType OwningApplication = CheckObjectOwnership(OBJECT_TASK,
    Task1);
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✗	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✓		
		ProtectionHook	✓		

See Also

CheckObjectAccess

2.12 CheckTaskMemoryAccess

Check if a memory region is read/write/execute/stack accessible by a specified Task.

Syntax

```
AccessType CheckTaskMemoryAccess(
    TaskType TaskID,
    MemoryStartAddressType Address,
    MemorySizeType Size
)
```

Parameters

Parameter	Mode	Description
TaskID	in	TaskType The task for which the memory access is being checked.
Address	in	MemoryStartAddressType A pointer to the start address of the memory area (the base).
Size	in	MemorySizeType The size of the memory area in bytes (the bound).

Return Values

The call returns values of type AccessType.

Description

If TaskID represents a valid task, then CheckTaskMemoryAccess() determines whether the inclusive range of memory addresses from Address to (Address+Size) is:

- readable by TaskID

- writeable by TaskID

- executable by TaskID

- represents stack space

If a memory access condition is not valid for the whole specified memory area, then CheckTaskMemoryAccess() reports no access for the type. That is, if any address in the range is not writeable, CheckTaskMemoryAccess() reports the range is not writeable.

A call to this service results in the OS calling Os_Cbk_CheckMemoryAccess().

The result of the call is encoded in an AccessType that can be decoded using the OS-MEMORY_IS_* macros.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

Example

```
TASK(MyTask){
  if (OSMEMORY_IS_WRITEABLE(CheckTaskMemoryAccess(MyTask, &datum,
      sizeof(datum)))) {
    datum = ...
  }
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✓		
		ProtectionHook	✓		

See Also

CheckISRMemoryAccess
OSMEMORY_IS_EXECUTABLE
OSMEMORY_IS_READABLE
OSMEMORY_IS_STACKSPACE
OSMEMORY_IS_WRITEABLE
Os_Cbk_CheckMemoryAccess
Os_Cbk_GetSetProtection
Os_Cbk_SetMemoryAccess

2.13 ClearEvent

Clear one (or more) events from the task's event mask.

Syntax

```
StatusType ClearEvent(
    EventMaskType Mask
)
```

Parameters

Parameter	Mode	Description
Mask	in	EventMaskType The event(s) to be cleared.

Return Values

The call returns values of type StatusType.

Value	Build	Description
E_OK	all	No error.
E_OS_ACCESS	extended	Not called from an extended task.
E_OS_CALLEVEL	extended	Called from interrupt level.
E_OS_CALLEVEL	extended	Called from an invalid context (only when Service Protection is configured).
E_OS_DISABLEDINT	extended	Called while interrupts are disabled (only when Service Protection is configured).

Description

The events of the extended task calling ClearEvent are cleared according to the event mask Mask.

Any events that are not set in the event mask remain unchanged.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

Example

```
TASK(MyExtendedTask){
  EventMaskType WhatHappened;
  while (WaitEvent(Event1 | Event2 | Event3) == E_OK ) {
    GetEvent(MyExtendedTask, &WhatHappened);
    if (WhatHappened & Event1) {
      ClearEvent(Event1);
      /* Take action on Event1 */
      ...
    } else if (WhatHappened & (Event2 | Event3) {
      ClearEvent(Event2 | Event3);
      /* Take action on Event2 or Event3 */
```

```
        ...
      }
    }
  }
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✗	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

See Also

DeclareEvent
GetEvent
SetEvent
WaitEvent

2.14 ClearPendingInterrupt

Clears the pending status for the Category 2 interrupt.

Syntax

```
StatusType ClearPendingInterrupt(
    ISRType ISRID
)
```

Parameters

Parameter	Mode	Description
ISRID	in	ISRType The ISR whose source pending status is to be cleared.

Return Values

The call returns values of type StatusType.

Value	Build	Description
E_OK	all	No error.
E_OS_ID	extended	ISRID is not a valid ISR.
E_OS_CALLEVEL	extended	Called from an invalid context (no running Task or ISR).
E_OS_ACCESS	extended	The calling OS-Application does not own the ISR (only when Service Protection is configured).

Description

Clear the pending status of the interrupt associated with ISRID. Typically will only be used when the interrupt is disabled, because otherwise the interrupt would have already fired.

Some targets have ISRs where the pending status cannot be manipulated. Where this is the case it will be documented in the Port Guide and this API will return E_OS_ID.

On some multicore targets each core has its own interrupt controller that is inaccessible from the other cores. In this case E_OS_ID will be returned if a cross-core request is made.

Note that this API is not supported for all targets. A build warning is emitted if the target does not support it.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✓	✓	✗

Example

```
TASK(MyTask){
  ...
  ClearPendingInterrupt(MyISR);
  ...
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

See Also

DisableInterruptSource
EnableInterruptSource

2.15 ControlIdle

Sets the idle mode for a core.

Syntax

```
StatusType ControlIdle(
    CoreIdType CoreID,
    IdleModeType IdleMode
)
```

Parameters

Parameter	Mode	Description
CoreID	in	CoreIdType The core ID.
IdleMode	in	IdleModeType The mode to set.

Return Values

The call returns values of type StatusType.

Value	Build	Description
E_OK	all	No error.
E_OS_ID	extended	Core is not valid.

Description

This call sets the idle mode for a core. The value that has been set for the currently running core can be read using the OS_CURRENT_IDLEMODE() macro. Note that this API was added for AUTOSAR version 4.1.x, but RTA-OS allows it to be used in earlier version too.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

Example

```
/* Set user-defined value */
ControlIdle(OS_CORE_ID_MASTER, 3);
/* Set AUTOSAR-defined value */
ControlIdle(OS_CORE_ID_1, IDLE_NO_HALT);

/* Read mode for this core */
idle_mode = OS_CURRENT_IDLEMODE();
```

See Also

OS_CURRENT_IDLEMODE

2.16 DisableAllInterrupts

Disables (masks) all interrupts for which the hardware supports disabling.

Syntax

```
void DisableAllInterrupts(void)
```

Description

This call is intended to start a (short) critical section of the code. This critical section must be finished by calling EnableAllInterrupts(). No API calls are allowed within the critical section.

The call does not support nesting. If nesting is needed for critical sections, e.g. for libraries, then SuspendAllInterrupts()/ResumeAllInterrupts() should be used.

In a multicore environment, this call only affects the interrupts on the core that it is called from.

It may be called before StartOS().

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

Example

```
TASK(MyTask){
  ...
  DisableAllInterrupts();
    /* Critical section */
    /* No RTA-OS API calls allowed */
  EnableAllInterrupts();
  ...
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✓	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✓		
		ProtectionHook	✓		

See Also

EnableAllInterrupts
ResumeAllInterrupts
ResumeOSInterrupts
SuspendAllInterrupts
SuspendOSInterrupts

2.17 DisableInterruptSource

Disables the Category 2 interrupt.

Syntax

```
StatusType DisableInterruptSource(
    ISRType ISRID
)
```

Parameters

Parameter	Mode	Description
ISRID	in	ISRType The ISR whose source is to be disabled.

Return Values

The call returns values of type StatusType.

Value	Build	Description
E_OK	all	No error.
E_OS_ID	extended	ISRID is not a valid ISR.
E_OS_CALLEVEL	extended	Called from an invalid context (no running Task or ISR).
E_OS_NOFUNC	extended	The ISR is already disabled.
E_OS_ACCESS	extended	The calling OS-Application does not own the ISR (only when Service Protection is configured).

Description

Disables the source of the interrupt associated with ISRID. When disabled, no interrupts will occur for ISRID, regardless of the CPU interrupt priority value.

Some targets have ISRs that cannot be disabled. Where this is the case it will be documented in the Port Guide and this API will return E_OS_ID if one is specified.

On some multicore targets each core has its own interrupt controller that is inaccessible from the other cores. In this case E_OS_ID will be returned if a cross-core request is made.

Note that this API is not supported for all targets. A build warning is emitted if the target does not support it.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✓	✓	✗

Example

```
TASK(MyTask){
  ...
  DisableInterruptSource(MyISR);
  ...
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

See Also

ClearPendingInterrupt
EnableInterruptSource

2.18 EnableAllInterrupts

Enables (unmasks) all interrupts.

Syntax

```
void EnableAllInterrupts(void)
```

Description

This API call marks the end of a critical section that is protected from any maskable interrupt occurring. The critical section must have been entered using the DisableAllInterrupts() call.

This call restores the state of the interrupt mask saved by DisableAllInterrupts().

In a multicore environment, this call only affects the interrupts on the core that it is called from.

It may be called before StartOS().

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

Example

```
TASK(MyTask){
  ...
  DisableAllInterrupts();
  /* Critical section */
  /* No RTA-OS API calls allowed */
  EnableAllInterrupts();
  ...
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✓	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✓		
		ProtectionHook	✓		

See Also

DisableAllInterrupts
ResumeAllInterrupts
ResumeOSInterrupts
SuspendAllInterrupts
SuspendOSInterrupts

2.19 EnableInterruptSource

Enables the Category 2 interrupt. Will also clear its pending status if ClearPending is TRUE.

Syntax

```
StatusType EnableInterruptSource(
    ISRType ISRID,
    boolean ClearPending
)
```

Parameters

Parameter	Mode	Description
ISRID	in	ISRType The ISR whose source is to be enabled.
ClearPending	in	boolean TRUE to clear the interrupt pending status.

Return Values

The call returns values of type StatusType.

Value	Build	Description
E_OK	all	No error.
E_OS_ID	extended	ISRID is not a valid ISR.
E_OS_CALLEVEL	extended	Called from an invalid context (no running Task or ISR).
E_OS_NOFUNC	extended	The ISR is already enabled.
E_OS_ACCESS	extended	The calling OS-Application does not own the ISR (only when Service Protection is configured).

Description

Enables the source of the interrupt associated with ISRID. Usually only done after previously explicitly disabling the source, because interrupts are enabled by default after StartOS().

If ClearPending is TRUE, then any pending status will be cleared before enabling the interrupt.

Some targets have ISRs that cannot be disabled. Where this is the case it will be documented in the Port Guide and this API will return E_OS_ID if one is specified.

Some targets have ISRs where the pending status cannot be manipulated. Where this is the case it will be documented in the Port Guide and this API will ignore the ClearPending argument.

On some multicore targets each core has its own interrupt controller that is inaccessible from the other cores. In this case E_OS_ID will be returned if a cross-core request is made.

Note that this API is not supported for all targets. A build warning is emitted if the target does not support it.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✓	✓	✗

Example

```
TASK(MyTask){
  ...
  EnableInterruptSource(MyISR, FALSE);
  ...
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

See Also

ClearPendingInterrupt
DisableInterruptSource

2.20 GetActiveApplicationMode

Get the currently active application mode.

Syntax

```
AppModeType GetActiveApplicationMode(void)
```

Return Values

The call returns values of type AppModeType.

Description

The call returns the currently active application mode (i.e. the value of parameter that was passed to StartOS()). The call can be used to write application-mode dependent code.

It will return OS_NOAPPMODE if the OS is not running.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

Example

```
TASK(MyTask){
  ...
  if (GetActiveApplicationMode() == DiagnosticsMode) {
    /* Send diagnostic message */
  }
  ...
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✓		
		ProtectionHook	✗		

See Also

StartOS

2.21 GetAlarm

Get the number of ticks before an alarm expires.

Syntax

```
StatusType GetAlarm(
    AlarmType AlarmID,
    TickRefType Tick
)
```

Parameters

Parameter	Mode	Description
AlarmID	in	AlarmType Name of the alarm of interest.
Tick	out	TickRefType Reference to a TickType variable.

Return Values

The call returns values of type StatusType.

Value	Build	Description
E_OK	all	No error.
E_OS_NOFUNC	all	AlarmID is not currently set.
E_OS_ILLEGAL_ADDRESS	all	Tick is an address that is not writable by the current OS-Application (only when there are access-restricted OS-Applications).
E_OS_ID	extended	AlarmID is not a valid alarm.
E_OS_ACCESS	extended	AlarmID is not accessible from the calling OS-Application.
E_OS_CALLEVEL	extended	Called from an invalid context (only when Service Protection is configured).
E_OS_DISABLEDINT	extended	Called while interrupts are disabled (only when Service Protection is configured).

Description

Returns the relative number of ticks from the point at which the call was made before the alarm AlarmID is due to expire.

Note that between making this call and evaluating the out parameter Tick the task may have been preempted and the alarm may have already expired. Exercise caution when making program decisions based on the value of Tick.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

Example

```
TASK(MyTask){
  TickType  TicksToExpiry;
  ...
  GetAlarm(MyAlarm, &TicksToExpiry);
  ...
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✓		
		ProtectionHook	✗		

See Also

CancelAlarm
DeclareAlarm
GetAlarmBase
SetAbsAlarm
SetRelAlarm

2.22 GetAlarmBase

Get properties of the counter associated with an alarm.

Syntax

```
StatusType GetAlarmBase(
    AlarmType AlarmID,
    AlarmBaseRefType Info
)
```

Parameters

Parameter	Mode	Description
AlarmID	in	AlarmType Name of the alarm of interest.
Info	out	AlarmBaseRefType Reference to an AlarmBaseType structure.

Return Values

The call returns values of type StatusType.

Value	Build	Description
E_OK	all	No error.
E_OS_ILLEGAL_ADDRESS	all	Info is an address that is not writable by the current OS-Application (only when there are access-restricted OS-Applications).
E_OS_ID	extended	AlarmID is not a valid alarm.
E_OS_ACCESS	extended	AlarmID is not accessible from the calling OS-Application.
E_OS_CALLEVEL	extended	Called from an invalid context (only when Service Protection is configured).
E_OS_DISABLEDINT	extended	Called while interrupts are disabled (only when Service Protection is configured).

Description

GetAlarmBase() reads the alarm base characteristics. These are the static properties of the counter with which AlarmID is associated.

The out parameter Info refers to a structure in which the information of data type AlarmBaseType gets stored.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

Example

```
TASK(MyTask){
  AlarmBaseType Info;
  TickType maxallowedvalue;
  TickType ticksperbase;
  TickType mincycle;

  GetAlarmBase(MyAlarm, &Info);
  maxallowedvalue = Info.maxallowedvalue;
  ticksperbase = Info.ticksperbase;
  mincycle = Info.mincycle;
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✓		
		ProtectionHook	✗		

See Also

CancelAlarm
DeclareAlarm
GetAlarm
SetAbsAlarm
SetRelAlarm

2.23 GetApplicationID

Get the identifier of the OS-Application that owns the currently running TASK, ISR or Hook.

Syntax

```
ApplicationType GetApplicationID(void)
```

Return Values

The call returns values of type `ApplicationType`.

Description

The call returns the OS-Application that owns the currently running hook, task or Category 2 ISR.

The call will return INVALID_OSAPPLICATION if no OS-Application is active.

The value returned does not change during the execution of CallTrustedFunction.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

Example

```
if (GetApplicationID() == App1) {
  ...
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✓		
		ProtectionHook	✓		

See Also

CallTrustedFunction
GetCurrentApplicationID
GetISRID
GetTaskID

2.24 GetApplicationState

Get the state of the specified OS-Application.

Syntax

```
StatusType GetApplicationState(
    ApplicationType Application,
    ApplicationStateRefType Value
)
```

Parameters

Parameter	Mode	Description
Application	in	ApplicationType The OS-Application from which the state is requested.
Value	out	ApplicationStateRefType The current state of the application.

Return Values

The call returns values of type StatusType.

Value	Build	Description
E_OK	all	No error.
E_OS_ID	all	Application is not a valid OS-Application.
E_OS_CALLEVEL	extended	Called from an invalid context (only when Service Protection is configured).

Description

The call returns the state of the specified OS-Application. The state is set to APPLICATION_ACCESSIBLE during StartOS.

It gets changed changed to APPLICATION_TERMINATED if the application gets terminated without being restarted.

It can be APPLICATION_RESTARTING following a ProtectionHook or TerminateApplication with RESTART.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✓	✓	✗

Example

```
ApplicationStateType appState;
if (GetApplicationState(App1, &appState) == E_OK) {
  switch (appState) {
    case APPLICATION_ACCESSIBLE:
      ...
    case APPLICATION_RESTARTING:
      ...
```

```
      case APPLICATION_TERMINATED:
        ...
    }
  }
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✓		
		ProtectionHook	✓		

See Also

AllowAccess
TerminateApplication

2.25 GetCoreID

GetCoreID returns the unique logical CoreID of the core on which the function is called.

Syntax

```
CoreIdType GetCoreID(void)
```

Return Values

The call returns values of type `CoreIdType`.

Description

GetCoreID returns the unique logical CoreID of the core on which the function is called.

It may be called before StartOS().

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✓	✗

Example

```
CoreIdType core_id = GetCoreID();
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✓	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✓		
		ProtectionHook	✓		

See Also

StartCore
StartNonAutosarCore

2.26 GetCounterValue

Get the value of a counter.

Syntax

```
StatusType GetCounterValue(
    CounterType CounterID,
    TickRefType Value
)
```

Parameters

Parameter	Mode	Description
CounterID	in	CounterType The counter to read.
Value	out	TickRefType The current value of the counter.

Return Values

The call returns values of type StatusType.

Value	Build	Description
E_OK	all	No error.
E_OS_ILLEGAL_ADDRESS	all	Value is an address that is not writable by the current OS-Application (only when there are access-restricted OS-Applications).
E_OS_ID	extended	CounterID is not a valid counter.
E_OS_ACCESS	extended	CounterID is not accessible from the calling OS-Application.
E_OS_CALLEVEL	extended	Called from an invalid context (only when Service Protection is configured).
E_OS_DISABLEDINT	extended	Called while interrupts are disabled (only when Service Protection is configured).

Description

Returns the current value of the specified counter CounterID in Value.

The Operating System ensures that the lowest value is zero and consecutive reads return an increasing count value until the counter wraps.

If CounterID is a hardware counter, then the user callback Os_Cbk_Now_<CounterID> will be called.

The value returned may be unreliable if this API is called from a core other than the one that the counter runs on.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

Example

```
Task(MyTask){
  TickType Value;
  ...
  GetCounterValue(MyCounter,&Value);
  ...
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

See Also

GetElapsedCounterValue
GetElapsedValue
IncrementCounter
Os_AdvanceCounter
Os_AdvanceCounter_<CounterID>
Os_IncrementCounter_<CounterID>

2.27 GetCurrentApplicationID

Get the identifier of the currently running OS-Application.

Syntax

```
ApplicationType GetCurrentApplicationID(void)
```

Return Values

The call returns values of type `ApplicationType`.

Description

The call returns the currently running OS-Application. This is the OS-Application that owns the currently running hook, task, Category 2 ISR or Trusted Function.

The call will return INVALID_OSAPPLICATION if no OS-Application is active.

The value returned is affected by the execution of CallTrustedFunction - it reports the OS-Application that owns the TrustedFunction.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

Example

```
if (GetCurrentApplicationID() == App1) {
  ...
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✓		
		ProtectionHook	✓		

See Also

CallTrustedFunction
GetApplicationID
GetISRID
GetTaskID

2.28 GetElapsedCounterValue

Returns the number of elapsed ticks since the given <Value> value via <Elapsed-Value>.

Syntax

```
StatusType GetElapsedCounterValue(
    CounterType CounterID,
    TickRefType Value,
    TickRefType ElapsedValue
)
```

Parameters

Parameter	Mode	Description
CounterID	in	CounterType Name of the counter.
Value	in	TickRefType A previous counter value.
Value	out	TickRefType The current value of the counter.
ElapsedValue	out	TickRefType The difference from the in Value.

Return Values

The call returns values of type StatusType.

Value	Build	Description
E_OK	all	No error.
E_OS_ILLEGAL_ADDRESS	all	Value or ElapsedValue is an address that is not writable by the current OS-Application (only when there are access-restricted OS-Applications).
E_OS_ID	extended	CounterID is not a valid counter.
E_OS_ACCESS	extended	CounterID is not accessible from the calling OS-Application.
E_OS_CALLEVEL	extended	Called from an invalid context (only when Service Protection is configured).
E_OS_DISABLEDINT	extended	Called while interrupts are disabled (only when Service Protection is configured).
E_OS_VALUE	extended	Value of ElapsedValue is above maxallowed-value for CounterID.

Description

Returns the number of ticks that have elapsed on the counter current since Value.

Value is updated with the current value of the counter when the call returns.

Note that the call can only return a value up to maxallowedvalue ticks in length.

If the counter has ticked more than maxallowedvalue ticks since Value then ElapsedValue will be Value modulo maxallowedvalue.

Note that this API was renamed to GetElapsedValue for AUTOSAR version 4. For compatibility reasons, RTA-OS allows you to use either name in AUTOSAR version 4 applications. Macros are used to map to the actual implementation name.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

Example

```
Task(MyTask){
  TickType Value;
  TickType ElapsedValue;
  ...
  GetCounterValue(MyCounterID,&Value);
  /* Value => current count */
  ...
  GetElapsedCounterValue(MyCounter,&Value,&ElapsedValue);
  /* ElapsedValue => ticks since original Value, Value => current count */
  ...
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

See Also

GetCounterValue
GetElapsedValue

2.29 GetElapsedValue

Returns the number of elapsed ticks since the given <Value> value via <Elapsed-Value>.

Syntax

```
StatusType GetElapsedValue(
    CounterType CounterID,
    TickRefType Value,
    TickRefType ElapsedValue
)
```

Parameters

Parameter	Mode	Description
CounterID	in	CounterType Name of the counter.
Value	in	TickRefType A previous counter value.
Value	out	TickRefType The current value of the counter.
ElapsedValue	out	TickRefType The difference from the in Value.

Return Values

The call returns values of type StatusType.

Value	Build	Description
E_OK	all	No error.
E_OS_ILLEGAL_ADDRESS	all	Value of ElapsedValue is an address that is not writable by the current OS-Application (only when there are access-restricted OS-Applications).
E_OS_ID	extended	CounterID is not a valid counter.
E_OS_ACCESS	extended	CounterID is not accessible from the calling OS-Application.
E_OS_CALLEVEL	extended	Called from an invalid context (only when Service Protection is configured).
E_OS_DISABLEDINT	extended	Called while interrupts are disabled (only when Service Protection is configured).

Description

Returns the number of ticks that have elapsed on the counter current since Value.

Value is updated with the current value of the counter when the call returns.

Note that the call can only return a value up to maxallowedvalue ticks in length.

If the counter has ticked more than maxallowedvalue ticks since Value then ElapsedValue will be Value modulo maxallowedvalue.

The value returned may be unreliable if this API is called from a core other than the one that the counter runs on.

Note that this API was called GetElapsedCounterValue for AUTOSAR version 3. For compatibility reasons, RTA-OS allows you to use either name in AUTOSAR version 4 applications. Macros are used to map to the actual implementation name.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

Example

```
Task(MyTask){
  TickType Value;
  TickType ElapsedValue;
  ...
  GetCounterValue(MyCounterID,&Value);
  /* Value => current count */
  ...
  GetElapsedValue(MyCounter,&Value,&ElapsedValue);
  /* ElapsedValue => ticks since original Value, Value => current count */
  ...
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

See Also

GetCounterValue
GetElapsedCounterValue

2.30 GetEvent

Get the state of all event bits for a task.

Syntax

```
StatusType GetEvent(
    TaskType TaskID,
    EventMaskRefType Event
)
```

Parameters

Parameter	Mode	Description
TaskID	in	TaskType Name of the Task of interest.
Event	out	EventMaskRefType Reference to an event mask.

Return Values

The call returns values of type StatusType.

Value	Build	Description
E_OK	all	No error.
E_OS_ILLEGAL_ADDRESS	all	Event is an address that is not writable by the current OS-Application (only when there are access-restricted OS-Applications).
E_OS_ID	extended	TaskID is not a valid task.
E_OS_ACCESS	extended	TaskID is not accessible from the calling OS-Application.
E_OS_ACCESS	extended	TaskID is not an extended task.
E_OS_STATE	extended	TaskID is in the suspended state.
E_OS_CALLEVEL	extended	Called from an invalid context (only when Service Protection is configured).
E_OS_DISABLEDINT	extended	Called while interrupts are disabled (only when Service Protection is configured).

Description

This call returns all events that are set for the extended task TaskID.

Note that all set events are returned, regardless of which events the task may have been waiting for.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

Example

```
TASK(MyExtendedTask){
  EventMaskType WhatHappened;
  while (WaitEvent(Event1 | Event2 | Event3) == E_OK ) {
    GetEvent(MyExtendedTask, &WhatHappened);
    if (WhatHappened & Event1) {
      ClearEvent(Event1);
      /* Take action on Event1 */
      ...
    } else if (WhatHappened & (Event2 | Event3) {
      ClearEvent(Event2 | Event3);
      /* Take action on Event2 or Event3 */
      ...
    }
  }
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✓		
		ProtectionHook	✗		

See Also

ClearEvent
DeclareEvent
SetEvent
WaitEvent

2.31 GetISRID

Get the identifier of the currently running ISR.

Syntax

```
ISRType GetISRID(void)
```

Return Values

The call returns values of type ISRType.

Description

The call returns the ID of the currently running Category2 ISR or INVALID_ISR if no ISR is running.

The main use of the call is to identify which ISR is running in hook functions.

In a multicore environment, this call only looks at the core that it is called from.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

Example

```
FUNC(void, {memclass}) ErrorHook(StatusType Error){
  ISRType ISRInError;
  TaskType TaskInError;

  ISRInError = GetISRID();
  if (ISRInError != INVALID_ISR) {
    /* Must be an ISR in error */
  } else {
    /* Maybe it's a task in error */
    GetTaskID(&TaskInError);
    ...
  }
  ...
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✗	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✓		
		ProtectionHook	✓		

See Also

GetTaskID
OS_ISRTYPE_TO_INDEX

2.32 GetNumberOfActivatedCores

Get the number of cores activated by the StartCore function.

Syntax

```
uint32 GetNumberOfActivatedCores(void)
```

Return Values

The call returns values of type uint32.

Description

GetNumberOfActivatedCores() returns the number of cores that have been activated using StartCore(). These must be cores that are configured to run the OS.

RTA-OS allows GetNumberOfActivatedCores() to be called before StartOS(), as this is the only time that its return value can be less than the number configured.

This API is only provided with multicore project configurations.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✓	✗

Example

```
OS_MAIN(){
  StatusType status;
  CoreIdType core_id = GetCoreID()
  ...
  if (core_id == OS_CORE_ID_MASTER) {
    while (GetNumberOfActivatedCores() < OS_NUM_CORES) {
      StartCore(++core_id, &status);
    }
  }
  ...
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

See Also

StartCore
StartOS

2.33 GetResource

Get (lock) a resource to enter a critical section.

Syntax

```
StatusType GetResource(
    ResourceType ResID
)
```

Parameters

Parameter	Mode	Description
ResID	in	ResourceType The resource to get.

Return Values

The call returns values of type StatusType.

Value	Build	Description
E_OK	all	No error.
E_OS_ID	extended	ResID is not a valid resource.
E_OS_ACCESS	extended	ResID is not accessible from the calling OS-Application.
E_OS_ACCESS	extended	Attempt to get a resource which is (a) already locked by another task or ISR, or (b) the priority of the calling task or interrupt routine is higher than the actual priority of ResID.
E_OS_CALLEVEL	extended	Called from an invalid context (only when Service Protection is configured).
E_OS_DISABLEDINT	extended	Called while interrupts are disabled (only when Service Protection is configured).

Description

This call enters a named critical section (the resource), protecting the code inside the critical section against concurrent access by any other tasks and ISRs that are configured to be able to access the resource.

A critical section must always be left using ReleaseResource().

Nested resource occupation is allowed, but only where the inner critical sections are completely executed within the surrounding critical section as shown in the example.

Nested occupation of the same resource is not allowed, although you can use linked resources to achieve this effect.

Calls that put the running task into any other state must not be used in critical sections. (e.g. as ChainTask(), Schedule(), TerminateTask() or WaitEvent().)

A system where Category 2 ISRs can lock a resource has slightly higher run-time overheads than one where only Tasks lock resources.

Resources can not be configured to work across different cores.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

Example

```
TASK(MyTask){
  ...
  GetResource(Outer);
    /* Outer Critical Section */
    ...
    GetResource(Inner);
      /* Inner Critical Section */
    ReleaseResource(Inner);
    ...
  ReleaseResource(Outer);
  ...
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

See Also

DeclareResource
ReleaseResource

2.34 GetScheduleTableStatus

Get the status of a schedule table.

Syntax

```
StatusType GetScheduleTableStatus(
    ScheduleTableType ScheduleTableID,
    ScheduleTableStatusRefType ScheduleStatus
)
```

Parameters

Parameter	Mode	Description
ScheduleTableID	in	ScheduleTableType Schedule table for which the status is required.
ScheduleStatus	out	ScheduleTableStatusRefType Reference to the schedule table status.

Return Values

The call returns values of type StatusType.

Value	Build	Description
E_OK	all	No error.
E_OS_ILLEGAL_ADDRESS	all	ScheduleStatus is an address that is not writable by the current OS-Application (only when there are access-restricted OS-Applications).
E_OS_ID	extended	ScheduleTableID is not a valid ScheduleTable.
E_OS_ACCESS	extended	ScheduleTableID is not accessible from the calling OS-Application.
E_OS_CALLEVEL	extended	Called from an invalid context (only when Service Protection is configured).
E_OS_DISABLEDINT	extended	Called while interrupts are disabled (only when Service Protection is configured).

Description

This call returns the status of the ScheduleTableID.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

Example

```
TASK(MyTask){
  ScheduleTableStatusType Status;

  GetScheduleTableStatus(MyScheduleTable, &Status);
  if (Status != SCHEDULETABLE_RUNNING){
      StartScheduleTableAbs(MyScheduleTable,42);
  }
  ...
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

See Also

DeclareScheduleTable
NextScheduleTable
Os_SyncScheduleTableRel
SetScheduleTableAsync
StartScheduleTableAbs
StartScheduleTableRel
StartScheduleTableSynchron
StopScheduleTable
SyncScheduleTable

2.35 GetSpinlock

GetSpinlock tries to occupy a spin-lock variable.

Syntax

```
StatusType GetSpinlock(
    SpinlockIdType SpinlockId
)
```

Parameters

Parameter	Mode	Description
SpinlockId	in	SpinlockIdType The Spinlock to occupy.

Return Values

The call returns values of type StatusType.

Value	Build	Description
E_OK	all	No error.
E_OS_ID	extended	SpinlockId does not refer to a valid Spinlock.
E_OS_STATE	extended	The Spinlock is already occupied by the calling TASK or ISR (pre AUTOSAR version 4.0.3).
E_OS_INTERFERENCE_DEADLOCK	extended	The Spinlock is already occupied by a TASK/ISR on the same core. This would cause a deadlock.
E_OS_NESTING_DEADLOCK	extended	Attempt to occupy the Spinlock while already holding a different Spinlock in a way that may cause a deadlock.
E_OS_ACCESS	extended	SpinlockId is not accessible from the calling OS-Application.
E_OS_CALLEVEL	extended	Called from an invalid context (only when Service Protection is configured).

Description

GetSpinlock is used to give one core exclusive access to a shared resource - typically writeable data.

Only one core is able to occupy a specific spinlock at a time. A request to occupy a spinlock occupied by a different core results in the caller busy-waiting until the other core releases the lock.

ReleaseSpinlock has to be used to release the lock.

Note that higher priority Tasks or ISRs can preempt code that has a spinlock. Therefore code on other cores may have to wait for the preempted code to complete before they can obtain the lock. This can have bad effects on system timing, so you may also choose to use resources or SuspendOSInterrupts/ResumeOSInterrupts around spinlocks in order to prevent such behavior.

Further, if preempting code tries to occupy a spinlock that is already occupied by a lower priority Task/ISR then the OS will not perform the operation and will raise an error, because otherwise the second lock could never succeed and would permanently busy-wait.

Care has to be taken when there is nested access to spinlocks. You must configure the order in which nested spinlocks will be locked so that the OS can enforce this and prevent deadlock.

GetSpinlock only returns when the lock is successfully taken or an error has occurred.

When the spinlock has a lock method with LOCK_ALL_INTERRUPTS semantics, this API will exit with a call to SuspendAllInterrupts() in effect and the usage restrictions that apply to SuspendAllInterrupts() also apply to GetSpinlock().

When the spinlock has a lock method with LOCK_CAT2_INTERRUPTS semantics, this API will exit with a call to SuspendOSInterrupts() in effect and the usage restrictions that apply to SuspendOSInterrupts() also apply to GetSpinlock().

When the spinlock has a lock method with LOCK_WITH_RES_SCHEDULER semantics, this API will exit with a call to GetResource(RES_SCHEDULER) in effect and the usage restrictions that apply to GetResource(RES_SCHEDULER) also apply to GetSpinlock().

When the spinlock has a lock method with NESTABLE semantics, this API will indicate a successful lock if the lock is already held by a TASK or ISR on the same core. The lock is only released by the original locker, and only then when it has made the same number of Release calls as Get calls.

When the spinlock has a lock method with COMMONABLE semantics, it can be followed by any spinlock that has no successors of its own, and is not also COMMONABLE. It can follow a normal lock without being on its list of successors.

Note that the OS configuration option 'Force spinlock error checks' can be used to cause the error checks to be done in standard as well as extended status builds.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✓	✗

Example

```
TASK(MyTask){
  ...
  GetSpinlock(Spinlock1);
  ...
  ReleaseSpinlock(Spinlock1);
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

See Also

GetResource
ReleaseSpinlock
SuspendAllInterrupts
SuspendOSInterrupts
TryToGetSpinlock
UncheckedGetSpinlock
UncheckedReleaseSpinlock
UncheckedTryToGetSpinlock

2.36 GetSpinlockInfo

Get run-time statistics about a spinlock.

Syntax

```
StatusType GetSpinlockInfo(
    SpinlockIdType SpinlockId,
    Os_SpinlockInfoRefType Info
)
```

Parameters

Parameter	Mode	Description
SpinlockId	in	SpinlockIdType The Spinlock
Info	out	Os_SpinlockInfo Reference to the Spinlock statistics.

Return Values

The call returns values of type StatusType.

Value	Build	Description
E_OK	all	No error.
E_OS_ILLEGAL_ADDRESS	all	Info is an address that is not writable by the current OS-Application (only when there are access-restricted OS-Applications).
E_OS_ID	extended	SpinlockId does not refer to a valid Spinlock.

Description

GetSpinlockInfo is an optional API that is only available when there are spinlocks in the application and the OS option 'Provide spinlock statistics' is set to true.

It is used to retrieve run-time locking statistics for the specified SpinlockId.

When the spinlock is locked at the time of the call:

- The CurrentLockTime field contains the number of ticks that the lock has been locked for. If it is not currently locked, the value will be zero.

- The CurrentLocker field contains the TaskType or ISRType TASK/ISR that took the lock. If it is not currently locked, the value will be INVALID_TASK.

- The CurrentLockingCore field contains the number of the core that has the lock. If it is not locked, the value will be OS_NUM_CORES.

The field LockAttempts contains the number of Try/Get attempts made for the spinlock (per core).

The field LockSucceeds contains the number of Try/Get attempts that resulted in the lock being taken (per core).

The field LockFails contains the number of Try attempts that failed, plus the number of retries on a Get (per core).

The field MaxLockTime shows for each core the max number of ticks that the spinlock was locked for. MaxLockTimeLocker contains the TaskType or ISRType TASK/ISR that took the lock

The field MaxSpinTime shows for each core the max number of ticks that it took to get the spinlock. This time is typically the time spent within GetSpinlock waiting for the other core to release the lock. MaxSpinTimeLocker contains the TaskType or ISRType TASK/ISR that wanted the lock

Note that the OS configuration option 'Force spinlock error checks' can be used to cause the error checks to be done in standard as well as extended status builds.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
TASK(MyTask){
  ...
  Os_SpinlockInfo Info;
  GetSpinlockInfo(Spinlock1, &Info);
  if ((TaskType)Info.CurrentLocker == MyTask) {
    ...
  }
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

See Also

GetSpinlock
Os_SpinlockInfo
ReleaseSpinlock
ResetSpinlockInfo
TryToGetSpinlock

2.37 GetTaskID

Identify the currently running task.

Syntax

```
StatusType GetTaskID(
    TaskRefType TaskID
)
```

Parameters

Parameter	Mode	Description
TaskID	out	TaskRefType A reference to the running task.

Return Values

The call returns values of type StatusType.

Value	Build	Description
E_OK	all	No error.
E_OS_ILLEGAL_ADDRESS	all	TaskID is an address that is not writable by the current OS-Application (only when there are access-restricted OS-Applications).
E_OS_CALLEVEL	extended	Called from an invalid context (only when Service Protection is configured).
E_OS_DISABLEDINT	extended	Called while interrupts are disabled (only when Service Protection is configured).

Description

The call returns a reference to the currently running Task.

If the call is made from a task, then it will return the identifier of that task.

If the call is made from an ISR, then it will return the identifier of the task that was running when the interrupt occurred.

The main use of the call is to identify which task is running in hook functions.

In a multicore environment, this call only looks at the core that it is called from.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

Example

```
FUNC(void, {memclass}) ErrorHook(StatusType Error){
  TaskType TaskInError;
  GetTaskID(&TaskInError);
  if (TaskInError == INVALID_TASK) {
    /* Must be an ISR in error */
  } else if (TaskInError == MyTask) {
    /* Do something */
  }
  ...
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✓		
		ProtectionHook	✓		

See Also

DeclareTask
GetISRID
GetTaskID
GetTaskState
OS_TASKTYPE_TO_INDEX
TerminateTask

2.38 GetTaskState

Get the current state (suspended, ready, running, waiting) of a specified task.

Syntax

```
StatusType GetTaskState(
    TaskType TaskID,
    TaskStateRefType State
)
```

Parameters

Parameter	Mode	Description
TaskID	in	TaskType The task of interest.
State	out	TaskStateRefType Reference to the task state.

Return Values

The call returns values of type StatusType.

Value	Build	Description
E_OK	all	No error.
E_OS_ILLEGAL_ADDRESS	all	State is an address that is not writable by the current OS-Application (only when there are access-restricted OS-Applications).
E_OS_ID	extended	TaskID is not a valid TaskType.
E_OS_ACCESS	extended	TaskID is not accessible from the calling OS-Application.
E_OS_CALLEVEL	extended	Called from an invalid context (only when Service Protection is configured).
E_OS_DISABLEDINT	extended	Called while interrupts are disabled (only when Service Protection is configured).

Description

The call returns the state of the task at the point GetTaskState() was called.

The main use of this API is to check that an extended task is not in the suspended state before setting an event.

A task that is preempted by an ISR remains in the running state.

Note that when called from a preemptive task or from an ISR the state may already be incorrect at the time it is evaluated because preemption may have occurred between the call returning and the result being evaluated.

In a multicore environment, the state of all tasks can be examined - even where the task is assigned to a different core.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

Example

```
TASK(MyTask){
  ...
  TaskStateType CurrentState;
  ...
  GetTaskState(YourTask, &CurrentState);
  switch (CurrrentState) {
    case SUSPENDED:
      /* YourTask is suspended */
    case READY:
      /* YourTask is ready to run */
    case WAITING:
      /* YourTask is waiting (for an event) */
    case RUNNING:
      /* YourTask is running. Not possible as MyTask must be running to
          make the call */
  }
  ...
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✓		
		ProtectionHook	✗		

See Also

DeclareTask
GetTaskID
GetTaskState
TerminateTask

2.39 GetVersionInfo

Get the version information for the OS

Syntax

```
void GetVersionInfo(
    Std_VersionInfoType *versioninfo
)
```

Parameters

Parameter	Mode	Description
versioninfo	out	Std_VersionInfoType Pointer to variable used to get the OS Version information

Description

The content of the structure 'Std_VersionInfoType' is defined in Std_Types.h

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

Example

```
Std_VersionInfoType ver;
GetVersionInfo(&ver);
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✓		
		ProtectionHook	✓		

See Also

None.

2.40 IncrementCounter

Increment a software counter.

Syntax

```
StatusType IncrementCounter(
    CounterType CounterID
)
```

Parameters

Parameter	Mode	Description
CounterID	in	CounterType Name of the counter to increment.

Return Values

The call returns values of type StatusType.

Value	Build	Description
E_OK	all	No error.
E_OS_ID	extended	CounterID is not a valid Counter or is not a software counter.
E_OS_ACCESS	extended	CounterID is not accessible from the calling OS-Application.
E_OS_CALLEVEL	extended	Called from an invalid context (only when Service Protection is configured).
E_OS_DISABLEDINT	extended	Called while interrupts are disabled (only when Service Protection is configured).
E_OS_CORE	extended	Called from a core other than the core that owns CounterID

Description

This call increments (adds one to) CounterID. CounterID must be a software counter.

If any alarms on the counter are triggered by the increment then the alarm actions will be executed before the call returns.

Note that if an error occurs during the expiry of an alarm (for example, a task activation raises E_OS_LIMIT), the error hook(s) are called for each error that occurs.

However, the IncrementCounter() service itself will still return E_OK.

The API call may cause re-scheduling to take place.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

Example

```
ISR(MillisecondTimerInterrupt){
  ...
  IncrementCounter(MillisecondCounter);
  ...
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

See Also

Os_AdvanceCounter
Os_AdvanceCounter_<CounterID>
Os_IncrementCounter_<CounterID>

2.41 ModifyPeripheral16

Modifies the 16-bit value at Address by ANDing it with the Clearmask and ORing it with the Setmask.

Syntax

```
StatusType ModifyPeripheral16(
    AreaIdType Area,
    uint16* Address,
    uint16 Clearmask,
    uint16 Setmask
)
```

Parameters

Parameter	Mode	Description
Area	in	AreaIdType ID of the PeripheralArea.
Address	in	uint16 * Address of the value to modify.
Clearmask	in	uint16 Value to bitwise AND with the addressed value.
Setmask	in	uint16 Value to bitwise OR with the addressed value.

Return Values

The call returns values of type StatusType.

Value	Build	Description
E_OK	all	No error.
E_OS_ID	extended	Area does not match the ID of a configured PeripheralArea.
E_OS_VALUE	extended	Address does not belong to the specified area.
E_OS_CALLEVEL	extended	Called from an invalid context (no running Task or ISR).
E_OS_ACCESS	extended	The area is not accessible from the calling OS-Application.

Description

Reads a 16 bit value from the specified address, then does a bitwise AND with the Clearmask and a bitwise OR with the Setmask, before writing the value back.

Typically used to allow code in untrusted OS Applications to modify data from areas of memory that would otherwise be inaccessible because of memory protection settings. The read is done from a trusted context.

Peripheral Areas must be declared in the configuration. Their legal address range must be specified, and also the OS-Applications that can access them.

All 16 bits of the modified value must be within the configured address range.

The OS does not prevent two different cores from accessing peripherals at the same time.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✓	✓	✗

Example

```
TASK(MyTask){
  ...
  ModifyPeripheral16(MyArea, LED_STATUS_PORT, 0xfffe, 0x0001);
  ...
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

See Also

ModifyPeripheral32
ModifyPeripheral8
ReadPeripheral16
ReadPeripheral32
ReadPeripheral8
WritePeripheral16
WritePeripheral32
WritePeripheral8

2.42 ModifyPeripheral32

Modifies the 32-bit value at Address by ANDing it with the Clearmask and ORing it with the Setmask.

Syntax

```
StatusType ModifyPeripheral32(
    AreaIdType Area,
    uint32* Address,
    uint32 Clearmask,
    uint32 Setmask
)
```

Parameters

Parameter	Mode	Description
Area	in	AreaIdType ID of the PeripheralArea.
Address	in	uint32 * Address of the value to modify.
Clearmask	in	uint32 Value to bitwise AND with the addressed value.
Setmask	in	uint32 Value to bitwise OR with the addressed value.

Return Values

The call returns values of type StatusType.

Value	Build	Description
E_OK	all	No error.
E_OS_ID	extended	Area does not match the ID of a configured PeripheralArea.
E_OS_VALUE	extended	Address does not belong to the specified area.
E_OS_CALLEVEL	extended	Called from an invalid context (no running Task or ISR).
E_OS_ACCESS	extended	The area is not accessible from the calling OS-Application.

Description

Reads a 32 bit value from the specified address, then does a bitwise AND with the Clearmask and a bitwise OR with the Setmask, before writing the value back.

Typically used to allow code in untrusted OS Applications to modify data from areas of memory that would otherwise be inaccessible because of memory protection settings. The read is done from a trusted context.

Peripheral Areas must be declared in the configuration. Their legal address range must be specified, and also the OS-Applications that can access them.

All 32 bits of the modified value must be within the configured address range.

The OS does not prevent two different cores from accessing peripherals at the same time.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✓	✓	✗

Example

```
TASK(MyTask){
  ...
  ModifyPeripheral32(MyArea, LED_STATUS_PORT, 0xfffe, 0x0001);
  ...
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

See Also

ModifyPeripheral16
ModifyPeripheral8
ReadPeripheral16
ReadPeripheral32
ReadPeripheral8
WritePeripheral16
WritePeripheral32
WritePeripheral8

2.43 ModifyPeripheral8

Modifies the 8-bit value at Address by ANDing it with the Clearmask and ORing it with the Setmask.

Syntax

```
StatusType ModifyPeripheral8(
    AreaIdType Area,
    uint8* Address,
    uint8 Clearmask,
    uint8 Setmask
)
```

Parameters

Parameter	Mode	Description
Area	in	AreaIdType ID of the PeripheralArea.
Address	in	uint8 * Address of the value to modify.
Clearmask	in	uint8 Value to bitwise AND with the addressed value.
Setmask	in	uint8 Value to bitwise OR with the addressed value.

Return Values

The call returns values of type StatusType.

Value	Build	Description
E_OK	all	No error.
E_OS_ID	extended	Area does not match the ID of a configured PeripheralArea.
E_OS_VALUE	extended	Address does not belong to the specified area.
E_OS_CALLEVEL	extended	Called from an invalid context (no running Task or ISR).
E_OS_ACCESS	extended	The area is not accessible from the calling OS-Application.

Description

Reads an 8 bit value from the specified address, then does a bitwise AND with the Clearmask and a bitwise OR with the Setmask, before writing the value back.

Typically used to allow code in untrusted OS Applications to modify data from areas of memory that would otherwise be inaccessible because of memory protection settings. The read is done from a trusted context.

Peripheral Areas must be declared in the configuration. Their legal address range must be specified, and also the OS-Applications that can access them.

The OS does not prevent two different cores from accessing peripherals at the same time.

Note: Not supported of targets that use 16-bit char types.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✓	✓	✗

Example

```
TASK(MyTask){
  ...
  ModifyPeripheral8(MyArea, LED_STATUS_PORT, 0xfe, 0x01);
  ...
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

See Also

ModifyPeripheral16
ModifyPeripheral32
ReadPeripheral16
ReadPeripheral32
ReadPeripheral8
WritePeripheral16
WritePeripheral32
WritePeripheral8

2.44 NextScheduleTable

Change the execution pattern from one ScheduleTable to another.

Syntax

```
StatusType NextScheduleTable(
    ScheduleTableType ScheduleTableID_From,
    ScheduleTableType ScheduleTableID_To
)
```

Parameters

Parameter	Mode	Description
ScheduleTableID_From	in	ScheduleTableType Schedule table to switch from.
ScheduleTableID_To	in	ScheduleTableType Schedule table to switch into.

Return Values

The call returns values of type StatusType.

Value	Build	Description
E_OK	all	No error.
E_OS_NOFUNC	all	ScheduleTableID_From is not started.
E_OS_ID	extended	ScheduleTableID_From or ScheduleTableID_To is not a valid ScheduleTable.
E_OS_ID	extended	ScheduleTableID_From and ScheduleTableID_To have different Synchronization strategies.
E_OS_ACCESS	extended	ScheduleTableID_From or ScheduleTableID_To is not accessible from the calling OS-Application.
E_OS_STATE	extended	ScheduleTableID_To is already started or nexted.
E_OS_CALLEVEL	extended	Called from an invalid context (only when Service Protection is configured).
E_OS_DISABLEDINT	extended	Called while interrupts are disabled (only when Service Protection is configured).

Description

This call starts the processing of schedule table of ScheduleTableID_To ScheduleTableID_From.FinalDelay ticks after the Final Expiry Point on ScheduleTableID_From has been processed.

The Initial Expiry Point on ScheduleTableID_To is processed ScheduleTable_To.InitialOffset ticks after the start of ScheduleTableID_To.

If ScheduleTableID_From already has a 'nexted' schedule table then ScheduleTableID_To replaces the previous 'nexted' schedule table and that previous table is set to state SCHEDULETABLE_STOPPED.

If either schedule table is not valid or they are driven by different counters then the states of both tables remain unchanged.

The synchronization strategy of ScheduleTableID_To comes into effect when the OS processes the first expiry point of ScheduleTableID_To.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

Example

```
TASK(MyTask){
    /* Stop MyScheduleTable at the end and start YourScheduleTable */
    NextScheduleTableAbs(MyScheduleTable, YourScheduleTable);
    ...
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

See Also

DeclareScheduleTable
GetScheduleTableStatus
Os_SyncScheduleTableRel
SetScheduleTableAsync
StartScheduleTableAbs
StartScheduleTableRel
StartScheduleTableSynchron
StopScheduleTable
SyncScheduleTable

2.45 Os_AddDelayedTasks

Add one or more tasks to the set of tasks that are subject to delayed execution for the calling core.

Syntax

```
StatusType Os_AddDelayedTasks(
    Os_TasksetType Taskset
)
```

Parameters

Parameter	Mode	Description
Taskset	in	Os_TasksetType The set of tasks to be added.

Return Values

The call returns values of type StatusType.

Value	Build	Description
E_OK	all	No error.
E_OS_ACCESS	extended	Delayed tasks cannot be changed by untrusted code
E_OS_ID	extended	A task is being added that has the same priority as another task on the same core. You have to add all of the tasks that share the priority.

Description

When delayed task execution is configured for the project, you can tell RTA-OS to delay execution of a set of tasks. These tasks can be activated, but will not actually run until they are removed from the set.

Os_AddDelayedTasks is used to add tasks to the existing set of delayed tasks. Adding a task multiple times is permissible but has no effect.

Note that where tasks share priorities on a specific core, you must either add all of the tasks sharing a priority or none.

You must only add tasks that run on the calling core.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
TASK(LowPrioTask){
  Os_AddDelayedTasks(TASK_MASK(Task1) | TASK_MASK(Task3));
  ... /* Tasks 1 and 3 can be activated, but will not run */

  Os_SetDelayedTasks(OS_NO_TASKS);
  ... /* Tasks 1 and 3 can run */
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✓		

See Also

OS_ADD_TASK
OS_NO_TASKS
Os_RemoveDelayedTasks
Os_SetDelayedTasks
Os_TasksetType
TASK_MASK

2.46 Os_AdvanceCounter

Inform the OS that a hardware counter has reached the previously programmed value.

Syntax

```
StatusType Os_AdvanceCounter(
    CounterType CounterID
)
```

Parameters

Parameter	Mode	Description
CounterID	in	CounterType Name of the counter that has reached a programmed value.

Return Values

The call returns values of type StatusType.

Value	Build	Description
E_OK	all	No error.
E_OS_ID	extended	CounterID is not a valid Counter or is not a hardware counter.
E_OS_ACCESS	extended	CounterID is not accessible from the calling OS-Application.
E_OS_STATE	extended	CounterID is not running.
E_OS_CALLEVEL	extended	Called from an invalid context (only when Service Protection is configured).
E_OS_DISABLEDINT	extended	Called while interrupts are disabled (only when Service Protection is configured).

Description

This call tells the OS that the counter value has matched the value most recently set using the Os_Cbk_Set_<CounterID> callback. It must be called on the core that owns the counter.

Typically it gets called from within an interrupt that occurs because the counter match value has been reached.

It causes the OS to process any alarm or expiry point actions that are due. It then either sets a new match value (via Os_Cbk_Set_<CounterID>) or cancels the counter (via Os_Cbk_Cancel_<CounterID>).

Note that it is possible for the new counter match value to be reached before leaving any interrupt that is being used to drive the counter. It is important that this occurrence is not missed because otherwise the counter will not be awoken again until a complete wrap of the underlying hardware counter value has occurred.

On some hardware platforms no special action is needed because the interrupt will simply get reasserted when the existing instance exits.

On other platforms, the interrupt has to be reasserted in software or, where this is not possible, the code must loop as shown in the example. In either case great care has to be taken to avoid missing matches that occur while the driver is executing.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
/* For systems where the interrupt will be re-entered automatically if the
    match occurs before leaving the ISR: */
ISR(SimpleCounterDriver){
  Os_AdvanceCounter(MyHWCounter);
}
/* For systems where the software can force the interrupt to get
    re-entered if the match occurs before leaving the ISR: */
ISR(RetriggeringCounterDriver){
  Os_CounterStatusRefType CurrentState;
  Os_AdvanceCounter(MyHWCounter);
  Os_Cbk_State_MyHWCounter(&CurrentState);
  if (CurrentState.Running && CurrentState.Pending) {
    /* Retrigger this interrupt */
  }
}
/* For systems where the software has to loop if the match occurs before
    leaving the ISR: */
ISR(LoopingCounterDriver){
  Os_CounterStatusRefType CurrentState;
  do {
    Os_AdvanceCounter(MyHWCounter);
    Os_Cbk_State_MyHWCounter(&CurrentState);
  } while (CurrentState.Running && CurrentState.Pending);
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

See Also

IncrementCounter
Os_AdvanceCounter_<CounterID>
Os_Cbk_Cancel_<CounterID>
Os_Cbk_Set_<CounterID>
Os_Cbk_State_<CounterID>
Os_IncrementCounter_<CounterID>

2.47 Os_AdvanceCounter_<CounterID>

Inform the OS that a hardware counter has reached a programmed value.

Syntax

```
StatusType Os_AdvanceCounter_CounterID(void)
```

Return Values

The call returns values of type StatusType.

Value	Build	Description
E_OK	all	No error.
E_OS_STATE	extended	CounterID is not running.

Description

This call has the same behavior as Os_AdvanceCounter(CounterID) but is customized for a specific counter. This makes the call faster and more suitable for use in interrupt handlers.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
/* For systems where the interrupt will be re-entered automatically if the
    match occurs before leaving the ISR: */
ISR(SimpleCounterDriver){
  Os_AdvanceCounter_MyHWCounter();
}
/* For systems where the software can force the interrupt to get
    re-entered if the match occurs before leaving the ISR: */
ISR(RetriggeringCounterDriver){
  Os_CounterStatusRefType CurrentState;
  Os_AdvanceCounter_MyHWCounter();
  Os_Cbk_State_MyHWCounter(&CurrentState);
  if (CurrentState.Running && CurrentState.Pending) {
    /* Retrigger this interrupt */
  }
}
/* For systems where the software has to loop if the match occurs before
    leaving the ISR: */
ISR(LoopingCounterDriver){
  Os_CounterStatusRefType CurrentState;
  do {
    Os_AdvanceCounter_MyHWCounter();
    Os_Cbk_State_MyHWCounter(&CurrentState);
  } while (CurrentState.Running && CurrentState.Pending);
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

See Also

IncrementCounter
Os_AdvanceCounter
Os_Cbk_Cancel_<CounterID>
Os_Cbk_Set_<CounterID>
Os_Cbk_State_<CounterID>
Os_IncrementCounter_<CounterID>

2.48 Os_GetCurrentIMask

Get the current interrupt priority/mask.

Syntax

```
Os_imaskType Os_GetCurrentIMask(void)
```

Return Values

The call returns values of type Os_imaskType.

Description

Returns the current interrupt priority (or interrupt mask for some targets).

It may not be called before StartOS().

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
Os_imaskType imask = Os_GetCurrentIMask();
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✓	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✗		
		ErrorHook	✓		
		ProtectionHook	✓		

See Also

OS_IMASK_FOR_<TaskOrISR>
OS_IMASK_FOR_ISR
OS_IMASK_FOR_TASK
Os_GetCurrentTPL

2.49 Os_GetCurrentTPL

Get the current internal task priority.

Syntax

```
uint32 Os_GetCurrentTPL(void)
```

Return Values

The call returns values of type uint32.

Description

Returns the currently running internal TASK priority.

Each TASK on a core has a unique TPL based on its internal priority. The number is not the same as the declared priority, but the ordering will be similar.

This will be OS_INVALID_TPL when no TASK is running.

This may be higher than the OS_TPL value for the running TASK if it has locked a resource or shared a priority with another TASK .

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
uint32 tpl = Os_GetCurrentTPL();
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✗		
		ErrorHook	✓		
		ProtectionHook	✓		

See Also

OS_INVALID_TPL
OS_TPL_FOR_<Task>
OS_TPL_FOR_TASK
Os_GetCurrentIMask

2.50 Os_GetElapsedTime

Get the cumulative execution time consumed by the calling TASK/ISR or Idle callback.

Syntax

```
Os_StopwatchTickType Os_GetElapsedTime(void)
```

Return Values

The call returns values of type `Os_StopwatchTickType`.

Description

Returns the cumulative time spent running the calling TASK, ISR or Idle callback since StartOS() (or an explicit reset of the accumulated time).

If the total time exceeds the range of Os_StopwatchTickType the reported value will saturate at the maximum Os_StopwatchTickType value (typically 0xffffffff).

The time reported includes the time accrued in the current invocation of the TASK/ISR/Idle callback.

The value is valid in PreTaskHook(), Os_Cbk_TaskStart() and Os_Cbk_ISRStart().

Although you can call this API in PostTaskHook(), Os_Cbk_TaskTerminated(), Os_Cbk_TaskEnd() and Os_Cbk_ISREnd(), you will get a value greater than the value that is used to compute the total execution time, because the total is computed before these callbacks run.

Time Monitoring and Elapsed Time Recording must be enabled for this API to give meaningful results. It returns zero if time monitoring is not enabled.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
Os_StopwatchTickType Total_Idle_Time;
Os_StopwatchTickType Total_Task_Time;
Os_StopwatchTickType Total_ISR_Time;

FUNC(boolean, {memclass}) Os_Cbk_Idle(void) {
  Total_Idle_Time = Os_GetElapsedTime();
}

TASK(MyTask){
  Total_Task_Time = Os_GetElapsedTime();
}

ISR(MyISR){
  Total_ISR_Time = Os_GetElapsedTime();
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✓		
		ProtectionHook	✓		

See Also

Os_GetISRElapsedTime
Os_GetIdleElapsedTime
Os_GetTaskElapsedTime
Os_ResetISRElapsedTime
Os_ResetIdleElapsedTime
Os_ResetTaskElapsedTime

2.51 Os_GetExecutionTime

Get the execution time consumed by the calling Task/ISR.

Syntax

```
Os_StopwatchTickType Os_GetExecutionTime(void)
```

Return Values

The call returns values of type Os_StopwatchTickType.

Description

Returns the net execution time consumed (i.e. excluding all preemptions) since the start of the Task or ISR.

In the case of an extended task, execution time restarts on return from a WaitEvent() call.

The value is valid in PreTaskHook(), Os_Cbk_TaskStart() and Os_Cbk_ISRStart().

Although you can call this API in PostTaskHook(), Os_Cbk_TaskTerminated(), Os_Cbk_TaskEnd() and Os_Cbk_ISREnd(), you will get a value greater than the value that is used to compute the maximum execution time, because the maximum is computed before these callbacks run.

If the value overflows, then the returned value will be the wrapped value.

Time monitoring must be enabled for this API to give meaningful results. It returns zero if time monitoring is not enabled.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
TASK(MyTask){
  Os_StopwatchTickType Start, Finish, Used, APICallCorrection;
  Start = GetExecutionTime();
  Finish = GetExecutionTime();
  APICallCorrection = Finish - Start;  /* Get time for GetExecutionTime()
      call itself. */
  Start = GetExecutionTime();
  Call3rdPartyLibraryFunction();       /* Measure 3rd Party Library Code
      Execution Time */
  Finish = GetExecutionTime();
  Used = Finish - Start - APICallCorrection; /* Calculate the amount of
      time used. */
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

See Also

OS_ISRTYPE_TO_INDEX
OS_TASKTYPE_TO_INDEX
Os_Cbk_ISREnd
Os_Cbk_ISRStart
Os_Cbk_TaskEnd
Os_Cbk_TaskStart
Os_Cbk_TaskTerminated
Os_GetISRMaxExecutionTime
Os_GetTaskMaxExecutionTime
Os_ResetISRMaxExecutionTime
Os_ResetTaskMaxExecutionTime

2.52 Os_GetISRElapsedTime

Get the cumulative execution time consumed by the specified ISR.

Syntax

```
Os_StopwatchTickType Os_GetISRElapsedTime(
    ISRType ISRID
)
```

Parameters

Parameter	Mode	Description
ISRID	in	ISRType The ISR of interest.

Return Values

The call returns values of type Os_StopwatchTickType.

Description

Returns the cumulative time spent running the specified category 2 ISR since StartOS() (or an explicit reset of the accumulated time).

If the total time exceeds the range of Os_StopwatchTickType the reported value will wrap around and appear to get smaller.

The elapsed time gets updated when an ISR finishes or is pre-empted. Calling Os_GetISRElapsedTime before the ISR completes may not account for some of the execution time of the current invocation.

Zero is returned if the ISRID is not valid.

Time Monitoring and Elapsed Time Recording must be enabled for this API to give meaningful results. It returns zero if time monitoring is not enabled.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
Os_StopwatchTickType Total_ISR_Time;
Total_ISR_Time = Os_GetISRMaxExecutionTime(MyISR);
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✓		
		ProtectionHook	✓		

See Also

Os_GetElapsedTime
Os_GetIdleElapsedTime
Os_GetTaskElapsedTime
Os_ResetISRElapsedTime
Os_ResetIdleElapsedTime
Os_ResetTaskElapsedTime

2.53 Os_GetISRMaxExecutionTime

Get the longest observed execution time consumed by an ISR.

Syntax

```
Os_StopwatchTickType Os_GetISRMaxExecutionTime(
    ISRType ISRID
)
```

Parameters

Parameter	Mode	Description
ISRID	in	ISRType The ISR of interest.

Return Values

The call returns values of type Os_StopwatchTickType.

Description

Returns the maximum observed execution time for the Category 2 ISR identified by ISRID.

This maximum value is over all complete invocations of the Category 2 ISR that have completed since the previous call to ResetISRMaxExecutionTime() for that Category 2 ISR or to StartOS().

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
TASK(LoggingTask){
  Os_StopwatchTickType ExecutionTimes[MAXISRS];
  ...
  ExecutionTimes[0] = GetISRMaxExecutionTime(ISR1);
  ExecutionTimes[1] = GetISRMaxExecutionTime(ISR2);
  ...
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✓		
		ProtectionHook	✗		

See Also

OS_ISRTYPE_TO_INDEX
Os_GetExecutionTime
Os_GetTaskMaxExecutionTime
Os_ResetISRMaxExecutionTime
Os_ResetTaskMaxExecutionTime

2.54 Os_GetISRMaxStackUsage

Get the maximum observed stack usage of an ISR.

Syntax

```
Os_StackSizeType Os_GetISRMaxStackUsage(
    ISRType ISRID
)
```

Parameters

Parameter	Mode	Description
ISRID	in	ISRType The ISR of interest.

Return Values

The call returns values of type Os_StackSizeType.

Description

Returns the maximum observed stack usage for the Category 2 ISR identified by ISRID.

This maximum value is over all invocations of the Category 2 ISR since the previous call to ResetISRMaxStackUsage() for that Category 2 ISR or to StartOS().

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
TASK(LoggingTask){
  Os_StackSizeType StackUsages[MAXISRS];
  ...
  StackUsages[0] = GetISRMaxStackUsage(ISR1);
  StackUsages[1] = GetISRMaxStackUsage(ISR2);
  ...
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✓		
		ProtectionHook	✗		

See Also

OS_ISRTYPE_TO_INDEX
Os_GetStackUsage
Os_GetTaskMaxStackUsage
Os_ResetISRMaxStackUsage
Os_ResetTaskMaxStackUsage

2.55 Os_GetIdleElapsedTime

Get the cumulative execution time consumed when idle on the specified core.

Syntax

```
Os_StopwatchTickType Os_GetIdleElapsedTime(
    Os_IdleType IdleID
)
```

Parameters

Parameter	Mode	Description
OS_CORE_CURRENT	in	Os_IdleType Idle on the calling core.
OS_CORE_ID_0	in	Os_IdleType Idle on Core 0
OS_CORE_ID_n	in	Os_IdleType Idle on Core n

Return Values

The call returns values of type Os_StopwatchTickType.

Description

Returns the cumulative time spent 'idle' since StartOS() (or an explicit reset of the accumulated time).

If the total time exceeds the range of Os_StopwatchTickType the reported value will wrap around and appear to get smaller.

The time reported includes the time accrued in the current invocation of the Idle callback if this is on the calling core.

Zero is returned if the IdleID is not valid.

Time Monitoring and Elapsed Time Recording must be enabled for this API to give meaningful results. It returns zero if time monitoring is not enabled.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
Os_StopwatchTickType Total_Idle_Time;
Total_Idle_Time = Os_IdleType IdleID(OS_CORE_CURRENT);
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✓		
		ProtectionHook	✓		

See Also

Os_GetElapsedTime
Os_GetISRElapsedTime
Os_GetTaskElapsedTime
Os_ResetISRElapsedTime
Os_ResetIdleElapsedTime
Os_ResetTaskElapsedTime

2.56 Os_GetStackSize

Get the difference between 2 stack values.

Syntax

```
Os_StackSizeType Os_GetStackSize(
    Os_StackValueType Base,
    Os_StackValueType Sample
)
```

Parameters

Parameter	Mode	Description
Base	in	Os_StackValueType The position to measure the stack from.
Sample	in	Os_StackValueType The position to measure the stack to.

Return Values

The call returns values of type Os_StackSizeType.

Description

Returns the difference between 2 Os_StackValueType values. To obtain a correct value, it is important that 'Base' represents an instant when the stack size was smaller than (or the same as) the point at which 'Sample' was measured.

It may not be called before StartOS().

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
Os_StackValueType start_position;
Os_StackValueType end_position;
Os_StackSizeType stack_size;
TASK(MyTask){
  start_position = Os_GetStackValue();
  nested_call();
  stack_size = Os_GetStackSize(start_position, end_position);
}
void nested_call(void) {
  end_position = Os_GetStackValue();
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✓	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✓		
		ProtectionHook	✓		

See Also

Os_GetStackUsage
Os_GetStackValue

2.57 Os_GetStackUsage

Get the amount of stack consumed by the calling Task/ISR.

Syntax

```
Os_StackSizeType Os_GetStackUsage(void)
```

Return Values

The call returns values of type Os_StackSizeType.

Description

Returns the amount of stack used by the calling Task or ISR at the point of the call.

The value is measured from the point at which the OS kernel starts to run the Task or ISR, and it includes overheads within the kernel so that the values returned can be used directly in the configuration of the stack allocation budget for a Task or ISR.

Calling this API has the side-effect of updating the recorded maximum stack usage for the calling Task or ISR (where necessary).

If the Task/ISR has a stack allocation budget, then a stack overrun may be reported before this API returns.

Stack monitoring must be enabled in general OS configuration for this API to give meaningful results. It returns zero if stack monitoring is not enabled.

The Os_Cbk_CheckStackDepth callback will be called from within this API if Stack Sampling is enabled. (It will be called even if stack monitoring is not enabled)

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
TASK(MyTask){
  Os_StackSizeType stack_size;
  stack_size = Os_GetStackUsage();
  nested_call();
}
void nested_call(void) {
  Os_GetStackUsage(); /* Identifies a possible max stack usage location */
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

See Also

Os_Cbk_CheckStackDepth
Os_Cbk_StackOverrunHook
Os_GetISRMaxStackUsage
Os_GetTaskMaxStackUsage
Os_ResetISRMaxStackUsage
Os_ResetTaskMaxStackUsage

2.58 Os_GetStackValue

Get the current stack value.

Syntax

```
Os_StackValueType Os_GetStackValue(void)
```

Return Values

The call returns values of type Os_StackValueType.

Description

Returns the current position of the stack pointer (or pointers).

It may not be called before StartOS().

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
Os_StackValueType start_position;
Os_StackValueType end_position;
Os_StackSizeType stack_size;
TASK(MyTask){
  start_position = Os_GetStackValue();
  nested_call();
  stack_size = Os_GetStackSize(start_position, end_position);
}
void nested_call(void) {
  end_position = Os_GetStackValue();
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✓	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✓		
		ProtectionHook	✓		

See Also

Os_GetStackSize
Os_GetStackUsage

2.59 Os_GetTaskActivationTime

Get the timestamp of the most recent successful activation of the Task.

Syntax

```
StatusType Os_GetTaskActivationTime(
    TaskType TaskID,
    Os_StopwatchTickRefType Value
)
```

Parameters

Parameter	Mode	Description
TaskID	in	TaskType The Task of interest.
Value	out	Os_StopwatchTickRefType The timestamp of the most recent activation for the task.

Return Values

The call returns values of type StatusType.

Value	Build	Description
E_OK	all	No error.
E_OS_ID	extended	TaskID is not a valid task.
E_OS_NOFUNC	extended	TaskID has not yet been activated.

Description

Returns the timestamp of the most recent activation of TaskID. (Zero if the task has not been activated.)

In this context, activation can be via ActivateTask, SetEvent or an Alarm/ExpiryPoint.

The timestamp is the value returned by the user-supplied function Os_Cbk_GetStopwatch() when it is called at an activation point.

NOTE: This API (and recording of activation times) is only available when OS option Activation Monitoring is enabled.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
TASK(Task1){
  Os_StopwatchTickType MyLastActivation;
  ...
  GetTaskActivationTime(Task1, &MyLastActivation);
  ...
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✓		
		ProtectionHook	✓		

See Also

OS_TASKTYPE_TO_INDEX
Os_Cbk_GetStopwatch

2.60 Os_GetTaskElapsedTime

Get the cumulative execution time consumed by the specified TASK.

Syntax

```
Os_StopwatchTickType Os_GetTaskElapsedTime(
    TaskType TaskID
)
```

Parameters

Parameter	Mode	Description
TaskID	in	TaskType The TASK of interest.

Return Values

The call returns values of type Os_StopwatchTickType.

Description

Returns the cumulative time spent running the specified TASK since StartOS() (or an explicit reset of the accumulated time).

If the total time exceeds the range of Os_StopwatchTickType the reported value will wrap around and appear to get smaller.

The elapsed time gets updated when a TASK finishes or is pre-empted. Calling Os_GetTaskElapsedTime before the TASK completes may not account for some of the execution time of the current invocation.

Zero is returned if the TaskID is not valid.

Time Monitoring and Elapsed Time Recording must be enabled for this API to give meaningful results. It returns zero if time monitoring is not enabled.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
Os_StopwatchTickType Total_Task_Time;
Total_Task_Time = Os_GetTaskElapsedTime(MyTask);
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✓		
		ProtectionHook	✓		

See Also

Os_GetElapsedTime
Os_GetISRElapsedTime
Os_GetIdleElapsedTime
Os_ResetISRElapsedTime
Os_ResetIdleElapsedTime
Os_ResetTaskElapsedTime

2.61 Os_GetTaskMaxExecutionTime

Get the longest observed execution time consumed by a Task.

Syntax

```
Os_StopwatchTickType Os_GetTaskMaxExecutionTime(
    TaskType TaskID
)
```

Parameters

Parameter	Mode	Description
TaskID	in	TaskType The Task of interest.

Return Values

The call returns values of type Os_StopwatchTickType.

Description

Returns the maximum observed execution time for TaskID.

This maximum value is over all complete invocations of TaskID that have completed since the previous call to ResetTaskMaxExecutionTime() for TaskID or to StartOS().

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
TASK(LoggingTask){
  Os_StopwatchTickType ExecutionTimes[MAXTASKS];
  ...
  ExecutionTimes[0] = GetTaskMaxExecutionTime(Task1);
  ExecutionTimes[1] = GetTaskMaxExecutionTime(Task2);
  ExecutionTimes[2] = GetTaskMaxExecutionTime(Task3);
  ExecutionTimes[3] = GetTaskMaxExecutionTime(Task4);
  ...
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✓		
		ProtectionHook	✗		

See Also

OS_TASKTYPE_TO_INDEX
Os_GetExecutionTime
Os_GetISRMaxExecutionTime
Os_ResetISRMaxExecutionTime
Os_ResetTaskMaxExecutionTime

2.62 Os_GetTaskMaxStackUsage

Get the maximum observed stack usage of a Task.

Syntax

```
Os_StackSizeType Os_GetTaskMaxStackUsage(
    TaskType TaskID
)
```

Parameters

Parameter	Mode	Description
TaskID	in	TaskType The Task of interest.

Return Values

The call returns values of type Os_StackSizeType.

Description

Returns the maximum observed stack usage for TaskID.

This maximum value is over all invocations of TaskID since the previous call to ResetTaskMaxStackUsage() for TaskID or to StartOS().

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
TASK(LoggingTask){
  Os_StackSizeType StackUsages[MAXTASKS];
  ...
  StackUsages[0] = GetTaskMaxStackUsage(Task1);
  StackUsages[1] = GetTaskMaxStackUsage(Task2);
  StackUsages[2] = GetTaskMaxStackUsage(Task3);
  StackUsages[3] = GetTaskMaxStackUsage(Task4);
  ...
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✓		
		ProtectionHook	✗		

See Also

OS_TASKTYPE_TO_INDEX
Os_GetISRMaxStackUsage
Os_GetStackUsage
Os_ResetISRMaxStackUsage
Os_ResetTaskMaxStackUsage

2.63 Os_IncrementCounter_<CounterID>

Increment a software counter.

Syntax

```
StatusType IncrementCounter_<CounterID>(void)
```

Return Values

The call returns values of type StatusType.

Value	Build	Description
E_OK	all	No error.

Description

This call has the same behavior as IncrementCounter(CounterID) but is customized for a named counter. This makes the call faster and more suitable for use in interrupt handlers. Note that error checks are not performed.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
ISR(MillisecondTimerInterrupt){
  ...
  Os_IncrementCounter_MillisecondCounter();
  ...
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

See Also

IncrementCounter
Os_AdvanceCounter
Os_AdvanceCounter_<CounterID>

2.64 Os_Metrics_Reset

Resets the OS metrics (for the calling core) to all zeroes.

Syntax

```
void Os_Metrics_Reset(void)
```

Description

When the OS option 'Collect OS usage metrics' is enabled, the OS collects data at run-time in a structure called 'Os_Metrics'. The content of this structure can be determined by looking at the file Os_Metrics.h. You can use this information yourself, or you can send a dump of it to ETAS support in order to help with performance tuning of your configuration.

The OS will collect counts of the following items: calls to each OS API, activations and starts of each TASK, starts of each ISR, Spinlock accesses, cross-core interrupts, cross-core TASK activations, cross-core IOC activations, Counter increments/advances, ECC TASK wait data, user-counters and possibly target-specific values.

The Os_Metrics_Reset() API can be called at any time to reset the counts to zero.

In a multicore environment, there is one copy of 'Os_Metrics' per core. A core only updates its own copy of the data.

Note that to keep the run-time overhead low, mutual exclusion and range checking code is not included when incrementing or resetting the values in Os_Metrics. Collection of OS usage metrics should only be used during testing and must not be used in production systems.

It may be called before StartOS().

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
TASK(MyTask){
  ...
#ifdef OS_METRICS_ENABLED
  Os_Metrics_Reset();
#endif /* OS_METRICS_ENABLED */
  ...
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✓	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✓		
		ProtectionHook	✓		

See Also

OS_COUNT_USER_n
OS_COUNT_cat1isrname

2.65 Os_RemoveDelayedTasks

Remove one or more tasks from the set of tasks that are subject to delayed execution.

Syntax

```
StatusType Os_RemoveDelayedTasks(
    Os_TasksetType Taskset
)
```

Parameters

Parameter	Mode	Description
Taskset	in	Os_TasksetType The set of tasks to be removed.

Return Values

The call returns values of type StatusType.

Value	Build	Description
E_OK	all	No error.
E_OS_ACCESS	extended	Delayed tasks cannot be changed by untrusted code
E_OS_ID	extended	A task is being removed that has the same priority as another task on the same core. You have to remove all of the tasks that share the priority.

Description

When delayed task execution is configured for the project, you can tell RTA-OS to delay execution of a set of tasks. These tasks can be activated, but will not actually run until they are removed from the set.

Os_RemoveDelayedTasks is used to remove tasks from the set of tasks being delayed.

Removed tasks will execute before this call returns (assuming they have higher priority than the caller).

Note that where tasks share priorities on a specific core, you must either remove all of the tasks sharing a priority or none.

You must only remove tasks that run on the calling core.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
TASK(LowPrioTask){
  Os_SetDelayedTasks(TASK_MASK(Task1) | TASK_MASK(Task3));
  ... /* Tasks 1 and 3 (only) can be activated, but will not run*/

  Os_RemoveDelayedTasks(TASK_MASK(Task1) | TASK_MASK(Task3));
  ... /* Tasks 1 and 3 can run*/
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✓		

See Also

OS_ADD_TASK
OS_NO_TASKS
Os_AddDelayedTasks
Os_SetDelayedTasks
Os_TasksetType
TASK_MASK

2.66 Os_ResetISRElapsedTime

Reset the cumulative execution time for an ISR.

Syntax

```
StatusType Os_ResetISRElapsedTime(
    ISRType ISRID
)
```

Parameters

Parameter	Mode	Description
ISRID	in	ISRType Name of the ISR to reset.

Return Values

The call returns values of type StatusType.

Value	Build	Description
E_OK	all	No error.
E_OS_ID	extended	ISRID is not a valid ISR.
E_OS_ACCESS	extended	ISRID is not accessible from the calling OS-Application.

Description

Reset the cumulative execution time for ISRID to zero.

Note that in a multicore application you can only reliably call this from the core that owns IsrID. Full cross-core protection has not been implemented for reasons of efficiency.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
Os_ResetISRElapsedTime(MyISR);
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✓		
		ProtectionHook	✓		

See Also

Os_GetElapsedTime
Os_GetISRElapsedTime
Os_GetIdleElapsedTime
Os_GetTaskElapsedTime
Os_ResetIdleElapsedTime
Os_ResetTaskElapsedTime

2.67 Os_ResetISRMaxExecutionTime

Reset the maximum observed execution time for an ISR.

Syntax

```
StatusType Os_ResetISRMaxExecutionTime(
    ISRType ISRID
)
```

Parameters

Parameter	Mode	Description
ISRID	in	ISRType Name of the ISR to reset.

Return Values

The call returns values of type StatusType.

Value	Build	Description
E_OK	all	No error.
E_OS_ID	extended	ISRID is not a valid Category 2 ISR.
E_OS_ACCESS	extended	ISRID is not accessible from the calling OS-Application.

Description

Reset the maximum observed execution time for the Category 2 ISR identified by ISRID to zero.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
TASK(ProfilingTask){
  Os_StopwatchTickType ExecutionTimeLog[SAMPLES];
  ...
  ExecutionTimeLog[index++] = Os_GetISRMaxExecutionTime(ISR1);
  Os_ResetISRMaxExecutionTime(ISR1);
  ...
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✓		
		ProtectionHook	✗		

See Also

Os_GetExecutionTime
Os_GetISRMaxExecutionTime
Os_GetTaskMaxExecutionTime
Os_ResetTaskMaxExecutionTime

2.68 Os_ResetISRMaxStackUsage

Reset the maximum observed stack usage for an ISR.

Syntax

```
StatusType Os_ResetISRMaxStackUsage(
    ISRType ISRID
)
```

Parameters

Parameter	Mode	Description
ISRID	in	ISRType Name of the ISR to reset.

Return Values

The call returns values of type StatusType.

Value	Build	Description
E_OK	all	No error.
E_OS_ID	extended	ISRID is not a valid Category 2 ISR.
E_OS_ACCESS	extended	ISRID is not accessible from the calling OS-Application.

Description

Reset the maximum observed stack usage for ISRID to zero.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
TASK(ProfilingTask){
  Os_StackSizeType StackUsageLog[SAMPLES];
  ...
  StackUsageLog[index++] = Os_GetISRMaxStackUsage(ISR1);
  Os_ResetISRMaxStackUsage(ISR1);
  ...
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✓		
		ProtectionHook	✗		

See Also

Os_GetISRMaxStackUsage
Os_GetStackUsage
Os_GetTaskMaxStackUsage
Os_ResetTaskMaxStackUsage

2.69 Os_ResetIdleElapsedTime

Reset the cumulative idle execution time for a core.

Syntax

```
StatusType Os_ResetIdleElapsedTime(
    Os_IdleType IdleID
)
```

Parameters

Parameter	Mode	Description
OS_CORE_CURRENT	in	Os_IdleType Idle on the calling core.
OS_CORE_ID_0	in	Os_IdleType Idle on Core 0
OS_CORE_ID_n	in	Os_IdleType Idle on Core n

Return Values

The call returns values of type StatusType.

Value	Build	Description
E_OK	all	No error.
E_OS_ID	extended	IdleID is not valid.

Description

Reset the cumulative idle execution time the core ID to zero.

Note that in a multicore application you can only reliably call this from the core that owns IdleID. Full cross-core protection has not been implemented for reasons of efficiency.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
Os_ResetIdleElapsedTime(OS_CORE_CURRENT);
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✓		
		ProtectionHook	✓		

See Also

Os_GetElapsedTime
Os_GetISRElapsedTime
Os_GetIdleElapsedTime
Os_GetTaskElapsedTime
Os_ResetISRElapsedTime
Os_ResetTaskElapsedTime

2.70 Os_ResetTaskElapsedTime

Reset the cumulative execution time for a task.

Syntax

```
StatusType Os_ResetTaskElapsedTime(
    TaskType TaskID
)
```

Parameters

Parameter	Mode	Description
TaskID	in	TaskType Name of the task to reset.

Return Values

The call returns values of type StatusType.

Value	Build	Description
E_OK	all	No error.
E_OS_ID	extended	TaskID is not a valid task.
E_OS_ACCESS	extended	TaskID is not accessible from the calling OS-Application.

Description

Reset the cumulative execution time for TaskID to zero.

Note that in a multicore application you can only reliably call this from the core that owns TaskID. Full cross-core protection has not been implemented for reasons of efficiency.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
Os_ResetTaskElapsedTime(MyTask);
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✓		
		ProtectionHook	✓		

See Also

Os_GetElapsedTime
Os_GetISRElapsedTime
Os_GetIdleElapsedTime
Os_GetTaskElapsedTime
Os_ResetISRElapsedTime
Os_ResetIdleElapsedTime

2.71 Os_ResetTaskMaxExecutionTime

Reset the maximum observed execution time for a task.

Syntax

```
StatusType Os_ResetTaskMaxExecutionTime(
    TaskType TaskID
)
```

Parameters

Parameter	Mode	Description
TaskID	in	TaskType Name of the task to reset.

Return Values

The call returns values of type StatusType.

Value	Build	Description
E_OK	all	No error.
E_OS_ID	extended	TaskID is not a valid task.
E_OS_ACCESS	extended	TaskID is not accessible from the calling OS-Application.

Description

Reset the maximum observed execution time for TaskID to zero.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
TASK(ProfilingTask){
  Os_StopwatchTickType ExecutionTimeLog[SAMPLES];
  ...
  ExecutionTimeLog[index++] = Os_GetTaskMaxExecutionTime(Task1);
  Os_ResetTaskMaxExecutionTime(Task1);
  ...
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✓		
		ProtectionHook	✗		

See Also

Os_GetExecutionTime
Os_GetISRMaxExecutionTime
Os_GetTaskMaxExecutionTime
Os_ResetISRMaxExecutionTime

2.72 Os_ResetTaskMaxStackUsage

Reset the maximum observed stack usage for a task.

Syntax

```
StatusType Os_ResetTaskMaxStackUsage(
    TaskType TaskID
)
```

Parameters

Parameter	Mode	Description
TaskID	in	TaskType Name of the task to reset.

Return Values

The call returns values of type StatusType.

Value	Build	Description
E_OK	all	No error.
E_OS_ID	extended	TaskID is not a valid task.
E_OS_ACCESS	extended	TaskID is not accessible from the calling OS-Application.

Description

Reset the maximum observed stack usage for TaskID to zero.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
TASK(ProfilingTask){
  Os_StackSizeType StackUsageLog[SAMPLES];
  ...
  StackUsageLog[index++] = Os_GetTaskMaxStackUsage(Task1);
  Os_ResetTaskMaxStackUsage(Task1);
  ...
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✓		
		ProtectionHook	✗		

See Also

Os_GetISRMaxStackUsage
Os_GetStackUsage
Os_GetTaskMaxStackUsage
Os_ResetISRMaxStackUsage

2.73 Os_Restart

Restart the OS by jumping to a previously specified location.

Syntax

```
StatusType Os_Restart(void)
```

Return Values

The call returns values of type StatusType.

Value	Build	Description
E_OS_SYS_RESTART	all	The call was not made from the ShutdownHook.
E_OS_SYS_NO_RESTART	all	No restart point has been set.

Description

The call re-initializes any necessary context and branches to the restart point set by Os_SetRestartPoint. The call does not return to the calling context.

The restart point must occur before a call to StartOS(), so that all OS re-initialization happens with the subsequent call to StartOS().

Os_Restart() can be used in multicore systems. You must take care to call it for *all* AUTOSAR cores, because the synchronization code in StartOS() will prevent individual cores from restarting.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
FUNC(void, {memclass}) ShutdownHook(StatusType Error){
  ...
  Os_Restart();
  ...
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✗	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✗	StartupHook	✗		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✗		

See Also

Os_SetRestartPoint
ShutdownAllCores
ShutdownOS
StartOS

2.74 Os_SetDelayedTasks

Specify exactly which tasks are subject to delayed execution.

Syntax

```
StatusType Os_SetDelayedTasks(
    Os_TasksetType Taskset
)
```

Parameters

Parameter	Mode	Description
Taskset	in	Os_TasksetType The set of tasks on the calling core that should have their execution delayed.

Return Values

The call returns values of type StatusType.

Value	Build	Description
E_OK	all	No error.
E_OS_ACCESS	extended	Delayed tasks cannot be changed by untrusted code
E_OS_ID	extended	A task is being added or removed that has the same priority as another task on the same core. You have to add/remove all of the tasks that share the priority.

Description

When delayed task execution is configured for the project, you can tell RTA-OS to delay execution of a set of tasks. These tasks can be activated, but will not actually run until they are removed from the set.

Os_SetDelayedTasks is used to specify which tasks should be delayed.

If a task was being delayed before the call but it is not in the new delayed task set, then it will execute before this call returns (assuming it has a higher priority than the caller).

Note that where tasks share priorities on a specific core, you must either specify all of the tasks sharing a priority or none.

You must only set tasks that run on the calling core.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
TASK(LowPrioTask){
  Os_SetDelayedTasks(TASK_MASK(Task1) | TASK_MASK(Task3));
  ... /* Tasks 1 and 3 (only) can be activated, but will not run*/

  Os_SetDelayedTasks(OS_NO_TASKS);
  ... /* Tasks 1 and 3 can run*/
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✓		

See Also

OS_ADD_TASK
OS_NO_TASKS
Os_AddDelayedTasks
Os_RemoveDelayedTasks
Os_TasksetType
TASK_MASK

2.75 Os_SetRestartPoint

Mark a location in code before StartOS() from where a restart of the OS can be made.

Syntax

```
StatusType Os_SetRestartPoint(void)
```

Return Values

The call returns values of type StatusType.

Value	Build	Description
E_OS_SYS_NO_RESTART	all	The call was not made before StartOS.

Description

The call marks the location from which the code should resume following a call to Os_Restart(). The location must be outside of OS control, i.e. at a point before StartOS() was called. Making the call when a restart point is already sets the restart point to the new location.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
OS_MAIN() {
  ...
  Os_SetRestartPoint();
  ...
  StartOS(OSDEFAULTAPPMODE);

}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✗	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✗	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

See Also

Os_Restart
ShutdownAllCores
ShutdownOS
StartOS

2.76 Os_SyncScheduleTableRel

Provide an adjustment value for an explicitly synchronized schedule table.

Syntax

```
StatusType Os_SyncScheduleTableRel(
    ScheduleTableType ScheduleTableID,
    SignedTickType RelativeValue
)
```

Parameters

Parameter	Mode	Description
ScheduleTableID	in	ScheduleTableType Name of the schedule table to synchronize.
RelativeValue	in	SignedTickType Amount to adjust the synchronizing counter.

Return Values

The call returns values of type StatusType.

Value	Build	Description
E_OK	all	No error.
E_OS_ID	all	ScheduleTableID is not a valid ScheduleTable or is not explicitly synchronized.
E_OS_ACCESS	extended	ScheduleTableID is not accessible from the calling OS-Application.
E_OS_VALUE	all	Value exceeds the duration of the table.
E_OS_STATE	all	The status of ScheduleTableID is SCHEDULETABLE_WAITING, SCHEDULETABLE_STOPPED or SCHEDULETABLE_NEXT.
E_OS_CALLEVEL	extended	Called from an invalid context (only when Service Protection is configured).
E_OS_DISABLEDINT	extended	Called while interrupts are disabled (only when Service Protection is configured).

Description

This call provides the synchronization RelativeValue for an explicitly synchronized table ScheduleTableID. ScheduleTableID must be running - you must use the SyncScheduleTable API to start it.

Control of and knowledge about the synchronizing counter is outside the domain of the OS. The OS assumes that the synchronizing counter has a duration equal to ScheduleTableID and that the resolution of the synchronizing counter is equal to the resolution of the OS counter used to drive ScheduleTableID. It is your responsibility to verify that your application satisfies these constraints.

When the ScheduleTableID is in the state SCHEDULETABLE_RUNNING or SCHEDULETABLE_RUNNING_AND_SYNCHRONOUS then RelativeValue is taken to be the current deviation between the notional position on ScheduleTableID and the desired value.

The state of ScheduleTableID is set according to the statically configured precision as follows:

- if RelativeValue <= precision then the state will be set to SCHEDULETABLE_RUNNING_AND_SYNCHRONOUS

- if RelativeValue > precision then the state will be set to SCHEDULETABLE_RUNNING

A positive RelativeValue that exceeds the configured precision will result in the entering the OS_SYNC_RETARDING state (ticks are removed).

A negative RelativeValue that exceeds the configured precision will result in the entering the OS_SYNC_ADVANCING state (extra ticks are inserted).

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
TASK(MyTask){
  StartScheduleTableSynchron(MyScheduleTable);
  ...
  Os_SyncScheduleTable(MyScheduleTable, 0); /* Start */
  ...
  Os_SyncScheduleTableRel(MyScheduleTable, -2); /* Adjust drift */
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

See Also

DeclareScheduleTable
GetScheduleTableStatus
NextScheduleTable
SetScheduleTableAsync
StartScheduleTableAbs
StartScheduleTableRel
StartScheduleTableSynchron
StopScheduleTable
SyncScheduleTable

2.77 Os_TimingFaultDetected

Report detection of a timing protection fault.

Syntax

```
void Os_TimingFaultDetected(void)
```

Description

When timing protection is configured and a timing interrupt is being used to enforce time limits, the timing interrupt must call this API whenever it runs.

The timing interrupt must run whenever the time limit that was set by the most recent call to Os_Cbk_SetTimeLimit() has been reached - unless a subsequent call to Os_Cbk_SuspendTimeLimit() has occurred to cancel it.

The timing interrupt must be a Category 1 ISR, and it should have priority higher than the highest Category 2 ISR. It is recommended that no other Category 1 ISRs are used. If you must have some, you should ensure that the timing interrupt cannot preempt them.

The OS responds to this call by calling ProtectionHook which means that it will normally not return to the timing interrupt. You must therefore perform any interrupt cleanup code that is needed before calling Os_TimingFaultDetected().

In a multicore system, one timing interrupt is needed for each core that has timing limits.

It may not be called before StartOS().

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
CAT1_ISR(timing_interrupt) {
  /* Reset pending interrupt flags here if needed */
  Os_TimingFaultDetected();
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✗	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✓	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✗	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

See Also

Os_Cbk_SetTimeLimit
Os_Cbk_SuspendTimeLimit
ProtectionHook

2.78 ReadPeripheral16

Reads the 16-bit value from memory location at Address into the variable addressed by ReadValue.

Syntax

```
StatusType ReadPeripheral16(
    AreaIdType Area,
    const uint16* Address,
    uint16* ReadValue
)
```

Parameters

Parameter	Mode	Description
Area	in	AreaIdType ID of the PeripheralArea.
Address	in	**const** uint16 * Address of the value to read.
ReadValue	out	uint16 * Reference to a uint16 to update.

Return Values

The call returns values of type StatusType.

Value	Build	Description
E_OK	all	No error.
E_OS_ID	extended	Area does not match the ID of a configured PeripheralArea.
E_OS_VALUE	extended	Address does not belong to the specified area.
E_OS_CALLEVEL	extended	Called from an invalid context (no running Task or ISR).
E_OS_ACCESS	extended	The area is not accessible from the calling OS-Application.

Description

Reads a 16 bit value from the specified address into a variable addressed by ReadValue.

Typically used to allow code in untrusted OS Applications to read data from areas of memory that would otherwise be inaccessible because of memory protection settings. The read is done from a trusted context.

Peripheral Areas must be declared in the configuration. Their legal address range must be specified, and also the OS-Applications that can access them.

All 16 bits of the read must be within the configured address range.

The OS does not prevent two different cores from accessing peripherals at the same time.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✓	✓	✗

Example

```
TASK(MyTask){
  uint16 result;
  ...
  ReadPeripheral16(MyArea, LED_STATUS_PORT, &result);
  ...
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

See Also

ModifyPeripheral16
ModifyPeripheral32
ModifyPeripheral8
ReadPeripheral32
ReadPeripheral8
WritePeripheral16
WritePeripheral32
WritePeripheral8

2.79 ReadPeripheral32

Reads the 32-bit value from memory location at Address into the variable addressed by ReadValue.

Syntax

```
StatusType ReadPeripheral32(
    AreaIdType Area,
    const uint32* Address,
    uint32* ReadValue
)
```

Parameters

Parameter	Mode	Description
Area	in	AreaIdType ID of the PeripheralArea.
Address	in	const uint32 * Address of the value to read.
ReadValue	out	uint32 * Reference to a uint32 to update.

Return Values

The call returns values of type StatusType.

Value	Build	Description
E_OK	all	No error.
E_OS_ID	extended	Area does not match the ID of a configured PeripheralArea.
E_OS_VALUE	extended	Address does not belong to the specified area.
E_OS_CALLEVEL	extended	Called from an invalid context (no running Task or ISR).
E_OS_ACCESS	extended	The area is not accessible from the calling OS-Application.

Description

Reads a 32 bit value from the specified address into a variable addressed by ReadValue.

Typically used to allow code in untrusted OS Applications to read data from areas of memory that would otherwise be inaccessible because of memory protection settings. The read is done from a trusted context.

Peripheral Areas must be declared in the configuration. Their legal address range must be specified, and also the OS-Applications that can access them.

All 32 bits of the read must be within the configured address range.

The OS does not prevent two different cores from accessing peripherals at the same time.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✓	✓	✗

Example

```
TASK(MyTask){
  uint32 result;
  ...
  ReadPeripheral32(MyArea, LED_STATUS_PORT, &result);
  ...
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

See Also

ModifyPeripheral16
ModifyPeripheral32
ModifyPeripheral8
ReadPeripheral16
ReadPeripheral8
WritePeripheral16
WritePeripheral32
WritePeripheral8

2.80 ReadPeripheral8

Reads the 8-bit value from memory location at Address into the variable addressed by ReadValue.

Syntax

```
StatusType ReadPeripheral8(
    AreaIdType Area,
    const uint8* Address,
    uint8* ReadValue
)
```

Parameters

Parameter	Mode	Description
Area	in	AreaIdType ID of the PeripheralArea.
Address	in	const uint8 * Address of the value to read.
ReadValue	out	uint8 * Reference to a uint8 to update.

Return Values

The call returns values of type StatusType.

Value	Build	Description
E_OK	all	No error.
E_OS_ID	extended	Area does not match the ID of a configured PeripheralArea.
E_OS_VALUE	extended	Address does not belong to the specified area.
E_OS_CALLEVEL	extended	Called from an invalid context (no running Task or ISR).
E_OS_ACCESS	extended	The area is not accessible from the calling OS-Application.

Description

Reads an 8 bit value from the specified address into a variable addressed by ReadValue.

Typically used to allow code in untrusted OS Applications to read data from areas of memory that would otherwise be inaccessible because of memory protection settings. The read is done from a trusted context.

Peripheral Areas must be declared in the configuration. Their legal address range must be specified, and also the OS-Applications that can access them.

The OS does not prevent two different cores from accessing peripherals at the same time.

Note: Not supported of targets that use 16-bit char types.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✓	✓	✗

Example

```
TASK(MyTask){
  uint8 result;
  ...
  ReadPeripheral8(MyArea, LED_STATUS_PORT, &result);
  ...
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

See Also

ModifyPeripheral16
ModifyPeripheral32
ModifyPeripheral8
ReadPeripheral16
ReadPeripheral32
WritePeripheral16
WritePeripheral32
WritePeripheral8

2.81 ReleaseResource

Release (unlock) a previously held resource to leave a critical section.

Syntax

```
StatusType ReleaseResource(
    ResourceType ResID
)
```

Parameters

Parameter	Mode	Description
ResID	in	ResourceType The resource to release.

Return Values

The call returns values of type StatusType.

Value	Build	Description
E_OK	all	No error.
E_OS_ID	extended	ResID is not a valid resource.
E_OS_ACCESS	extended	ResID is not accessible from the calling OS-Application.
E_OS_ACCESS	extended	Attempt to release a resource which has a lower ceiling priority than the configured priority of the calling task/ISR.
E_OS_CALLEVEL	extended	Called from an invalid context (only when Service Protection is configured).
E_OS_DISABLEDINT	extended	Called while interrupts are disabled (only when Service Protection is configured).
E_OS_RESOURCE	extended	Called where a new Spinlock was acquired after this resource was locked. Spinlocks and Resources can only be locked and unlocked in strict LIFO order.
E_OS_NOFUNC	extended	The resource is not occupied.

Description

ReleaseResource is the counterpart of GetResource and serves to quit a critical section in the code.

Resources can not be configured to work across different cores.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

Example

```
TASK(MyTask){
  ...
  GetResource(Outer);
    /* Outer Critical Section */
    ...
    GetResource(Inner);
      /* Inner Critical Section */
    ReleaseResource(Inner);
    ...
  ReleaseResource(Outer);
  ...
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

See Also

DeclareResource
GetResource

2.82 ReleaseSpinlock

ReleaseSpinlock is used to release a spin-lock variable.

Syntax

```
StatusType ReleaseSpinlock(
    SpinlockIdType SpinlockId
)
```

Parameters

Parameter	Mode	Description
SpinlockId	in	SpinlockIdType The Spinlock to release.

Return Values

The call returns values of type StatusType.

Value	Build	Description
E_OK	all	No error.
E_OS_ID	extended	SpinlockId does not refer to a valid Spinlock.
E_OS_STATE	extended	The Spinlock is not occupied by the calling TASK or ISR.
E_OS_ACCESS	extended	SpinlockId is not accessible from the calling OS-Application.
E_OS_NOFUNC	extended	A different Spinlock must be released first. Nesting order of Spinlock GetSpinlock/ReleaseSpinlock must be maintained.
E_OS_RESOURCE	extended	A Resource must be released first. Nesting order of Spinlocks and Resources must be maintained.
E_OS_CALLEVEL	extended	Called from an invalid context (only when Service Protection is configured).
E_OS_RESOURCE	extended	Called where a new Resource was locked after this spinlock was acquired. Spinlocks and Resources can only be locked and unlocked in strict LIFO order.

Description

ReleaseSpinlock is used to release a spinlock that was previously occupied using GetSpinLock or TryToGetSpinLock.

All spinlocks must be released before a Task or ISR terminate.

All spinlocks must be released before an ECC Task calls WaitEvent.

When the spinlock has a lock method with LOCK_ALL_INTERRUPTS semantics, this API will exit with a call to ResumeAllInterrupts().

When the spinlock has a lock method with LOCK_CAT2_INTERRUPTS semantics, this API will exit with a call to ResumeOSInterrupts().

When the spinlock has a lock method with LOCK_WITH_RES_SCHEDULER semantics, this API will exit with a call to ReleaseResource(RES_SCHEDULER).

When the spinlock has a lock method with NESTABLE semantics, this API will only perform the unlock when called by the original locker, and only then when it has made the same number of Release calls as Get calls.

Note that the OS configuration option 'Force spinlock error checks' can be used to cause the error checks to be done in standard as well as extended status builds.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✓	✗

Example

```
TASK(MyTask){
  ...
  GetSpinlock(Spinlock1);
  ...
  ReleaseSpinlock(Spinlock1);
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

See Also

GetSpinLock
ReleaseResource
ResumeAllInterrupts
ResumeOSInterrupts
TryToGetSpinlock
UncheckedGetSpinlock
UncheckedReleaseSpinlock
UncheckedTryToGetSpinlock

2.83 ResetSpinlockInfo

Reset run-time statistics for a Spinlock.

Syntax

```
StatusType ResetSpinlockInfo(
    SpinlockIdType SpinlockId
)
```

Parameters

Parameter	Mode	Description
SpinlockId	in	SpinlockIdType The Spinlock

Return Values

The call returns values of type StatusType.

Value	Build	Description
E_OK	all	No error.
E_OS_ID	extended	SpinlockId does not refer to a valid Spinlock.

Description

ResetSpinlockInfo is an optional API that is only available when there are spinlocks in the application and the OS option 'Provide spinlock statistics' is set to true.

It is used to reset the statistics belonging to the specified spinlock to all zero values.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
TASK(MyTask){
  ...
  Os_SpinlockInfo Info;
  ResetSpinlockInfo(Spinlock1);
  ...
  GetSpinlock(Spinlock1);
  GetSpinlockInfo(Spinlock1, &Info);
  if ((TaskType)Info.LockFails > 0) {
    ...
  }
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

See Also

GetSpinlockInfo
Os_SpinlockInfo

2.84 ResumeAllInterrupts

Resume recognition of Category 1 and Category 2 interrupts.

Syntax

```
void ResumeAllInterrupts(void)
```

Description

This API call marks the end of a critical section that is protected from any maskable interrupt occurring. The critical section must have been entered using the SuspendAllInterrupts() call.

No API calls beside SuspendAllInterrupts()/ResumeAllInterrupts() pairs and SuspendOSInterrupts()/ResumeOSInterrupts() pairs are allowed within this critical section.

Interrupt processing is restored to that in effect before the immediately prior SuspendAllInterrupts() call.

When calls to SuspendAllInterrupts() and ResumeAllInterrupts() are nested then the interrupt recognition status saved by the first call of SuspendAllInterrupts() is restored by the last call of the ResumeAllInterrupts().

In a multicore environment, this call only affects the interrupts on the core that it is called from.

It may be called before StartOS().

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

Example

```
TASK(MyTask){
  ...
  SuspendAllInterrupts():
    /* Critical Section 1 */
    FunctionWithNestedCriticalSection();
  ResumeAllInterrupts():
  ...
}
void FunctionWithNestedCriticalSection(void) {
  ...
  SuspendAllInterrupts():
  /* Critical Section 2 */
  ResumeAllInterrupts():
  ...
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✓	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✓		
		ProtectionHook	✓		

See Also

DisableAllInterrupts
EnableAllInterrupts
ResumeOSInterrupts
SuspendAllInterrupts
SuspendOSInterrupts

2.85 ResumeOSInterrupts

Resume recognition of Category 2 interrupts

Syntax

```
void ResumeOSInterrupts(void)
```

Description

This API call marks the end of a critical section that is protected from any Category 2 (OS level) interrupt occurring. The critical section must have been entered using the SuspendOSInterrupts() call.

No API calls beside SuspendAllInterrupts()/ResumeAllInterrupts() pairs and SuspendOSInterrupts()/ResumeOSInterrupts() pairs are allowed within this critical section.

Interrupt processing is restored to that in effect before the immediately prior SuspendOSInterrupts() call.

When calls to SuspendOSInterrupts() and ResumeOSInterrupts() are nested then the interrupt recognition status saved by the first call of SuspendOSInterrupts() is restored by the last call of the ResumeOSInterrupts().

In a multicore environment, this call only affects the interrupts on the core that it is called from.

It may not be called before StartOS().

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

Example

```
TASK(MyTask){
  ...
  SuspendOSInterrupts():
    /* Longer Critical Section */
    SuspendAllInterrupts();
      /* Shorter Critical Section  */
    ResumeAllInterrupts();
  ResumeOSInterrupts():
  ...
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✗
Category 1 ISR	✓	PostTaskHook	✓	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✓		
		ProtectionHook	✓		

See Also

DisableAllInterrupts
EnableAllInterrupts
ResumeAllInterrupts
SuspendAllInterrupts
SuspendOSInterrupts

2.86 Schedule

Forces the OS to check if a higher priority task can be run.

Syntax

```
StatusType Schedule(void)
```

Return Values

The call returns values of type `StatusType`.

Value	Build	Description
E_OK	all	No error.
E_OS_RESOURCE	extended	Calling task still holds resources.
E_OS_CALLEVEL	extended	Called at interrupt level.
E_OS_CALLEVEL	extended	Called from an invalid context (only when Service Protection is configured).
E_OS_DISABLEDINT	extended	Called while interrupts are disabled (only when Service Protection is configured).
E_OS_SPINLOCK	extended	Called while the task holds a Spinlock

Description

The call allows a non-preemptive task or a task/ISR that uses an internal resource to offer a preemption point.

Rescheduling occurs if:

1. The calling task is non-preemptive and a higher priority task has been activated while the calling task was in the running state.

2. The calling task/ISR shares an internal resource with a higher priority task/ISR and that higher priority task/ISR has been activated.

If no higher-priority task/ISR is in the ready state the calling task/ISR resumes.

This service has no influence on preemptive tasks or ISRs that do not use internal resources.

Note that allowing ISRs to share internal resources is an RTA-OS specific feature.

If the Optimize Schedule() OS option is selected, the code may do a quick plausibility check before calling the OS, in order to reduce overheads in cases where no rescheduling could occur. In this case the return type of Schedule() becomes void, so should not be checked.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

Example

```
TASK(MyTask){
  CooperativeProcessA();
  Schedule();
  CooperativeProcessB();
  Schedule();
  CooperativeProcessC();
  Schedule();
  ...
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✗	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

See Also

DeclareTask
GetTaskID
GetTaskState
TerminateTask

2.87 SetAbsAlarm

Set an alarm for an absolute counter value.

Syntax

```
StatusType SetAbsAlarm(
    AlarmType AlarmID,
    TickType start,
    TickType cycle
)
```

Parameters

Parameter	Mode	Description
AlarmID	in	AlarmType Name of the alarm to set.
start	in	TickType Absolute tick value at which the alarm is first triggered.
cycle	in	TickType Ticks before the alarm is triggered subsequently..

Return Values

The call returns values of type StatusType.

Value	Build	Description
E_OK	all	No error.
E_OS_STATE	all	AlarmID already running.
E_OS_ID	extended	AlarmID is not a valid alarm.
E_OS_ACCESS	extended	AlarmID is not accessible from the calling OS-Application.
E_OS_VALUE	extended	The value of start or cycle is outside the permitted range. 0 <= start <= maxallowedvalue. cycle = 0 or mincycle <= cycle <= maxallowedvalue.
E_OS_CALLEVEL	extended	Called from an invalid context (only when Service Protection is configured).
E_OS_DISABLEDINT	extended	Called while interrupts are disabled (only when Service Protection is configured).

Description

This call starts an alarm running and sets the match value with the associated counter that triggers the alarm.

If cycle is equal to zero then the alarm will be triggered once only. If cycle is nonzero then the alarm will be triggered every cycle ticks after start.

When the alarm expires, the statically configured action (activate a task / set an event / run an alarm callback / increment a counter) occurs.

You must cancel an alarm if it is running before you can restart it with different values.

Note that if the value of start is less than or equal to the current counter value then AlarmID will not be triggered until a full wrap of the underlying counter.

In particular, note that if an absolute alarm is set at startup with a start of zero - SetAbsAlarm(MyAlarm,0,x) - then the alarm will not be triggered until maxallowedvalue+1 ticks of the counter have elapsed.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

Example

```
TASK(MyTask){
  ...
  /* SingleShotAlarm at tick 42 */
  SetAbsAlarm(SingleShotAlarm, 42, 0);
  ...
  /* PeriodicAlarm at 10, 60, 110, 160,... */
  SetAbsAlarm(PeriodicAlarm, 10, 50);
  ...
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

See Also

CancelAlarm
DeclareAlarm
GetAlarm
GetAlarmBase
SetRelAlarm

2.88 SetEvent

Set event(s) for a task.

Syntax

```
StatusType SetEvent(
    TaskType TaskID,
    EventMaskType Mask
)
```

Parameters

Parameter	Mode	Description
TaskID	in	TaskType Name of the Task to set the Event for.
Mask	in	EventMaskType A mask of events to set.

Return Values

The call returns values of type StatusType.

Value	Build	Description
E_OK	all	No error.
E_OS_CORE	all	The task belongs to a core that is stopped (in Shutdown).
E_OS_SYS_XCORE_QFULL	all	Only when OS option 'Asynchronous TASK activation' is active: the queue allocated for sending cross-core event notifications is full. The event has not been set.
E_OS_ID	extended	TaskID is not a valid task.
E_OS_ACCESS	extended	TaskID is not accessible from the calling OS-Application.
E_OS_ACCESS	extended	TaskID is not an extended task.
E_OS_STATE	extended	TaskID is in the suspended state.
E_OS_CALLEVEL	extended	Called from an invalid context (only when Service Protection is configured).
E_OS_DISABLEDINT	extended	Called while interrupts are disabled (only when Service Protection is configured).

Description

This API call sets events for task TaskID according to Mask.

If the task is waiting for any event in Event, it is immediately transferred to the ready state and re-scheduling can occur.

Multiple events can be set simultaneously by logically bitwise or-ing events.

Any unset events in the event mask remain unchanged.

Events cannot be set for extended tasks that are in the suspended state. In extended status this results in the error E_OS_STATE. In standard status, setting an event for a suspended task has no effect.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

Example

```
TASK(MyTask) {
  ...
  /* Set a single event */
  SetEvent(MyExtendedTask, Event1);
  ...
  /* Set multiple events */
  SetEvent(MyOtherExtendedTask, Event1 | Event2 | Event3);
  ...
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

See Also

ClearEvent
DeclareEvent
SetEventAsyn
WaitEvent

2.89 SetEventAsyn

Set event(s) for a task asynchronously.

Syntax

```
void SetEventAsyn(
    TaskType TaskID,
    EventMaskType Mask
)
```

Parameters

Parameter	Mode	Description
TaskID	in	TaskType Name of the Task to set the Event for.
Mask	in	EventMaskType A mask of events to set.

Description

This is a version of SetEvent where cross-core task activation is performed by signalling a request to the appropriate core. The API does not block while waiting for the request to be acted upon.

This API does not return any error status values but the errors that can be raised by SetEvent are still passed to ErrorHook.

The errors that are reported on the calling core are E_OS_CORE, E_OS_ID, E_OS_ACCESS, E_OS_CALLEVEL and E_OS_DISABLEDINT.

The errors that are reported on the core of the target task are E_OS_LIMIT and E_OS_SYS_XCORE_QFULL.

The OS option 'Support ActivateTaskAsyn and SetEventAsyn' has to be set to TRUE used to cause this API to be generated because it adds a small amount of extra run-time overhead. Without this, SetEventAsyn may be mapped to be the same as SetEvent.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

Example

```
TASK(MyTask) {
  ...
  /* Set a single event */
  SetEventAsyn(MyExtendedTask, Event1);
  ...
  /* Set multiple events */
  SetEventAsyn(MyOtherExtendedTask, Event1 | Event2 | Event3);
  ...
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

See Also

ClearEvent
DeclareEvent
SetEvent
WaitEvent

2.90 SetRelAlarm

Set an alarm for a relative counter value.

Syntax

```
StatusType SetRelAlarm(
    AlarmType AlarmID,
    TickType increment,
    TickType cycle
)
```

Parameters

Parameter	Mode	Description
AlarmID	in	AlarmType Name of the alarm to set.
increment	in	TickType Relative number of ticks before the alarm is first triggered.
cycle	in	TickType Ticks before the alarm is triggered subsequently.

Return Values

The call returns values of type StatusType.

Value	Build	Description
E_OK	all	No error.
E_OS_STATE	all	AlarmID already running.
E_OS_ID	extended	AlarmID is not a valid alarm.
E_OS_ACCESS	extended	AlarmID is not accessible from the calling OS-Application.
E_OS_VALUE	extended	The value of increment or cycle is outside the permitted range. 0 < increment <= maxallowedvalue. cycle =0 or mincycle <= cycle <= maxallowedvalue.
E_OS_CALLEVEL	extended	Called from an invalid context (only when Service Protection is configured).
E_OS_DISABLEDINT	extended	Called while interrupts are disabled (only when Service Protection is configured).

Description

This call starts an alarm running and sets the match value with the associated counter that triggers the alarm. The match value is equal to the current counter value plus the increment.

If cycle is equal to zero then the alarm will be triggered once only. If cycle is nonzero then the alarm will be triggered every cycle ticks after start.

When the alarm expires, the statically configured action (activate a task / set an event / run an alarm callback / increment a counter) occurs.

You must cancel an alarm if it is running before you can restart it with different values.

Care must be taken when the value of increment is small because the outcome of SetRelAlarm() can produce different results depending on whether the counter has ticked past the match value before the call completes. It will either result in the alarm expiring almost immediately or when the value is reached again (after the next wrap of the underlying counter).

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

Example

```
TASK(MyTask){
  ...
  /* SingleShotAlarm in Now+123 ticks */
  SetRelAlarm(SingleShotAlarm, 123, 0);
  ...
  /* PeriodicAlarm at Now+42, Now+142, Now+242... */
  SetRelAlarm(PeriodicAlarm, 42, 100);
  ...
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

See Also

CancelAlarm
DeclareAlarm
GetAlarm
GetAlarmBase
SetAbsAlarm

2.91 SetScheduleTableAsync

Cancels synchronization on a schedule table.

Syntax

```
StatusType SetScheduleTableAsync(
    ScheduleTableType ScheduleTableID
)
```

Parameters

Parameter	Mode	Description
ScheduleTableID	in	ScheduleTableType Name of the schedule table.

Return Values

The call returns values of type StatusType.

Value	Build	Description
E_OK	all	No error.
E_OS_ID	extended	ScheduleTableID is invalid or is not an explicitly synchronized table.
E_OS_STATE	extended	The current state of ScheduleTableID is not SCHEDULETABLE_STOPPED, SCHEDULETABLE_NEXT or SCHEDULETABLE_WAITING.
E_OS_ACCESS	extended	ScheduleTableID is not accessible from the calling OS-Application.
E_OS_CALLEVEL	extended	Called from an invalid context (only when Service Protection is configured).
E_OS_DISABLEDINT	extended	Called while interrupts are disabled (only when Service Protection is configured).

Description

This call sets the status of ScheduleTableID to SCHEDULETABLE_RUNNING if and only if ScheduleTableID is running and is configured as explicitly synchronized.

The OS will continue to process expiry points on ScheduleTableID, but will stop expiry point synchronization until a SyncScheduleTable() or SyncScheduleTableRel() call is subsequently made.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

Example

```
TASK(MyTask){
  StartScheduleTableRel(MyScheduleTable, 2U);
  ...
  SyncScheduleTable(MyScheduleTable, 12U);
  ...
  SetScheduleTableAsync(MyScheduleTable);
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

See Also

DeclareScheduleTable
GetScheduleTableStatus
NextScheduleTable
Os_SyncScheduleTableRel
StartScheduleTableAbs
StartScheduleTableRel
StartScheduleTableSynchron
StopScheduleTable
SyncScheduleTable

2.92 ShutdownAllCores

Shutdown all AUTOSAR cores.

Syntax

```
void ShutdownAllCores(
    StatusType Error
)
```

Parameters

Parameter	Mode	Description
Error	in	StatusType The reason for the shutdown.

Description

In a multicore configuration, ShutdownAllCores causes a synchronized shutdown of all AUTOSAR cores.

It forces ShutdownOS to be executed on each AUTOSAR core.

The cores are synchronized just before calling the global ShutdownHook.

The system can be restarted if Os_Restart() is called in the global ShutdownHook (by each core).

ShutdownAllCores is not effective when called by untrusted code, and will simply return to the caller.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✓	✗

Example

```
TASK(MyTask){
  ...
  if (ErrorCondition != E_OK) {
    ShutdownAllCores(ErrorCondition);
  }
  ...
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✗		
		ErrorHook	✓		
		ProtectionHook	✗		

See Also

Os_Restart
Os_SetRestartPoint
ShutdownOS
StartOS

2.93 ShutdownOS

Shutdown the operating system.

Syntax

```
void ShutdownOS(
    StatusType Error
)
```

Parameters

Parameter	Mode	Description
Error	in	StatusType The reason for the shutdown.

Description

This API causes the OS to shut down. Task scheduling, all Category 2 interrupts, alarms and schedule tables are stopped immediately.

PostTaskHook (if configured) is not called when ShutdownOS() occurs.

ShutdownHook is called (if configured) and is passed the Error argument as the OS shuts down.

If ShutdownHook() returns, or is absent, the operating system then disables all Category 1 interrupts and enters an endless loop.

ShutdownOS() can be called internally by the operating system in response to an unrecoverable error.

ShutdownOS() is not effective when called by untrusted code, and will simply return to the caller.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

Example

```
TASK(MyTask){
  ...
  if (ErrorCondition != E_OK) {
    ShutdownOS(ErrorCondition);
  }
  ...
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✗		
		ErrorHook	✓		
		ProtectionHook	✗		

See Also

Os_Cbk_InShutdown
Os_Restart
Os_SetRestartPoint
ShutdownAllCores
ShutdownHook
StartOS

2.94 StartCore

Start a core that is configured to run AUTOSAR.

Syntax

```
void StartCore(
    CoreIdType CoreID,
    StatusType* Status
)
```

Parameters

Parameter	Mode	Description
CoreID	in	CoreIdType The core to activate.
Status	out	StatusType The return status.

Return Values

The call returns values of type StatusType.

Value	Build	Description
E_OK	all	No error.
E_OS_ID	extended	The core does not exist or it is not configured to run AUTOSAR.
E_OS_ACCESS	extended	The API was called after StartOS().
E_OS_STATE	extended	The core is already started.

Description

In a multicore configuration, StartCore is used to start cores that are configured to run the AUTOSAR OS. It must be called before StartOS().

It must be called for each of the AUTOSAR cores, including the master core if it is so configured.

All cores start execution at OS_MAIN(), so take care not to start cores more than once.

StartCore can be invoked from the master core or from slave cores.

StartOS() must be called for each started AUTOSAR OS core. Each call must be passed either the same AppModeType or DONOTCARE.

This API is only provided with multicore project configurations.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✓	✗

Example

```
OS_MAIN(){
  StatusType status;
  ...
  if (GetCoreID() == OS_CORE_ID_MASTER) {
    StartCore(OS_CORE_ID_0, &status);
    StartCore(OS_CORE_ID_1, &status);
  }
  StartOS(OSDEFAULTAPPMODE);
  ...
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✗	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✗	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

See Also

GetCoreID
GetNumberOfActivatedCores
StartNonAutosarCore
StartOS

2.95 StartNonAutosarCore

Start a core that is not configured to run AUTOSAR.

Syntax

```
void StartNonAutosarCore(
    CoreIdType CoreID,
    StatusType* Status
)
```

Parameters

Parameter	Mode	Description
CoreID	in	CoreIdType The core to activate.
Status	out	StatusType The return status.

Return Values

The call returns values of type StatusType.

Value	Build	Description
E_OK	all	No error.
E_OS_ID	extended	The core does not exist or it is configured to run AUTOSAR.
E_OS_ACCESS	extended	The API was called after StartOS().
E_OS_STATE	extended	The core is already started.

Description

In a multicore configuration, StartNonAutosarCore is used to start cores that are NOT configured to run the AUTOSAR OS. It may be called for each of the non-AUTOSAR cores, including the master core if it is so configured.

All cores start execution at OS_MAIN(), so take care not to start cores more than once.

StartNonAutosarCore can be invoked from the master core, or from slave cores.

This API is only provided with multicore project configurations where there are non-AUTOSAR cores.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✓	✗

Example

```
OS_MAIN(){
  StatusType status;
  ...
  if (GetCoreID() == OS_CORE_ID_MASTER) {
    StartNonAutosarCore(OS_CORE_ID_0, &status);
    StartNonAutosarCore(OS_CORE_ID_1, &status);
  }
  ...
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✗	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✗	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

See Also

GetCoreID
StartCore

2.96 StartOS

Start the operating system in a specified mode.

Syntax

```
void StartOS(
    AppModeType Mode
)
```

Parameters

Parameter	Mode	Description
Mode	in	AppModeType The application mode to use for startup.

Description

StartOS() initializes all internal OS data structures and starts the OS in the specified Mode.

Any tasks that are autostarted in the specified Mode are set to the ready state.

Any alarms or schedule tables that are autostarted in the specified Mode are initialized appropriately.

Software counters are initialized to zero.

The Mode OSDEFAULTAPPMODE must always exist, but other names can be configured as needed.

StartOS() is only allowed outside the context of the OS. It has no effect if called while the OS is already running.

Restarting the OS can be achieved using Os_SetRestartPoint() to set a restart point before the call the StartOS() and jumping to the point using Os_Restart().

If StartOS() is called with invalid preconditions, it may call ShutdownOS(E_OS_STATE). The preconditions are port-specific, so are documented in the port user guide. They may include issues such as the CPU being in the wrong mode, or the stack not being set up correctly.

Where there is more than one core configured to use AUTOSAR OS, each OS core must call StartOS. At least one of these must provide a Mode value that is different to DONOTCARE. Other than DONOTCARE, all calls to StartOS must use the same Mode value.

When called correctly by an OS core, StartOS() does not return to the caller.

When called by a non-OS core, StartOS() returns without any effect.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

Example

```
OS_MAIN() {
  /* Initialize target hardware before starting OS */
  StartOS(OSDEFAULTAPPMODE);
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✗	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✗	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

See Also

DONOTCARE
Os_Cbk_Idle
Os_Restart
Os_SetRestartPoint
ShutdownAllCores
ShutdownOS

2.97 StartScheduleTableAbs

Set the counter tick at which a schedule table starts.

Syntax

```
StatusType StartScheduleTableAbs(
    ScheduleTableType ScheduleTableID,
    TickType Start
)
```

Parameters

Parameter	Mode	Description
ScheduleTableID	in	ScheduleTableType Name of the schedule table to start.
Start	in	TickType Absolute counter tick value at which the schedule table starts.

Return Values

The call returns values of type StatusType.

Value	Build	Description
E_OK	all	No error.
E_OS_STATE	all	ScheduleTableID already running.
E_OS_ID	extended	ScheduleTableID is not a valid ScheduleTable.
E_OS_ACCESS	extended	ScheduleTableID is not accessible from the calling OS-Application.
E_OS_VALUE	extended	Start > maxallowedvalue of the underlying counter.
E_OS_CALLEVEL	extended	Called from an invalid context (only when Service Protection is configured).
E_OS_DISABLEDINT	extended	Called while interrupts are disabled (only when Service Protection is configured).

Description

If the parameters are valid, this call starts ScheduleTableID running and sets the state of ScheduleTableID to SCHEDULETABLE_RUNNING.

The first expiry point is processed at Start+InitialOffset ticks, where InitialOffset is the numerically lowest of the statically configured offsets defined for expiry points on ScheduleTableID.

Note that if this gives a value less than or equal to the current counter value then the first expiry will not happen until a full modulus wrap of the underlying counter has occurred.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

Example

```
TASK(MyTask){
  /* Start MyScheduleTable when the associated counter reaches 100 */
  StartScheduleTableAbs(MyScheduleTable, 100);
  ...
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

See Also

DeclareScheduleTable
GetScheduleTableStatus
NextScheduleTable
Os_SyncScheduleTableRel
SetScheduleTableAsync
StartScheduleTableRel
StartScheduleTableSynchron
StopScheduleTable
SyncScheduleTable

2.98 StartScheduleTableRel

Set the number of counter ticks before a schedule table starts.

Syntax

```
StatusType StartScheduleTableRel(
    ScheduleTableType ScheduleTableID,
    TickType Offset
)
```

Parameters

Parameter	Mode	Description
ScheduleTableID	in	ScheduleTableType Name of the schedule table to start.
Offset	in	TickType Relative number of ticks before the schedule table starts.

Return Values

The call returns values of type StatusType.

Value	Build	Description
E_OK	all	No error.
E_OS_STATE	all	ScheduleTableID is not in the state SCHEDULETABLE_STOPPED.
E_OS_ID	extended	ScheduleTableID is not a valid ScheduleTable.
E_OS_ACCESS	extended	ScheduleTableID is not accessible from the calling OS-Application.
E_OS_VALUE	extended	Offset == zero or Offset > maxallowedvalue - InitialOffset.
E_OS_CALLEVEL	extended	Called from an invalid context (only when Service Protection is configured).
E_OS_DISABLEDINT	extended	Called while interrupts are disabled (only when Service Protection is configured).

Description

If the parameters are valid, this call starts ScheduleTableID running and sets the state of ScheduleTableID to SCHEDULETABLE_RUNNING.

The first expiry point on ScheduleTableID is processed after Offset+InitialOffset ticks have elapsed, where InitialOffset is the numerically lowest of the statically configured offsets defined for expiry points on ScheduleTableID.

The call is not permitted for a schedule table that is configured as implicitly synchronized. If ScheduleTableID is an implicitly synchronized schedule table then the call will return E_OS_ID.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

Example

```
TASK(MyTask){
  ...
  /* Start MyScheduleTable at Now+42 ticks */
  StartScheduleTableRel(MyScheduleTable, 42);
  ...
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

See Also

DeclareScheduleTable
GetScheduleTableStatus
NextScheduleTable
Os_SyncScheduleTableRel
StartScheduleTableAbs
StartScheduleTableSynchron
StopScheduleTable
SyncScheduleTable

2.99 StartScheduleTableSynchron

Start an explicitly synchronized schedule table and wait for a synchronization call.

Syntax

```
StatusType StartScheduleTableSynchron(
    ScheduleTableType ScheduleTableID
)
```

Parameters

Parameter	Mode	Description
ScheduleTableID	in	ScheduleTableType Name of the schedule table to start.

Return Values

The call returns values of type StatusType.

Value	Build	Description
E_OK	all	No error.
E_OS_ID	extended	ScheduleTableID is not a valid ScheduleTable.
E_OS_ACCESS	extended	ScheduleTableID is not accessible from the calling OS-Application.
E_OS_STATE	extended	The state of ScheduleTableID is not SCHEDULETABLE_STOPPED.
E_OS_CALLEVEL	extended	Called from an invalid context (only when Service Protection is configured).
E_OS_DISABLEDINT	extended	Called while interrupts are disabled (only when Service Protection is configured).

Description

This call primes the explicitly synchronized ScheduleTableID to start synchronously once a synchronization count to be provided by the call SyncScheduleTable(). The call returns E_OS_ID if ScheduleTableID is not explicitly synchronized.

A successful call results in ScheduleTableID entering the state SCHEDULETABLE_WAITING. Expiry point processing for ScheduleTableID does not start until a call to SyncScheduleTable() is made while the schedule table is in state SCHEDULETABLE_WAITING.

Note that if no call to SyncScheduleTable() (or StopScheduleTable()) is made after ScheduleTableID is started synchronously, then it will remain in the state SCHEDULETABLE_WAITING indefinitely.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

Example

```
TASK(MyTask){
  StartScheduleTableSynchron(MyScheduleTable);
  ...
  SyncScheduleTable(MyScheduleTable, 12U);
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

See Also

DeclareScheduleTable
GetScheduleTableStatus
NextScheduleTable
Os_SyncScheduleTableRel
StartScheduleTableAbs
StartScheduleTableRel
StopScheduleTable
SyncScheduleTable

2.100 StopScheduleTable

Stop a schedule table.

Syntax

```
StatusType StopScheduleTable(
    ScheduleTableType ScheduleTableID
)
```

Parameters

Parameter	Mode	Description
ScheduleTableID	in	ScheduleTableType Name of the schedule table to stop.

Return Values

The call returns values of type StatusType.

Value	Build	Description
E_OK	all	No error.
E_OS_NOFUNC	all	ScheduleTableID is not running.
E_OS_ID	extended	ScheduleTableID is not a valid ScheduleTable.
E_OS_ACCESS	extended	ScheduleTableID is not accessible from the calling OS-Application.
E_OS_CALLEVEL	extended	Called from an invalid context (only when Service Protection is configured).
E_OS_DISABLEDINT	extended	Called while interrupts are disabled (only when Service Protection is configured).

Description

This call stops ScheduleTableID immediately. A call to StartScheduleTableAbs(), StartScheduleTableRel() or StartScheduleTableSynchron() (where appropriate) will re-start ScheduleTableID at the start.

Note that any schedule table that was nexted from ScheduleTableID will not start and will remain in the state SCHEDULETABLE_NEXT. StopScheduleTable() will need to be called on such tables in order to reset their state.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

Example

```
TASK(MyTask){
  ...
  StopScheduleTable(MyScheduleTable);
  ...
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

See Also

DeclareScheduleTable
GetScheduleTableStatus
NextScheduleTable
StartScheduleTableAbs
StartScheduleTableRel
StartScheduleTableSynchron

2.101 SuspendAllInterrupts

Suspend recognition of Category 1 and Category 2 interrupts.

Syntax

```
void SuspendAllInterrupts(void)
```

Description

This API call marks the start of a critical section that is protected from any maskable Category 1 or Category 2 interrupt occurring. The critical section must be left by using the ResumeAllInterrupts() call.

No API calls beside SuspendAllInterrupts()/ResumeAllInterrupts() pairs and SuspendOSInterrupts()/ResumeOSInterrupts() pairs are allowed within this critical section.

The call saves the current interrupt mask so that it can be restored later by the ResumeAllInterrupts() call.

When calls to SuspendAllInterrupts() and ResumeAllInterrupts() are nested then the interrupt recognition status saved by the first call of SuspendAllInterrupts() is restored by the last call of the ResumeAllInterrupts().

In a multicore environment, this call only affects the interrupts on the core that it is called from.

It may be called before StartOS().

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

Example

```
TASK(MyTask){
  ...
  SuspendAllInterrupts();
  ...
  ResumeAllInterrupts();
  ...
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✓	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✓		
		ProtectionHook	✓		

See Also

DisableAllInterrupts
EnableAllInterrupts
ResumeAllInterrupts
ResumeOSInterrupts
SuspendOSInterrupts

2.102 SuspendOSInterrupts

Suspend recognition of Category 2 interrupts.

Syntax

```
void SuspendOSInterrupts(void)
```

Description

This API call marks the start of a critical section that is protected from any Category 2 interrupt occurring. Category 1 interrupts may still occur. The critical section must be left using the ResumeOSInterrupts() call.

No API calls beside SuspendAllInterrupts()/ResumeAllInterrupts() pairs and SuspendOSInterrupts()/ResumeOSInterrupts() pairs are allowed within this critical section.

The call saves the current interrupt mask so that it can be restored later by the ResumeOSInterrupts() call.

When calls to SuspendOSInterrupts() and ResumeOSInterrupts() are nested then the interrupt recognition status saved by the first call of SuspendOSInterrupts() is restored by the last call of the ResumeOSInterrupts().

In a multicore environment, this call only affects the interrupts on the core that it is called from.

It may not be called before StartOS().

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

Example

```
TASK(MyTask){
  ...
  SuspendOSInterrupts();
    /* Longer Critical Section */
    ...
    SuspendAllInterrupts();
    /* Shorter Critical Section  */
    ResumeAllInterrupts();
    ...
  ResumeOSInterrupts();
  ...
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✗
Category 1 ISR	✓	PostTaskHook	✓	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✓		
		ProtectionHook	✓		

See Also

DisableAllInterrupts
EnableAllInterrupts
ResumeAllInterrupts
ResumeOSInterrupts
SuspendOSInterrupts

2.103 SyncScheduleTable

Provide the synchronization count for an explicitly synchronized schedule table.

Syntax

```
StatusType SyncScheduleTable(
    ScheduleTableType ScheduleTableID,
    TickType Value
)
```

Parameters

Parameter	Mode	Description
ScheduleTableID	in	ScheduleTableType Name of the schedule table to start.
Value	in	TickType Absolute value of the synchronizing counter.

Return Values

The call returns values of type StatusType.

Value	Build	Description
E_OK	all	No error.
E_OS_ID	all	ScheduleTableID is not a valid ScheduleTable or is not explicitly synchronized.
E_OS_ACCESS	extended	ScheduleTableID is not accessible from the calling OS-Application.
E_OS_VALUE	all	Value exceeds the duration of the table.
E_OS_STATE	all	The status of ScheduleTableID is SCHEDULETABLE_STOPPED or SCHEDULETABLE_NEXT.
E_OS_CALLEVEL	extended	Called from an invalid context (only when Service Protection is configured).
E_OS_DISABLEDINT	extended	Called while interrupts are disabled (only when Service Protection is configured).

Description

This call provides the synchronization Value for an explicitly synchronized table ScheduleTableID. ScheduleTableID must be either waiting for a synchronization value or be running.

Control of and knowledge about the synchronizing counter is outside the domain of the OS. The OS assumes that the synchronizing counter has a duration equal to ScheduleTableID and that the resolution of the synchronizing counter is equal to the resolution of the OS counter used to drive ScheduleTableID. It is your responsibility to verify that your application satisfies these constraints.

If ScheduleTableID is in the state SCHEDULETABLE_WAITING then SyncScheduleTable() causes ScheduleTableID to change state to SCHEDULETABLE_RUNNING_AND_SYNCHRONOUS and the OS to start processing expiry points. The current deviation between ScheduleTableID and the synchronization count will be zero.

The first expiry point that will be processed is the one with the smallest statically configured offset. The smallest offset is known as the InitialOffset. The point at which the first expiry point is processed is determined as follows:

- if Value is less than the InitialOffset, then the first expiry point will be processed when InitialOffset-Value ticks have elapsed on the counter driving ScheduleTableID.

- if Value is greater than or equal to InitialOffset, then the first expiry point will be processed when (Duration-Value)+InitialOffset ticks have elapsed. This may require a full wrap of the underlying drive counter before the first expiry point is processed.

This means that calling SyncScheduleTable() when ScheduleTableID is in the state SCHEDULETABLE_WAITING has behavior that is logically equivalent to calling StartScheduleTableRel() with an Offset equal to InitialOffset-Value or (Duration-Value)+InitialOffset accordingly.

If the ScheduleTableID is in the state SCHEDULETABLE_RUNNING or SCHEDULETABLE_RUNNING_AND_SYNCHRONOUS then SyncScheduleTable() will calculate the current deviation between the notional position on ScheduleTableID and Value. The deviation is equal to P-Value mod Duration. The state of ScheduleTableID is set according to the difference between the calculated deviation and the statically configured precision as follows:

- if deviation <= precision then the state will be set to SCHEDULETABLE_RUNNING_AND_SYNCHRONOUS

- if deviation > precision then the state will be set to SCHEDULETABLE_RUNNING

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

Example

```
TASK(MyTask){
  StartScheduleTableSynchron(MyScheduleTable);
  ...
  SyncScheduleTable(MyScheduleTable, 12U);
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

See Also

DeclareScheduleTable
GetScheduleTableStatus
NextScheduleTable
Os_SyncScheduleTableRel
SetScheduleTableAsync
StartScheduleTableAbs
StartScheduleTableRel
StartScheduleTableSynchron
StopScheduleTable

2.104 TerminateApplication

Terminates an OS-Application

Syntax

```
StatusType TerminateApplication(
    ApplicationType Application /* AUTOSAR R4.x only */,
    RestartType RestartOption
)
```

Parameters

Parameter	Mode	Description
Application	in	ApplicationType (AUTOSAR R4.x only) The OS-Application from which the state is requested.
RestartOption	in	RestartType RESTART or NO_RESTART.

Return Values

The call returns values of type StatusType.

Value	Build	Description
E_OK	all	No error.
E_OS_VALUE	all	RestartOption is neither RESTART nor NO_RESTART.
E_OS_CALLEVEL	all	Called from an invalid context (only when Service Protection is configured).
E_OS_STATE	all	(AUTOSAR R4.x) Called when the state of the OS-Application is already APPLICATION_TERMINATED, or APPLICATION_RESTARTING and RestartOption is RESTART, or APPLICATION_RESTARTING and the caller does not belong to the OS-Application.
E_OS_ACCESS	all	(AUTOSAR R4.x) Called from a different OS-Application that is untrusted.
E_OS_ACCESS	all	Called for a Trusted OS-Application when optimization 'Only Terminate Untrusted Applications' is configured.
E_OS_ID	all	(AUTOSAR R4.x) Application is not a valid OS-Application.

Description

NOTE: Prior to AUTOSAR R4.0, this call operates only on the OS-Application that owns the calling task or ISR.

NOTE: From AUTOSAR R4.0 onwards, trusted tasks and interrupts can specify which OS-Application to terminate.

This call terminates the OS-Application. Any running task/ISR of the OS-Application is forcibly terminated. The ready tasks/ISRs of the OS-Application are forcibly terminated before they can resume.

The interrupt sources for all Category 2 ISRs owned by the OS-Application are disabled by the OS calling Os_Cbk_Disable_<ISRName>() for each ISRName owned by the OS-Application.

All alarms owned by the OS-Application are cancelled. All schedule tables owned by the OS-Application are stopped.

If any of the tasks/ISRs hold any resources (whether standard, linked or internal) then the resources are released. Similarly, if any of the tasks/ISRs had masked interrupts using the Suspend[All|OS]Interrupts() or DisableAllInterrupts() service calls then OS will automatically call the services Resume[All|OS]Interrupts() or EnableAllInterrupts() as appropriate.

If any of the tasks/ISRs hold any Spinlocks then they are released.

If the RestartOption is RESTART, the OS-Application's restart task will be activated.

Applications should take note of the following race conditions when using TerminateApplication():

- if resources had been locked and/or interrupts masked to protect a critical section shared between OS-Applications, then be aware that the forced termination of tasks/ISRs may leave the data which is manipulated in the critical section in an unknown state. It is the application's responsibility to protect the system against the impact of this possibility.

- there is no guarantee that the restart task will execute before any other task in the OS-Application unless the restart task has the highest priority

- counters that are accessed by other OS-Applications will cease to be operational after termination until the interrupts that drive them are re-enabled. This may cause failures to propagate to other OS-Applications.

- tasks (and events set for them) in other OS-Applications that are triggered by alarms or schedule in terminated OS-Application will not be activated (or set). This may cause other OS-Applications to fail.

- (AUTOSAR R3.x) other OS-Applications that have access to any of the terminated OS-Application's objects may use those objects even if the application has been terminated. For example, another OS-Application may activate a task in a terminated OS-Application.

TerminateApplication can be used to terminate application on other cores if needed. However there may be subtle timing issues involved in doing this, so it is recommended that TerminateApplication is called on the core that contains the application.

This API is not provided if the OS option 'Omit TerminateApplication' is TRUE. Similarly it is omitted if the OS option 'Only Terminate Untrusted Applications' is TRUE but there are no untrusted OS Applications.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✓	✓	✗

Example

```
TASK(MyTask){
  ...
  if (ErrorDetected == TRUE) {
#if (OS_AR_MAJOR_VERSION < (4U))
    TerminateApplication(RESTART);
#else
    TerminateApplication(GetApplicationID(), RESTART);
#endif
  }
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✓		
		ProtectionHook	✗		

See Also

AllowAccess
GetApplicationState
Os_Cbk_Disable_<ISRName>
Os_Cbk_Terminated_<ISRName>
TerminateTask

2.105 TerminateTask

Terminates the calling task

Syntax

```
StatusType TerminateTask(void)
```

Return Values

The call returns values of type StatusType.

Value	Build	Description
E_OK	all	No error.
E_OS_RESOURCE	extended	Calling task still holds resources.
E_OS_CALLEVEL	extended	Called at interrupt level.
E_OS_CALLEVEL	extended	Called from an invalid context (only when Service Protection is configured).
E_OS_DISABLEDINT	extended	Called while interrupts are disabled (only when Service Protection is configured).
E_OS_SPINLOCK	extended	Called while the task holds a Spinlock
E_OS_MISSINGEND	extended	A task finished without calling TerminateTask or ChainTask (when protection is configured)

Description

This call terminates the calling task. This transfers the calling task from the running state to the suspended state. The call does not return to the calling context if successful.

If the calling task has queued activations pending then the next instance of the task is automatically transferred into the ready state.

Internal resources are released automatically.

Standard or linked resources are also released automatically and this is reported as an error condition in extended status.

TerminateTask() always causes re-scheduling.

If the 'Fast Terminate' is enabled in Optimizations for RTA-OS then TerminateTask() must only be called from the task entry function and the return status should not be checked (ErrorHook, when configured, will be called if there is an error). This optimization saves memory and execution time. For further savings, you can actually omit the call to TerminateTask() in SC1 and SC2.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

Example

```
TASK(MyTask){
  ...
  TerminateTask():
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✗	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

See Also

DeclareTask
GetTaskID
GetTaskState
TerminateTask

2.106 TryToGetSpinlock

TryToGetSpinlock attempts to occupy a spin-lock variable.

Syntax

```
StatusType TryToGetSpinlock(
    SpinlockIdType SpinlockId,
    TryToGetSpinlockType* Success
)
```

Parameters

Parameter	Mode	Description
SpinlockId	in	SpinlockIdType The Spinlock to occupy.
Success	out	Os_TryToGetSpinlockRefType Result of the call: TRYTOGETSPINLOCK_SUCCESS or TRYTOGETSPINLOCK_NOSUCCESS

Return Values

The call returns values of type StatusType.

Value	Build	Description
E_OK	all	No error.
E_OS_ID	extended	SpinlockId does not refer to a valid Spinlock.
E_OS_STATE	extended	The Spinlock is already occupied by the calling TASK or ISR (pre 4.0.3).
E_OS_INTERFERENCE_DEADLOCK	extended	The Spinlock is already occupied by a TASK/ISR on the same core. This would cause a deadlock.
E_OS_NESTING_DEADLOCK	extended	Attempt to occupy the Spinlock while already holding a different Spinlock in a way that may cause a deadlock.
E_OS_ACCESS	extended	SpinlockId is not accessible from the calling OS-Application.
E_OS_CALLEVEL	extended	Called from an invalid context (only when Service Protection is configured).

Description

TryToGetSpinlock has the same behavior as GetSpinlock, except that if the spinlock is already occupied by a different core, TryToGetSpinlock does not wait for the spinlock but returns E_OK and sets *Success to TRYTOGETSPINLOCK_NOSUCCESS.

If it is able to occupy the spinlock then it returns E_OK and sets *Success to TRYTOGETSPINLOCK_SUCCESS.

For a spinlock with a lock method with LOCK_ALL_INTERRUPTS, LOCK_CAT2_INTERRUPTS or LOCK_WITH_RES_SCHEDULER semantics, the lock method is only in effect if the lock succeeds. If the lock is not successful then no change is made to system schedulability.

When the spinlock has a lock method with NESTABLE semantics, this API will indicate a successful lock if the lock is already held by a TASK or ISR on the same core. The lock is only released by the original locker, and only then when it has made the same number of Release calls as Get calls.

When the spinlock has a lock method with COMMONABLE semantics, it can be followed by any spinlock that has no successors of its own, and is not also COMMONABLE. It can follow a normal lock without being on its list of successors.

Note that the OS configuration option 'Force spinlock error checks' can be used to cause the error checks to be done in standard as well as extended status builds.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✓	✗

Example

```
TASK(MyTask){
  TryToGetSpinlockType try_result;
  ...
  TryToGetSpinlock(Spinlock1, &try_result);
  if (TRYTOGETSPINLOCK_SUCCESS == try_result) {
    ...
    ReleaseSpinlock(Spinlock1);
  }
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

See Also

GetSpinlock
ReleaseSpinlock
UncheckedGetSpinlock
UncheckedReleaseSpinlock
UncheckedTryToGetSpinlock

2.107 UncheckedGetSpinlock

UncheckedGetSpinlock tries to occupy a spin-lock variable.

Syntax

```
StatusType UncheckedGetSpinlock(
    SpinlockIdType SpinlockId
)
```

Parameters

Parameter	Mode	Description
SpinlockId	in	SpinlockIdType The Spinlock to occupy.

Return Values

The call returns values of type StatusType.

Value	Build	Description
E_OK	all	No error.

Description

This is a version of GetSpinlock that can be called (with care) from Category 1 ISRs. Few checks can be done within Category 1 ISRs, so it is sensible to only use this API for spinlocks with LOCK_ALL behavior.

This API is only available if you set the OS option 'Add Spinlock APIs for CAT1 ISRs' to TRUE.

On PPC target ports, the "OS Locks disable Cat1" option should also be selected if the SEMA4 unit is being used for locking.

It may not be called before StartOS().

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
CAT1ISR(MyISR){
  ...
  UncheckedGetSpinlock(Spinlock1);
  ...
  UncheckedReleaseSpinlock(Spinlock1);
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✓	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✓		
		ProtectionHook	✓		

See Also

GetSpinlock
ReleaseSpinlock
TryToGetSpinlock
UncheckedReleaseSpinlock
UncheckedTryToGetSpinlock

2.108 UncheckedReleaseSpinlock

UncheckedReleaseSpinlock is used to release a spin-lock variable.

Syntax

```
StatusType UncheckedReleaseSpinlock(
    SpinlockIdType SpinlockId
)
```

Parameters

Parameter	Mode	Description
SpinlockId	in	SpinlockIdType The Spinlock to release.

Return Values

The call returns values of type StatusType.

Value	Build	Description
E_OK	all	No error.

Description

This is a version of ReleaseSpinlock that can be called (with care) from Category 1 ISRs. Few checks can be done within Category 1 ISRs, so it is sensible to only use this API for spinlocks with LOCK_ALL behavior.

This API is only available if you set the OS option 'Add Spinlock APIs for CAT1 ISRs' to TRUE.

On PPC target ports, the "OS Locks disable Cat1" option should also be selected if the SEMA4 unit is being used for locking.

It may not be called before StartOS().

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✓	✗

Example

```
CAT1ISR(MyISR){
  ...
  UncheckedGetSpinlock(Spinlock1);
  ...
  UncheckedReleaseSpinlock(Spinlock1);
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✓	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✓		
		ProtectionHook	✓		

See Also

GetSpinlock
ReleaseSpinlock
TryToGetSpinlock
UncheckedGetSpinlock
UncheckedTryToGetSpinlock

2.109 UncheckedTryToGetSpinlock

UncheckedTryToGetSpinlock attempts to occupy a spin-lock variable.

Syntax

```
StatusType UncheckedTryToGetSpinlock(
    SpinlockIdType SpinlockId,
    TryToGetSpinlockType* Success
)
```

Parameters

Parameter	Mode	Description
SpinlockId	in	SpinlockIdType The Spinlock to occupy.
Success	out	Os_TryToGetSpinlockRefType Result of the call: TRYTOGETSPINLOCK_SUCCESS or TRYTOGETSPINLOCK_NOSUCCESS

Return Values

The call returns values of type StatusType.

Value	Build	Description
E_OK	all	No error.

Description

This is a version of TryToGetSpinlock that can be called (with care) from Category 1 ISRs. Few checks can be done within Category 1 ISRs, so it is sensible to only use this API for spinlocks with LOCK_ALL behavior.

This API is only available if you set the OS option 'Add Spinlock APIs for CAT1 ISRs' to TRUE.

On PPC target ports, the "OS Locks disable Cat1" option should also be selected if the SEMA4 unit is being used for locking.

It may not be called before StartOS().

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✓	✗

Example

```
CAT1ISR(MyISR){
  UncheckedTryToGetSpinlockType try_result;
  ...
  UncheckedTryToGetSpinlock(Spinlock1, &try_result);
  if (TRYTOGETSPINLOCK_SUCCESS == try_result) {
    ...
    UncheckedReleaseSpinlock(Spinlock1);
  }
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✓	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✓		
		ProtectionHook	✓		

See Also

GetSpinlock
ReleaseSpinlock
TryToGetSpinlock
UncheckedGetSpinlock
UncheckedReleaseSpinlock

2.110 WaitEvent

Wait for one or more events.

Syntax

```
StatusType WaitEvent(
    EventMaskType Mask
)
```

Parameters

Parameter	Mode	Description
Mask	in	EventMaskType The event(s) to be waited upon.

Return Values

The call returns values of type StatusType.

Value	Build	Description
E_OK	all	No error.
E_OS_ACCESS	extended	Not called from an extended task.
E_OS_CALLEVEL	extended	Called from interrupt level.
E_OS_RESOURCE	extended	The calling task holds a resource.
E_OS_CALLEVEL	extended	Called from an invalid context (only when Service Protection is configured).
E_OS_DISABLEDINT	extended	Called while interrupts are disabled (only when Service Protection is configured).
E_OS_SPINLOCK	extended	Called while the task holds a Spinlock

Description

Puts the calling task into the waiting state until one of the specified events is set.

If one or more of the events is already set, then the task remains in the running state.

The API call may cause re-scheduling to take place.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

Example

```
TASK(MyExtendedTask){
  ...
  WaitEvent(Event1);
  /* Task resumes here when Event1 is set */
  ...
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✗	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

See Also

ClearEvent
DeclareEvent
GetEvent
SetEvent

2.111 WritePeripheral16

Writes the 16-bit value to the memory location at Address.

Syntax

```
StatusType WritePeripheral16(
    AreaIdType Area,
    uint16* Address,
    uint16 WriteValue
)
```

Parameters

Parameter	Mode	Description
Area	in	AreaIdType ID of the PeripheralArea.
Address	in	uint16 * Address of the value to write.
WriteValue	in	uint16 Value to write.

Return Values

The call returns values of type StatusType.

Value	Build	Description
E_OK	all	No error.
E_OS_ID	extended	Area does not match the ID of a configured PeripheralArea.
E_OS_VALUE	extended	Address does not belong to the specified area.
E_OS_CALLEVEL	extended	Called from an invalid context (no running Task or ISR).
E_OS_ACCESS	extended	The area is not accessible from the calling OS-Application.

Description

Writes a 16 bit value to the specified address.

Typically used to allow code in untrusted OS Applications to write data to areas of memory that would otherwise be inaccessible because of memory protection settings. The write is done from a trusted context.

Peripheral Areas must be declared in the configuration. Their legal address range must be specified, and also the OS-Applications that can access them.

All 16 bits of the write must be within the configured address range.

The OS does not prevent two different cores from accessing peripherals at the same time.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✓	✓	✗

Example

```
TASK(MyTask){
  ...
  WritePeripheral16(MyArea, LED_STATUS_PORT, 0x1234);
  ...
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

See Also

ModifyPeripheral16
ModifyPeripheral32
ModifyPeripheral8
ReadPeripheral16
ReadPeripheral32
ReadPeripheral8
WritePeripheral32
WritePeripheral8

2.112 WritePeripheral32

Writes the 32-bit value to the memory location at Address.

Syntax

```
StatusType WritePeripheral32(
    AreaIdType Area,
    uint32* Address,
    uint32 WriteValue
)
```

Parameters

Parameter	Mode	Description
Area	in	AreaIdType ID of the PeripheralArea.
Address	in	uint32 * Address of the value to write.
WriteValue	in	uint32 Value to write.

Return Values

The call returns values of type StatusType.

Value	Build	Description
E_OK	all	No error.
E_OS_ID	extended	Area does not match the ID of a configured PeripheralArea.
E_OS_VALUE	extended	Address does not belong to the specified area.
E_OS_CALLEVEL	extended	Called from an invalid context (no running Task or ISR).
E_OS_ACCESS	extended	The area is not accessible from the calling OS-Application.

Description

Writes a 32 bit value to the specified address.

Typically used to allow code in untrusted OS Applications to write data to areas of memory that would otherwise be inaccessible because of memory protection settings. The write is done from a trusted context.

Peripheral Areas must be declared in the configuration. Their legal address range must be specified, and also the OS-Applications that can access them.

All 32 bits of the write must be within the configured address range.

The OS does not prevent two different cores from accessing peripherals at the same time.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✓	✓	✗

Example

```
TASK(MyTask){
  ...
  WritePeripheral32(MyArea, LED_STATUS_PORT, &0x12345678);
  ...
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

See Also

ModifyPeripheral16
ModifyPeripheral32
ModifyPeripheral8
ReadPeripheral16
ReadPeripheral32
ReadPeripheral8
WritePeripheral16
WritePeripheral8

2.113 WritePeripheral8

Writes the 8-bit value to the memory location at Address.

Syntax

```
StatusType WritePeripheral8(
    AreaIdType Area,
    uint8* Address,
    uint8 WriteValue
)
```

Parameters

Parameter	Mode	Description
Area	in	AreaIdType ID of the PeripheralArea.
Address	in	uint8 * Address of the value to write.
WriteValue	in	uint8 Value to write.

Return Values

The call returns values of type StatusType.

Value	Build	Description
E_OK	all	No error.
E_OS_ID	extended	Area does not match the ID of a configured PeripheralArea.
E_OS_VALUE	extended	Address does not belong to the specified area.
E_OS_CALLEVEL	extended	Called from an invalid context (no running Task or ISR).
E_OS_ACCESS	extended	The area is not accessible from the calling OS-Application.

Description

Writes an 8 bit value to the specified address.

Typically used to allow code in untrusted OS Applications to write data to areas of memory that would otherwise be inaccessible because of memory protection settings. The write is done from a trusted context.

Peripheral Areas must be declared in the configuration. Their legal address range must be specified, and also the OS-Applications that can access them.

The OS does not prevent two different cores from accessing peripherals at the same time.

Note: Not supported of targets that use 16-bit char types.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✓	✓	✗

Example

```
TASK(MyTask){
  ...
  WritePeripheral8(MyArea, LED_STATUS_PORT, 0x12);
  ...
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✓	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

See Also

ModifyPeripheral16
ModifyPeripheral32
ModifyPeripheral8
ReadPeripheral16
ReadPeripheral32
ReadPeripheral8
WritePeripheral16
WritePeripheral32

3 RTA-OS Callbacks

3.1 Guide to Descriptions

Callbacks are code that is required by the OS but must be provided by the user. This section documents all the callbacks in RTA-OS. The descriptions have the following structure:

Syntax

```
/* C function prototype for the callback */
ReturnValue NameOfCallback(Parameter Type, ...)
```

Parameters

A list of parameters for each callback and their mode:

in The parameter is passed in to the callback by the OS

out The parameter is passed out of the API callback by passing a reference (pointer) to the parameter into the call.

inout The parameter is passed into the callback and then (updated) and passed out.

Return Values

A description of the return value of the callback,

Description

A detailed description of the required functionality of the callback.

Portability

The portability of the call between OSEK OS (ISO 17356-3), AUTOSAR OS, RTA-OS and RTA-TRACE.

Example Code

```
A C code listing showing how to implement the callback.
```

Configuration Condition

The configuration of RTA-OS that requires user code to implement the callback.

See Also

A list of related callbacks.

3.2 ErrorHook

Callback routine used for trapping errors resulting from incorrect use of the OS API.

Syntax

```
FUNC(void, {memclass}) ErrorHook(
    StatusType Error
)
```

Parameters

Parameter	Mode	Description
Error	in	StatusType The type of the error that has occurred.

Description

This is called when an API call returns a StatusType not equal to E_OK. The StatusType is passed into ErrorHook().

Macros are provided for obtaining information about the source of the error ErrorHook(), but they are only available if the OS has been configured to generate them.

The macros should only be used within ErrorHook().

(1) The macro OSErrorGetServiceId() returns an OSServiceIdType that indicates the API that raised the error. The values take the form OSServiceId_xxx where xxx is the name of an API call. e.g. OSServiceId_ActivateTask.

(2) Macros of the form OSError_<APIName>_<ParameterName>() return the values of the parameters were passed to API. e.g. OSError_ActivateTask_TaskID() (Note: The ModifyPeripheral APIs only report the first 2 parameters.)

ErrorHook runs at OS level and will not be preempted by Tasks or Category 2 ISRs.

A sample ErrorHook can be generated automatically by rtaosgen. See the RTA-OS User Guide for further details.

Note: memclass is OS_APPL_CODE for AUTOSAR 3.x, OS_ERRORHOOK_CODE for AUTOSAR 4.1 and OS_CALLOUT_CODE otherwise.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

Example

```
FUNC(void, {memclass}) ErrorHook(StatusType Error){
  switch (Error){
    case E_OS_ID:
      /* Handle illegal identifier error */
      break;
    case E_OS_VALUE:
      /* Handle illegal value error */
      break;
    case E_OS_STATE:
      /* Handle illegal state error */
      break;
    default:
      /* Handle all other types of error */
      break;
  }
}
```

Configuration Condition

Required when the ErrorHook is configured.

3.3 Os_Cbk_Cancel_<CounterID>

Callback routine to cancel the interrupt from a hardware counter.

Syntax

```
FUNC(void, {memclass}) Os_Cbk_Cancel_<CounterID>(void)
```

Description

This callback is called by the OS from within Os_AdvanceCounter when there are no Alarms or ScheduleTables running.

It should be used to stop the hardware counter from asserting matches.

This is normally done by disabling the interrupt source. You may also need to clear any related interrupt pending bits.

The hardware counter itself should continue counting.

Note: memclass is OS_APPL_CODE for AUTOSAR 3.x, OS_OS_CBK_CANCEL_<COUNTERID>_CODE for AUTOSAR 4.1 and OS_CALLOUT_CODE otherwise.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
FUNC(void, {memclass}) Os_Cbk_Cancel_MyCounter(void) {
  DISABLE_HW_COUNTER_INTERRUPT_SOURCE;
  CLEAR_HW_COUNTER_PENDING_INTERRUPT;
}
```

Configuration Condition

Required for each hardware counter configured.

See Also

Os_Cbk_Now_<CounterID>
Os_Cbk_Set_<CounterID>
Os_Cbk_State_<CounterID>
Os_AdvanceCounter

3.4 Os_Cbk_CheckMemoryAccess

Check if a memory region is read/write/execute/stack accessible by a specified OS-Application.

Syntax

```
FUNC(AccessType, {memclass}) Os_Cbk_CheckMemoryAccess(
    ApplicationType Application,
    TaskType TaskID,
    ISRType ISRID,
    MemoryStartAddressType Address,
    MemorySizeType Size
)
```

Parameters

Parameter	Mode	Description
Application	in	ApplicationType The OS-Application to which the task or ISR belongs. INVALID_OSAPPLICATION if there are no OS-Applications.
TaskID	in	TaskType If not INVALID_TASK, the task for which the memory access is being checked.
ISRID	in	ISRType If not INVALID_ISR, the ISR for which the memory access is being checked.
Address	in	MemoryStartAddressType The start address of the memory area.
Size	in	MemorySizeType The size in bytes of the memory area.

Return Values

The call returns values of type AccessType.

Description

This callback is provided so that you have control over the memory access permissions that you apply to specific objects within a particular project. For example, you may choose to limit write-access for a Task but allow it read access.

It gets called by RTA-OS when it needs to determine whether access to a particular memory location is allowable.

Note that this mechanism is independent of the the memory protection system - it is present even when memory protection is disabled.

RTA-OS will call Os_Cbk_CheckMemoryAccess whenever the AUTOSAR CheckTaskMemoryAccess() or CheckISRMemoryAccess() service calls are made. This effectively delegates the decision about access permissions to you, because RTA-OS cannot know how you have chosen to partition memory.

RTA-OS will also call Os_Cbk_CheckMemoryAccess when there are access-restricted OS applications (untrusted or trusted-with-protection) in the configuration and it needs to check that addresses passed to 'Get' APIs are legal to write to. For example, GetAlarm() is passed a TickRefType into which it writes a value. The address is validated before use by calling Os_Cbk_CheckMemoryAccess.

The implementation of the callback needs to determine whether the memory locations bounded by Address and (Address + Size) are available for read/write/execute/stack access by the OS-Application (and optionally the Task or ISR that is specified).

When called in response to a CheckTaskMemoryAccess() service call, then ISRID will be set to INVALID_ISR. Similarly, if called in response to a CheckISRMemoryAccess() call, the OS will set TaskID to INVALID_TASK.

When called in response to a pointer check (such as for GetAlarm()), Application will be set to the calling OS-Application, ISRID will be set to INVALID_ISR and TaskID will be set to INVALID_TASK.

The returned AccessType can be constructed using the following constants:

OS_ACCESS_READ - the memory range is readable

OS_ACCESS_EXECUTE - the memory range is executable

OS_ACCESS_WRITE - the memory range is writeable

OS_ACCESS_STACK - the memory range is stack

Note: memclass is OS_APPL_CODE for AUTOSAR 3.x, OS_OS_CBK_CHECKMEMORYACCESS_CODE for AUTOSAR 4.1 and OS_CALLOUT_CODE otherwise.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
FUNC(AccessType, {memclass}) Os_Cbk_CheckMemoryAccess(ApplicationType
    Application, TaskType TaskID, ISRType ISRID, MemoryStartAddressType
    Address, MemorySizeType Size) {
  AccessType Access = OS_ACCESS_EXECUTE;
  /* Check for stack space in address range */
  if ((Address >= STACK_BASE) && (Address + Size < STACK_BASE +
      STACK_SIZE) ) {
      Access |= OS_ACCESS_STACK;
  }
  /* Address range is read/write if it is in RAM */
  if ((Address >= RAM_BASE) && (Address + Size < RAM_BASE + RAM_SIZE) ) {
      Access |= (OS_ACCESS_WRITE | OS_ACCESS_READ);
  }
```

```
  switch (Application) {
    case APP1:
      /* Trusted application - no further restrictions */
      break;
    case APP2:
      /* Access-restricted application - write restrictions */
      if ((Address <= APP2_BASE) || (Address + Size > APP2_BASE +
          APP2_SIZE) ) {
        Access &= ~OS_ACCESS_WRITE;
      }
      break;
  }
  return Access;
}
```

Configuration Condition

Required when memory access checking is used.

See Also

CheckISRMemoryAccess
CheckTaskMemoryAccess
Os_Cbk_SetMemoryAccess
Os_Cbk_GetSetProtection

3.5 Os_Cbk_CheckStackDepth

Callback routine used to monitor system stack usage at run time.

Syntax

```
FUNC(void, {memclass}) Os_Cbk_CheckStackDepth(
    CoreIdType Core_id,
    Os_StackSizeType Depth,
    Os_StackSizeType CurrentPos
)
```

Parameters

Parameter	Mode	Description
Core_id	in	CoreIdType The ID of the calling core (OS_CORE_ID_MASTER for single core configurations).
Depth	in	Os_StackSizeType The amount of stack used since StartOS.
CurrentPos	in	Os_StackOverrunType The value of the stack pointer(s) at the point of the call.

Description

This hook routine is only called if the OS option 'Stack Sampling' is configured.

It is intended to be used to help track the stack usage of the overall system, rather than of individual TASKs and ISRs.

It gets called just before any TASK or ISR starts.

It also gets called from within the Os_GetStackUsage API, so that you can insert monitoring points in your application code.

The APIs GetISRID and GetTaskID can be used within the callback to check what TASK/ID is running (or just about to run).

Note: memclass is OS_APPL_CODE for AUTOSAR 3.x, OS_OS_CBK_CHECKSTACKDEPTH_CODE for AUTOSAR 4.1 and OS_CALLOUT_CODE otherwise.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Configuration Condition

The callback must be provided if OS option 'Stack Sampling' is enabled.

See Also

Os_GetStackUsage
Os_GetISRMaxStackUsage
Os_GetTaskMaxStackUsage
Os_ResetISRMaxStackUsage
Os_ResetTaskMaxStackUsage
GetISRID
GetTaskID

3.6 Os_Cbk_CrosscoreISREnd

Callback routine indicating the end of a cross-core ISR.

Syntax

```
FUNC(void, {memclass}) Os_Cbk_CrosscoreISREnd(
    CoreIdType core_id
)
```

Description

Os_Cbk_CrosscoreISREnd() must be implemented in a multi-core configuration if OS option 'Additional ISR Hooks' is enabled. It gets called at the point that a cross-core ISR handler finishes execution.

Note: memclass is OS_APPL_CODE for AUTOSAR 3.x, OS_OS_CBK_CROSSCOREISREND_CODE for AUTOSAR 4.1 and OS_CALLOUT_CODE otherwise.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
uint32 CrosscoreEndCount[OS_NUM_CORES];
FUNC(void, {memclass}) Os_Cbk_CrosscoreISREnd(CoreIdType core_id) {
  CrosscoreEndCount[core_id] += 1;
}
```

Configuration Condition

The callback must be provided in a multi-core configuration if OS option 'Additional ISR Hooks' is enabled.

See Also

Os_Cbk_ISRStart
Os_Cbk_ISREnd
Os_Cbk_CrosscoreISRStart

3.7 Os_Cbk_CrosscoreISRStart

Callback routine indicating the start of a cross-core ISR.

Syntax

```
FUNC(void, {memclass}) Os_Cbk_CrosscoreISRStart(
    CoreIdType core_id
)
```

Description

Os_Cbk_CrosscoreISRStart() must be implemented in a multi-core configuration if OS option 'Additional ISR Hooks' is enabled. It gets called at the point that a cross-core ISR handler starts execution.

Note: memclass is OS_APPL_CODE for AUTOSAR 3.x, OS_OS_CBK_CROSSCOREISRSTART_CODE for AUTOSAR 4.1 and OS_CALLOUT_CODE otherwise.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
uint32 CrosscoreStartCount[OS_NUM_CORES];
FUNC(void, {memclass}) Os_Cbk_CrosscoreISRStart(CoreIdType core_id) {
  CrosscoreStartCount[core_id] += 1;
}
```

Configuration Condition

The callback must be provided in a multi-core configuration if OS option 'Additional ISR Hooks' is enabled.

See Also

Os_Cbk_ISREnd
Os_Cbk_ISRStart
Os_Cbk_CrosscoreISREnd

3.8 Os_Cbk_Disable_<ISRName>

Callback routine indicating that the ISR <ISRName> must be disabled.

Syntax

```
FUNC(void, {memclass}) Os_Cbk_Disable_<ISRName>(void)
```

Description

The OS calls this function during TerminateApplication to request that the interrupt source associated with the named ISR is disabled.

AUTOSAR requires that all interrupts belonging to an OS Application are disabled when it is terminated.

You would normally re-enable an OS Application's interrupts in its Restart Task.

Note: memclass is OS_APPL_CODE for AUTOSAR 3.x, OS_OS_CBK_DISABLE_<ISRNAME>_CODE for AUTOSAR 4.1 and OS_CALLOUT_CODE otherwise.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
FUNC(void, {memclass}) Os_Cbk_Disable_App2Isr1(void) {
    disable_interrupt_source(_App2Isr1_);
}
```

Configuration Condition

Required for each ISR if TerminateApplication is supported.

See Also

ProtectionHook
TerminateApplication

3.9 Os_Cbk_GetSetProtection

Callback routine used to control the activation of the memory protection system.

Syntax

```
FUNC(boolean, {memclass}) Os_Cbk_GetSetProtection(
    boolean enable
)
```

Return Values

The call returns values of type `boolean`.

Description

This callback is used in some systems that have OS Applications where TrustedApplicationWithProtection is true. It must return the state of the memory protection hardware at the point it was called (TRUE if enabled, FALSE otherwise). It must then enable or disable memory protection based on the incoming 'enable' value. It is used to switch between Trusted and TrustedApplicationWithProtection modes.

This callback is not needed on all target platforms. Platforms such as the TriCore can provide separate memory protection sets for untrusted, trusted and trusted-with-protection modes. In this case the callback is not used. Platforms such as PowerPC only have 2 effective protection sets, so transitioning from trusted-with-protection to trusted mode is most efficiently performed by disabling the memory protection hardware while running trusted code. The target Port Guide will say if this callback has to be implemented.

Note: memclass is OS_APPL_CODE for AUTOSAR 3.x, OS_OS_CBK_GETSETPROTECTION_CODE for AUTOSAR 4.1 and OS_CALLOUT_CODE otherwise.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
FUNC(boolean {memclass}) Os_Cbk_GetSetProtection(boolean enable) {
    boolean initial = MPU.enabled;
    MPU.enabled = enable.
    return initial;
}
```

Configuration Condition

The callback must be provided on some targets when memory protection is selected and there are OS Applications where TrustedApplicationWithProtection is true.

See Also

Os_Cbk_CheckMemoryAccess
Os_UntrustedContextType
CallTrustedFunction
CallAndProtectFunction
Os_Cbk_SetMemoryAccess

3.10 Os_Cbk_GetStopwatch

Callback routine to get the current value of a free-running counter.

Syntax

```
FUNC(Os_StopwatchTickType, {memclass}) Os_Cbk_GetStopwatch(void)
```

Return Values

The call returns values of type Os_StopwatchTickType.

Description

Os_Cbk_GetStopwatch() must return the current value of a free-running timer which increments and overflows at the end of its range.

This timer provides the timebase for execution time and trace measurements.

This callback gets accessed very frequently when the OS is making timing measurements. It can improve performance to provide an implementation that can be inlined into the compiled code. This can be done by implementing Os_Cbk_GetStopwatch as a C preprocessor macro in a header file, and then injecting this header file into the build via the –using command-line option. (e.g. –using:MyStopwatch.h in the example.)

Note that files injected to the build via the –using command-line option appear before all other OS header files, so you may need to provide a hard-coded peripheral address rather than addressing the peripheral by name.

Note: memclass is OS_APPL_CODE for AUTOSAR 3.x, OS_OS_CBK_GETSTOPWATCH_CODE for AUTOSAR 4.1 and OS_CALLOUT_CODE otherwise.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
FUNC(Os_StopwatchTickType, {memclass}) Os_Cbk_GetStopwatch(void){
  return (Os_StopwatchTickType)HARDWARE_TIMER_CHANNEL;
}

..or in file MyStopwatch.h:
#define Os_Cbk_GetStopwatch()
    ((Os_StopwatchTickType)HARDWARE_TIMER_CHANNEL)
```

Configuration Condition

The callback must be provided if time monitoring or tracing is configured in the OS.

See Also

Os_GetExecutionTime
Os_GetISRMaxExecutionTime
Os_GetTaskMaxExecutionTime
Os_ResetISRMaxExecutionTime
Os_ResetTaskMaxExecutionTime

3.11 Os_Cbk_ISREnd

Callback routine indicating the end of a Category 2 ISR.

Syntax

```
FUNC(void, {memclass}) Os_Cbk_ISREnd(
    ISRType isr
)
```

Description

Os_Cbk_ISREnd() must be implemented if OS option 'Additional ISR Hooks' is enabled. It gets called at the point that a Category 2 ISR finishes execution.

It does not get called when a ISR is pre-empted. It runs at or above the ISR's interrupt priority.

Note: memclass is OS_APPL_CODE for AUTOSAR 3.x, OS_OS_CBK_ISREND_CODE for AUTOSAR 4.1 and OS_CALLOUT_CODE otherwise.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
Os_StopwatchTickType isr_exe_time[OS_NUM_ISRS];
FUNC(void, {memclass}) Os_Cbk_ISREnd(ISRType isr) {
  isr_exe_time[OS_ISRTYPE_TO_INDEX(isr)] = GetExecutionTime();
}
```

Configuration Condition

The callback must be provided if OS option 'Additional ISR Hooks' is enabled.

See Also

OS_ISRTYPE_TO_INDEX
Os_Cbk_TaskStart
Os_Cbk_ISRStart
Os_Cbk_TaskEnd
Os_Cbk_CrosscoreISRStart
Os_Cbk_CrosscoreISREnd

3.12 Os_Cbk_ISRStart

Callback routine indicating the start of a Category 2 ISR.

Syntax

```
FUNC(void, {memclass}) Os_Cbk_ISRStart(
    ISRType isr
)
```

Description

Os_Cbk_ISRStart() must be implemented if OS option 'Additional ISR Hooks' is enabled. It gets called at the point that a Category 2 ISR is about to start execution.

It does not get called when a ISR is pre-empted. It runs at or above the ISR's interrupt priority.

Note: memclass is OS_APPL_CODE for AUTOSAR 3.x, OS_OS_CBK_ISRSTART_CODE for AUTOSAR 4.1 and OS_CALLOUT_CODE otherwise.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
uint32 isr_starts[OS_NUM_ISRS];
FUNC(void, {memclass}) Os_Cbk_ISRStart(ISRType isr) {
  isr_starts[OS_ISRTYPE_TO_INDEX(isr)] += 1;
}
```

Configuration Condition

The callback must be provided if OS option 'Additional ISR Hooks' is enabled.

See Also

OS_ISRTYPE_TO_INDEX
Os_Cbk_TaskEnd
Os_Cbk_TaskStart
Os_Cbk_ISREnd
Os_Cbk_CrosscoreISRStart
Os_Cbk_CrosscoreISREnd

3.13 Os_Cbk_Idle

Runs when the OS becomes idle.

Syntax

```
FUNC(boolean, {memclass}) Os_Cbk_Idle(void)
```

Return Values

The call returns values of type boolean.

Description

Os_Cbk_Idle() is called when the OS first becomes idle after startup. Any autostarted tasks will have run before it gets called.

If Os_Cbk_Idle() exits with a return value TRUE then it will be called again immediately. If Os_Cbk_Idle() exits with a return value FALSE then it will not be called again and the OS will busy wait when there are no tasks or ISRs ready to run.

A default implementation is supplied in the library that returns FALSE.

The default version is not provided if OS option "Omit Default Implementations" is TRUE.

Note: memclass is OS_APPL_CODE for AUTOSAR 3.x, OS_OS_CBK_IDLE_CODE for AUTOSAR 4.1 and OS_CALLOUT_CODE otherwise.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
FUNC(boolean, {memclass}) Os_Cbk_Idle(void) {
  sleep();
  return TRUE;
}
```

Configuration Condition

Optional in user code. No configuration required.

See Also

StartOS
ShutdownOS
ShutdownAllCores

3.14 Os_Cbk_InShutdown

Runs when OS is in shutdown.

Syntax

```
FUNC(void, {memclass}) Os_Cbk_InShutdown(void)
```

Description

Os_Cbk_InShutdown() is called when the OS is in shutdown, after any shutdown hooks have run.

It will be called repeatedly in an infinite loop, because ShutdownOS does not return.

It can be called by each core that is in shutdown.

A default implementation is supplied in the library that that does nothing.

The default version is not provided if OS option "Omit Default Implementations" is TRUE.

In most systems you will perform any shutdown actions in a shutdown hook before this is ever called.

Note: memclass is OS_APPL_CODE for AUTOSAR 3.x, OS_OS_CBK_INSHUTDOWN_CODE for AUTOSAR 4.1 and OS_CALLOUT_CODE otherwise.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
FUNC(void, {memclass}) Os_Cbk_InShutdown(void) {
  reset_chip();
}
```

Configuration Condition

Optional in user code. No configuration required.

See Also

ShutdownOS
ShutdownHook

3.15 Os_Cbk_IsSystemTrapAllowed

Callback routine to determine if the caller is allowed to access the system trap.

Syntax

```
FUNC(boolean, {memclass}) Os_Cbk_IsSystemTrapAllowed(
    MemoryStartAddressType Caller
)
```

Return Values

The call returns values of type boolean.

Description

Most RTA-OS targets use the system trap to switch the processor from trusted to untrusted modes. Although access to this can be somewhat limited (using the 'Guard supervisor access' target option), it is still possible for untrusted code to call the trap and wrongly gain elevated permissions.

The Os_Cbk_IsSystemTrapAllowed() can be used to prevent this. It gets called by the OS just before it performs the switch to trusted mode. It is passed the return address of the code that made the trap call. You can use this address to determine if the code is allowed to switch to trusted mode.

If it returns TRUE then the switch to trusted is allowed.

If it returns FALSE then no switch is made and the trap returns without doing anything.

It is assumed that the implementation of the callback is provided by a secure part of the system that is able to police valid accesses to the trap. The implementation will typically use the caller address to decide if the trap is allowed. It cannot just rely on GetTaskID/GetISRID because the OS has to use the trap to get back to trusted mode when untrusted TASKs and ISRs complete. It has to do this before it can change the running TASK/ISR id.

In most applications it is only the OS code that needs to perform these mode switches, so a simple implementation will ensure that all OS APIs are located in a small contiguous block or memory and that the code simple checks that the caller is within this range.

Be careful not to implement any 'helper functions' that use the system trap to change to trusted mode. If they can be called from untrusted code then this will defeat the protection, because the caller will be the helper function, not the untrusted code.

Note: memclass is OS_APPL_CODE for AUTOSAR 3.x, OS_OS_CBK_ISSYSTEMTRAPALLOWED_CODE for AUTOSAR 4.1 and OS_CALLOUT_CODE otherwise.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
FUNC(boolean, {memclass})
    Os_Cbk_IsSystemTrapAllowed(MemoryStartAddressType Caller) {
  return ( (Caller >= MyTrustedCodeStartAddress) && (Caller <=
     MyTrustedCodeEndAddress));
}
```

Configuration Condition

The callback must be provided if the 'Enhanced Isolation' target option is TRUE. Note that not all targets support this option.

See Also

Os_Cbk_RestoreGlobalRegisters
Os_Cbk_IsUntrustedCodeOK
Os_Cbk_IsUntrustedTrapOK

3.16 Os_Cbk_IsUntrustedCodeOK

Callback routine used to check the validity of untrusted code.

Syntax

```
FUNC(ProtectionReturnType, {memclass}) Os_Cbk_IsUntrustedCodeOK(
    { parameters are target-dependent }
)
```

Return Values

The call returns values of type `ProtectionReturnType`.

Description

When the 'Enhanced Isolation' target option is TRUE, the OS assumes that untrusted code might somehow corrupt its registers and any system context that it is able to access.

In this case any interrupt that preempts untrusted code gets intercepted by the OS. Details of the registers in use by the untrusted code are passed to this callback which must decide whether they show that the code has misbehaved.

A return value of PRO_IGNORE means that the code will be allowed to continue after the interrupt completes.

A return value of PRO_TERMINATETASKISR means that the untrusted code will be terminated cleanly, the interrupt will run and then the system will continue as if the untrusted code had completed correctly.

A return value of PRO_TERMINATEAPPL does the same as PRO_TERMINATETASKISR, but also terminates any other TASKs or ISRs that are part of the OS Application owning the untrusted code.

A return value of PRO_TERMINATEAPPL_RESTART does the same as PRO_TERMINATEAPPL, but also then activates the relevant restart TASK.

Note: memclass is OS_APPL_CODE for AUTOSAR 3.x, OS_OS_CBK_ISUNTRUSTEDCODEOK_CODE for AUTOSAR 4.1 and OS_CALLOUT_CODE otherwise.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
FUNC(ProtectionReturnType, {memclass}) Os_Cbk_IsUntrustedCodeOK({
    parameters are target-dependent }) {
    return PRO_IGNORE;
}
```

Configuration Condition

The callback must be provided if the ’Enhanced Isolation’ target option is TRUE. Note that not all targets support this option.

See Also

Os_Cbk_RestoreGlobalRegisters
Os_Cbk_IsSystemTrapAllowed
ProtectionHook
Os_Cbk_IsUntrustedTrapOK

3.17 Os_Cbk_IsUntrustedTrapOK

Callback routine used to check what to do when untrusted code causes a processor trap/exception.

Syntax

```
FUNC(ProtectionReturnType, {memclass}) Os_Cbk_IsUntrustedTrapOK(
    { parameters are target-dependent }
)
```

Return Values

The call returns values of type `ProtectionReturnType`.

Description

When the 'Enhanced Isolation' target option is TRUE, any trap that is caused by untrusted code gets intercepted by the OS. Details of the trap are passed to this callback which must decide the action to be taken.

A return value of PRO_IGNORE means that the code will be allowed to continue to execute the normal trap handling code.

A return value of PRO_TERMINATETASKISR means that the untrusted code will be terminated cleanly and the rest of the system will continue operating unaffected.

A return value of PRO_TERMINATEAPPL does the same as PRO_TERMINATETASKISR, but also terminates any other TASKs or ISRs that are part of the OS Application owning the untrusted code.

A return value of PRO_TERMINATEAPPL_RESTART does the same as PRO_TERMINATEAPPL, but also then activates the relevant restart TASK.

The callback runs on the 'Enhanced Isolation' stack. It may not make any OS API calls because interrupts will be globally disabled.

Note: memclass is OS_APPL_CODE for AUTOSAR 3.x, OS_OS_CBK_ISUNTRUSTEDTRAPOK_CODE for AUTOSAR 4.1 and OS_CALLOUT_CODE otherwise.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
FUNC(ProtectionReturnType, {memclass}) Os_Cbk_IsUntrustedTrapOK({
    parameters are target-dependent }) {
    return PRO_IGNORE;
}
```

Configuration Condition

The callback must be provided if the ’Enhanced Isolation’ target option is TRUE. Note that not all targets support this option.

See Also

Os_Cbk_RestoreGlobalRegisters
Os_Cbk_IsSystemTrapAllowed
ProtectionHook
Os_Cbk_IsUntrustedCodeOK

3.18 Os_Cbk_Now_<CounterID>

Callback routine that returns the current tick value of the counter.

Syntax

```
FUNC(TickType, {memclass}) Os_Cbk_Now_<CounterID>(void)
```

Return Values

The call returns values of type TickType.

Description

The callback must return the current value of the hardware counter.

It can be called on any core in a multi-core system.

Note: memclass is OS_APPL_CODE for AUTOSAR 3.x, OS_OS_CBK_NOW_<COUNTERID>_CODE for AUTOSAR 4.1 and OS_CALLOUT_CODE otherwise.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
FUNC(TickType, {memclass}) Os_Cbk_Now_MyCounter(void){
  return (TickType) HW_COUNTER_NOW_VALUE;
}
```

Configuration Condition

Required for each hardware counter configured.

See Also

Os_Cbk_Cancel_<CounterID>
Os_Cbk_Set_<CounterID>
Os_Cbk_State_<CounterID>

3.19 Os_Cbk_RegSetRestore_<RegisterSetID>

Callback routine requiring that the context for register set <RegisterSetID> gets restored.

Syntax

```
FUNC(void, {memclass}) Os_Cbk_RegSetRestore_<RegisterSetID>(
    Os_RegSetDepthType Depth
)
```

Description

This callback is provided so that the application can restore the current context for register set <RegisterSetID>.

Depth gives the position in the application-provided save buffer from which the context must be read. It ranges from zero to (OS_REGSET_<RegisterSetID>_SIZE - 1).

The RegisterSet implementation is multicore aware. Note that it is possible for the callback to be running on 2 cores at the same time.

Note: memclass is OS_APPL_CODE for AUTOSAR 3.x, OS_OS_CBK_REGSETRESTORE_<REGISTERSETID>_CODE for AUTOSAR 4.1 and OS_CALLOUT_CODE otherwise.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
#ifdef OS_REGSET_FP_SIZE
static fp_context_save_area fpsave[OS_REGSET_FP_SIZE];
FUNC(void, {memclass}) Os_Cbk_RegSetRestore_FP(Os_RegSetDepthType Depth){
  ... = fpsave[Depth];
}
#endif /* OS_REGSET_FP_SIZE */
```

Configuration Condition

The callback must be provided if Register Set <RegisterSetID> exists and preemption may require its context to be restored.

See Also

OS_REGSET_<RegisterSetID>_SIZE
Os_Cbk_RegSetSave_<RegisterSetID>

3.20 Os_Cbk_RegSetSave_<RegisterSetID>

Callback routine requiring that the context for register set <RegisterSetID> gets saved.

Syntax

```
FUNC(void, {memclass}) Os_Cbk_RegSetSave_<RegisterSetID>(
    Os_RegSetDepthType Depth
)
```

Description

This callback is provided so that the application can save the current context for register set <RegisterSetID>.

Depth gives the position in the application-provided save buffer into which the context must be stored. It ranges from zero to (OS_REGSET_<RegisterSetID>_SIZE - 1).

The RegisterSet implementation is multicore aware. It saves context from different cores in different locations. RegisterSets cannot protect shared context across cores, however. Note that it is possible for the callback to be running on 2 cores at the same time.

Note: memclass is OS_APPL_CODE for AUTOSAR 3.x, OS_OS_CBK_REGSETSAVE_<REGISTERSETID>_CODE for AUTOSAR 4.1 and OS_CALLOUT_CODE otherwise.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
#ifdef OS_REGSET_FP_SIZE
static fp_context_save_area fpsave[OS_REGSET_FP_SIZE];
FUNC(void, {memclass}) Os_Cbk_RegSetSave_FP(Os_RegSetDepthType Depth){
  fpsave[Depth] = ...;
}
#endif /* OS_REGSET_FP_SIZE */
```

Configuration Condition

The callback must be provided if Register Set <RegisterSetID> exists and preemption may require its context to be saved.

See Also

OS_REGSET_<RegisterSetID>_SIZE
Os_Cbk_RegSetRestore_<RegisterSetID>

3.21 Os_Cbk_RestoreGlobalRegisters

Callback routine used to ensure that any system-global registers contain correct values.

Syntax

```
FUNC(void, {memclass}) Os_Cbk_RestoreGlobalRegisters(void)
```

Return Values

The call returns values of type boolean.

Description

When the 'Enhanced Isolation' target option is TRUE, the OS assumes that untrusted code might somehow corrupt its registers and any system context that it is able to access.

It inserts extra checking / recovery code into the OS at all points where untrusted code terminates or gets preempted. This code runs on a special safe stack, and runs with all interrupts disabled.

It calls Os_Cbk_RestoreGlobalRegisters() very early in the checks so that the callback can reload any system-global registers or other system context. Typically this might mean reloading any registers used as base registers for small-data areas.

The callback runs on the 'Enhanced Isolation' stack and can be implemented in C provided that the generated processor instructions do not try to use system-global registers before they have been restored. It may not make any OS API calls because interrupts will be globally disabled.

It is not necessary to check whether the values were corrupted. It is simpler simply to reset them.

Note: on some targets one or more of these global registers might be deliberately corrupted by the Enhanced Isolation code because it needs them as part of the stack-swapping code.

Note: memclass is OS_APPL_CODE for AUTOSAR 3.x, OS_OS_CBK_RESTOREGLOBALREGISTERS_CODE for AUTOSAR 4.1 and OS_CALLOUT_CODE otherwise.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
FUNC(void, {memclass}) Os_Cbk_RestoreGlobalRegisters(void) {
  ASM("mov A0, _sda_RAM_base");
  ASM("mov A1, _sda_ROM_base");
}
```

Configuration Condition

The callback must be provided if the ’Enhanced Isolation’ target option is TRUE. Note that not all targets support this option.

See Also

Os_Cbk_IsSystemTrapAllowed
Os_Cbk_IsUntrustedCodeOK
Os_Cbk_IsUntrustedTrapOK

3.22 Os_Cbk_SetMemoryAccess

Callback routine used to prepare the memory protection system for a switch from trusted to access-restricted code (untrusted or trusted-with-protection).

Syntax

```
FUNC(void, {memclass}) Os_Cbk_SetMemoryAccess(
    Os_UntrustedContextRefType ApplicationContext
)
```

Parameters

Parameter	Mode	Description
ApplicationContext	in	Os_UntrustedContextRefType A reference to an Os_UntrustedContextType that describes the Untrusted context.

Description

This callback is provided so that you have full control over the memory protection hardware on your device, and so that you can decide the degree of protection that you want to apply on a particular project. For example, you may choose to limit write-access for access-restricted code but allow any read and execute access. Alternatively you may wish to limit read/write and execute access for access-restricted code.

In an AUTOSAR OS, code that runs in the context of a Trusted OS Application is assumed to have full access to any area of RAM, ROM or IO space that is available. Such code runs in a privileged mode. On the other hand, code that runs in the context of an access-restricted OS Application may have restrictions placed on it that prevent it from being able access certain areas. Such code typically runs in 'user' (untrusted) mode, but there is also an AUTOSAR option to configure trusted OS Applications to run with memory protection enabled. Trusted-with-protection code behaves in most ways like trusted code. The only difference is that it runs with restricted access to memory.

Whenever RTA-OS is about to switch from trusted to access-restricted code, it makes a call to Os_Cbk_SetMemoryAccess. It passes in a reference to an Os_UntrustedContextType data structure that you can use to determine what permissions to set for access-restricted code. The Os_UntrustedContextType structure contains information about the OS Application, Task/ISR and stack region that applies to the code that is about to be executed. Depending on the context of the switch, some of these may contain NULL values. Os_Cbk_SetMemoryAccess is only called from trusted code.

Os_Cbk_SetMemoryAccess gets called in the following cases:

1) Before calling a TASK that belongs to an access-restricted OS-Application.

2) Before calling a Category 2 ISR that belongs to an access-restricted OS-Application.

3) Before calling an access-restricted OS-Application Startup, Shutdown or Error hook.

4) Before calling a 'TrustedFunction' that belongs to an access-restricted OS-Application. (This extends the AUTOSAR concept, and allows a core trusted task to call out to access-restricted code supplied by third parties.)

When using memory protection features, you must initialize the memory protection hardware before calling StartOS(). You can choose what hardware to use, how many regions to protect and what restrictions to apply.

If you want to run all access-restricted code with the same memory protection settings, then you can set the 'Single Memory Protection Zone' OS option. In this case Os_Cbk_SetMemoryAccess will not be called. You must set up the MPU before running any access-restricted code.

If you want to run all access-restricted code with the same basic memory protection settings but apply protection to the stack, then you can set the 'Stack Only Memory Protection' OS option. In this case Os_Cbk_SetMemoryAccess will only be passed the stack-related fields (Address and Size) plus Application. You must ensure that the memory protection settings limit the stack to the specified range.

* Note *

On certain target processors supported by RTA-OS, there are restrictions on the addresses that can be used to configure MPU protection regions. For example they may have to be aligned on a 64 byte boundary. If you wish to fully protect the stack in these cases, RTA-OS supports an extra field in the Os_UntrustedContextType called 'AlignedAddress'.

When 'AlignedAddress' is present, its value is initially set to the same value as 'Address'. You may change its value so that it reflects the next address on the stack that would be legal for the MPU. For example you might change it from 0x580 to 0x500 if the region has to start on a 256-byte boundary (and the stack grows to lower addresses!).

On targets that support this, RTA-OS will detect the change in 'AlignedAddress' and ensure that the stack is moved to this position just before the access-restricted code is run so that it operates in the memory protection region that you set up.

You will have to account for these adjustments in any stack budgets that you declare.

You must not attempt to move the stack to a position that would not be on the normal stack. This will invalidate many of the assumptions and optimizations in RTA-OS.

This mechanism is only available on the RTA-OS target ports that support it and provide the command-line option 'Enable stack repositioning' ('AlignUntrustedStacks' in the ARXML).

Note that with Stack repositioning in effect, the stack budgets that you need to assign are typically larger than normal by a value equal to the positioning granularity.

* Note *

'FunctionID' and 'FunctionParams' are only present when there are access-restricted functions. The value of 'FunctionID' will be INVALID_FUNCTION except when the callback is for an access-restricted function. In this case, 'FunctionID' contains the function identifier and 'FunctionParams' is a copy of the pointer to the parameters of the function.

* Note *

'CoreID' is only present where there are multiple AUTOSAR cores, and it holds the number of the current core.

Note: memclass is OS_APPL_CODE for AUTOSAR 3.x, OS_OS_CBK_SETMEMORYACCESS_CODE for AUTOSAR 4.1 and OS_CALLOUT_CODE otherwise.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
FUNC(void, {memclass}) Os_Cbk_SetMemoryAccess(Os_UntrustedContextRefType
    ApplicationContext) {
  /*
   * When called for an access-restricted TASK:
   *  ApplicationContext->Application contains the ID of the OS
       Application that the TASK belongs to.
   *  ApplicationContext->TaskID is the ID of the TASK
   *  ApplicationContext->ISRID is INVALID_ISR
   *  ApplicationContext->Address is the starting address for the TASK's
       stack.
   *  ApplicationContext->Size is the stack budget configured for the
       TASK. (Zero if no budget.)
   *  ApplicationContext->Trusted is true if the OS Application is trusted
       (TrustedApplicationWithProtection)
   *
   * When called for an access-restricted ISR:
   *  ApplicationContext->Application contains the ID of the OS
       Application that the ISR belongs to.
   *  ApplicationContext->TaskID is INVALID_TASK
   *  ApplicationContext->ISRID is the ID of the ISR
   *  ApplicationContext->Address is the starting address for the ISR's
       stack.
   *  ApplicationContext->Size is the stack budget configured for the ISR.
       (Zero if no budget.)
   *  ApplicationContext->Trusted is true if the OS Application is trusted
       (TrustedApplicationWithProtection)
   *
   * When called for:
   *    - an access-restricted Function
   *    - an access-restricted OS Application error hook
```

```
*    - an access-restricted OS Application startup hook
*    - an access-restricted OS Application shutdown hook
*  ApplicationContext->Application contains the ID of the OS
    Application that the function/hook belongs to.
*  ApplicationContext->TaskID is INVALID_TASK
*  ApplicationContext->ISRID is INVALID_ISR
*  ApplicationContext->Address is the value of the stack pointer just
    before the access-restricted code gets called.
*  ApplicationContext->Size is zero
*  ApplicationContext->Trusted is true if the OS Application is trusted
    (TrustedApplicationWithProtection)
*
* Where there are access-restricted Functions, there are two more
    fields:
*  ApplicationContext->FunctionID contains the ID of the function
    (INVALID_FUNCTION unless being called for an access-restricted
    Function)
*  ApplicationContext->FunctionParams contains FunctionParams for the
    access-restricted Function call (undefined for INVALID_FUNCTION)
*
* On systems where the hardware does not allow protection regions to be
    set at any address/size combination,
* it may be necessary to adjust the stack to a position that can be
    protected efficiently.
* For example, the protection region may have to be aligned on a
    64-byte address boundary.
* In these cases, RTA-OS may provide the 'AlignUntrustedStacks'
    configuration option.
* When this is set, a further field
    'ApplicationContext->AlignedAddress' becomes available.
* Its initial value will be the same as ApplicationContext->Address.
    However you can
* change its value to signal to the OS that the access-restricted code
    should start at a different location.
* For the earlier example, if ApplicationContext->AlignedAddress
    initially has value 0x1020, you
* might change it to 0x1000 before returning so that the OS will start
    running the code at an
* address that is a multiple of 64. (This example assumes that the
    stack grows towards lower addresses.)
* You will have set the stack protection region to start from 0x1000.
*
* Be aware that on some target devices (Power PC, for example) the EABI
    might specify that a
* back link will be written before the stack pointer on entry.
* You will have to account for this in your calculations.
*
* For a multicore system, ApplicationContext->CoreID contains the ID of
    the calling core.
* This is omitted if the OS is only running on one core.
*/
```

```
  /*  Force AlignedAddress to the the next 64-byte value below Address */
  (uint32)ApplicationContext->AlignedAddress &=
      ((uint32)ApplicationContext->Address % 64U);
  SET_STACK_RANGE(ApplicationContext->AlignedAddress, STACK_ALLOWANCE);

  if (ApplicationContext->Application == App2) {
    /* Set memory protection regions that apply for the overall
       application 'App2' */
    SET_UNTRUSTED_WRITE_RANGE(App2_BASE, App2_SIZE); /* Example */
    if (ApplicationContext->TaskID == App2TaskB) {
      /* Extend or restrict ranges as desired for Task 'App2TaskB' */
    }
    if (ApplicationContext->ISRID == App2ISR1) {
      /* Extend or restrict ranges as desired for ISR 'App2ISR1' */
    }
    if (ApplicationContext->FunctionID == UTF1) {
      /* Extend or restrict ranges as desired for access-restricted
         Function 'tf1' */
    }
  }
  if (ApplicationContext->Application == App3) {
    /* Set memory protection regions that apply for the overall
       application 'App3' */
    SET_UNTRUSTED_WRITE_RANGE(App3_BASE, App3_SIZE); /* Example */
    if (ApplicationContext->TaskID == App3TaskB) {
      /* Extend or restrict ranges as desired for Task 'App3TaskB' */
    }
    if (ApplicationContext->FunctionID == UTF2) {
      /* Extend or restrict ranges as desired for access-restricted
         Function 'tf2' */
    }
    if (ApplicationContext->FunctionID == UTF3) {
      /* Extend or restrict ranges as desired for access-restricted
         Function 'tf3' */
    }
  }
  ...
}
OS_MAIN() {
  ...
  InitializeMemoryProtectionHardware();
  ...
  StartOS(OSDEFAULTAPPMODE);

}
```

Configuration Condition

The callback must be provided when memory protection is selected and there are access-restricted OS Applications.

See Also

Os_Cbk_CheckMemoryAccess
Os_UntrustedContextType
CallTrustedFunction
CallAndProtectFunction

3.23 Os_Cbk_SetTimeLimit

Callback routine to enable the timing interrupt and set a time limit for it.

Syntax

```
FUNC(void,{memclass}) Os_Cbk_SetTimeLimit(
    Os_TimeLimitType Limit
)
```

Return Values

The call returns values of type Os_TimeLimitType.

Description

Os_Cbk_SetTimeLimit() must be implemented if timing protection is configured and a timing interrupt is being used to enforce time limits.

You must use it to ensure that the timing interrupt is enabled and that it will fire after 'Limit' ticks from now, unless cancelled by Os_Cbk_SuspendTimeLimit().

Note that an Os_TimeLimitType tick is expected to have the same duration as a Stopwatch tick.

If called with a value zero, you may call Os_TimingFaultDetected() immediately and skip enabling the interrupt.

In a multicore system, one timing interrupt is needed for each core that has timing limits. Os_Cbk_SetTimeLimit() should only affect the timing interrupt on the core on which it gets called.

Note: memclass is OS_APPL_CODE for AUTOSAR 3.x, OS_OS_CBK_SETTIMELIMIT_CODE for AUTOSAR 4.1 and OS_CALLOUT_CODE otherwise.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
FUNC(void,{memclass}) Os_Cbk_SetTimeLimit(Os_TimeLimitType Limit) {
  Os_TimeLimitType now = <read current counter value>;
  if (Limit == 0) {
    Os_TimingFaultDetected();
  }
  <set current counter compare value>(now + Limit + 1);
}
```

Configuration Condition

The callback must be provided if timing protection is configured and a timing interrupt is being used to enforce time limits.

See Also

Os_TimingFaultDetected
Os_Cbk_SuspendTimeLimit
ProtectionHook
Os_TimeLimitType

3.24 Os_Cbk_Set_<CounterID>

Callback routine to set the next match value for a hardware counter.

Syntax

```
FUNC(void, {memclass}) Os_Cbk_Set_<CounterID>(
    TickType Match
)
```

Parameters

Parameter	Mode	Description
Match	in	TickType The next absolute match value.

Description

The callback must set up the hardware counter to raise the appropriate interrupt when its value reaches the new Match value.

Match is an absolute value at which the next counter action needs to be processed.

This is called from within Os_AdvanceCounter to set the match value appropriate for the next alarm or expiry point.

It can also be called as a result of starting an Alarm or ScheduleTable.

It will only be called on the core that owns the counter, and it does not need to be reentrant.

Care must be taken to cope with the following situations:

- Where intervals are short, it is possible for the hardware count to have already moved past the Match value at the point this get called. If so, it is important to ensure that the interrupt pending bit gets set in software.

- Where an Alarm or ScheduleTable can be started with an interval shorter than one already set, the code must be able to reduce the match value and detect if this means that the hardware count has already passed this point.

The callback does not normally initialize the underlying hardware. That is normally done in initialization code before the OS is started.

Note: memclass is OS_APPL_CODE for AUTOSAR 3.x, OS_OS_CBK_SET_<COUNTERID>_CODE for AUTOSAR 4.1 and OS_CALLOUT_CODE otherwise.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
FUNC(void, {memclass}) Os_Cbk_Set_MyCounter(TickType Match){
  /* Prevent match interrupts for maxallowedvalue+1 ticks*/
  HW_OUTPUT_COMPARE_VALUE = COUNTER - 1u;
  dismiss_interrupt();
  HW_OUTPUT_COMPARE = Match;
  enable_interrupt();
}
```

Configuration Condition

Required for each hardware counter configured.

See Also

Os_AdvanceCounter
SetAbsAlarm
SetRelAlarm
Os_Cbk_Cancel_<CounterID>
Os_Cbk_Now_<CounterID>
Os_Cbk_State_<CounterID>

3.25 Os_Cbk_StackOverrunHook

Callback routine to trap stack-related errors.

Syntax

```
FUNC(void, {memclass}) Os_Cbk_StackOverrunHook(
    Os_StackSizeType Overrun,
    Os_StackOverrunType Reason
)
```

Parameters

Parameter	Mode	Description
Overrun	in	Os_StackSizeType The amount of the overrun.
Reason	in	Os_StackOverrunType The cause of the overrun.

Description

This hook routine is called if configured and any of the following events occur:

(a) a stack allocation budget has been specified for a task/ISR and this budget has been exceeded.

(b) an ECC task failed to start because there was no space on the stack

(c) an ECC task failed to resume from wait because there was no space on the stack

(d) an ECC task failed to wait because it was using too much stack (and its state could not, therefore, be safely preserved)

GetTaskID() and GetISRID() can be used to determine which Task or ISR is involved.

A default version of the hook is present in the kernel that calls ProtectionHook() (if configured, otherwise ShutdownOS()) with the status E_OS_STACKFAULT. You can implement Os_Cbk_StackOverrunHook within your application to override this behavior.

The default version is not provided if OS option "Omit Default Implementations" is TRUE.

Budget overruns are detected at preemption points (or when Os_GetStackUsage() is called) and are only reported the first time that the overrun is first detected in a given run.

A budget overrun does not result in a Task/ISR being forcibly terminated. (Note that it is not permissible to call TerminateTask within the hook.)

ECC related overruns occur when lower priority tasks exceed their stack budget, or when the stack preemption overheads are set to values that are too small.

An ECC overrun does result in the Task being forcibly terminated.

OS_BUDGET can only occur when Stack Monitoring is configured.

OS_ECC_START, OS_ECC_RESUME and OS_ECC_WAIT can occur independently of whether Stack Monitoring is configured.

Note: memclass is OS_APPL_CODE for AUTOSAR 3.x, OS_OS_CBK_STACKOVERRUNHOOK_CODE for AUTOSAR 4.1 and OS_CALLOUT_CODE otherwise.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
FUNC(void, {memclass}) Os_Cbk_StackOverrunHook(Os_StackSizeType Overrun,
    Os_StackOverrunType Reason) {
  switch (Reason) {
    case OS_BUDGET:
      /* The currently running task or ISR has exceeded its stack budget */
      break;
    case OS_ECC_START:
      /* An ECC task has failed to start because there is insufficient
          room on the stack */
      break;
    case OS_ECC_RESUME:
      /* An ECC task has failed to resume from wait because there is
          insufficient room on the stack */
      break;
    case OS_ECC_WAIT:
      /* An ECC task has failed to enter the waiting state because it is
          exceeding its waiting stack budget */
      break;
  }
}
```

Configuration Condition

Optional when Stack Monitoring is configured and budgets are assigned, or when there are ECC tasks.

See Also

Os_GetStackUsage
Os_GetISRMaxStackUsage
Os_GetTaskMaxStackUsage
Os_ResetISRMaxStackUsage
Os_ResetTaskMaxStackUsage
GetISRID
GetTaskID

3.26 Os_Cbk_State_<CounterID>

Callback routine to read the current state of a hardware counter.

Syntax

```
FUNC(void, {memclass}) Os_Cbk_State_<CounterID>(
    Os_CounterStatusRefType State
)
```

Parameters

Parameter	Mode	Description
State	out	Os_CounterStatusRefType The counter state.

Description

This is called by the OS when it needs to decide whether to start a counter. It can also be called by user-code (typically the counter interrupt) to determine the state of the counter.

It must update the State structure to indicate if the counter is running, whether a match interrupt is pending, and (if needed) how long the interval is to the next match.

Note: memclass is OS_APPL_CODE for AUTOSAR 3.x, OS_OS_CBK_STATE_<COUNTERID>_CODE for AUTOSAR 4.1 and OS_CALLOUT_CODE otherwise.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
FUNC(void, {memclass}) Os_Cbk_State_MyCounter(Os_CounterStatusRefType
    State) {
  State->Delay = HW_OUTPUT_COMPARE_VALUE - HW_COUNTER_NOW_VALUE;
  State->Pending = counter_interrupt_pending();
  State->Running = counter_interrupt_enabled();
}
```

Configuration Condition

Required for each hardware counter configured.

See Also

Os_Cbk_Cancel_<CounterID>
Os_Cbk_Now_<CounterID>
Os_Cbk_Set_<CounterID>
Os_AdvanceCounter

3.27 Os_Cbk_SuspendTimeLimit

Callback routine to cancel the timing interrupt and determine how much time was left.

Syntax

```
FUNC(Os_TimeLimitType,{memclass}) Os_Cbk_SuspendTimeLimit(void)
```

Return Values

The call returns values of type Os_TimeLimitType.

Description

Os_Cbk_SuspendTimeLimit() must be implemented if timing protection is configured and a timing interrupt is being used to enforce time limits.

The OS calls it to cancel a previous call to Os_Cbk_SetTimeLimit(). You must ensure that the timing interrupt does not fire when the time limit is reached, and if it is currently pending, that its pending status is cleared.

The return value must be the number of ticks that were remaining to the limit at the point that the call was made.

In a multicore system, one timing interrupt is needed for each core that has timing limits. Os_Cbk_SuspendTimeLimit() should only affect the timing interrupt on the core on which it gets called.

Note: memclass is OS_APPL_CODE for AUTOSAR 3.x, OS_OS_CBK_SUSPENDTIMELIMIT_CODE for AUTOSAR 4.1 and OS_CALLOUT_CODE otherwise.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
FUNC(Os_TimeLimitType,{memclass}) Os_Cbk_SuspendTimeLimit(void) {
  Os_TimeLimitType now = <read current counter value>;
  <disable timing interrupt>;
  <clear timing interrupt pending flag>;
  return now - <read current counter compare value>;
}
```

Configuration Condition

The callback must be provided if timing protection is configured and a timing interrupt is being used to enforce time limits.

See Also

Os_TimingFaultDetected
Os_Cbk_SetTimeLimit
ProtectionHook
Os_TimeLimitType

3.28 Os_Cbk_TaskActivated

Callback routine indicating the activation of a TASK.

Syntax

```
FUNC(void, {memclass}) Os_Cbk_TaskActivated(
    TaskType task
)
```

Description

Os_Cbk_TaskActivated() must be implemented if OS option 'Task Activation Hook' is enabled. It gets called at the point that a TASK is successfully activated.

Activation can be via ActivateTask, ChainTask, Alarms or ScheduleTables.

Note: memclass is OS_APPL_CODE for AUTOSAR 3.x, OS_OS_CBK_TASKACTIVATED_CODE for AUTOSAR 4.1 and OS_CALLOUT_CODE otherwise.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
uint32 task_activations[OS_NUM_TASKS];
FUNC(void, {memclass}) Os_Cbk_TaskActivated(TaskType task) {
  task_activations[OS_TASKTYPE_TO_INDEX(task)] += 1;
}
```

Configuration Condition

The callback must be provided if OS option 'Task Activation Hook' is enabled.

See Also

PreTaskHook
OS_TASKTYPE_TO_INDEX
Os_Cbk_TaskStart
Os_Cbk_TaskEnd
Os_Cbk_ISRStart
Os_Cbk_ISREnd
Os_Cbk_TaskTerminated

3.29 Os_Cbk_TaskEnd

Callback routine indicating the end of a TASK.

Syntax

```
FUNC(void, {memclass}) Os_Cbk_TaskEnd(
    TaskType task
)
```

Description

Os_Cbk_TaskEnd() must be implemented if OS option 'Additional Task Hooks' is enabled. It gets called at the point that a TASK finishes execution.

For an ECC TASK, it is also called when starting to WAIT.

Unlike PostTaskHook, it does not get called when a TASK is pre-empted.

Note: memclass is OS_APPL_CODE for AUTOSAR 3.x, OS_OS_CBK_TASKEND_CODE for AUTOSAR 4.1 and OS_CALLOUT_CODE otherwise.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
Os_StopwatchTickType task_exe_time[OS_NUM_TASKS];
FUNC(void, {memclass}) Os_Cbk_TaskEnd(TaskType task) {
  task_exe_time[OS_TASKTYPE_TO_INDEX(task)] = GetExecutionTime();
}
```

Configuration Condition

The callback must be provided if OS option 'Additional Task Hooks' is enabled.

See Also

PostTaskHook
OS_TASKTYPE_TO_INDEX
Os_Cbk_TaskStart
Os_Cbk_ISRStart
Os_Cbk_ISREnd
Os_Cbk_TaskTerminated

3.30 Os_Cbk_TaskStart

Callback routine indicating the start of a TASK.

Syntax

```
FUNC(void, {memclass}) Os_Cbk_TaskStart(
    TaskType task
)
```

Description

Os_Cbk_TaskStart() must be implemented if OS option 'Additional Task Hooks' is enabled. It gets called at the point that a TASK is about to start execution.

For an ECC TASK, it is also called when resuming from WAITING.

Unlike PreTaskHook, it does not get called when a TASK is pre-empted.

Note: memclass is OS_APPL_CODE for AUTOSAR 3.x, OS_OS_CBK_TASKSTART_CODE for AUTOSAR 4.1 and OS_CALLOUT_CODE otherwise.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
uint32 task_starts[OS_NUM_TASKS];
FUNC(void, {memclass}) Os_Cbk_TaskStart(TaskType task) {
  task_starts[OS_TASKTYPE_TO_INDEX(task)] += 1;
}
```

Configuration Condition

The callback must be provided if OS option 'Additional Task Hooks' is enabled.

See Also

PreTaskHook
OS_TASKTYPE_TO_INDEX
Os_Cbk_TaskActivated
Os_Cbk_TaskEnd
Os_Cbk_ISRStart
Os_Cbk_ISREnd
Os_Cbk_TaskTerminated

3.31 Os_Cbk_TaskTerminated

Callback routine indicating the termination of a TASK.

Syntax

```
FUNC(void, {memclass}) Os_Cbk_TaskTerminated(
    TaskType task
)
```

Description

Os_Cbk_TaskTerminated() must be implemented if OS option 'Task Termination Hook' is enabled. It gets called at the point that a TASK is successfully terminated.

Termination can be via TerminateTask, forced termination or returning from the TASK body. It does not get called when an ECC task waits.

Note: memclass is OS_APPL_CODE for AUTOSAR 3.x, OS_OS_CBK_TASKTERMINATED_CODE for AUTOSAR 4.1 and OS_CALLOUT_CODE otherwise.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
uint32 task_terminations[OS_NUM_TASKS];
FUNC(void, {memclass}) Os_Cbk_TaskTerminated(TaskType task) {
  task_terminations[OS_TASKTYPE_TO_INDEX(task)] += 1;
}
```

Configuration Condition

The callback must be provided if OS option 'Task Termination Hook' is enabled.

See Also

PreTaskHook
OS_TASKTYPE_TO_INDEX
Os_Cbk_TaskStart
Os_Cbk_TaskEnd
Os_Cbk_ISRStart
Os_Cbk_ISREnd
Os_Cbk_TaskActivated

3.32 Os_Cbk_Terminated_<ISRName>

Callback routine indicating that the Category 2 ISR <ISRName> has been forcibly terminated.

Syntax

```
FUNC(void, {memclass}) Os_Cbk_Terminated_<ISRName>(void)
```

Description

This callback is provided so that the application can take appropriate action when a Category 2 ISR is forcibly terminated by the OS.

An ISR can be terminated in the following situations:

1) You call TerminateApplication() while the ISR is running (including when it has been interrupted by a higher priority interrupt).

2) There is a timing or memory protection violation while the ISR is running and you return PRO_TERMINATETASKISR from ProtectionHook().

3) There is a timing or memory protection violation while a preempting ISR is running and you return PRO_TERMINATEAPPL or PRO_TERMINATEAPPL_RESTART from ProtectionHook().

On target processors where you have to clear some 'interrupt pending' status for the interrupt source, you must use this callback to clear the status. If you fail to do this, the interrupt will be re-entered when the processor priority is subsequently lowered.

Note: memclass is OS_APPL_CODE for AUTOSAR 3.x, OS_OS_CBK_TERMINATED_<ISRNAME>_CODE for AUTOSAR 4.1 and OS_CALLOUT_CODE otherwise.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
FUNC(void, {memclass}) Os_Cbk_Terminated_App2Isr1(void) {
    clear_interrupt_source(_App2Isr1_);
}
```

Configuration Condition

Required for each Category 2 ISR if forced termination of interrupts is supported.

See Also

ProtectionHook
TerminateApplication

3.33 Os_Cbk_TimeOverrunHook

Callback routine to trap errors detected during time monitoring.

Syntax

```
FUNC(void, {memclass}) Os_Cbk_TimeOverrunHook(
    Os_StopwatchTickType Overrun
)
```

Parameters

Parameter	Mode	Description
Overrun	in	Os_StopwatchTickType The amount of the overrun in stopwatch ticks.

Description

This hook routine is called if an execution budget has been specified for a task/ISR and the execution time has exceeded this budget.

Budget overruns are detected at preemption points or when the Task/ISR terminated. This hook is called once, when the overrun is first detected.

A budget overrun does not result in a Task/ISR being forcibly terminated. (Note that it is not permissible to call TerminateTask within the hook.)

If tracing via RTA-TRACE is active, the error code E_OS_SYS_OVERRUN will be logged in the trace buffer.

Note: memclass is OS_APPL_CODE for AUTOSAR 3.x, OS_OS_CBK_TIMEOVERRUNHOOK_CODE for AUTOSAR 4.1 and OS_CALLOUT_CODE otherwise.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
FUNC(void, {memclass}) Os_Cbk_TimeOverrunHook(Os_StopwatchTickType
    Overrun) {
}
```

Configuration Condition

Required when Time Monitoring is configured and budgets are assigned.

See Also

Os_GetExecutionTime
Os_GetISRMaxExecutionTime
Os_GetTaskMaxExecutionTime
Os_ResetISRMaxExecutionTime
Os_ResetTaskMaxExecutionTime
GetISRID
GetTaskID

3.34 PostTaskHook

Callback routine called when context switching from a task.

Syntax

```
FUNC(void, {memclass}) PostTaskHook(void)
```

Description

This hook routine is called by the operating system immediately before it leaves the running state.

This means it is safe to evaluate the TaskID.

The PostTaskHook is not called if a task is leaving the running state because the ShutdownOS() call has been made.

A sample PostTaskHook can be generated automatically by rtaosgen. See the RTA-OS User Guide for further details.

Note: memclass is OS_APPL_CODE for AUTOSAR 3.x, OS_POSTTASKHOOK_CODE for AUTOSAR 4.1 and OS_CALLOUT_CODE otherwise.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

Example

```
FUNC(void, {memclass}) PostTaskHook(void){
  TaskType LeavingTask;
  GetTaskID(&LeavingTask);
  if (LeavingTask == TaskA) {
    /* Do action for leaving A */
  } else if (LeavingTask == TaskB) {
    /* Do action for leaving B */
  }
  ...
}
```

Configuration Condition

Required when the PostTaskHook is configured.

See Also

PreTaskHook
Os_Cbk_TaskEnd
Os_Cbk_ISREnd
Os_Cbk_TaskTerminated

3.35 PreTaskHook

Callback routine called when context switching into a task.

Syntax

```
FUNC(void, {memclass}) PreTaskHook(void)
```

Description

This hook routine is called by the operating system immediately after it enters the running state but before the task itself begins to execute.

This means it is safe to evaluate the TaskID.

A sample PreTaskHook can be generated automatically by rtaosgen. See the RTA-OS User Guide for further details.

Note: memclass is OS_APPL_CODE for AUTOSAR 3.x, OS_PRETASKHOOK_CODE for AUTOSAR 4.1 and OS_CALLOUT_CODE otherwise.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

Example

```
FUNC(void, {memclass}) PreTaskHook(void){
  TaskType EnteringTask;
  GetTaskID(&EnteringTask);
  if (EnteringTask == TaskA) {
    /* Do action for entering A */
  } else if (EnteringTask == TaskB) {
    /* Do action for entering B */
  }
  ...
}
```

Configuration Condition

Required when the PreTaskHook is configured.

See Also

PostTaskHook
Os_Cbk_TaskStart
Os_Cbk_TaskActivated
Os_Cbk_ISRStart

3.36 ProtectionHook

Callback routine used for trapping protection faults.

Syntax

```
FUNC(ProtectionReturnType, {memclass}) ProtectionHook(
    StatusType FatalError
)
```

Parameters

Parameter	Mode	Description
FatalError	in	StatusType The type of the error that has occurred.

Return Values

The call returns values of type ProtectionReturnType.

Description

This is called when a timing or memory protection fault occurs. The type of fault is passed into ProtectionHook().

The return type determines what action the OS takes after the callback:

PRO_IGNORE: The arrival is ignored and processing continues. Only allowed for E_OS_PROTECTION_ARRIVAL.

PRO_TERMINATETASKISR: The task, ISR or protected function that caused the fault is forcibly terminated. Only valid when memory or timing protection are configured.

PRO_TERMINATEAPPL: The OS Application that contains the faulting task or ISR is forcibly terminated. Only valid when memory or timing protection are configured.

PRO_TERMINATEAPPL_RESTART: The OS Application that contains the faulting task or ISR is forcibly terminated and then restarted. Only valid when memory or timing protection are configured.

PRO_SHUTDOWN: ShutdownOS() is called.

If any Category 2 ISR is terminated, the OS will use the callback Os_Cbk_Terminated_<ISRName>() to allow you to ensure that the interrupt source is dealt with appropriately.

ProtectionHook runs at OS level and will not be preempted by Tasks or Category 2 ISRs.

A sample ProtectionHook can be generated automatically by rtaosgen. See the RTA-OS User Guide for further details.

Note: memclass is OS_APPL_CODE for AUTOSAR 3.x, OS_PROTECTIONHOOK_CODE for AUTOSAR 4.1 and OS_CALLOUT_CODE otherwise.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

Example

```
FUNC(ProtectionReturnType, {memclass}) ProtectionHook(StatusType
    FatalError) {
  switch (FatalError) {
    case E_OS_PROTECTION_MEMORY:
         /* A memory protection error has been detected */
         break;
    case E_OS_PROTECTION_TIME:
         /* Task, Category 2 ISR or time-limited function exceeds its
            execution time */
         break;
    case E_OS_PROTECTION_ARRIVAL:
         /* Task/Category 2 arrives before its timeframe has expired */
         return PRO_IGNORE;  /* This is the only case where PRO_IGNORE is
            allowed */
    case E_OS_PROTECTION_LOCKED:
         /* Task/Category 2 ISR blocks for too long */
         break;
    case E_OS_PROTECTION_EXCEPTION:
         /* Trap occurred */
         break;
  }
  return PRO_SHUTDOWN;
}
```

Configuration Condition

Required when the ProtectionHook is configured. Should be configured when timing or memory protection are required.

See Also

Os_Cbk_Terminated_<ISRName>

3.37 ShutdownHook

Callback routine called during OS shutdown.

Syntax

```
FUNC(void, {memclass}) ShutdownHook(
    StatusType Error
)
```

Parameters

Parameter	Mode	Description
Error	in	StatusType The reason for the shutdown.

Description

If a ShutdownHook() is configured, this hook routine is called by the operating system when the OS API call ShutdownOS() has been called.

This routine is called during the operating system shutdown. The OS can be restarted from the ShutdownHook() using Os_Restart()

Note: memclass is OS_APPL_CODE for AUTOSAR 3.x, OS_SHUTDOWNHOOK_CODE for AUTOSAR 4.1 and OS_CALLOUT_CODE otherwise.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

Example

```
FUNC(void, {memclass}) ShutdownHook(StatusType Error){
  if (Error == E_OS_STACKFAULT) {
    /* Attempt recovery by restart */
    Os_Restart();
    /* Never reach here... */
  } else if (Error == E_OK) {
    /* Normal shutdown procedure */
  }
  ...
}
```

Configuration Condition

Required when the ShutdownHook is configured.

See Also

Os_Restart
StartupHook
ShutdownOS

3.38 StartupHook

Callback routine called during OS startup.

Syntax

```
FUNC(void, {memclass}) StartupHook(void)
```

Description

If a StartupHook() is configured, this hook routine is called by the OS at the end of the OS initialization, but before the scheduler is running.

The application can start tasks, initialize device drivers and so on within StartupHook().

StartupHook() runs with Category2 ISRs disabled so it is safe to enable interrupt sources from the hook.

Note: memclass is OS_APPL_CODE for AUTOSAR 3.x, OS_STARTUPHOOK_CODE for AUTOSAR 4.1 and OS_CALLOUT_CODE otherwise.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

Example

```
FUNC(void, {memclass}) StartupHook(void){
  /* Enable timer interrupt */
  CHANNEL0_CONTROL_REG |= ONE_MILLISECOND_TIMER;
  CHANNEL0_CONTROL_REG |= ENABLE;
}
```

Configuration Condition

Required when the StartupHook is configured.

See Also

ShutdownHook

4 RTA-OS Types

4.1 AccessType

An integral value that holds information about how a specific memory region can be accessed.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

Constants

```
OS_ACCESS_READ
OS_ACCESS_WRITE
OS_ACCESS_EXECUTE
OS_ACCESS_STACK
```

Example

```
FUNC(AccessType,{memclass}) Os_Cbk_CheckMemoryAccess(ApplicationType
    Application, TaskType TaskID, ISRType ISRID, MemoryStartAddressType
    Address, MemorySizeType Size) {
  AccessType Access = OS_ACCESS_EXECUTE;
  /* Address range is read/write if it is in RAM */
  if ((Address >= RAM_BASE) && (Address + Size < RAM_BASE + RAM_SIZE) ) {
      Access |= (OS_ACCESS_WRITE | OS_ACCESS_READ);
  }
  ...
  return Access;
}
```

4.2 AlarmBaseRefType

A pointer to an object of AlarmBaseType.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

Example

```
AlarmBaseType AlarmBase;
AlarmBaseRefType AlarmBaseRef = &AlarmBase;
```

4.3 AlarmBaseType

Defines the configuration of a counter. The type is a C struct that contains the fields maxallowedvalue, ticksperbase and mincycle.

maxallowedvalue is the maximum allowed count value in ticks.

ticksperbase is the number of ticks required to reach a counter-specific (significant) unit.

mincycle is the smallest allowed value for the cycle-parameter of SetRelAlarm/SetAbsAlarm.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

Values

All values are of type TickType.

Example

```
TickType         max,min,ticks;
AlarmBaseType    SomeAlarmBase;
AlarmBaseRefType PointerToSomeAlarmBase = &SomeAlarmBase;
max   = SomeAlarmBase.maxallowedvalue;
ticks = SomeAlarmBase.ticksperbase;
min   = SomeAlarmBase.mincycle;
```

4.4 AlarmType

The type of an Alarm.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

Example

```
AlarmType SomeAlarm;
```

4.5 AppModeType

The type of an application mode.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

Values

Symbolic names of the application modes declared at configuration time. (Must include OSDEFAULTAPPMODE)

Example

```
AppModeType SomeAppMode;
```

4.6 ApplicationStateRefType

A pointer to an object of ApplicationStateType.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✓	✓	✗

Example

```
ApplicationStateRefType SomeState = &state_variable;
GetApplicationState(MyApp, SomeState);
```

4.7 ApplicationStateType

Enumerated type defining the state of an OS-Application.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✓	✓	✗

Values

APPLICATION_ACCESSIBLE
APPLICATION_RESTARTING
APPLICATION_TERMINATED

Example

```
GetApplicationState(MyApp, &MyAppState);
if (MyAppState == APPLICATION_RESTARTING) {...}
```

4.8 ApplicationType

The type of an OS-Application.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

Values

Symbolic names of the OS-Applications declared at configuration time.

Constants

`INVALID_OSAPPLICATION`

Example

```
ApplicationType SomeOSApplication;
```

4.9 AreaIdType

The ID of a configured PeripheralArea.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

Values

Symbolic names of the PeripheralAreas declared at configuration time.

Example

```
AreaIdType area1;
```

4.10 CoreIdType

Scalar representing the ID of a processor core.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✓	✗

Example

```
CoreIdType core = GetCoreID();
```

4.11 CounterType

The type of a Counter.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

Example

```
CounterType SomeCounter;
```

4.12 EventMaskRefType

A pointer to an object of EventMaskType.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

Example

```
EventMaskRefType SomeEventRef;
```

4.13 EventMaskType

The type of an event.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

Values

Symbolic names of the EventMasks declared at configuration time.

Example

```
EventMaskType SomeEvent;
```

4.14 ISRRefType

A pointer to an object of ISRType.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

Example

```
ISRType SomeISR;
ISRRefType PointerToSomeISR = &SomeISR;
```

4.15 ISRType

The type of a ISR.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

Values

The symbolic names of ISRs declared at configuration time.

Constants

INVALID_ISR

Example

```
ISRType SomeISR;
```

4.16 MemorySizeType

This data type holds the size (in bytes) of a memory region.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

Example

```
MemorySizeType DatumSize = sizeof(datum);
CheckISRMemoryAccess(SomeISR, &datum, DatumSize);
```

4.17 MemoryStartAddressType

This data type is a pointer which is able to point to any location in the address space.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

Example

```
MemoryStartAddressType StartAddress  = &datum;
CheckISRMemoryAccess(SomeISR, StartAddress, sizeof(datum));
```

4.18 OSServiceIdType

The type of a OS API call. Used only in the ErrorHook(). The values take the form OSServiceId__APICallName_ where _APICallName_ represents the name of an API call (without any leading Os_).

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

Example

```
FUNC(void, {memclass}) ErrorHook(StatusType Error){
  OSServiceIdType ServiceExecuting;
  ServiceExecuting = OSError_GetServiceID();
  switch ( ServiceExecuting ) {
    case OSServiceId_None: /* Used for errors detected when an ISR exits
        with resources or interrupts locked */
      ...
      break;
    case OSServiceId_ActivateTask:
      ...
      break;
    case OSServiceId_CancelAlarm:
      ...
      break;
    case OSServiceId_ChainTask:
      ...
      break;
    ...
    default:
      ...
  }
}
```

4.19 ObjectAccessType

Enumerated type defining whether an OS-Application has access to an object.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

Values

ACCESS
NO_ACCESS

Example

```
if (ACCESS == CheckObjectAccess(MyOSApp, OBJECT_TASK, MyTask) {...}
```

4.20 ObjectTypeType

Enumerated type defining the type of an OS object.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

Values

OBJECT_TASK
OBJECT_ISR
OBJECT_ALARM
OBJECT_RESOURCE
OBJECT_COUNTER
OBJECT_SCHEDULETABLE

Example

```
if (ACCESS == CheckObjectAccess(MyOSApp, OBJECT_TASK, MyTask) {...}
```

4.21 Os_AnyType

A reference to an OS object.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

Example

```
CheckObjectOwnership(OBJECT_TASK, Task1);
```

4.22 Os_CounterStatusRefType

A pointer to an object of Os_CounterStatusType.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
Os_CounterStatusType MyHwCounterStatus;
do {
  Os_AdvanceCounter_MyHWCounter();
  Os_Cbk_State_MyHWCounter(&MyHwCounterStatus);
} while (MyHwCounterStatus.Running && MyHwCounterStatus.Pending);
```

4.23 Os_CounterStatusType

Defines the status of a hardware counter. The type is a C struct that contains the fields Running, Pending and Delay.

Running is TRUE only if the counter driver is running.

Pending is TRUE only if an expiry of an associated alarm and/or schedule table expiry point is pending.

Delay is a value that defines the number of ticks - relative to the last expiry - at which the next expiry is due. An Os_CounterStatusType.Delay value of zero represents max-allowedvalue+1 (the modulus) of the counter.

The Delay field is only valid when Running and Pending are TRUE.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
Os_CounterStatusType CounterStatus;
```

4.24 Os_LockerRefType

A pointer to a TASK or Category 2 ISR.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
TASK(MyTask){
  ...
  Os_SpinlockInfo Info;
  GetSpinlockInfo(Spinlock1, &Info);
  if ((TaskType)Info.CurrentLocker == MyTask) {
    ...
  }
}
```

4.25 Os_SpinlockInfo

Contains the spinlock statistics information that can be returned from the GetSpinlock-Info API. (The structure may contain additional values used for calculation. These are not intended to be part of the public API.)

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Values

uint32 LockAttempts[OS_NUM_CORES]
uint32 LockSucceeds[OS_NUM_CORES]
uint32 LockFails[OS_NUM_CORES]
Os_StopwatchTickType MaxLockTime[OS_NUM_CORES]
Os_LockerRefType MaxLockTimeLocker[OS_NUM_CORES]
Os_StopwatchTickType MaxSpinTime[OS_NUM_CORES]
Os_LockerRefType MaxSpinTimeLocker[OS_NUM_CORES]
CoreIdType CurrentLockingCore
Os_StopwatchTickType CurrentLockTime
Os_LockerRefType CurrentLocker

Example

```
Os_SpinlockInfo Info;
```

4.26 Os_SpinlockInfoRefType

A pointer to an Os_SpinlockInfo.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
TASK(MyTask){
  ...
  Os_SpinlockInfo Info;
  GetSpinlockInfo(Spinlock1, &Info);
  if ((TaskType)Info.CurrentLocker == MyTask) {
    ...
  }
}
```

4.27 Os_StackOverrunType

Enumerated type defining the reason for a stack overrun.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Values

OS_BUDGET
OS_ECC_START
OS_ECC_RESUME
OS_ECC_WAIT

Example

```
FUNC(void, {memclass}) Os_Cbk_StackOverrunHook(Os_StackSizeType Overrun,
    Os_StackOverrunType Reason) {
  switch (Reason) {
    case OS_BUDGET:
      /* The currently running task or ISR has exceeded its stack budget */
      break;
    case OS_ECC_START:
      /* An ECC task has failed to start because there is insufficient
          room on the stack */
      break;
    case OS_ECC_RESUME:
      /* An ECC task has failed to resume from wait because there is
          insufficient room on the stack */
      break;
    case OS_ECC_WAIT:
      /* An ECC task has failed to enter the waiting state because it is
          exceeding its wait-stack budget */
      break;
  }
}
```

4.28 Os_StackSizeType

An unsigned value representing an amount of stack in bytes.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
Os_StackSizeType stack_size;
stack_size = Os_GetStackSize(start_position, end_position);
```

4.29 Os_StackValueType

An unsigned value representing the position of the stack pointer (ESP).

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
Os_StackValueType start_position;
start_position = Os_GetStackValue();
```

4.30 Os_StatusRefType

A pointer to a StatusType.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
StatusType status;
Os_StatusRefType status_ref = &status;
...
StartCore(OS_CORE_ID_0, status_ref);
```

4.31 Os_StopwatchTickRefType

A pointer to an object of Os_StopwatchTickType.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
Os_StopwatchTickRefType SomeTick = &tickvalue;
Os_GetTaskActivationTime(MyTask, SomeTick);
```

4.32 Os_StopwatchTickType

Scalar representing ticks of a stopwatch (time monitoring or protection) counter.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
Os_StopwatchTickType Duration;
GetExecutionTime(&Duration);
```

4.33 Os_TasksetType

This type may only be used when delayed task execution is configured. It is used to contain a set of Tasks that reside on the same core.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Constants

OS_NO_TASKS

Example

```
Os_TasksetType t1 = TASK_MASK(Task1);
```

4.34 Os_TimeLimitType

Scalar representing an execution time limit, used with timing protection. The duration of one Os_TimeLimitType is the same as one Os_StopwatchTickType

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
Os_TimeLimitType limit = 100;
CallAndProtectFunction(Func3, &data, limit);
```

4.35 Os_UntrustedContextRefType

A pointer to an object of Os_UntrustedContextType.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
FUNC(void, {memclass}) Os_Cbk_SetMemoryAccess(Os_UntrustedContextRefType
    ApplicationContext) {}
```

4.36 Os_UntrustedContextType

Defines the context of the access-restricted code that is about to be executed (in an untrusted or trusted-with-protection OS Application). It is only used by the Os_Cbk_SetMemoryAccess() callback when memory protection features are configured.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Values

ApplicationType Application
TaskType TaskID
ISRType ISRID
MemoryStartAddressType Address
MemorySizeType Size

Example

```
FUNC(void, {memclass}) Os_Cbk_SetMemoryAccess(Os_UntrustedContextRefType
    ApplicationContext) {}
```

4.37 PhysicalTimeType

Scalar representing a units of physical (wall clock) time.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

Example

```
PhysicalTimeType Milliseconds = OS_TICKS2MS_MyCounter(42);
```

4.38 ProtectionReturnType

Enumerated type defining the action taken following a protection fault.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

Values

PRO_IGNORE
PRO_TERMINATETASKISR
PRO_TERMINATEAPPL
PRO_TERMINATEAPPL_RESTART
PRO_SHUTDOWN

Example

```
FUNC(ProtectionReturnType, {memclass}) ProtectionHook(StatusType
    FatalError) {
  switch (FatalError) {
    case E_OS_PROTECTION_MEMORY:
         /* A memory protection error has been detected */
         break;
    case E_OS_PROTECTION_TIME:
         /* Task, Category 2 ISR or time-limited function exceeds its
            execution time */
         break;
    case E_OS_PROTECTION_ARRIVAL:
         /* Task/Category 2 arrives before its timeframe has expired */
         return PRO_IGNORE;  /* This is the only case where PRO_IGNORE is
            allowed */
    case E_OS_PROTECTION_LOCKED:
         /* Task/Category 2 ISR blocks for too long */
         break;
    case E_OS_PROTECTION_EXCEPTION:
```

```
            /* Trap occurred */
            break;
    }
    return PRO_SHUTDOWN;
}
```

4.39 ResourceType

The type of a Resource.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

Values

RES_SCHEDULER plus the symbolic names of Resources declared at configuration time.

Constants

RES_SCHEDULER

Example

```
ResourceType SomeResource;
```

4.40 RestartType

Enumerated type defining the action to be taken in TerminateApplication().

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

Values

RESTART
NO_RESTART

4.41 ScheduleTableRefType

A pointer to an object of ScheduleTableType.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

Example

```
ScheduleTableType SomeScheduleTable;
ScheduleTableRefType PointerToSomeScheduleTable = &SomeScheduleTable;
```

4.42 ScheduleTableStatusRefType

A pointer to an object of ScheduleTableStatusType.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

Example

```
ScheduleTableStatusType SomeScheduleTableStatus;
GetScheduleTableStatus(&SomeScheduleTableStatus);
```

4.43 ScheduleTableStatusType

Enumerated type defining the runtime state of a schedule table.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

Values

SCHEDULETABLE_STOPPED
SCHEDULETABLE_NEXT
SCHEDULETABLE_WAITING
SCHEDULETABLE_RUNNING
SCHEDULETABLE_RUNNING_AND_SYNCHRONOUS

Example

```
ScheduleTableStatusType SomeScheduleTableStatus;
```

4.44 ScheduleTableType

The type of a ScheduleTable.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

Example

```
ScheduleTableType SomeScheduleTable;
```

4.45 SignedTickType

Signed Scalar representing a ticks of a counter.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
SignedTickType drift = -2;
Os_SyncScheduleTableRel(MyTable, drift);
```

4.46 SpinlockIdType

The type of a Spinlock.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✓	✗

Values

The symbolic names of spinlocks declared at configuration time.

Constants

```
INVALID_SPINLOCK
```

Example

```
TASK(MyTask){
  ...
  GetSpinlock(Spinlock1);
  ...
  ReleaseSpinlock(Spinlock1);
}
```

4.47 StatusType

Enumeration type defining the status of an API call.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

Values

E_OK
E_OS_ACCESS
E_OS_CALLEVEL
E_OS_ID
E_OS_LIMIT
E_OS_NOFUNC
E_OS_RESOURCE
E_OS_STATE
E_OS_VALUE
E_OS_SERVICEID
E_OS_ILLEGAL_ADDRESS
E_OS_MISSINGEND
E_OS_DISABLEDINT
E_OS_STACKFAULT
E_OS_PROTECTION_MEMORY
E_OS_PROTECTION_TIME
E_OS_PROTECTION_ARRIVAL
E_OS_PROTECTION_LOCKED
E_OS_PROTECTION_EXCEPTION
E_OS_CORE
E_OS_SPINLOCK
E_OS_INTERFERENCE_DEADLOCK
E_OS_NESTING_DEADLOCK
E_OS_PARAM_POINTER
E_OS_SYS_NO_RESTART
E_OS_SYS_RESTART
E_OS_SYS_OVERRUN
E_OS_SYS_XCORE_QFULL
E_OS_SYS_ERROR_LIMIT

Example

```
StatusType ErrorCode;
ErrorCode = ActivateTask(MyTask);
```

4.48 Std_ReturnType

AUTOSAR's standard API service return type. This is only used by AUTOSAR OS for the IOC API. The type is an 8-bit unsigned integer whose top 6 bits may encode module-specific error codes.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

Values

E_OK=0
E_NOT_OK
IOC_E_OK
IOC_E_NOK
IOC_E_LIMIT
IOC_E_LOST_DATA
IOC_E_NO_DATA

Example

```
Std_ReturnType ErrorCode;
ErrorCode = IocSend_OverTheRainbow(Dorothy);
if (ErrorCode == IOC_E_OK)  {
  /* call succeeded */
} else {
  /* call failed */
}
```

4.49 Std_VersionInfoType

A C struct whose fields contained AUTOSAR version information for a module. (Defined in Std_Types.h)

The fields are:

vendorID

moduleID

instanceID (AUTOSAR R3.x only)

sw_major_version

sw_minor_version

sw_patch_version

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

Values

The field vendorID for ETAS is 11
The field moduleID for AUTOSAR OS is 1

Example

```
Std_VersionInfoType Version;
GetVersionInfo(&Version);
if (Version.vendorID == 11) {
  /* Make ETAS-specific API call */
  AdvanceCounter(HardwareCounter);
}
```

4.50 TaskRefType

A pointer to an object of TaskType.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

Example

```
TaskType SomeTask;
TaskRefType TaskRef = &SomeTask;
```

4.51 TaskStateRefType

A pointer to an object of TaskStateType.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

Example

```
TaskStateType TaskState;
TaskStateRefType TaskStateRef = &TaskState;
```

4.52 TaskStateType

Enumerated type defining the current state of a task.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

Values

SUSPENDED
READY
WAITING
RUNNING

Example

```
TaskStateType TaskState;
GetTaskState(&TaskState);
```

4.53 TaskType

The type of a task.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

Values

The symbolic names of tasks declared at configuration time.

Constants

```
INVALID_TASK
```

Example

```
TaskType SomeTask;
```

4.54 TickRefType

A pointer to an object of TickType.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

Example

```
TickRefType SomeTick = &tickvalue;
GetCounterValue(MyCounter,SomeTick);
```

4.55 TickType

Scalar representing ticks of a counter.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

Example

```
TickType StartTime = 42;
TickType NoRepeat = 0;
SetAbsAlarm(MyAlarm,StartTime,NoRepeat);
```

4.56 TrustedFunctionIndexType

The index value of a Trusted function.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

Values

Symbolic names of the Trusted functions declared at configuration time.

Constants

`INVALID_FUNCTION`

Example

```
CallTrustedFunction(Func3, &data);
```

4.57 TrustedFunctionParameterRefType

A reference to the parameters for a Trusted function.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

Example

```
CallTrustedFunction(Func3, &data);
```

4.58 TryToGetSpinlockType

Enumerated type defining the result of a call to TryToGetSpinlock.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✓	✗

Values

TRYTOGETSPINLOCK_SUCCESS
TRYTOGETSPINLOCK_NOSUCCESS

Example

```
TryToGetSpinlock(MyLock, &retval);
if (retval == TRYTOGETSPINLOCK_SUCCESS) {...}
```

4.59 boolean

Addressable 8 bits only for use with TRUE/FALSE. (Defined in Platform_Types.h)

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

Values

0=FALSE
1=TRUE

Example

```
if (Condition == TRUE) {
  x = y;
}
```

4.60 float32

Single precision floating point number. (Defined in Platform_Types.h)

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

Example

```
float32 x;
```

4.61 float64

Double precision floating point number. (Defined in Platform_Types.h)

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

Example

```
float64 x;
```

4.62 sint16

Signed 16-bit integer. (Defined in Platform_Types.h)

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

Values

-32768..32767

Example

```
sint16 x;
```

4.63 sint16_least

Signed integer at least 16-bits wide. (Defined in Platform_Types.h)

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

Values

At least -32768..32767

Example

```
sint16_least x;
```

4.64 sint32

Signed 32-bit integer. (Defined in Platform_Types.h)

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

Values

-2147483648..2147483647

Example

```
sint32 x;
```

4.65 sint32_least

Signed integer at least 32-bits wide. (Defined in Platform_Types.h)

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

Values

At least -2147483648..2147483647

Example

```
sint32_least x;
```

4.66 sint8

Signed 8-bit integer. (Defined in Platform_Types.h)

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

Values

-128..127

Example

```
sint8 x;
```

4.67 sint8_least

Signed integer at least 8-bits wide. (Defined in Platform_Types.h)

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

Values

At least -128..127

Example

```
sint8_least x;
```

4.68 uint16

Unsigned 16-bit integer. (Defined in Platform_Types.h)

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

Values

0..65535

Example

```
uint16 x;
```

4.69 uint16_least

Unsigned integer at least 16-bits wide. (Defined in Platform_Types.h)

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

Values

At least 0..65535

Example

```
uint16_least x;
```

4.70 uint32

Unsigned 32-bit integer. (Defined in Platform_Types.h)

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

Values

0..4294967295

Example

```
uint32 x;
```

4.71 uint32_least

Unsigned integer at least 32-bits wide. (Defined in Platform_Types.h)

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

Values

At least 0..4294967295

Example

```
uint32_least x;
```

4.72 uint8

Unsigned 8-bit integer. (Defined in Platform_Types.h)

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

Values

0..255

Example

```
uint8 x;
```

4.73 uint8_least

Unsigned integer at least 8-bits wide. (Defined in Platform_Types.h)

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

Values

At least 0..255

Example

```
uint8_least x;
```

5 RTA-OS Macros

5.1 ALARMCALLBACK

Declares an alarm callback. The only OS API calls that can be made in an alarm callback are SuspendAllInterrupts() and ResumeAllInterrupts().

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

Example

```
ALARMCALLBACK(MyCallback){...}
```

5.2 CAT1_ISR

Macro that should be used to create a Category 1 ISR entry function. This macro exists to help make your code portable between targets.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
CAT1_ISR(MyISR) {...}
```

5.3 DONOTCARE

In a multicore system, all but one call to StartOS can pass DONOTCARE as the AppModeType.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✓	✗

5.4 DeclareAlarm

This is used to declare an alarm and works similarly to external declaration of variables in C. You will not normally need to use this because RTA-OS automatically declares all Alarms in your configuration.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

Example

```
DeclareAlarm(MyAlarm);
```

5.5 DeclareCounter

This is used to declare a Counter and works similarly to external declaration of variables in C. You will not normally need to use this because RTA-OS automatically declares all Counters in your configuration.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
DeclareCounter(MyCounter);
```

5.6 DeclareEvent

This is used to declare an Event and works similarly to external declaration of variables in C. You will not normally need to use this because RTA-OS automatically declares all Events in your configuration.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

Example

```
DeclareEvent(MyEvent);
```

5.7 DeclareISR

This is used to declare an ISR and works similarly to external declaration of variables in C. You will not normally need to use this because RTA-OS automatically declares all ISRs in your configuration.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
DeclareISR(MyISR);
```

5.8 DeclareResource

This is used to declare a Resource and works similarly to external declaration of variables in C. You will not normally need to use this because RTA-OS automatically declares all Resources in your configuration.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

Example

```
DeclareResource(MyResource);
```

5.9 DeclareScheduleTable

This is used to declare a ScheduleTable and works similarly to external declaration of variables in C. You will not normally need to use this because RTA-OS automatically declares all ScheduleTables in your configuration.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
DeclareScheduleTable(MyScheduleTable);
```

5.10 DeclareTask

This is used to declare a Task and works similarly to external declaration of variables in C. You will not normally need to use this because RTA-OS automatically declares all Tasks in your configuration.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

Example

```
DeclareTask(MyTask);
```

5.11 INVALID_SPINLOCK

Represents 'no spinlock'.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✓	✗

5.12 ISR

Macro that must be used to create a Category 2 ISR entry function.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

Example

```
ISR(MyISR) {...}
```

5.13 OSCYCLEDURATION

Duration of an instruction cycle in nanoseconds.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
time_in_ns = CycleMeasurement * OSCYCLEDURATION;
```

5.14 OSCYCLESPERSECOND

The number of instruction cycles per second.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
time_in_secs = CycleMeasurement / OSCYCLESPERSECOND;
```

5.15 OSErrorGetServiceId

Returns the identifier of the service that generated an error.

Values are of OSServiceIdType.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

Example

```
OSServiceIdType WhatServiceFailed = OSErrorGetServiceId();
```

5.16 OSMAXALLOWEDVALUE

Constant definition of the maximum possible value of the Counter called System-Counter in ticks.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

Example

```
SetAbsAlarm(MyAlarm,OSMAXALLOWEDVALUE,0)
```

5.17 OSMAXALLOWEDVALUE_<CounterID>

Constant definition of the maximum possible value of the Counter called CounterID in ticks.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

Example

```
SetAbsAlarm(MyAlarm,OSMAXALLOWEDVALUE_SomeCounter,0)
```

5.18 OSMEMORY_IS_EXECUTABLE

Check whether access rights indicate that memory is executable.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

Example

```
rights = CheckTaskMemoryAccess(MyTask, &datum, sizeof(datum));
if (OSMEMORY_IS_EXECUTABLE(rights)) {...}
```

5.19 OSMEMORY_IS_READABLE

Check whether access rights indicate that memory is readable.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

Example

```
rights = CheckTaskMemoryAccess(MyTask, &datum, sizeof(datum));
if (OSMEMORY_IS_READABLE(rights)) {...}
```

5.20 OSMEMORY_IS_STACKSPACE

Check whether access rights indicate that memory is stack space.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

Example

```
rights = CheckTaskMemoryAccess(MyTask, &datum, sizeof(datum));
if (OSMEMORY_IS_STACKSPACE(rights)) {...}
```

5.21 OSMEMORY_IS_WRITEABLE

Check whether access rights indicate that memory is writeable.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

Example

```
rights = CheckTaskMemoryAccess(MyTask, &datum, sizeof(datum));
if (OSMEMORY_IS_WRITEABLE(rights)) {...}
```

5.22 OSMINCYCLE

Constant definition of the minimum number of ticks for a cyclic alarm on the Counter called SystemCounter.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

Example

```
if (ComputedValue < OSMINCYCLE) {
    SetAbsAlarm(MyAlarm,42,OSMINCYCLE);
} else {
    SetAbsAlarm(MyAlarm,42,ComputedValue);
}
```

5.23 OSMINCYCLE_<CounterID>

Constant definition of the minimum number of ticks for a cyclic alarm on the Counter called CounterID.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

Example

```
if (ComputedValue < OSMINCYCLE_SomeCounter) {
    SetAbsAlarm(MyAlarm,42,OSMINCYCLE_SomeCounter);
} else {
    SetAbsAlarm(MyAlarm,42,ComputedValue);
}
```

5.24 OSSWTICKDURATION

Duration of a stopwatch tick in nanoseconds.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
time_in_ns = StopwatchMeasurement * OSSWTICKDURATION;
```

5.25 OSSWTICKSPERSECOND

The number of stopwatch ticks per second.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
time_in_secs = CycleMeasurement / OSSWTICKSPERSECOND;
```

5.26 OSTICKDURATION

Duration of a tick of the Counter called SystemCounter in nanoseconds.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

Example

```
uint32   RealTimeDeadline = 50000000; /* 50 ms */
TickType Deadline = (TickType)RealTimeDeadline / OSTICKDURATION;
SetRelAlarm(Timeout,Deadline,0);
```

5.27 OSTICKDURATION_<CounterID>

Duration of a tick of the Counter called CounterID in nanoseconds.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
uint32   RealTimeDeadline = 50000000; /* 50 ms */
TickType Deadline = (TickType)RealTimeDeadline /
    OSTICKDURATION_SomeCounter;
SetRelAlarm(Timeout,Deadline,0);
```

5.28 OSTICKSPERBASE

Constant definition of the ticks per base setting of the Counter called SystemCounter in ticks.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

5.29 OSTICKSPERBASE_<CounterID>

Constant definition of the ticks per base setting of the Counter called CounterID in ticks.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

5.30 OS_ACTIVATION_MONITORING

This macro is only defined if task activation monitoring is configured.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
#ifdef OS_ACTIVATION_MONITORING
Os_StopwatchTickType t;
Os_GetTaskActivationTime(MyTask,&t);
#endif
```

5.31 OS_ADD_TASK

This macro is used to add a Task to an Os_TasksetType. It may only be used when delayed task execution is configured.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
Os_TasksetType t1_and_t3 = TASK_MASK(Task1);
OS_ADD_TASK(t1_and_t3, Task3);
```

5.32 OS_CORE_CURRENT

The ID of the calling core for the APIs Os_GetIdleElapsedTime and Os_ResetIdleElapsedTime.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

5.33 OS_CORE_FOR_<TaskOrISR>

Returns the ID of the core on which the Task or ISR runs. Only available on multi-core.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
CoreIdType core;
core = OS_CORE_FOR_MyTask;
core = OS_CORE_FOR_MyCat2ISR;
core = OS_CORE_FOR_MyCat1ISR;
```

5.34 OS_CORE_FOR_ISR

Returns the ID of the core on which the ISR runs. Only available on multi-core. May only be passed the name of a category 2 ISR.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
CoreIdType core;
core = OS_CORE_FOR_ISR(MyCat2ISR);
core = OS_CORE_FOR_ISR(MyCat1ISR);
```

5.35 OS_CORE_FOR_TASK

Returns the ID of the core on which the Task runs. Only available on multi-core. Can be passed a TaskType. Note there is no checking on the value of the TaskType passed.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
CoreIdType core;
core = OS_CORE_FOR_TASK(MyTask);
```

5.36 OS_CORE_ID_0

The logical ID of CORE 0.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✓	✗

5.37 OS_CORE_ID_1

The logical ID of CORE 1. (etc.)

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✓	✗

5.38 OS_CORE_ID_MASTER

The ID of the master core.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✓	✗

5.39 OS_COUNT_USER_n

Ten user-counters (0 to 9) are available when the OS option 'Collect OS usage metrics' is enabled. You can place OS_COUNT_USER_0() to OS_COUNT_USER_9() in your code to cause the appropriate counter to increment.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
TASK(MyTask){
  ...
#ifdef OS_METRICS_ENABLED
  OS_COUNT_USER_5();
#endif /* OS_METRICS_ENABLED */
  ...
}
```

5.40 OS_COUNT_cat1isrname

The OS cannot instrument Category 1 ISRs, so when the OS option 'Collect OS usage metrics' is enabled it creates a macro that you can place in your handler in order to capture the number of interrupts seen.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
CAT1_ISR(MyCat1ISR) {
  ...
#ifdef OS_METRICS_ENABLED
  OS_COUNT_MyCat1ISR();
#endif /* OS_METRICS_ENABLED */
  ...
}
```

5.41 OS_CURRENT_IDLEMODE

Returns the idle mode for the calling core.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

5.42 OS_DURATION_<ScheduleTableID>

Constant definition of the duration of the ScheduleTable named ScheduleTableID in ticks of its Counter.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

5.43 OS_ELAPSED_TIME_RECORDING

This macro is only defined if elapsed time recording is configured.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
#ifdef OS_ELAPSED_TIME_RECORDING
Os_StopwatchTickType total_idle_time;
Os_StopwatchTickType total_taskA_time;
total_idle_time  = Os_GetIdleElapsedTime(OS_CORE_CURRENT);
total_taskA_time = Os_GetTaskElapsedTime(TaskA);
#endif
```

5.44 OS_EXTENDED_STATUS

Defined when extended status is configured.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
#ifdef OS_EXTENDED_STATUS
CheckStatusType = ActivateTask(Task1);
if (CheckStatusType == E_OS_LIMIT) {
   /* Log an error */
}
#else
ActivateTask(Task1);
#endif
```

5.45 OS_FAST_TASK_TERMINATION

Defined when Fast task termination is configured (STANDARD status only).

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

5.46 OS_IMASK_FOR_<TaskOrISR>

Returns the interrupt priority/mask for the Task or ISR. Although it may be emitted for CPU traps, the value may not be useful because trap priorities are often outside the normal controllable range.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
Os_imaskType imask;
imask = OS_IMASK_FOR_MyTask;
imask = OS_IMASK_FOR_MyCat2ISR;
imask = OS_IMASK_FOR_MyCat1ISR;
```

5.47 OS_IMASK_FOR_ISR

Returns the interrupt priority/mask for the ISR. Must be passed the name of an ISR. Although it may be emitted for CPU traps, the value may not be useful because trap priorities are often outside the normal controllable range.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
Os_imaskType imask;
imask = OS_IMASK_FOR_ISR(MyCat2ISR);
imask = OS_IMASK_FOR_ISR(MyCat1ISR);
```

5.48 OS_IMASK_FOR_TASK

Returns the interrupt priority/mask for the named Task. This typically only 'nonzero' when the TASK shares an internal RESOURCE with an ISR. Must be passed the name of a TASK.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
Os_imaskType imask;
imask = OS_IMASK_FOR_TASK(MyTask);
```

5.49 OS_INDEX_FOR_<Isr>

Returns a unique integral 0-based id for the Category2 ISR.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
switch (OS_ISRTYPE_TO_INDEX(isr_being_checked)) {
  case OS_INDEX_FOR_MyISR:
    ...
    break;
  case OS_INDEX_FOR_YourISR:
    ...
    break;
}
```

5.50 OS_INDEX_FOR_<Task>

Returns a unique integral 0-based id for the Task.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
switch (OS_TASKTYPE_TO_INDEX(task_being_checked)) {
  case OS_INDEX_FOR_MyTask:
    ...
    break;
  case OS_INDEX_FOR_YourTask:
    ...
    break;
}
```

5.51 OS_INDEX_TO_ISRTYPE

Macro that can convert an ISR-specific index in the range 0 to OS_NUM_ISRS-1 to a valid ISRType

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
ISRType isr = OS_INDEX_TO_ISRTYPE(2);
```

5.52 OS_INDEX_TO_TASKTYPE

Macro that can convert a task-specific index in the range 0 to OS_NUM_TASKS-1 to a valid TaskType

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
ActivateTask(OS_INDEX_TO_TASKTYPE(2));
```

5.53 OS_INVALID_TPL

Value returned from Os_GetCurrentTPL() when no TASK is running.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
if (OS_INVALID_TPL == Os_GetCurrentTPL()) { /* In idle */ }
```

5.54 OS_ISRTYPE_TO_INDEX

Macro that can convert a valid ISRType to an ISR-specific index in the range 0 to OS_NUM_ISRS-1

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
extern Os_StopwatchTickType isr_time[OS_NUM_ISRS];
isr_time[OS_ISRTYPE_TO_INDEX(GetISRID())] = GetExecutionTime();
```

5.55 OS_MAIN

Declare the main program. Use of OS_MAIN() rather than main() is preferred for portable code, because different compilers have different requirements on the parameters and return type of main().

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
#include "Os.h"
OS_MAIN() {
  /* Initialize target hardware */
  StartOS(OSDEFAULTAPPMODE);
}
```

5.56 OS_NOAPPMODE

The value returned by GetActiveApplicationMode() when the OS is not running.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

5.57 OS_NO_TASKS

This macro is used to initialize an empty Os_TasksetType. It can only be used when delayed task execution is configured.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
Os_TasksetType t1 = OS_NO_TASKS;
```

5.58 OS_NUM_ALARMS

The number of alarms declared.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

5.59 OS_NUM_APPLICATIONS

The number of OS Applications declared.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

5.60 OS_NUM_APPMODES

The number of AppModes declared.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

5.61 OS_NUM_CORES

The number of Cores declared (OsNumberOfCores).

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

5.62 OS_NUM_COUNTERS

The number of counters declared.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

5.63 OS_NUM_EVENTS

The number of Events declared.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

5.64 OS_NUM_IOC_CALLBACK_FUNCTIONS

The number of IOC callback functions reserved for –ioc:stub.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

5.65 OS_NUM_ISRS

The number of Category 2 ISRs declared.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

5.66 OS_NUM_OS_CORES

The number of Cores that are configured for the OS.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

5.67 OS_NUM_PERIPHERALAREAS

The number of PeripheralAreas declared.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

5.68 OS_NUM_RESOURCES

The number of resources declared (excludes internal).

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

5.69 OS_NUM_SCHEDULETABLES

The number of schedule tables declared.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

5.70 OS_NUM_SPINLOCKS

The number of Spinlocks declared.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✓	✗

5.71 OS_NUM_TASKS

The number of tasks declared.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

5.72 OS_NUM_TRUSTED_FUNCTIONS

The number of Trusted functions declared.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

5.73 OS_REGSET_<RegisterSetID>_SIZE

This macro defines the size of the buffer needed to preserve Register Set <RegisterSetID> at run time. If no buffer is needed, then it is not declared. This can happen if no task/ISR that uses the register set can be preempted by another one that also uses it.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
#ifdef OS_REGSET_FP_SIZE
  fp_context_save_area fpsave[OS_REGSET_FP_SIZE];
#endif /* OS_REGSET_FP_SIZE */
```

5.74 OS_SCALABILITY_CLASS_1

Defined when AUTOSAR Scalability Class 1 is configured.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
#ifdef OS_SCALABILITY_CLASS_1
ALARMCALLBACK(OnlyInSC1){
    ...
}
#endif
```

5.75 OS_SCALABILITY_CLASS_2

Defined when AUTOSAR Scalability Class 2 is configured.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
#if defined(OS_SCALABILITY_CLASS_2) || defined(OS_SCALABILITY_CLASS_4)
StartScheduleTableSynchron(Table);
#endif
```

5.76 OS_SCALABILITY_CLASS_3

Defined when AUTOSAR Scalability Class 3 is configured.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
#if defined(OS_SCALABILITY_CLASS_3) || defined(OS_SCALABILITY_CLASS_4)
FUNC(void, {memclass}) ErrorHook_MyApplication(StatusType Error){
 /* Handle OS-Application error */
}
#endif
```

5.77 OS_SCALABILITY_CLASS_4

Defined when AUTOSAR Scalability Class 4 is configured.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
#if defined(OS_SCALABILITY_CLASS_3) || defined(OS_SCALABILITY_CLASS_4)
FUNC(void, {memclass}) ErrorHook_MyApplication(StatusType Error){
 /* Handle OS-Application error */
}
#endif
```

5.78 OS_STACK_MONITORING

This macro is only defined if stack monitoring is configured.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
FUNC(boolean, {memclass}) Os_Cbk_Idle(void){
    #ifdef OS_STACK_MONITORING
        Os_StackSizeType Task1Stack, Task2Stack, Task3Stack;
        Task1Stack = Os_GetTaskMaxStackUsage(Task1);
        Task2Stack = Os_GetTaskMaxStackUsage(Task2);
        ...
        TaskNStack = Os_GetTaskMaxStackUsage(TaskN);
    #endif
    return TRUE;
}
```

5.79 OS_STANDARD_STATUS

Defined when standard status is configured.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
#ifdef OS_STANDARD_STATUS
ActivateTask(Task1);
#else
CheckStatusType = ActivateTask(Task1);
if (CheckStatusType == E_OS_LIMIT) {
   /* Log an error */
}
#endif
```

5.80 OS_SUPPORTS_TRUSTED_WITH_PROTECTION

Defined when there are OS Applications that are configured for TrustedApplicationWithProtection.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

5.81 OS_TASKTYPE_TO_INDEX

Macro that can convert a valid TaskType to a task-specific index in the range 0 to OS_NUM_TASKS-1

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
extern Os_StopwatchTickType task_time[OS_NUM_TASKS];
TaskType this;
GetTaskID(&this);
task_time[OS_TASKTYPE_TO_INDEX(this)] = GetExecutionTime();
```

5.82 OS_TICKS2<Unit>_<CounterID>(ticks)

Converts ticks on CounterID to Unit where Unit is: NS (nanosecond), MS (Millisecond), US (Microsecond), SEC(Second).

RTA-OS will try where possible to generate these macros using integer multiplication or division. However for some tick rates it is necessary to use floating-point numbers in the calculation in order to preserve accuracy. Where these macros are passed fixed values known at compile-time, the compiler will usually perform the calculation itself and embed the integral result. If the value passed is a variable, then the compiler will have to generate code that uses floating-point calculations at run time. You should check the file Os_Cfg.h to inspect the code for the macros if this might be a problem in your application.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✓	✓	✓	✗

Example

```
time_in_ms = OS_TICKS2MS_SystemCounter(time);
```

5.83 OS_TIME_MONITORING

This macro is only defined if time monitoring is configured.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
#ifdef OS_TIME_MONITORING
Os_StopwatchTickType start,end,function_duration;
start = Os_GetExecutionTime();
#endif
ThirdPartyFunction(x,y);
#ifdef OS_TIME_MONITORING
end = Os_GetExecutionTime();
function_duration = end - start;
#endif
```

5.84 OS_TPL_FOR_<Task>

Returns the internal TASK base priority for the Task.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
uint32 tpl;
tpl = OS_TPL_FOR_MyTask;
```

5.85 OS_TPL_FOR_TASK

Returns the internal TASK base priority for the named TASK. Must be passed the name of a TASK.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
uint32 tpl;
tpl = OS_TPL_FOR_TASK(MyTask);
```

5.86 Os_DisableAllConfiguredInterrupts

The Os_DisableAllConfiguredInterrupts macro will disable all configured interrupts. You will need to #include the file "Os_DisableInterrupts.h" if you want to use this macro. It may not be used by untrusted code.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
Os_DisableAllConfiguredInterrupts()
Os_Enable_Millisecond()
```

5.87 Os_Disable_x

The Os_Disable_x macro will disable the named interrupt. It is normally paired with a call to Os_Enable_x. The macro can be called using either the vector address or the RTA-OS configured vector name. In the example, this is Os_Disable_12() and Os_Disable_Millisecond() respectively. You will need to #include the file "Os_DisableInterrupts.h" if you want to use these macros. They may not be used by untrusted code.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
Os_Disable_12()
Os_Disable_Millisecond()
```

5.88 Os_EnableAllConfiguredInterrupts

The Os_EnableAllConfiguredInterrupts macro will enable all configured interrupts. You will need to #include the file "Os_DisableInterrupts.h" if you want to use this macro. It may not be used by untrusted code.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
Os_DisableAllConfiguredInterrupts()
...
Os_EnableAllConfiguredInterrupts()
```

5.89 Os_Enable_x

The Os_Enable_x macro will re-enable the named interrupt. It is normally paired with a call to Os_Disable_x. The macro can be called using either the vector address or the RTA-OS configured vector name. In the example, this is Os_Enable_12() and Os_Enable_Millisecond() respectively. You will need to #include the file "Os_DisableInterrupts.h" if you want to use these macros. They may not be used by untrusted code.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
Os_Enable_12()
Os_Enable_Millisecond()
```

5.90 TASK

Macro that must be used to create the task's entry function.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✓	✓	✓	✓	✗

Example

```
TASK(MyTask) {...}
```

5.91 TASK_MASK

This macro is used to convert a TaskType to an Os_TasksetType. It may only be used when delayed task execution is configured.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✗

Example

```
Os_TasksetType t1 = TASK_MASK(Task1);
```

6 RTA-TRACE API calls

6.1 Guide to Descriptions

All API calls have the following structure:

Syntax

```
/* C function prototype for the API call */
ReturnValue NameOfAPICall(Parameter Type, ...)
```

Parameters

A list of parameters for each API call and their mode:

in The parameter is passed in to the call

out The parameter is passed out of the API call by passing a reference (pointer) to the parameter into the call.

inout The parameter is passed into the call and then (updated) and passed out.

Return Values

Where API calls return a `StatusType` the values of the type returned and an indication of the reason for the error/warning are listed. The build column indicates whether the value is returned for both standard and extended status builds or for extended status build only.

Description

A detailed description of the behavior of the API call.

Portability

The RTA-OS API includes four classes of API calls:

OSEK calls are those specified by the OSEK OS standard. OSEK OS calls are portable to other implementations of OSEK OS and are portable to other implementations of AUTOSAR OS.

AUTOSAR R3.x calls are those specified by the AUTOSAR R3.x standards. AUTOSAR OS calls are portable to other implementations of AUTOSAR R3.x. The calls are portable to OSEK OS only if the call is also an OSEK OS call.

AUTOSAR R4.x calls are those specified by the AUTOSAR R4.x standards. AUTOSAR OS calls are portable to other implementations of AUTOSAR R4.x. The calls are portable to OSEK OS only if the call is also an OSEK OS call.

MultiCore calls are those specified by the AUTOSAR R4.0 multicore OS standards. These calls are portable to other implementations of AUTOSAR R4.x. They are only available when more than one core has been configured and the target variant supports more than one core.

RTA-TRACE calls are provided by RTA-OS for controlling the RTA-TRACE run-time profiling tool. These calls are only available when RTA-TRACE support has been configured.

RTA-OS calls include all those from the other three classes plus calls that provide extensions AUTOSAR OS functionality. These calls are unique to RTA-OS and are not portable to other implementations.

Example Code

```
A C code listing showing how to use the API calls
```

Calling Environment

The valid calling environment for the API call. A ✓ indicates that a call can be made in the indicated context. A ✗ indicates that the call cannot be made in the indicated context.

See Also

A list of related API calls.

6.2 Multicore notes

RTA-TRACE stores its control and trace data in separate per-core memory areas in a multicore application. All of the main RTA-TRACE APIs described here operate individually on the core on which they are called. You can, for example, select bursting tracing on one core and triggered tracing on another. Per-core behavior extends to the APIs that are concerned with the upload of the traced data - you use the same APIs and uploading strategy on multi and single core applications. The only difference is that you only need to initialize the trace communications hardware on one of the cores. RTA-OS ensures that only one core is actually sending trace data at a time.

6.3 Os_CheckTraceOutput

Checks for the presence of trace data.

Syntax

```
void Os_CheckTraceOutput(void)
```

Description

When tracing in free-running mode, this must be called regularly by the application. It is used to detect when the trace buffer has data to upload to RTA-TRACE.

It does not have to be called in Bursting or Triggering modes, though it is not harmful to do so.

It causes Os_Cbk_TraceCommDataReady() to be called when there is data to send.

This API will only check trace data for the core that calls it. If you have more than one core tracing data, each one needs to call this independently.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

Example

```
Os_CheckTraceOutput();
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✗	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✗	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✗	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

See Also

Os_Cbk_TraceCommDataReady
Os_Cbk_TraceCommTxByte
Os_Cbk_TraceCommTxEnd
Os_Cbk_TraceCommTxReady
Os_Cbk_TraceCommTxStart

6.4 Os_ClearTrigger

Clear all triggering conditions.

Syntax

```
void Os_ClearTrigger(void)
```

Description

This API call clears all trigger conditions that have been set using an Os_TriggerOnXXX() API.

Trace information will continue to be logged in the trace buffer, but no trace record will trigger the upload of the trace buffer to the host.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

Example

```
Os_ClearTrigger();
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

See Also

Os_SetTraceRepeat
Os_SetTriggerWindow
Os_StartBurstingTrace
Os_StartFreeRunningTrace

6.5 Os_DisableTraceCategories

Control which tracepoints are traced.

Syntax

```
void Os_DisableTraceCategories(
    Os_TraceCategoriesType CategoriesMask
)
```

Parameters

Parameter	Mode	Description
CategoriesMask	in	Os_TraceCategoriesType Mask of the trace categories to disable.

Description

Trace categories are used to filter whether tracepoints, task tracepoints and intervals get recorded and are typically used to control the volume of data that gets traced.

A category can be configured at build time to be active always, never or under run-time control. Categories that are under run-time control are enabled using Os_EnableTraceCategories and disabled using Os_DisableTraceCategories.

This call disables the specified run-time categories and therefore will inhibit the logging of all tracepoints, task tracepoints and intervals that are filtered by these categories.

Categories not listed in the call will be left in their current state. The category filter applies to all cores in a multicore application.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

Example

```
Os_DisableTraceCategories(DebugTracePoints | DataLogTracePoints); /*
    Disable DebugTracePoints and DataLogTracePoints*/
Os_LogTracepoint(tpTest, DebugTracePoints); /* tpTest is not recorded:
    DebugTracePoints is disabled */
Os_LogTracepoint(tpTest, OS_TRACE_CATEGORY_ALWAYS); /* tpTest is recorded
    here */
Os_DisableTraceCategories(OS_TRACE_ALL_CATEGORIES); /* Disable all
    categories except for OS_TRACE_CATEGORY_ALWAYS */
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

See Also

Os_EnableTraceCategories

6.6 Os_DisableTraceClasses

Control which types of objects are traced.

Syntax

```
void Os_DisableTraceClasses(
    Os_TraceClassesType ClassMask
)
```

Parameters

Parameter	Mode	Description
ClassMask	in	Os_TraceClassesType Mask of the trace classes to disable.

Description

Trace classes are used to filter whether complete types of trace events get recorded. They are typically used to control the volume of data that gets traced.

Trace classes can be configured at build time to be active always, never or under run-time control. Classes that are under run-time control are enabled using Os_EnableTraceClasses and disabled using Os_DisableTraceClasses.

This call disables the specified run-time classes and therefore will inhibit the tracing of events that are filtered by these classes.

Classes not listed in the call will be left in their current state. The class filter applies to all cores in a multicore application.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

Example

```
Os_DisableTraceClasses(OS_TRACE_TRACEPOINT_CLASS);
Os_LogTracepoint(tpTest, OS_TRACE_ALL_CATEGORIES);  /* Will not get
    recorded */
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

See Also

Os_EnableTraceClasses

6.7 Os_EnableTraceCategories

Control which tracepoints are traced.

Syntax

```
void Os_EnableTraceCategories(
    Os_TraceCategoriesType CategoriesMask
)
```

Parameters

Parameter	Mode	Description
CategoriesMask	in	Os_TraceCategoriesType Mask of the trace categories to enable.

Description

Trace categories are used to filter whether tracepoints, task tracepoints and intervals get recorded and are typically used to control the volume of data that gets traced.

A category can be configured at build time to be active always, never or under run-time control. Categories that are under run-time control are enabled using Os_EnableTraceCategories and disabled using Os_DisableTraceCategories.

This call enables the specified run-time categories.

Categories not listed in the call will be left in their current state. The category filter applies to all cores in a multicore application.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

Example

```
Os_EnableTraceCategories(DebugTracePoints | DataLogTracePoints);
Os_LogTracepoint(tpTest, DebugTracePoints); /* tpTest is recorded */
Os_LogTracepoint(tpTest, FunctionProfileTracePoints); /* tpTest is not
    recorded - FunctionProfileTracePoints not enabled */
Os_LogTracepoint(tpTest, OS_TRACE_ALL_CATEGORIES); /* tpTest is recorded */
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

See Also

Os_DisableTraceCategories

6.8 Os_EnableTraceClasses

Control which types of objects are traced.

Syntax

```
void Os_EnableTraceClasses(
    Os_TraceClassesType ClassMask
)
```

Parameters

Parameter	Mode	Description
ClassMask	in	Os_TraceClassesType Mask of the trace classes to enable.

Description

Trace classes are used to filter whether complete types of trace events get recorded. They are typically used to control the volume of data that gets traced.

Trace classes can be configured at build time to be active always, never or under run-time control. Classes that are under run-time control are enabled using Os_EnableTraceClasses and disabled using Os_DisableTraceClasses.

This call enables the specified run-time classes. The class filter applies to all cores in a multicore application.

Classes not listed in the call will be left in their current state.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

Example

```
Os_EnableTraceClasses(OS_TRACE_TRACEPOINT_CLASS);
Os_LogTracepoint(tpTest, OS_TRACE_ALL_CATEGORIES);  /* Will get recorded */
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

See Also

Os_DisableTraceClasses

6.9 Os_LogCat1ISREnd

Log the end of a Category 1 ISR.

Syntax

```
void Os_LogCat1ISREnd(
    ISRType ISRID
)
```

Parameters

Parameter	Mode	Description
ISRID	in	ISRType Category 1 ISR identifier.

Description

This call marks the end of a Category 1 ISR. This type of ISR is not controlled by the operating system so no automatic tracing of it can occur. If Category 1 ISRs need to be logged then it is necessary to do this manually using this call.

This event is only logged if the OS_TRACE_TASKS_AND_ISRS_CLASS trace class is active.

Take care to ensure that both the start and end of the Category 1 ISR logged, otherwise the resulting trace will be incorrect.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

Example

```
CAT1_ISR(Category1Handler) {
  Os_LogCat1ISRStart(Category1Handler);
  ...
  Os_LogCat1ISREnd(Category1Handler);
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✗	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✓	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✗	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

See Also

Os_LogCat1ISRStart

6.10 Os_LogCat1ISRStart

Log the start of a Category 1 ISR.

Syntax

```
void Os_LogCat1ISRStart(
    ISRType ISRID
)
```

Parameters

Parameter	Mode	Description
ISRID	in	ISRType Category 1 ISR identifier.

Description

This call marks the start of a Category 1 ISR. This type of ISR is not controlled by the operating system so no automatic tracing of it can occur. If Category 1 ISRs need to be logged then it is necessary to do this manually using this call.

This event is only logged if the OS_TRACE_TASKS_AND_ISRS_CLASS trace class is active.

Take care to ensure that both the start and end of the Category 1 ISR are logged, otherwise the resulting trace will be incorrect.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

Example

```
CAT1_ISR(Category1Handler) {
  Os_LogCat1ISRStart(Category1Handler);
  ...
  Os_LogCat1ISREnd(Category1Handler);
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✗	PreTaskHook	✗	StackOverrunHook	✗
Category 1 ISR	✓	PostTaskHook	✗	TimeOverrunHook	✗
Category 2 ISR	✗	StartupHook	✗		
		ShutdownHook	✗		
		ErrorHook	✗		
		ProtectionHook	✗		

See Also

Os_LogCat1ISREnd

6.11 Os_LogCriticalExecutionEnd

Log the completion of a critical execution event.

Syntax

```
void Os_LogCriticalExecutionEnd(
    Os_TraceInfoType CriticalExecutionID
)
```

Parameters

Parameter	Mode	Description
CriticalExecutionID	in	Os_TraceInfoType Critical execution profile identifier.

Description

Logs the end of a critical point of execution in the trace buffer. This is typically used to indicate that a task/ISR has completed a time-critical section of code. This might be needed if the deadline that needs to be met by the task/ISR occurs before the end of the task/ISR.

CriticalExecutionID is only logged if the OS_TRACE_TASKS_AND_ISRS_CLASS class is active.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

Example

```
TASK(MyTask){
  ...
  ReadSensor(X);
  Os_LogCriticalExecutionEnd(SensorRead);
  ...
  WriteActuator(Y);
  Os_LogCriticalExecutionEnd(SensorRead);
  ...
  TerminateTask();
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

See Also

None.

6.12 Os_LogIntervalEnd

Log the end of a measurement interval.

Syntax

```
void Os_LogIntervalEnd(
    Os_TraceIntervalIDType IntervalID,
    Os_TraceCategoriesType CategoryMask
)
```

Parameters

Parameter	Mode	Description
IntervalID	in	Os_TraceIntervalIDType Interval Identifier.
CategoryMask	in	Os_TraceCategoriesType A category mask.

Description

Log the end of an interval in the trace buffer.

The interval is only logged if the OS_TRACE_INTERVAL_CLASS class is active and one or more of the categories in CategoryMask are active.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

Example

```
Os_LogIntervalStart(EndToEndTime, SystemLoggingCategory);
...
Os_LogIntervalEnd(EndToEndTime, SystemLoggingCategory);
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

See Also

Os_LogIntervalEndData
Os_LogIntervalEndValue
Os_LogIntervalStart
Os_LogIntervalStartData
Os_LogIntervalStartValue

6.13 Os_LogIntervalEndData

Log the end of a measurement interval together with associated data.

Syntax

```
void Os_LogIntervalEndData(
    Os_TraceIntervalIDType IntervalID,
    Os_TraceDataPtrType DataPtr,
    Os_TraceDataLengthType Length,
    Os_TraceCategoriesType CategoryMask
)
```

Parameters

Parameter	Mode	Description
IntervalID	in	Os_TraceIntervalIDType Interval Identifier.
DataPtr	in	Os_TraceDataPtrType A pointer to the start address of the data block to log.
Length	in	Os_TraceDataLengthType The length of the data block in bytes.
CategoryMask	in	Os_TraceCategoriesType A category mask.

Description

Log the end of an interval in the trace buffer and associate some data with it.

The interval is only logged if the OS_TRACE_INTERVAL_CLASS class is active and one or more of the categories in CategoryMask are active.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

Example

```
Os_LogIntervalStart(EndToEndTime, SystemLoggingCategory);
...
Os_LogIntervalEndData(EndToEndTime, &DataBlock, 4,SystemLoggingCategory);
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

See Also

Os_LogIntervalEnd
Os_LogIntervalEndValue
Os_LogIntervalStart
Os_LogIntervalStartData
Os_LogIntervalStartValue

6.14 Os_LogIntervalEndValue

Log the end of a measurement interval together with an associated value.

Syntax

```
void Os_LogIntervalEndValue(
    Os_TraceIntervalIDType IntervalID,
    Os_TraceValueType Value,
    Os_TraceCategoriesType CategoryMask
)
```

Parameters

Parameter	Mode	Description
IntervalID	in	Os_TraceIntervalIDType Interval Identifier.
Value	in	Os_TraceValueType Numerical value to be logged with the interval.
CategoryMask	in	Os_TraceCategoriesType A category mask.

Description

Log the end of an interval in the trace buffer and associate a value with it.

The interval is only logged if the OS_TRACE_INTERVAL_CLASS class is active and one or more of the categories in CategoryMask are active.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

Example

```
Os_LogIntervalStart(EndToEndTime, SystemLoggingCategory);
...
Os_LogIntervalEndValue(EndToEndTime, 42, SystemLoggingCategory);
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

See Also

Os_LogIntervalEnd
Os_LogIntervalEndData
Os_LogIntervalEndValue
Os_LogIntervalStartData
Os_LogIntervalStartValue

6.15 Os_LogIntervalStart

Log the start of a measurement interval.

Syntax

```
void Os_LogIntervalStart(
    Os_TraceIntervalIDType IntervalID,
    Os_TraceCategoriesType CategoryMask
)
```

Parameters

Parameter	Mode	Description
IntervalID	in	Os_TraceIntervalIDType Interval Identifier.
CategoryMask	in	Os_TraceCategoriesType A category mask.

Description

Log the start of an interval in the trace buffer.

The interval is only logged if the OS_TRACE_INTERVAL_CLASS class is active and one or more of the categories in CategoryMask are active.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

Example

```
Os_LogIntervalStart(EndToEndTime, SystemLoggingCategory);
...
Os_LogIntervalEnd(EndToEndTime, SystemLoggingCategory);
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

See Also

Os_LogIntervalEnd
Os_LogIntervalEndData
Os_LogIntervalEndValue
Os_LogIntervalStartData
Os_LogIntervalStartValue

6.16 Os_LogIntervalStartData

Log the start of a measurement interval together with associated data.

Syntax

```
void Os_LogIntervalStartData(
    Os_TraceIntervalIDType IntervalID,
    Os_TraceDataPtrType DataPtr,
    Os_TraceDataLengthType Length,
    Os_TraceCategoriesType CategoryMask
)
```

Parameters

Parameter	Mode	Description
IntervalID	in	Os_TraceIntervalIDType Interval Identifier.
DataPtr	in	Os_TraceDataPtrType A pointer to the start address of the data block to log.
Length	in	Os_TraceDataLengthType The length of the data block in bytes.
CategoryMask	in	Os_TraceCategoriesType A category mask.

Description

Log the start of an interval in the trace buffer and associate some data with it.

The interval is only logged if the OS_TRACE_INTERVAL_CLASS class is active and one or more of the categories in CategoryMask are active.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

Example

```
Os_LogIntervalStartData(EndToEndTime, &DataBlock, 4,
    SystemLoggingCategory);
...
Os_LogIntervalEnd(EndToEndTimeSystemLoggingCategory);
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

See Also

Os_LogIntervalEnd
Os_LogIntervalEndData
Os_LogIntervalEndValue
Os_LogIntervalStart
Os_LogIntervalStartValue

6.17 Os_LogIntervalStartValue

Log the start of a measurement interval together with an associated value.

Syntax

```
void Os_LogIntervalStartValue(
    Os_TraceIntervalIDType IntervalID,
    Os_TraceValueType Value,
    Os_TraceCategoriesType CategoryMask
)
```

Parameters

Parameter	Mode	Description
IntervalID	in	Os_TraceIntervalIDType Interval Identifier.
Value	in	Os_TraceValueType Numerical value to be logged with the interval.
CategoryMask	in	Os_TraceCategoriesType A category mask.

Description

Log the start of an interval in the trace buffer and associate a value with it.

The interval is only logged if the OS_TRACE_INTERVAL_CLASS class is active and one or more of the categories in CategoryMask are active.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

Example

```
Os_LogIntervalStartValue(EndToEndTime, 42, SystemLoggingCategory);
...
Os_LogIntervalEnd(EndToEndTime, SystemLoggingCategory);
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

See Also

Os_LogIntervalEnd
Os_LogIntervalEndData
Os_LogIntervalEndValue
Os_LogIntervalStart
Os_LogIntervalStartData

6.18 Os_LogProfileStart

Log the start of a new execution profile.

Syntax

```
void Os_LogProfileStart(
    Os_TraceInfoType ProfileID
)
```

Parameters

Parameter	Mode	Description
ProfileID	in	Os_TraceInfoType Profile Identifier.

Description

Logs which execution profile is active in the trace buffer. Execution profiles can be used to identify which route is taken through a Task or ISR when this depends on external conditions.

The profile is only recorded if the OS_TRACE_TASKS_AND_ISRS_CLASS class is active.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

Example

```
TASK(MyTask){
  if (some_condition()) {
    Os_LogProfileStart(TrueRoute);
    ...
  } else {
    Os_LogProfileStart(FalseRoute);
    ...
  }
  TerminateTask();
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

See Also

None.

6.19 Os_LogTaskTracepoint

Log a tracepoint in the specified categories.

Syntax

```
void Os_LogTaskTracepoint(
    Os_TraceTracepointIDType TaskTracepointID,
    Os_TraceCategoriesType CategoryMask
)
```

Parameters

Parameter	Mode	Description
TaskTracepointID	in	Os_TraceTracepointIDType Task Tracepoint Identifier.
CategoryMask	in	Os_TraceCategoriesType A category mask.

Description

Log the task tracepoint event in the trace buffer.

TaskTracepointID is recorded only if the OS_TRACE_TASK_TRACEPOINT_CLASS class is active and one or more of the categories in CategoryMask are active.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

Example

```
Os_LogTaskTracepoint(MyTaskTracePoint, ACategory);
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

See Also

Os_LogTaskTracepointData
Os_LogTaskTracepointValue
Os_LogTracepoint
Os_LogTracepointData
Os_LogTracepointValue

6.20 Os_LogTaskTracepointData

Log a tracepoint in the specified categories together with associated data.

Syntax

```
void Os_LogTaskTracepointData(
    Os_TraceTracepointIDType TracepointID,
    Os_TraceDataPtrType DataPtr,
    Os_TraceDataLengthType Length,
    Os_TraceCategoriesType CategoryMask
)
```

Parameters

Parameter	Mode	Description
TracepointID	in	Os_TraceTracepointIDType Tracepoint Identifier.
DataPtr	in	Os_TraceDataPtrType A pointer to the start address of the data block to log.
Length	in	Os_TraceDataLengthType The length of the data block in bytes.
CategoryMask	in	Os_TraceCategoriesType A category mask.

Description

Log the task tracepoint event in the trace buffer and associate some data with it.

TaskTracepointID is recorded only if the OS_TRACE_TASK_TRACEPOINT_CLASS class is active and one or more of the categories in CategoryMask are active.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

Example

```
Os_LogTaskTracepointData(MyTracePoint, &DataBlock, 4, ACategory);
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

See Also

Os_LogTaskTracepoint
Os_LogTaskTracepointValue
Os_LogTracepoint
Os_LogTracepointData
Os_LogTracepointValue

6.21 Os_LogTaskTracepointValue

Log a tracepoint in the specified categories together with an associated value.

Syntax

```
void Os_LogTaskTracepointValue(
    Os_TraceTracepointIDType TracepointID,
    Os_TraceValueType Value,
    Os_TraceCategoriesType CategoryMask
)
```

Parameters

Parameter	Mode	Description
TracepointID	in	Os_TraceTracepointIDType Tracepoint Identifier.
Value	in	Os_TraceValueType Numerical value to be logged with the tracepoint.
CategoryMask	in	Os_TraceCategoriesType A category mask.

Description

Log the task tracepoint event in the trace buffer and associate a value with it.

TaskTracepointID is recorded only if the OS_TRACE_TASK_TRACEPOINT_CLASS class is active and one or more of the categories in CategoryMask are active.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

Example

```
Os_LogTaskTracepointValue(MyTracePoint, 99, ACategory);
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

See Also

Os_LogTaskTracepoint
Os_LogTaskTracepointData
Os_LogTracepoint
Os_LogTracepointData
Os_LogTracepointValue

6.22 Os_LogTracepoint

Log a tracepoint in the specified categories.

Syntax

```
void Os_LogTracepoint(
    Os_TraceTracepointIDType TracepointID,
    Os_TraceCategoriesType CategoryMask
)
```

Parameters

Parameter	Mode	Description
TracepointID	in	Os_TraceTracepointIDType Tracepoint Identifier.
CategoryMask	in	Os_TraceCategoriesType A category mask.

Description

Log the tracepoint event in the trace buffer.

TracepointID is recorded only if the OS_TRACE_TRACEPOINT_CLASS class is active and one or more of the categories in CategoryMask are active.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

Example

```
Os_LogTracepoint(MyTracepoint, ACategory);
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

See Also

Os_LogTracepoint
Os_LogTracepointData
Os_LogTracepointData
Os_LogTracepointValue
Os_LogTracepointValue

6.23 Os_LogTracepointData

Log a tracepoint in the specified categories together with associated data.

Syntax

```
void Os_LogTracepointData(
    Os_TraceTracepointIDType TracepointID,
    Os_TraceDataPtrType DataPtr,
    Os_TraceDataLengthType Length,
    Os_TraceCategoriesType CategoryMask
)
```

Parameters

Parameter	Mode	Description
TracepointID	in	Os_TraceTracepointIDType Tracepoint Identifier.
DataPtr	in	Os_TraceDataPtrType A pointer to the start address of the data block to log.
Length	in	Os_TraceDataLengthType The length of the data block in bytes.
CategoryMask	in	Os_TraceCategoriesType A category mask.

Description

Log the tracepoint event in the trace buffer and associate some data with it.

TracepointID is recorded only if the OS_TRACE_TRACEPOINT_CLASS class is active and one or more of the categories in CategoryMask are active.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

Example

```
Os_LogTracepointData(MyTracePoint, &DataBlock, 4, ACategory);
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

See Also

Os_LogTracepoint
Os_LogTracepoint
Os_LogTracepointData
Os_LogTracepointValue
Os_LogTracepointValue

6.24 Os_LogTracepointValue

Log a tracepoint in the specified categories together with an associated value.

Syntax

```
void Os_LogTracepointValue(
    Os_TraceTracepointIDType TracepointID,
    Os_TraceValueType Value,
    Os_TraceCategoriesType CategoryMask
)
```

Parameters

Parameter	Mode	Description
TracepointID	in	Os_TraceTracepointIDType Tracepoint Identifier.
Value	in	Os_TraceValueType Numerical value to be logged with the tracepoint.
CategoryMask	in	Os_TraceCategoriesType A category mask.

Description

Log the tracepoint event in the trace buffer and associate a value with it.

TracepointID is recorded only if the OS_TRACE_TRACEPOINT_CLASS class is active and one or more of the categories in CategoryMask are active.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

Example

```
Os_LogTracepointValue(MyTracePoint, 99, ACategory);
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

See Also

Os_LogTracepoint
Os_LogTracepoint
Os_LogTracepointData
Os_LogTracepointData
Os_LogTracepointValue

6.25 Os_SetTraceRepeat

Control whether trace repeats or not.

Syntax

```
void Os_SetTraceRepeat(
    boolean Repeat
)
```

Parameters

Parameter	Mode	Description
Repeat	in	boolean Control whether bursting/triggering traces repeat.

Description

When TRUE, bursting and triggering trace modes automatically restart once the most recent trace content has been transmitted from the trace buffer to the RTA-TRACE client.

The API has no effect in free-running trace mode.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

Example

```
Os_SetTraceRepeat(TRUE);
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

See Also

Os_StartBurstingTrace
Os_StartTriggeringTrace

6.26 Os_SetTriggerWindow

Set the size of the trace buffer window to be uploaded in triggering mode.

Syntax

```
void Os_SetTriggerWindow(
    Os_TraceIndexType Before,
    Os_TraceIndexType After
)
```

Parameters

Parameter	Mode	Description
Before	in	Os_TraceIndexType Number of records to be recorded before the trigger event.
After	in	Os_TraceIndexType Number of records to record after the trigger event.

Description

This call sets the number of records to be recorded before and after a trigger event.

When the trigger occurs, tracing events continue to be logged until After trace records have been written to the trace buffer, and the data is then uploaded.

The total number of records uploaded (Before + After) is limited by the size of the trace buffer.

Note that a trace event that contains data values may require multiple records to be written to the trace buffer. This means that the number of complete events seen before or after the trigger point may be less than the number of records requested.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

Example

```
FUNC(void, {memclass}) StartupHook(){
  ...
  Os_SetTriggerWindow(100,50);
  Os_StartTriggeringTrace();
  ...
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

See Also

Os_StartBurstingTrace
Os_StartFreeRunningTrace

6.27 Os_StartBurstingTrace

Starts tracing in bursting mode.

Syntax

```
void Os_StartBurstingTrace(void)
```

Description

Bursting trace mode logs trace information into the trace buffer until the buffer is full. When the trace buffer is full, tracing stops and data transfer begins. No attempt is made to upload data to the host until the trace buffer has filled.

Where Os_SetTraceRepeat() has been used to enable repeated bursting traces, tracing resumes once the buffer is empty (i.e. once data transfer is complete).

The trace buffer is cleared and tracing restarts again if this call is made while tracing.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

Example

```
FUNC(void, {memclass}) StartupHook(){
  ...
  Os_StartBurstingTrace();
  ...
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

See Also

Os_SetTraceRepeat
Os_StartFreeRunningTrace
Os_StartTriggeringTrace

6.28 Os_StartFreeRunningTrace

Starts tracing in free-running mode.

Syntax

```
void Os_StartFreeRunningTrace(void)
```

Description

Free running trace mode logs trace information while there is space in the trace buffer. Data is uploaded to the host from the buffer as soon as it is available, concurrently with capture.

If the trace buffer becomes full, logging of trace data is suspended until there is space in the buffer. When space in the buffer is available again, tracing resumes. The buffer might become full if the communications link is too slow for the desired volume of trace data.

The trace buffer is cleared and tracing restarts again if this call is made while tracing.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

Example

```
FUNC(void, {memclass}) StartupHook(){
  ...
  Os_StartFreeRunningTrace();
  ...
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

See Also

Os_StartBurstingTrace
Os_StartTriggeringTrace

6.29 Os_StartTriggeringTrace

Starts tracing in triggering mode.

Syntax

```
void Os_StartTriggeringTrace(void)
```

Description

Triggering trace mode logs trace information into the buffer continuously, waiting for a trigger condition. If the buffer overflows, then new trace information overwrites existing information.

A pre- and post-trigger number of buffer records must be specified using Os_SetTriggerWindow() so that only the set of events before and after the trigger event can be seen. Unpredictable behavior may occur if the trigger window is not set.

Trigger events are set using the Os_TriggerOnXXX() APIs.

When a triggering event occurs (for example, when a task starts executing), data collection continues until post-trigger number of trace records are logged. Data transfer to the host then begins.

Tracing resumes after the data transfer completes if Os_SetTraceRepeat() permits this.

The trace buffer is cleared and tracing restarts again if this call is made while tracing.

Each core can have different trigger settings in a multicore application.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

Example

```
FUNC(void, {memclass}) StartupHook(){
  ...
  Os_SetTriggerWindow(100,50);
  Os_StartTriggeringTrace();
  ...
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

See Also

Os_SetTraceRepeat
Os_SetTriggerWindow
Os_StartBurstingTrace
Os_StartFreeRunningTrace

6.30 Os_StopTrace

Stops tracing.

Syntax

```
void Os_StopTrace(void)
```

Description

Stops data logging to the trace buffer. Any data remaining in the trace buffer is uploaded to the host.

Note that the call does not stop the data link.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

Example

```
Os_StopTrace();
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

See Also

Os_StartBurstingTrace
Os_StartFreeRunningTrace
Os_StartTriggeringTrace

6.31 Os_TraceCommInit

Initializes external communication support for tracing.

Syntax

```
Os_TraceStatusType Os_TraceCommInit(void)
```

Return Values

The call returns values of type Os_TraceStatusType.

Description

This function is used to initialize a trace communications link. It should not be used if you use a debugger link to extract trace data.

It calls the callback Os_Cbk_TraceCommInitTarget() to initialize the appropriate target hardware and its return value indicates the return value from the call to Os_Cbk_TraceCommInitTarget().

RTA-OS will call this during StartOS if automatic tracing is configured.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

Example

```
FUNC(void, {memclass}) StartupHook() {
  ...
  Os_TraceCommInit();
  Os_StartFreeRunningTrace();
  ...
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

See Also

Os_Cbk_TraceCommInitTarget

6.32 Os_TraceDumpAsync

Uses an asynchronous communication to upload trace data in a single operation.

Syntax

```
void Os_TraceDumpAsync(
    Os_AsyncPushCallbackType fn
)
```

Description

This API is normally called in response to Os_Cbk_TraceCommDataReady(). It gets passed a reference to a function that can transmit a single character. It will call this function for each character that needs to be transmitted before returning to the caller.

An appropriate asynchronous serial device must be available and previously initialized. A typical serial link might be set to 115200bps, 8 data bits, no parity and 1 stop bit.

This API will only upload trace data for the core that calls it. If you have more than one core tracing data, each one needs to call this independently.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

Example

```
FUNC(void, OS_CODE) push_async_io(Os_TraceDataType val) {
  while(!async_tx_ready) {/* wait for room */}
  async_transmit(val)  ;
}
FUNC(void, {memclass}) Os_Cbk_TraceCommDataReady(void) {
  Os_TraceDumpAsync(push_async_io);
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

See Also

Os_Cbk_TraceCommDataReady

6.33 Os_TriggerNow

Trigger upload of the trace buffer.

Syntax

```
void Os_TriggerNow(void)
```

Description

This API call forces a trigger condition to occur. This will cause the trace buffer to be uploaded, regardless of any other trigger conditions.

The call does not modify the state of the trigger conditions.

The call only has an effect in triggering trace mode.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

Example

```
Os_TriggerNow();
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

See Also

None.

6.34 Os_TriggerOnActivation

Trigger when a task is activated.

Syntax

```
void Os_TriggerOnActivation(
    TaskType TaskID
)
```

Parameters

Parameter	Mode	Description
TaskID	in	TaskType Identifier of the task to trigger on.

Description

Causes a trace trigger to occur when specified task is activated.

TaskID can be set to OS_TRIGGER_ANY, in which case activation of any task will cause the trigger to occur.

The trigger will occur when a task is activated through ActivateTask, StartOS, Alarms or ScheduleTables.

Note that ChainTask(TaskID) does not cause an activation trigger; see Os_TriggerOnChain().

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

Example

```
Os_TriggerOnActivation(InterestingTask);
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

See Also

Os_TriggerOnChain

6.35 Os_TriggerOnAdvanceCounter

Trigger when a counter is advanced.

Syntax

```
void Os_TriggerOnAdvanceCounter(
    CounterType CounterID
)
```

Parameters

Parameter	Mode	Description
CounterID	in	CounterType Identifier of the hardware counter that triggers on advance.

Description

Causes a trace trigger to occur when a specified hardware counter is advanced.

CounterID can be set to OS_TRIGGER_ANY, in which case advancing any counter will cause the trigger to occur.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

Example

```
Os_TriggerOnAdvanceCounter(HWCounter);
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

See Also

Os_TriggerOnIncrementCounter

6.36 Os_TriggerOnAlarmExpiry

Trigger when an alarm expires.

Syntax

```
void Os_TriggerOnAlarmExpiry(
    AlarmType AlarmID
)
```

Parameters

Parameter	Mode	Description
AlarmID	in	AlarmType Identifier of the alarm.

Description

Causes a trace trigger to occur when a specified alarm expires.

AlarmID can be set to OS_TRIGGER_ANY, in which case any alarm expiry or *expiry point* will cause the trigger to occur.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

Example

```
Os_TriggerOnAlarmExpiry(Alarm_10ms);
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

See Also

None.

6.37 Os_TriggerOnCat1ISRStart

Trigger when a Category 1 ISR starts.

Syntax

```
void Os_TriggerOnCat1ISRStart(
    ISRType ISRID
)
```

Parameters

Parameter	Mode	Description
ISRID	in	ISRType Identifier of the Category 1 ISR to trigger on.

Description

Causes a trace trigger to occur when a specified Category 1 ISR starts running.

ISRID can be set to OS_TRIGGER_ANY, in which case any such ISR will cause the trigger to occur.

Note that Category 1 ISRs are not controlled by RTA-OS, so you are responsible for calling Os_LogCat1ISRStart() at the beginning of your interrupt handler.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

Example

```
Os_TriggerOnCat1ISRStart(InterestingCat1ISR);
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

See Also

Os_LogCat1ISREnd
Os_TriggerOnCat1ISRStop

6.38 Os_TriggerOnCat1ISRStop

Trigger when a Category 1 ISR stops.

Syntax

```
void Os_TriggerOnCat1ISRStop(
    ISRType ISRID
)
```

Parameters

Parameter	Mode	Description
ISRID	in	ISRType Identifier of the Category 1 ISR to trigger on.

Description

Causes a trace trigger to occur when a specified Category 1 ISR stops running.

ISRID can be set to OS_TRIGGER_ANY, in which case any such ISR will cause the trigger to occur.

Note that Category 1 ISRs are not controlled by RTA-OS, so you are responsible for calling Os_LogCat1ISREnd() at the end of your interrupt handler.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

Example

```
Os_TriggerOnCat1ISRStop(InterestingCat1ISR);
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

See Also

Os_LogCat1ISREnd
Os_LogCat1ISRStart
Os_TriggerOnCat1ISRStart

6.39 Os_TriggerOnCat2ISRStart

Trigger when a Category 2 ISR starts.

Syntax

```
void Os_TriggerOnCat2ISRStart(
    ISRType ISRID
)
```

Parameters

Parameter	Mode	Description
ISRID	in	ISRType Identifier of the Category 2 ISR to trigger on.

Description

Causes a trace trigger to occur when a specified Category 2 ISR starts running.

ISRID can be set to OS_TRIGGER_ANY, in which case any such ISR will cause the trigger to occur.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

Example

```
Os_TriggerOnCat2ISRStart(InterestingCat2ISR);
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

See Also

Os_TriggerOnCat2ISRStop

6.40 Os_TriggerOnCat2ISRStop

Trigger when a Category 2 ISR stops.

Syntax

```
void Os_TriggerOnCat2ISRStop(
    ISRType ISRID
)
```

Parameters

Parameter	Mode	Description
ISRID	in	ISRType Identifier of the Category 2 ISR to trigger on.

Description

Causes a trace trigger to occur when a specified Category 2 ISR stops running.

ISRID can be set to OS_TRIGGER_ANY, in which case any such ISR will cause the trigger to occur.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

Example

```
Os_TriggerOnCat2ISRStop(InterestingCat2ISR);
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

See Also

Os_TriggerOnCat2ISRStart

6.41 Os_TriggerOnChain

Trigger when a task is chained.

Syntax

```
void Os_TriggerOnChain(
    TaskType TaskID
)
```

Parameters

Parameter	Mode	Description
TaskID	in	TaskType Identifier of the task to trigger on.

Description

Causes a trace trigger to occur when an attempt is made to chain a specified task. (Noting that chain attempts can fail.)

TaskID can be set to OS_TRIGGER_ANY, in which case chaining of any task will cause the trigger to occur.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

Example

```
Os_TriggerOnChain(InterestingTask);
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

See Also

Os_TriggerOnActivation

6.42 Os_TriggerOnError

Trigger when an error occurs.

Syntax

```
void Os_TriggerOnError(
    StatusType Error
)
```

Parameters

Parameter	Mode	Description
Error	in	StatusType Identifier of the error to trigger on.

Description

Causes a trace trigger to occur when a specified error is raised.

Error can be set to OS_TRIGGER_ANY, in which case any error will cause the trigger to occur.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

Example

```
Os_TriggerOnError(E_OS_LIMIT);
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

See Also

None.

6.43 Os_TriggerOnGetResource

Trigger when a resource is locked.

Syntax

```
void Os_TriggerOnGetResource(
    ResourceType ResourceID
)
```

Parameters

Parameter	Mode	Description
ResourceID	in	ResourceType Identifier of the resource to trigger on.

Description

Causes a trace trigger to occur when a specified resource is locked.

ResourceID can be set to OS_TRIGGER_ANY, in which case any resource lock will cause the trigger to occur.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

Example

```
Os_TriggerOnGetResource(CriticalSection);
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

See Also

Os_TriggerOnReleaseResource

6.44 Os_TriggerOnIncrementCounter

Trigger when a counter is incremented.

Syntax

```
void Os_TriggerOnIncrementCounter(
    CounterType CounterID
)
```

Parameters

Parameter	Mode	Description
CounterID	in	CounterType Identifier of the software counter.

Description

Causes a trace trigger to occur when a specified counter is incremented.

CounterID can be set to OS_TRIGGER_ANY, in which case any counter increment will cause the trigger to occur.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

Example

```
Os_TriggerOnIncrementCounter(SWCounter);
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

See Also

Os_TriggerOnAdvanceCounter

6.45 Os_TriggerOnIntervalEnd

Trigger when a trace interval ends.

Syntax

```
void Os_TriggerOnIntervalEnd(
    Os_TraceIntervalIDType IntervalID
)
```

Parameters

Parameter	Mode	Description
IntervalID	in	Os_TraceIntervalIDType Identifier of the interval to trigger on.

Description

Causes a trace trigger to occur when a specified interval ends.

IntervalID can be set to OS_TRIGGER_ANY, in which case any interval end will cause the trigger to occur.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

Example

```
Os_TriggerOnIntervalEnd(EndToEndTimeMeasurement);
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

See Also

Os_TriggerOnIntervalStart
Os_TriggerOnIntervalStop

6.46 Os_TriggerOnIntervalStart

Trigger when a trace interval is started.

Syntax

```
void Os_TriggerOnIntervalStart(
    Os_TraceIntervalIDType IntervalID
)
```

Parameters

Parameter	Mode	Description
IntervalID	in	Os_TraceIntervalIDType Identifier of the interval to trigger on.

Description

Causes a trace trigger to occur when a specified interval starts.

IntervalID can be set to OS_TRIGGER_ANY, in which case any interval start will cause the trigger to occur.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

Example

```
Os_TriggerOnIntervalStart(EndToEndTimeMeasurement);
```

See Also

Os_TriggerOnIntervalEnd
Os_TriggerOnIntervalStop

6.47 Os_TriggerOnIntervalStop

Trigger when a trace interval ends.

Syntax

```
void Os_TriggerOnIntervalStop(
    Os_TraceIntervalIDType IntervalID
)
```

Parameters

Parameter	Mode	Description
IntervalID	in	Os_TraceIntervalIDType Identifier of the interval to trigger on.

Description

This call is a synonym for Os_TriggerOnIntervalEnd.

It causes a trace trigger to occur when a specified interval ends.

IntervalID can be set to OS_TRIGGER_ANY, in which case any interval end will cause the trigger to occur.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

Example

```
Os_TriggerOnIntervalStop(EndToEndTimeMeasurement);
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

See Also

Os_TriggerOnIntervalEnd

6.48 Os_TriggerOnReleaseResource

Trigger when a resource is unlocked.

Syntax

```
void Os_TriggerOnReleaseResource(
    ResourceType ResourceID
)
```

Parameters

Parameter	Mode	Description
ResourceID	in	ResourceType Identifier of the resource to trigger on.

Description

Causes a trace trigger to occur when a specified resource is unlocked.

ResourceID can be set to OS_TRIGGER_ANY, in which case any resource unlock will cause the trigger to occur.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

Example

```
Os_TriggerOnReleaseResource(CriticalSection);
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

See Also

Os_TriggerOnGetResource

6.49 Os_TriggerOnScheduleTableExpiry

Trigger when a specified expiry point expires.

Syntax

```
void Os_TriggerOnScheduleTableExpiry(
    ExpiryID
)
```

Parameters

Parameter	Mode	Description
ExpiryID	in	Os_TraceExpiryIDType Identifier of the expiry to trigger on. The ExpiryID is formed by combining the name of the ScheduleTable and Expiry with an underscore character.

Description

Causes a trace trigger to occur when a specific expiry point is reached.

ExpiryID can be set to OS_TRIGGER_ANY, in which case any expiry *or alarm* will cause the trigger to occur.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

Example

```
StartScheduleTableRel(SchedTable, 1);
Os_TriggerOnScheduleTableExpiry(SchedTable_ep1);
IncrementCounter(SystemCounter);
...
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

See Also

None.

6.50 Os_TriggerOnSetEvent

Trigger when an event is set for a task.

Syntax

```
void Os_TriggerOnSetEvent(
    TaskType TaskID
)
```

Parameters

Parameter	Mode	Description
TaskID	in	TaskType Identifier of the task to trigger on.

Description

Causes a trace trigger to occur when an event is set for a specified task.

TaskID can be set to OS_TRIGGER_ANY, in which case any event setting will cause the trigger to occur.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

Example

```
Os_TriggerOnSetEvent(ExtendedTask);
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

See Also

None.

6.51 Os_TriggerOnShutdown

Trigger when the OS is shutdown.

Syntax

```
void Os_TriggerOnShutdown(
    StatusType Status
)
```

Parameters

Parameter	Mode	Description
Status	in	StatusType Identifier of the shutdown exit code.

Description

Causes a trace trigger to occur when a specific status is passed to ShutdownOS.

Status can be set to OS_TRIGGER_ANY, in which case status value passed to ShutdownOS will cause the trigger to occur.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

Example

```
Os_TriggerOnShutdown(E_OK); /* Trigger on normal shutdown */
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

See Also

ShutdownOS

6.52 Os_TriggerOnTaskStart

Trigger when a task is started.

Syntax

```
void Os_TriggerOnTaskStart(
    TaskType TaskID
)
```

Parameters

Parameter	Mode	Description
TaskID	in	TaskType Identifier of the task to trigger on.

Description

Causes a trace trigger to occur when a specified task starts running.

TaskID can be set to OS_TRIGGER_ANY, in which case any task start will cause the trigger to occur.

Note that a TaskID is started when its entry function is called, or when it resumes from the WAITING state.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

Example

```
Os_TriggerOnTaskStart(InterestingTask);
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

See Also

Os_TriggerOnTaskStop

6.53 Os_TriggerOnTaskStop

Trigger when a task is stopped.

Syntax

```
void Os_TriggerOnTaskStop(
    TaskType TaskID
)
```

Parameters

Parameter	Mode	Description
TaskID	in	TaskType Identifier of the task to trigger on.

Description

Causes a trace trigger to occur when a specified task stops running.

TaskID can be set to OS_TRIGGER_ANY, in which case any task stop will cause the trigger to occur.

Note that a TaskID is stopped when its entry function is called, or when it enters the WAITING state.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

Example

```
Os_TriggerOnTaskStop(InterestingTask);
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

See Also

Os_TriggerOnTaskStart

6.54 Os_TriggerOnTaskTracepoint

Trigger when a task tracepoint is logged.

Syntax

```
void Os_TriggerOnTaskTracepoint(
    Os_TraceTracepointIDType TaskTracepointID,
    TaskType TaskID
)
```

Parameters

Parameter	Mode	Description
TaskTracepointID	in	Os_TraceTracepointIDType Identifier of the tracepoint to trigger on.
TaskID	in	TaskType Identifier of the task.

Description

Causes a trace trigger to occur when a specified task-tracepoint for a specified task is logged.

TaskID can be set to OS_TRIGGER_ANY, in which any task-tracepoint with the specified value will cause the trigger to occur.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

Example

```
Os_TriggerOnTaskTracepoint(MyTaskTracepoint,InterestingTask);
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

See Also

Os_TriggerOnTracepoint

6.55 Os_TriggerOnTracepoint

Trigger when a tracepoint is logged.

Syntax

```
void Os_TriggerOnTracepoint(
    Os_TraceTracepointIDType TracepointID
)
```

Parameters

Parameter	Mode	Description
TracepointID	in	Os_TraceTracepointIDType Identifier of the tracepoint to trigger on.

Description

Causes a trace trigger to occur when a specified tracepoint is logged.

TracepointID can be set to OS_TRIGGER_ANY, in which any tracepoint will cause the trigger to occur.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

Example

```
Os_TriggerOnTracepoint(MyTracepoint);
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

See Also

Os_TriggerOnTaskTracepoint

6.56 Os_UploadTraceData

Uses asynchronous communication to upload trace data a byte at a time.

Syntax

```
void Os_UploadTraceData(void)
```

Description

This API is responsible for sending individual bytes of trace data over a serial communications link. It uses callbacks into the application code to manage access to the actual communications link.

In polled mode, it is necessary to call this function frequently enough to ensure data is transmitted in a timely manner.

As a special case in interrupt mode, this function should be called from the Os_Cbk_TraceCommDataReady() callback and the transmit-interrupt handler.

An appropriate asynchronous serial device must be available and previously initialized. A typical serial link might be set to 115200bps, 8 data bits, no parity and 1 stop bit.

NOTE: This API should only be called from trusted OS Application code. This is for reasons of efficiency.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

Example

```
/* This callback occurs when a new frame is ready for upload */
FUNC(void, {memclass}) Os_Cbk_TraceCommDataReady(void) {
  Os_UploadTraceData(); /* Causes call to Os_Cbk_TraceCommTxStart() */
}
ISR(asyncio) {
  Os_UploadTraceData();
}
FUNC(void, {memclass}) Os_Cbk_TraceCommTxStart(void) {
  /* Called from UploadTraceData when the first byte of a frame is ready
     to send.
   * It is immediately followed by a call to Os_Cbk_TraceCommTxByte().
   * In interrupt mode, this is used to enable the transmit interrupt.
   */
  enable_asyncio_interrupt();
}
FUNC(void, {memclass}) Os_Cbk_TraceCommTxByte(Os_TraceDataType val) {
  /* Called from UploadTraceData when there is a byte ready to send */
  async_transmit(val);
}
FUNC(void, {memclass}) Os_Cbk_TraceCommTxEnd(void)  {
  /* Called from UploadTraceData when the last byte of data has been sent*/
```

```
  disable_asyncio_interrupt();
}
FUNC(boolean, {memclass}) Os_Cbk_TraceCommTxReady(void) {
  /* Called from UploadTraceData to determine whether there is room in the
     transmit buffer */
  /* This should always return true in interrupt mode, because the
     interrupt should only
   * fire when there is room to send the next byte. */
  return async_tx_ready();
}
```

Calling Environment

Tasks/ISRs		AUTOSAR OS Hooks		RTA-OS Hooks	
Task	✓	PreTaskHook	✓	StackOverrunHook	✓
Category 1 ISR	✗	PostTaskHook	✓	TimeOverrunHook	✓
Category 2 ISR	✓	StartupHook	✓		
		ShutdownHook	✓		
		ErrorHook	✗		
		ProtectionHook	✓		

See Also

Os_Cbk_TraceCommDataReady
Os_Cbk_TraceCommTxByte
Os_Cbk_TraceCommTxEnd
Os_Cbk_TraceCommTxReady
Os_Cbk_TraceCommTxStart
Os_CheckTraceOutput

7 RTA-TRACE Callbacks

7.1 Guide to Descriptions

Callbacks are code that is required by RTA-TRACE but must be provided by the user. This section documents all the callbacks required for RTA-TRACE. The descriptions have the following structure:

Syntax

```
/* C function prototype for the callback */
ReturnValue NameOfCallback(Parameter Type, ...)
```

Parameters

A list of parameters for each callback and their mode:

in The parameter is passed in to the callback by the OS

out The parameter is passed out of the API callback by passing a reference (pointer) to the parameter into the call.

inout The parameter is passed into the callback and then (updated) and passed out.

Return Values

A description of the return value of the callback,

Description

A detailed description of the required functionality of the callback.

Portability

The portability of the call between OSEK OS, AUTOSAR OS, RTA-OS and RTA-TRACE.

Example Code

```
A C code listing showing how to implement the callback.
```

Configuration Condition

The configuration of RTA-TRACE that requires user code to implement the callback.

See Also

A list of related callbacks.

7.2 Os_Cbk_TraceCommDataReady

Callback routine that signals when there is trace data ready to be sent.

Syntax

```
FUNC(void, {memclass}) Os_Cbk_TraceCommDataReady(void)
```

Description

When tracing in Bursting or Triggering modes, this gets called automatically when there is a new frame of data to be uploaded to RTA-TRACE.

When tracing in Free-running mode, this gets called from Os_CheckTraceOutput(), which must be called regularly by the application.

Note: memclass is OS_APPL_CODE for AUTOSAR 3.x, OS_OS_CBK_TRACECOMMDATAREADY_CODE for AUTOSAR 4.1 and OS_CALLOUT_CODE otherwise.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

Example

```
FUNC(void, {memclass}) Os_Cbk_TraceCommDataReady(void) {
  Os_UploadTraceData(); /* Causes call to Os_Cbk_TraceCommTxStart() */
}
```

Configuration Condition

The callback may be provided if a communications link is used with tracing. A default version is present in the kernel library.

See Also

Os_UploadTraceData
Os_CheckTraceOutput
Os_Cbk_TraceCommTxStart
Os_Cbk_TraceCommTxByte
Os_Cbk_TraceCommTxEnd
Os_Cbk_TraceCommTxReady

7.3 Os_Cbk_TraceCommInitTarget

Callback routine used to allow the application to perform initialization of external communication for tracing.

Syntax

```
FUNC(Os_TraceStatusType, {memclass}) Os_Cbk_TraceCommInitTarget(void)
```

Return Values

The call returns values of type Os_TraceStatusType.

Description

Os_Cbk_TraceCommInitTarget supports the Os_TraceCommInit by providing application-specific code to initialize the communication link to RTA-TRACE. Typically it sets up an RS232 link.

E_OK should be returned if the initialization succeeded. Any other value will result in trace communication being disabled.

Note: memclass is OS_APPL_CODE for AUTOSAR 3.x, OS_OS_CBK_TRACECOMMINITTARGET_CODE for AUTOSAR 4.1 and OS_CALLOUT_CODE otherwise.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

Example

```
FUNC(Os_TraceStatusType, {memclass}) Os_Cbk_TraceCommInitTarget(void){
  initialize_uart();
  return E_OK;
}
```

Configuration Condition

The callback must be provided if Os_TraceCommInit is used to initialize tracing using an external communications link.

See Also

Os_TraceCommInit

7.4 Os_Cbk_TraceCommTxByte

Callback routine that supplies a byte of trace data for sending.

Syntax

```
FUNC(void, {memclass}) Os_Cbk_TraceCommTxByte(
    Os_TraceDataType val
)
```

Description

This is called from UploadTraceData when there is a byte of data to send.

Note: memclass is OS_APPL_CODE for AUTOSAR 3.x, OS_OS_CBK_TRACECOMMTXBYTE_CODE for AUTOSAR 4.1 and OS_CALLOUT_CODE otherwise.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

Example

```
FUNC(void, {memclass}) Os_Cbk_TraceCommTxByte(Os_TraceDataType val) {
  /* Called from UploadTraceData when there is a byte ready to send */
  async_transmit(val);
}
```

Configuration Condition

The callback must be provided if Os_UploadTraceData is used.

See Also

Os_UploadTraceData
Os_CheckTraceOutput
Os_Cbk_TraceCommDataReady
Os_Cbk_TraceCommTxStart
Os_Cbk_TraceCommTxEnd
Os_Cbk_TraceCommTxReady

7.5 Os_Cbk_TraceCommTxEnd

Callback routine that signals that the last byte of trace data has been sent.

Syntax

```
FUNC(void, {memclass}) Os_Cbk_TraceCommTxEnd(void)
```

Description

This is called from UploadTraceData when the last byte of a frame has been sent.

In interrupt mode, this is used to disable the transmit interrupt.

Note: memclass is OS_APPL_CODE for AUTOSAR 3.x, OS_OS_CBK_TRACECOMMTXEND_CODE for AUTOSAR 4.1 and OS_CALLOUT_CODE otherwise.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

Example

```
FUNC(void, {memclass}) Os_Cbk_TraceCommTxEnd(void)  {
  disable_asyncio_interrupt();
}
```

Configuration Condition

The callback must be provided if Os_UploadTraceData is used.

See Also

Os_UploadTraceData
Os_CheckTraceOutput
Os_Cbk_TraceCommDataReady
Os_Cbk_TraceCommTxStart
Os_Cbk_TraceCommTxByte
Os_Cbk_TraceCommTxReady

7.6 Os_Cbk_TraceCommTxReady

Callback routine used to discover if there is room to send the next trace data byte.

Syntax

```
FUNC(boolean, {memclass}) Os_Cbk_TraceCommTxReady(void)
```

Return Values

The call returns values of type boolean.

Description

This is called from UploadTraceData to determine whether there is room in the transmit buffer to send the next byte.

This should always return true in interrupt mode, because the interrupt should only fire when there is room to send the next byte.

Note: memclass is OS_APPL_CODE for AUTOSAR 3.x, OS_OS_CBK_TRACECOMMTXREADY_CODE for AUTOSAR 4.1 and OS_CALLOUT_CODE otherwise.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

Example

```
FUNC(boolean, {memclass}) Os_Cbk_TraceCommTxReady(void) {
  return async_tx_ready();
}
```

Configuration Condition

The callback must be provided if Os_UploadTraceData is used.

See Also

Os_UploadTraceData
Os_CheckTraceOutput
Os_Cbk_TraceCommDataReady
Os_Cbk_TraceCommTxStart
Os_Cbk_TraceCommTxByte
Os_Cbk_TraceCommTxEnd

7.7 Os_Cbk_TraceCommTxStart

Callback routine that signals that the first byte of trace data is ready to be sent.

Syntax

```
FUNC(void, {memclass}) Os_Cbk_TraceCommTxStart(void)
```

Description

This is called from UploadTraceData when the first byte of a frame is ready to send.

It is immediately followed by a call to Os_Cbk_TraceCommTxByte().

In interrupt mode, this is used to enable the transmit interrupt.

Note: memclass is OS_APPL_CODE for AUTOSAR 3.x, OS_OS_CBK_TRACECOMMTXSTART_CODE for AUTOSAR 4.1 and OS_CALLOUT_CODE otherwise.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

Example

```
FUNC(void, {memclass}) Os_Cbk_TraceCommTxStart(void) {
  enable_asyncio_interrupt();
}
```

Configuration Condition

The callback must be provided if Os_UploadTraceData is used.

See Also

Os_UploadTraceData
Os_CheckTraceOutput
Os_Cbk_TraceCommDataReady
Os_Cbk_TraceCommTxByte
Os_Cbk_TraceCommTxEnd
Os_Cbk_TraceCommTxReady

8 RTA-TRACE Types

8.1 Os_AsyncPushCallbackType

Type that represents a pointer to a void function that gets passed a single Os_TraceDataType value. Used by Os_TraceDumpAsync()

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

8.2 Os_TraceCategoriesType

Type that is used to contain mask values relating to user-defined trace filter categories. An all and a non category are defined by default.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

Values

OS_TRACE_NO_CATEGORIES
OS_TRACE_ALL_CATEGORIES

Example

```
Os_TraceCategoriesType ExtraTracing = DebugTracePoints |
    DataLogTracePoints;
```

8.3 Os_TraceClassesType

Type that is used to contain mask values relating to trace filter classes.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

Values

OS_TRACE_ACTIVATIONS_CLASS
OS_TRACE_RESOURCES_CLASS
OS_TRACE_INTERRUPT_LOCKS_CLASS
OS_TRACE_SWITCHING_OVERHEADS_CLASS
OS_TRACE_TASKS_AND_ISRS_CLASS
OS_TRACE_ERRORS_CLASS
OS_TRACE_TASK_TRACEPOINT_CLASS
OS_TRACE_TRACEPOINT_CLASS
OS_TRACE_INTERVALS_CLASS
OS_TRACE_MESSAGE_DATA_CLASS
OS_TRACE_STARTUP_AND_SHUTDOWN_CLASS
OS_TRACE_ALARMS_CLASS
OS_TRACE_SCHEDULETABLES_CLASS
OS_TRACE_OSEK_EVENTS_CLASS
OS_TRACE_EXPIRY_POINTS_CLASS
OS_TRACE_NO_CLASSES
OS_TRACE_ALL_CLASSES

Example

```
Os_TraceClassesType AllTracepoints = OS_TRACE_TRACEPOINT_CLASS |
    OS_TRACE_TASK_TRACEPOINT_CLASS;
```

8.4 Os_TraceDataLengthType

The length of a data block (in bytes).

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

Example

```
Os_TraceDataLengthType BlockLength = 8;
```

8.5 Os_TraceDataPtrType

A pointer to a block of data to log at a trace point or interval.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

Example

```
Os_TraceDataPtrType DataPtr;
Os_TraceDataType DataValues[10];
...
 DataPtr = &DataValue;
```

8.6 Os_TraceExpiryIDType

Enumerated type that defines Expiry points.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

Values

The names of expiry points. These are generated using the pattern <schedeuletable_name>_<expiry_name>.

8.7 Os_TraceIndexType

An unsigned integer value of at least 16 bits representing a number of trace records.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

Example

```
Os_TraceIndexType PreTriggerRecords = 100;
```

8.8 Os_TraceInfoType

An unsigned integer value representing a traced object.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

8.9 Os_TraceIntervalIDType

Enumerated type that defines RTA-TRACE trace intervals.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

Values

The names of user defined trace intervals.

8.10 Os_TraceStatusType

Type containing the status of a trace API call.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

Values

OS_TRACE_STATUS_OK
OS_TRACE_STATUS_COMM_INIT_FAILURE

8.11 Os_TraceTracepointIDType

Enumerated type that defines RTA-TRACE tracepoints.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

Values

The names of user defined trace points.

8.12 Os_TraceValueType

An unsigned integer value representing either 16 or 32 bits depending on the configuration of compact time.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

9 RTA-TRACE Macros

9.1 OS_NUM_INTERVALS

The number of Trace Intervals declared.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

9.2 OS_NUM_TASKTRACEPOINTS

The number of TaskTracepoints declared.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

9.3 OS_NUM_TRACECATEGORIES

The number of Trace Categories declared.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

9.4 OS_NUM_TRACEPOINTS

The number of Tracepoints declared.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

9.5 OS_TRACE

This macro is only defined if tracing is enabled.

Portability

RTA-OS	OSEK	R3.x	R4.x	MultiCore	RTA-TRACE
✓	✗	✗	✗	✗	✓

Example

```
#ifdef OS_TRACE
  ...
#endif
```

10 Coding Conventions

10.1 Namespace

The C programming language provides a single namespace for all global names. This means that different names must be used for each function, variable, constant and type even if the names are declared in different compilation units. The AUTOSAR standard defines a naming convention for every basic software module to avoid problems with namespace clashes. This is defined by the “AUTOSAR General Requirements on Basic Software Modules”. RTA-OS has been implemented to satisfy these requirements. The namespace used by RTA-OS therefore reserves all names that are prefixed by:

- `Os*`
- `os*`

Note however, that the API interface provided by AUTOSAR OS does not comply with the AUTOSAR naming convention for historical reasons. This means that the names used by AUTOSAR OS for types, API calls, macros, constants, callbacks etc. are also reserved names and should not be duplicated in user code

Integration Guidance 10.1:*RTA-OS defines OS API calls and macros internally according to the AUTOSAR general requirements and provides the AUTOSAR OS names to the user through C macros. This does not apply to standard callbacks which retain their standard name, for example `ErrorHook()`, `ShutdownHook()` etc.*

This means the following forms are identical:

```
Os_StatusType Os_ActivateTask(Os_TaskType, Os_TaskId)

StatusType ActivateTask(TaskType, TaskId)
```

The two forms can be used interchangeably in user code if required, but only the second form represents standard AUTOSAR OS API.

11 Configuration Language

11.1 Configuration Files

RTA-OS is configured using AUTOSAR's ECU Parameter description language. This section gives a short overview of the AUTOSAR R4.x basic software module configuration in AUTOSAR XML and the extensions made by ETAS to the description language.

11.2 Understanding AUTOSAR XML Configuration

AUTOSAR uses eXtensible Markup Language (XML) as its configuration file format. AUTOSAR defines the tags and their semantics using an XML schema definition.

Every AUTOSAR XML file needs to reference the AUTOSAR schema instance that defines the structure of the XML elements for AUTOSAR XML files. In the simple case this is done as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<AUTOSAR xmlns="http://autosar.org/schema/r4.0" xsi:schemaLocation="
    http://autosar.org/schema/r4.0 autosar_4-0-1.xsd">
  ...
</AUTOSAR>
```

All elements between <AUTOSAR> and </AUTOSAR> have the form <**ELEMENT**-NAME>. Only one AUTOSAR element is allowed per XML configuration file. All other AUTOSAR definitions are contained within this element.

11.2.1 Packages

The <AUTOSAR> element is a container for exactly one <AR-PACKAGES> element. The <AR-PACKAGES> element represents the root of an XML object tree from which all objects in all configuration files can be accessed. The <AR-PACKAGES> itself then contains one or more packages each defined with the <AR-PACKAGE> element. Each <AR-PACKAGE> defines a group of AUTOSAR elements or a set of sub-packages related to some part of AUTOSAR configuration.

Each <AR-PACKAGE> package definition is named using the <SHORT-NAME> element. Each package should have a unique name so that the elements contained within the package can be referenced by other packages. If two packages share the same name then they are assumed to be parts of the same package.

```
<AUTOSAR xmlns="http://autosar.org/schema/r4.0" xsi:schemaLocation="
    http://autosar.org/schema/r4.0 autosar_4-0-1.xsd">
  <AR-PACKAGES>
    <AR-PACKAGE>
      <SHORT-NAME>MyPackage</SHORT-NAME>
      <DESC>This is one of my packages</DESC>
    </AR-PACKAGE>
    ...
    <AR-PACKAGE>
```

```
      <SHORT-NAME>MyOtherPackage</SHORT-NAME>
      <DESC>This is another</DESC>
    </AR-PACKAGE>
  </AR-PACKAGES>
</AUTOSAR>
```

The <AR-PACKAGE> element is used to define the package name as well as acting as a container for other elements.

Non basic software configuration can only be split at the <AR-PACKAGES> level. When you need to work with multiple XML files you must therefore split them at the <AR-PACKAGES> level. In the previous example, we might have decided to split this file into two different files, in which case in File 1 we would have:

```
<?xml version="1.0" encoding="UTF-8"?>
<AUTOSAR xmlns="http://autosar.org/schema/r4.0" xsi:schemaLocation="
    http://autosar.org/schema/r4.0 autosar_4-0-1.xsd">
  <AR-PACKAGES>
    <AR-PACKAGE>
      <SHORT-NAME>SWCs</SHORT-NAME>
      <DESC>This is one of my packages</DESC>
      ...
    </AR-PACKAGE>
  </AR-PACKAGES>
</AUTOSAR>
```

In File 2 we would have the second AR-PACKAGE:

```
<?xml version="1.0" encoding="UTF-8"?>
<AUTOSAR xmlns="http://autosar.org/schema/r4.0" xsi:schemaLocation="
    http://autosar.org/schema/r4.0 autosar_4-0-1.xsd">
  <AR-PACKAGES>
    <AR-PACKAGE>
      <SHORT-NAME>Interfaces</SHORT-NAME>
      <DESC>This is another</DESC>
      ...
    </AR-PACKAGE>
  </AR-PACKAGES>
</AUTOSAR>
```

11.3 ECU Configuration Description

AUTOSAR basic software uses a different configuration concept to the rest of AUTOSAR. Configuration uses an ECU configuration description file. This file is also an XML file, but the use of XML is significantly different to the rest of AUTOSAR configuration.

Rather than define a dedicated set of XML tags for the configuration of each basic software module, the ECU configuration description defines a module configuration that contains CONTAINERS that hold configuration data in a <ECUC-CONTAINER-VALUE>.

Each <ECUC-CONTAINER-VALUE> holds <PARAMETER-VALUES>, <REFERENCE-VALUES> and <SUB-CONTAINERS>. <SUB-CONTAINERS> hold <ECUC-CONTAINER-VALUE> definitions, allowing a hierarchy of configuration containers to be formed.

This structure is common to all AUTOSAR basic software modules. The same format is used for the OS as for COM, NM, etc. The structure is customized for different basic software modules using a <DEFINITION-REF>. Each module configuration and <ECUC-CONTAINER-VALUE> has a <DEFINITION-REF> which references the AUTOSAR ECU Configuration Definition. The <DEFINITION-REF> is an absolute reference to the definition of a configuration item in the AUTOSAR ECU Configuration Definition. This is also an XML file and defines the type of the container and what configuration elements are allowed.

By default, references are rooted at /AUTOSAR. For the OS there are things like:

- /AUTOSAR/EcuDefs/Os/OsTask
- /AUTOSAR/EcuDefs/Os/OsTask/OsTaskPriority
- /AUTOSAR/EcuDefs/Os/OsResource
- /AUTOSAR/EcuDefs/Os/OsIsr

Each definition in the definition file specifies:

- how many instance of the <ECUC-CONTAINER-VALUE> can exist in the module configuration
- how many of each of the <PARAMETER-VALUES>, <REFERENCE-VALUES> and <SUB-CONTAINERS> the container can hold. This is called the *multiplicity* and the definition file specifies a <LOWER-MULTIPLICITY> and an <UPPER-MULTIPLICITY>.
- the definitions of the <PARAMETER-VALUES>, <REFERENCE-VALUES> and <SUB-CONTAINERS> the <ECUC-CONTAINER-VALUE> can hold

The description files used to configure AUTOSAR OS is written according to the rules specified in the definition file. The following example shows a valid description file for the OS that includes a single task called MyTask:

```
<ELEMENTS>
<ECUC-MODULE-CONFIGURATION-VALUES>
  <SHORT-NAME>OsRTA</SHORT-NAME>
  <DEFINITION-REF DEST="ECUC-MODULE-DEF">/AUTOSAR/EcucDefs/Os</
     DEFINITION-REF>
```

```
  <CONTAINERS>
    <ECUC-CONTAINER-VALUE>
      <SHORT-NAME>MyTask</SHORT-NAME>
      <DEFINITION-REF DEST="ECUC-PARAM-CONF-CONTAINER-DEF">/AUTOSAR/
          EcucDefs/Os/OsTask</DEFINITION-REF>
      <PARAMETER-VALUES>
        <ECUC-NUMERICAL-PARAM-VALUE>
          <DEFINITION-REF DEST="ECUC-INTEGER-PARAM-DEF">/AUTOSAR/
              EcucDefs/Os/OsTask/OsTaskPriority</DEFINITION-REF>
          <VALUE>27</VALUE>
        </ECUC-NUMERICAL-PARAM-VALUE>
        <ECUC-TEXTUAL-PARAM-VALUE>
          <DEFINITION-REF DEST="ECUC-ENUMERATION-PARAM-DEF">/AUTOSAR/
              EcucDefs/Os/OsTask/OsTaskSchedule</DEFINITION-REF>
          <VALUE>FULL</VALUE>
        </ECUC-TEXTUAL-PARAM-VALUE>
        <ECUC-NUMERICAL-PARAM-VALUE>
          <DEFINITION-REF DEST="ECUC-INTEGER-PARAM-DEF">/AUTOSAR/
              EcucDefs/Os/OsTask/OsTaskActivation</DEFINITION-REF>
          <VALUE>1</VALUE>
        </ECUC-NUMERICAL-PARAM-VALUE>
      </PARAMETER-VALUES>
      <REFERENCE-VALUES />
      <SUB-CONTAINERS />
    </ECUC-CONTAINER-VALUE>
  </CONTAINERS>
</ECUC-MODULE-CONFIGURATION-VALUES>
</ELEMENTS>
```

Standard AUTOSAR configuration elements for the OS are documented in the *AUTOSAR Specification of Operating System*.

11.4 RTA-OS Configuration Language Extensions

In addition to the standard AUTOSAR configuration elements, each AUTOSAR OS vendor will also define their own pieces of ECU configuration to capture things that are not standardized in AUTOSAR - for example the allocation of vector addresses and priorities to interrupts.

Vendor extensions to AUTOSAR configuration take the standard AUTOSAR Standard Module Definition (called the StMD) and produce a Vendor Specific Module Definition (VSMD). This includes all the elements from AUTOSAR plus those defined by the vendor. More information about this process can be found in *AUTOSAR Specification of ECU Configuration*.

The following sections define the extensions to the standard AUTOSAR configuration attributes that are supported by RTA-OS. Each section defines (or extends) a

<ECUC-CONTAINER-VALUE> and the <PARAMETER-VALUES>, <REFERENCE-VALUES> and <SUB-CONTAINERS> that the <ECUC-CONTAINER-VALUE> can hold.

Portability Note 11.1:*The presence of vendor specific extensions to AUTOSAR is portable to 3rd party AUTOSAR configuration tooling. However, this applies only to the syntax of extensions. The semantics of extensions is, of course, not portable. For example, if one vendor defines a configuration element called `OsEnableSpecialOptimization` then another vendor will not be able to do anything with this configuration because their implementation cannot know the meaning of a "special optimization".*

11.4.1 Container: OsAppMode

Integer Parameters

Name	Occurs	Description
OsAppModeId	1..1	Internal ID of an AppMode. Necessary in 3.1 only, so that an AppMode is addressable by other modules. **Range:** ..maxint

11.4.2 Container: OsRTATarget

Description

Parameters to represent a specific piece of target hardware.

Multiplicity

0..1

String Parameters

Name	Occurs	Description
OsRTATargetName	1..1	The name of the target system.
OsRTATargetVersion	0..1	The version number of the OS on the target system.
OsRTATargetVariant	0..1	The variant of the OS for this target system.

Sub-container: Param

Description

Target-specific parameter representation.

Multiplicity

0..*

String Parameters

Name	Occurs	Description
Value	1..1	Value of the parameter

11.4.3 Container: OsCounter

String Parameters

Name	Occurs	Description
OsFormat	0..1	A string that specifies a format for each tracepoint.

11.4.4 Container: OsIsr

Enumeration Parameters

Name	Occurs	Description
OsTraceFilter	0..1	Describes whether this ISR is traced with RTA-TRACE. Permitted values are: **ALWAYS** Always trace this ISR **NEVER** Never trace this ISR **RUNTIME** Allow the user to control tracing of this ISR at runtime.

Integer Parameters

Name	Occurs	Description
OsIsrPriority	1..1	The Interrupt Priority **Range:** 0..maxint

String Parameters

Name	Occurs	Description
OsIsrBudget	0..1	Execution budget expressed as a float, then time-base name, then units.
OsIsrStackAllocation	0..1	ISR manual stack allocation in bytes
OsIsrAddress	1..1	The Interrupt Vector

Reference Parameters

Name	Occurs	Destination
OsRegSetRef	0..*	/AUTOSAR/Os/OsRegSet

11.4.5 Container: OsOS

Boolean Parameters

Name	Occurs	Description
OsSuppressVectorGen	0..1	Suppresses generation of the vector table.

Integer Parameters

Name	Occurs	Description
OsCyclesPerSecond	0..1	Defines the clock speed of the target **Range:** 0..maxint
OsTicksPerSecond	0..1	Defines the stopwatch speed of the target **Range:** 0..maxint

String Parameters

Name	Occurs	Description
OsDefTaskStack	0..1	Default stack values
OsDefCat1Stack	0..1	Default category 1 stack values
OsDefCat2Stack	0..1	Default category 2 stack values

Sub-container: Param

Description

Representation of parameters

Multiplicity

0..*

String Parameters

Name	Occurs	Description
Value	1..1	Value of the parameter

Sub-container: OsHooks

Boolean Parameters

Name	Occurs	Description
OsStackFaultHook	0..1	Use stack fault hook

11.4.6 Container: OsRegSet

Description

Target specific register sets that can be associated with a task or ISR. By association with a task or ISR, the integrator is specifying that a specific task or ISR uses this register set. Having no association defined allows potential optimization.

Multiplicity

0..*

11.4.7 Container: OsSpinlock

Enumeration Parameters

Name	Occurs	Description
OsSpinlockLockMethod	0..1	Determines the locking method for the Spinlock. Permitted values are: **LOCK_ALL_INTERRUPTS** Lock all interrupts when spinlock is taken. **LOCK_CAT2_INTERRUPTS** Lock OS interrupts when spinlock is taken. **LOCK_NOTHING** No lock when spinlock is taken. **LOCK_WITH_RES_SCHEDULER** Lock RES_SCHEDULER when spinlock is taken. **NESTABLE_LOCK_ALL_INTERRUPTS** Lock all interrupts when spinlock is taken. Allow nesting. **NESTABLE_LOCK_CAT2_INTERRUPTS** Lock OS interrupts when spinlock is taken. Allow nesting. **NESTABLE_LOCK_NOTHING** No lock when spinlock is taken. Allow nesting. **NESTABLE_LOCK_WITH_RES_SCHEDULER** Lock RES_SCHEDULER when spinlock is taken. Allow nesting. **COMMONABLE_LOCK_ALL_INTERRUPTS** Lock all interrupts when spinlock is taken. Lock is commonable. **COMMONABLE_LOCK_CAT2_INTERRUPTS** Lock OS interrupts when spinlock is taken. Lock is commonable. **COMMONABLE_LOCK_NOTHING** No lock when spinlock is taken. Lock is commonable. **COMMONABLE_LOCK_WITH_RES_SCHEDULER** Lock RES_SCHEDULER when spinlock is taken. Lock is commonable. **NESTABLE_COMMONABLE_LOCK_ALL_INTERRUPTS** Lock all interrupts when spinlock is taken. Allow nesting. Lock is commonable. **NESTABLE_COMMONABLE_LOCK_CAT2_INTERRUPTS** Lock OS interrupts when spinlock is taken.

11.4.8 Container: OsTask

Enumeration Parameters

Name	Occurs	Description
OsTraceFilter	0..1	Describes whether this Task is traced with RTA-TRACE. Permitted values are: **ALWAYS** Always trace this ISR **NEVER** Never trace this ISR **RUNTIME** Allow the user to control tracing of this ISR at runtime.

String Parameters

Name	Occurs	Description
OsTaskStackAllocation	0..1	Task manual stack allocation
OsTaskWaitStack	0..1	Task stack usage when invoking WaitEvent
OsTaskBudget	0..1	Execution budget expressed as a float, then timebase name, then units.

Reference Parameters

Name	Occurs	Destination
OsRegSetRef	0..*	/AUTOSAR/Os/OsRegSet

11.4.9 Container: OsTrace

Description

RTA-TRACE Data

Multiplicity

0..1

Boolean Parameters

Name	Occurs	Description
OsTraceEnabled	0..1	Enables or disables tracing.
OsTraceCompactID	0..1	Trace Compact Identifiers
OsTraceCompactTime	1..1	Use compact time format
OsTraceTgtStack	1..1	Enable stack recording.
OsTraceTgtTrigger	1..1	Runtime target triggering.
OsTraceAutoComms	1..1	Initialise trace comms link at startup
OsTraceAutoRepeat	1..1	Call set trace repeat at startup

Enumeration Parameters

Name	Occurs	Description
OsTraceAuto	1..1	The autostart type for RTA-TRACE Permitted values are: **NONE** Don't automatically start tracing **BURSTING** Start tracing in bursting mode (wait till buffer is full before uploading) **TRIGGERING** Start tracing, waiting for a trigger **FREE_RUNNING** Start tracing continuously

Integer Parameters

Name	Occurs	Description
OsTraceBufferSize	1..1	The trace buffer size (in number of trace records) **Range:** 0..maxint

Sub-container: OsEnumeration

Description

Specifies an enumeration for tracing.

Multiplicity

0..*

Sub-container: OsEnumeration/Param

Description

Representation of name-value pairs

Multiplicity

0..*

String Parameters

Name	Occurs	Description
Value	1..1	Value of the parameter

Sub-container: OsTraceTracepoint

Description

Specifies a tracepoint

Multiplicity

0..*

Integer Parameters

Name	Occurs	Description
OsTraceTracepointID	1..1	Specifies a tracepoint ID (1-n, 0 indicates auto) **Range:** 0..maxint

String Parameters

Name	Occurs	Description
OsTraceTracepointFormat	0..1	A string that specifies a format for each tracepoint.

Sub-container: OsTraceTaskTracepoint

Description

Specifies a task tracepoint

Multiplicity

0..*

Integer Parameters

Name	Occurs	Description
OsTraceTaskTracepointID	1..1	Specifies a tracepoint ID (1-n, 0 indicates auto) **Range:** 0..maxint

String Parameters

Name	Occurs	Description
OsTraceTaskTracepointFormat	0..1	A string that specifies a format for each tracepoint.

Reference Parameters

Name	Occurs	Destination
OsTaskRef	0..1	/AUTOSAR/Os/OsTask
OsIsrRef	0..1	/AUTOSAR/Os/OsIsr

Sub-container: OsInterval

Description

Specifies a named interval.

Multiplicity

0..*

Integer Parameters

Name	Occurs	Description
OsIntervalID	1..1	Specifies a interval identifier (1-n, 0 indicates auto) **Range:** 0..maxint

String Parameters

Name	Occurs	Description
OsIntervalFormat	0..1	A string that specifies a format for each interval

Sub-container: Param

Description

Representation of name-value pairs

Multiplicity

0..*

String Parameters

Name	Occurs	Description
Value	1..1	Value of the parameter

Sub-container: OsClass

Description

Specifies an unnamed trace class.

Multiplicity

0..*

Boolean Parameters

Name	Occurs	Description
OsClassAutostart	0..1	For a run-time trace class, this determines whether it is started automatically at runtime.

Enumeration Parameters

Name	Occurs	Description
OsClassFilter	1..1	Specifies the filtering for a class. Permitted values are: **ALWAYS** Always trace this class **NEVER** Never trace this class **RUNTIME** Allow the user to control tracing of this class at runtime.

Sub-container: OsCategory

Description

Specifies a named trace class.

Multiplicity

0..*

Boolean Parameters

Name	Occurs	Description
OsCategoryAutostart	0..1	For a run-time trace category, this determines whether it is started automatically at runtime.

Enumeration Parameters

Name	Occurs	Description
OsCategoryFilter	1..1	Specifies the filtering for a category. Permitted values are: **ALWAYS** Always trace this category **NEVER** Never trace this category **RUNTIME** Allow the user to control tracing of this category at runtime.

Integer Parameters

Name	Occurs	Description
OsCategoryMask	1..1	Specifies a category mask. 0 represents auto. **Range:** 0..maxint

11.5 Project Description Files

A single logical OS configuration can be split across multiple XML configuration files. The files can be edited individually or simultaneously by the **rtaoscfg** configuration tool.

To help with the management of large, complex, configurations, RTA-OS provides a convenient shorthand for you to group a set of multiple files that represent a single logical OS configuration. This is called a “project”. The files that comprise the project are referenced from a project file.

Portability Note 11.2:*Project files are specific to the RTA-OS tools and may not be portable to third party AUTOSAR tooling.*

A project file is an XML file that has the following structure:

```
    file ::=  <?xml version="1.0"?>
              <RTAOS_Project version="1.0">
              [<Working name="filename"/>]
              {<File name="filename"/>}
              [options]
              </RTAOS_Project>
 options ::=  <Options>
              {<Option name="filename">value</Option>}
              </Options>
   value ::=  booleanvalue | stringvalue |integervalue
```

12 Command Line

The tools shipped with RTA-OS can be invoked from the command line, making them easy to integrate into a build process. All commands accept any number of XML input files together with tool-specific options as parameters. The ordering of command line parameters is largely unimportant: options and XML files can be mixed freely.

Some command line options can be specified using either short or long (POSIX style) names. The two options forms provide identical functionality and can be used interchangeably.

When a command line option takes an argument, the argument appears immediately following short name options and after a colon following long name options. For example, an option with argument arg could appear as either

`command -oarg` or `command --option:arg`

The two forms are equivalent and can be mixed on the command line.

Optional settings for arguments are placed in brackets immediately before the argument itself. For example, assuming argument arg had a setting s, it would appear as either:

`command -o[s]arg` or `command --option:[s]arg`

12.1 rtaoscfg

The command **rtaoscfg** runs the graphical RTA-OS configuration editor.

```
rtaoscfg [options] <files>
```

12.1.1 Options

Option	Description
@<FILE>	Read command line parameters from <FILE>. Each command in <FILE> must appear on a separate line. Quotation marks are not required to escape white space for filenames inside a command file. The @<FILE> option can itself appear multiple times inside <FILE>.

Option	Description
`--diagnostic`	Display the diagnostic information on the standard output. Diagnostic information includes: • The version of the tool executable • The names and versions of all tool plug-ins • The names and version of all target plug-ins • The location and contents of the license file
`--env:<VALUE>`	Adds <VALUE> to the front of the PATH environment variable seen by the tool or any program that it invokes. This is typically used to specify the path to your compiler if it is not already on the path.
`--env:[<NAME>]<VALUE>`	Sets environment variable <NAME> to value <VALUE> for the tool or any program that it invokes. This can be used to specify compiler-related information such as library paths if not already in your environment.
`-h, -?, --help`	Display usage information on the standard output.
`--nomsgbox`	Do not prompt the user with a message box when an error causes the configuration tool to exit.
`--os_option:<NAME=VALUE>`	Override OS option <NAME> with <VALUE>. A list of options is obtained using `--os_option:?`.
`-o[<EXPS>]<DIR>` `--output:[<EXPS>]<DIR>`	Place all generated output files into the directory <DIR>. The optional <EXPS> clause places all generated files whose names match the comma-separated lost of expressions in <EXPS> in the directory <DIR>. Expressions can include the following wildcards: **?** matches a single character * matches a sequence of 1 or more characters

Option	Description
`--status:<STATUS>`	Generate a kernel library for the specified <STATUS> level. <STATUS> has two valid options: 1. STANDARD 2. EXTENDED If the OsStatus value is set in the input configuration then this option overrides the setting.
`--target:[<VARIANT>]<TARGET>`	Generate a kernel library for the specified <TARGET>. If multiple versions of <TARGET> are installed then the most recent version of the <TARGET> is selected. Selection of a specific version is possible using <TARGET>_<VERSION>. The option <VARIANT> selects a variant of <TARGET>. Both <TARGET> and <VARIANT> override the OsTarget and OsTargetVariant settings in the configuration file. A list of available targets and their associated versions and variants can be generated using `--target:?`
`--target_option:<NAME=VALUE>`	Override target option <NAME> with <VALUE>. A list of options is obtained using `--target_option:?`.
`--target_include:<PATH>`	Add the directory <PATH> to the locations which are searched for target DLLs. e.g. `--target_include:..\MyTargets`
`--trace:<OPTION>`	Enable or disable RTA-TRACE. <OPTION> may be one of: **on** enables RTA-TRACE (equivalent to setting `OsTraceEnabled` to `true`) **off** disables RTA-TRACE (equivalent to setting `OsTraceEnabled` to `false`
`--xml:<OPTION>`	Control the behavior of the XML processor when reading <files>. <OPTION> can be one of: **Novalidate** do not validate the input against the XML schema.

Option	Description
`--xmlschema:<SCHEMA>`	If validating the XML against a schema (`--xml:novalidate` is not set) then use the <SCHEMA> for the validation.

12.1.2 Generated Files

rtaoscfg does not generate any files directly. When the Builder is used in rtaoscfg this calls **rtaosgen**. See Section 12.2.2 for details of the files generated by **rtaosgen**.

12.1.3 Examples

Open a single file Config.arxml for editing:

```
rtaoscfg Config.arxml
```

Open an RTA-OS project file for editing:

```
rtaoscfg MyProject.rtaos
```

12.2 rtaosgen

The command **rtaosgen** runs the RTA-OS kernel library generator.

```
rtaosgen [options] <files>
```

12.2.1 Options

Option	Description
`@<FILE>`	Read command line parameters from <FILE>. Each command in <FILE> must appear on a separate line. Quotation marks are not required to escape white space for filenames inside a command file. The @<FILE> option can itself appear multiple times inside <FILE>.
`--ar_version:x.y.z`	Specify the AUTOSAR version to use, regardless of any version present in the ARXML configuration. e.g. `--ar_version:4.5.0`.

Option	Description
`--build:<OPTION>`	Pass `<OPTION>` to the build environment. `<OPTION>` may be one of: **verbose** display all build messages on the standard output **quiet** display no build messages on the standard output **clean** clean the build directory before building
`--debug:<OPTION>`	Option used to help debug build issues. `<OPTION>` may be one of: **assembler** Emits assembler listing file **source** Emits source code[1] **compact_source** As above, but compacted source **build_info** Emits `<projectname>_build_info.bat` containing information about how the OS builds the source code.
`--diagnostic`	Display the diagnostic information on the standard output. Diagnostic information includes: • The version of the tool executable • The names and versions of all tool plug-ins • The names and version of all target plug-ins • The location and contents of the license file
`--env:<VALUE>`	Adds `<VALUE>` to the front of the PATH environment variable seen by the tool or any program that it invokes. This is typically used to specify the path to your compiler if it is not already on the path.

[1] Keeping source code is only possible with a valid source code license

Option	Description
`--env:[<NAME>]<VALUE>`	Sets environment variable <NAME> to value <VALUE> for the tool or any program that it invokes. This can be used to specify compiler-related information such as library paths if not already in your environment.
`-h, -?, --help`	Display usage information on the standard output.
`-I<PATH>` `--include:<PATH>`	Add the directory in the value <PATH> to the include path for the builder.
`--ioc:stub`	In normal operation, **rtaosgen** will place any IOC-related code in the RTAOS library file. However there are use-cases where it is desired to generate the IOC code at a later stage in the build process, without modifying the core OS library. The `--ioc:stub` option tells **rtaosgen** to place code stubs (hooks) in the RTAOS library instead of the IOC code. The **rtaosgen** option `--ioc:impl` is used later to generate a RTAIOC library that contains only the IOC code and the necessary code to satisfy these stubs. It is necessary to reference both of these libraries at link time in order to be able to build the ECU. The configuration data fed to **rtaosgen** does not need to contain any IOC data (it will be ignored if present). The configuration data fed to **rtaosgen** when using option `--ioc:impl` must be the IOC data plus exactly the same non-IOC data that was fed to the original call of **rtaosgen**. If this is not done then the IOC implementation could fail in unexpected ways, because code optimizations rely on knowledge of the OS configuration.
`--ioc:impl`	This option is the counterpart of `--ioc:stub`, and is used to tell **rtaosgen** to generate the IOC implementation to match the IOC stubs.
`--ioc:std`	This option tells **rtaosgen** not to use the stub/impl mechanism for the IOC code, and to place the IOC code in the RTAOS library. This is the default option for **rtaosgen**.

Option	Description
`--mcsd:<FILE>`	This option is used to tell **rtaosgen** the name of the MCSD file created by RTA-RTE when it is generating a system that requires measurement of signals that are implemented in the IOC. **rtaosgen** will detect extra information passed from RTA-RTE and then update the content of the MCSD file so that it references the correct symbol in the IOC implementation.
`--nobuild`	Perform checks on the input configuration but does not build an RTA-OS library. If you have a source code license then `--debug:source` will still cause the source files to be created, regardless of whether you have a target compiler available.
`--noinfo`	Suppress all information messages.
`--nowarnings`	Suppress all warning messages.
`-o[<EXPS>]<DIR>` `--output:[<EXPS>]<DIR>`	Place all generated output files into the directory <DIR>. The optional <EXPS> clause places all generated files whose names match the comma-separated list of expressions in <EXPS> in the directory <DIR>. Expressions can include the following wildcards: **?** matches a single character * matches a sequence of 1 or more characters
`--os_option:<NAME=VALUE>`	Override OS option <NAME> with <VALUE>. A list of options is obtained using `--os_option:?`.
`--report:<REPORT>`	Generate the REPORT. A list of available reports is displayed on the standard output using `--report:?`
`--samples:[<SAMPLE>]<OPTION>`	Generate example code for <SAMPLE>. Use `--samples:[<SAMPLE>]overwrite` to write over existing samples. Use `--samples:?` to view available samples.

Option	Description
`--status:<STATUS>`	Generate a kernel library for the specified <STATUS> level. <STATUS> has two valid options: 1. STANDARD 2. EXTENDED If the OsStatus value is set in the input configuration then this option overrides the setting.
`--target:[<VARIANT>]<TARGET>`	Generate a kernel library for the specified <TARGET>. If multiple versions of <TARGET> are installed then the most recent version of the <TARGET> is selected. Selection of a specific version is possible using <TARGET>_<VERSION>. The option <VARIANT> selects a variant of <TARGET>. Both <TARGET> and <VARIANT> override the OsTarget and OsTargetVariant settings in the configuration file. A list of available targets and their associated versions and variants can be generated using `--target:?`
`--target_include:<PATH>`	Add the directory <PATH> to the locations which are searched for target DLLs. e.g. `--target_include:..\MyTargets`
`--target_option:<NAME=VALUE>`	Override target option <NAME> with <VALUE>. A list of options is obtained using `--target_option:?`.
`--trace:<OPTION>`	Enable or disable RTA-TRACE. <OPTION> may be one of: **on** enables RTA-TRACE (equivalent to setting `OsTraceEnabled` to `true`) **off** disables RTA-TRACE (equivalent to setting `OsTraceEnabled` to `false`
`--using:<FILES>`	**`#include`** each file in the comma-separated list <FILES> at the start of each library source file.
`--verbose`	Generate additional information when running.
`--version`	Show version information in compact form. More detailed information can be obtained using `--diagnostic`.

Option	Description
`--xml:<OPTION>`	Control the behavior of the XML processor when reading `<files>`. `<OPTION>` can be one of: **Novalidate** do not validate the input against the XML schema.
`--xmlschema:<SCHEMA>`	If validating the XML against a schema (`--xml:novalidate` is not set) then use the `<SCHEMA>` for the validation.

12.2.2 Generated Files

When **rtaosgen** runs and terminates without generating any errors or fatal messages then it will have generated the following files:

Filename	Contents
`Os.h`	The main include file for the OS.
`Os_Cfg.h`	Declarations of the objects you have configured. This is included by `Os.h`.
`Os_MemMap.h`	AUTOSAR memory mapping configuration used by RTA-OS to merge with the system-wide `MemMap.h` file in AUTOSAR versions 4.0 and earlier. From AUTOSAR version 4.1, `Os_MemMap.h` is used by the OS instead of `MemMap.h`.
`RTAOS.<lib>`	The RTA-OS library for your application. The extension `<lib>` depends on your target.
`RTAOS.<lib>.sig`	A signature file for the library for your application. This is used by **rtaosgen** to work out which parts of the kernel library need to be rebuilt if the configuration has changed. The extension `<lib>` depends on your target.
`<projectname>.log`	A log file that contains a copy of the text that the tool and compiler sent to the screen during the build process.

There may be other files which are generated that are specific to your port. A list of additional files that can be generated can be found in the *Target/Compiler Port Guide* for your port.

12.2.3 Examples

Display the usage information

```
rtaosgen --help
```

Generate the OS described by `Config.arxml` and generate sample AUTOSAR header files that will work with the OS. Create the library including both these generated files and the OS-specific generated files that are placed in the current directory. This is the

standard command line you use when you will *not* be integrating RTA-OS with 3rd party AUTOSAR software:

```
rtaosgen --samples:[Includes] --include:Samples\Includes Config.arxml
```

Generate the OS described in BigConfig.rtaos using the AUTOSAR header files located at `PathToAutosarHeaderFiles` and the OS-specific header files that will be generated in the current directory. This is the standard command line you use when integrating RTA-OS with 3rd party AUTOSAR software:

```
rtaosgen --include:PathToAutosarHeaderFiles BigConfig.rtaos
```

List which sample files can be generated for the ManchesterMk1 target:

```
rtaosgen --target:ManchesterMk1 --samples:?
```

List which reports can be generated for the ManchesterMk1 target:

```
rtaosgen --target:ManchesterMk1 --report:?
```

Generate the OS as in the first example, but overwrite the existing sample includes files and override the target to be ManchesterMk1:

```
rtaosgen --samples:[Includes]overwrite --include:Samples\Includes
    --target:ManchesterMk1 Config.arxml
```

Generate the OS from a description split between `CoreConfig.arxml` and `TargetConfig.arxml`:

```
rtaosgen --include:PathToAutosarHeaderFiles CoreConfig.arxml
    TargetConfig.arxml
```

Generate the OS described in `Config.arxml`, place the header files in `C:\working\OS\inc` and the library (plus the associated signature file) in `C:\working\OS\lib`

```
rtaosgen --include:PathToAutosarHeaderFiles
    --output:[*.h]C:\working\OS\inc
    --output:[*.lib,*.sig]C:\working\OS\inc Config.arxml
```

12.3 Target Options

Target options can be set using the `--target_option` command-line option and can also be saved in the AUTOSAR XML configuration.

12.3.1 Stack used for C-startup

XML name SpPreStartOS

Description

The amount of stack already in use at the point that StartOS() is called. This value is simply added to the total stack size that the OS needs to support all tasks and interrupts at run-time. Typically you use this to obtain the amount of stack that the linker must allocate. The value does not normally change if the OS configuration changes.

12.3.2 Stack used when idle

XML name SpStartOS

Description

The amount of stack used when the OS is in the idle state (typically inside Os_Cbk_Idle()). This is just the difference between the stack used at the point that Os_StartOS() is called and the stack used when no task or interrupt is running. This can be zero if Os_Cbk_Idle() is not used. It must include the stack used by any function called while in the idle state. The value does not normally change if the OS configuration changes.

12.3.3 Stack overheads for ISR activation

XML name SpIDisp

Description

The extra amount of stack needed to activate a task from within an ISR. If a task is activated within a Category 2 ISR, and that task has a higher priority than any currently running task, then for some targets the OS may need to use marginally more stack than if it activates a task that is of lower priority. This value accounts for that. On most targets this value is zero. This value is used in worst-case stack size calculations. The value may change if significant changes are made to the OS configuration. e.g. STANDARD/EXTENDED, SC1/2/3/4.

12.3.4 Stack overheads for ECC tasks

XML name SpECC

Description

The extra amount of stack needed to start an ECC task. ECC tasks need to save slightly more state on the stack when they are started than BCC tasks. This value contains the difference. The value may change if significant changes are made to the OS configuration. e.g. STANDARD/EXTENDED, SC1/2/3/4. Also note that if you are using stack repositioning (to align the stack of untrusted code to suit the MPU) then you will need to reduce the value by the amount of the adjustment.

12.3.5 Stack overheads for ISR

XML name SpPreemption

Description

The amount of stack used to service a Category 2 ISR. When a Category 2 ISR interrupts a task, it usually places some data on the stack. If the ISR measures the stack to determine if the preempted task has exceeded its stack budget, then it will overestimate the stack usage unless this value is subtracted from the measured size. The value is also used when calculating the worst-case stack usage of the system. Be careful to set this value accurately. If its value is too high then when the subtraction occurs, 32-bit underflow can occur and cause the OS to think that a budget overrun has been detected. The value may change if significant changes are made to the OS configuration. e.g. STANDARD/EXTENDED, SC1/2/3/4.

12.3.6 ORTI22

XML name Orti22

Description

Select an ORTI generation based on the the Lauterbach debugger.

Settings

Value	Description
TRUE	Generate ORTI
FALSE	No ORTI

12.3.7 Number of Virtual Cores

XML name CoreCount

Description

Specify the number of cores to create. Defaults to 1.

12.3.8 MultiCore interrupts

XML name MC_Interrupt

Description

Select interrupt reserved for multi-core implementation.

Settings

Value	Description
v1	v1
v2	v2
v3	v3
v4	v4
v5	v5
v6	v6
v7	v7
v8	v8
v9	v9
v10	v10
v11	v11
v12	v12
v13	v13
v14	v14
v15	v15
v16	v16
v17	v17
v18	v18
v19	v19
v20	v20
v21	v21
v22	v22
v23	v23
v24	v24
v25	v25
v26	v26
v27	v27
v28	v28
v29	v29
v30	v30
v31	v31
v32	v32

12.3.9 Run-time license name

XML name LicName

Description

Override the name of the run-time license. Default LD_RTA-OS_VRTA.

12.3.10 Instrument MinGW for Code Coverage

XML name MinGWCoverage

Description

Build with gcov code coverage on MinGW

Settings

Value	Description
TRUE	Instrument for Code coverage
FALSE	Normal compilation (default)

12.4 OS Options

OS options can be set using the `--os_option` command-line option and can also be saved in the AUTOSAR XML configuration.

12.4.1 Optimize for core-local memory

XML name core_local

Description

If true, multicore applications will have their core-local data put into separate MemMap sections so that they can be located locally to each appropriate processor core. Defaults to FALSE.

12.4.2 Fast Terminate

XML name lightweight

Description

Declare that ChainTask() and TerminateTask() are only used in the Task entry function and their return value is not checked. This allows the OS to optimize task termination. Defaults to FALSE.

12.4.3 Disallow upwards activation

XML name no_upward_activation

Description

Declare that no task activates a higher priority task using ActivateTask or SetEvent. This allows the OS to optimize the time taken to activate tasks. Defaults to FALSE.

12.4.4 Disallow ChainTask()

XML name no_chain

Description

Declare that ChainTask will not be used. This allows the OS to optimize task switching time. Defaults to FALSE.

12.4.5 Disallow Schedule()

XML name no_schedule

Description

Declare that Schedule() will not be used. This can reduce the worst-case stack size of a system. Defaults to FALSE.

12.4.6 Optimize Schedule()

XML name optimize_schedule

Description

Changes the implementation of the Schedule() macro in order to avoid the run-time overhead of making a full OS call in cases where the OS can quickly determine that the OS call would have no effect. The return type of Schedule() should be assumed to be void when this is selected. Defaults to FALSE.

12.4.7 Allow STANDARD Status in SC3/SC4

XML name std_sc34_checks

Description

Allow STANDARD Status to be selected in Scalability Class 3 or 4 to reduce error checking overheads. AUTOSAR does not normally allow this. Defaults to FALSE.

12.4.8 Allow Alarm Callbacks in SC2/SC3/SC4

XML name sc234_alarmcallbacks

Description

Allow Alarm Callbacks in any Scalability Class. AUTOSAR normally only allows these in SC1. Defaults to FALSE.

12.4.9 Omit activation checks for WAITING state

XML name no_ecc_activate_in_wait

Description

ECC Tasks are normally activated once and do not terminate. When this is the case, RTA-OS can optimize task activation code by omitting the extra run-time checks to see if a task is WAITING. Defaults to TRUE.

12.4.10 Asynchronous TASK activation

XML name async_tasks

Description

The ActivateTask and SetEvent APIs change so that cross-core invocations no longer use an internal spinlock to coordinate between cores. Instead a message is sent to a queue on the core that owns the Task, and that core performs the required operation. When this option is in effect, the E_OS_LIMIT error can not be detected for cross-core activations. A different error E_OS_SYS_XCORE_QFULL may be seen if the queue on the receiving core is full. See the option 'AsyncQ' for how to avoid this. Defaults to FALSE.

12.4.11 Support ActivateTaskAsyn and SetEventAsyn

XML name asyn_apis

Description

These APIs were added in AUTOSAR V4.4.0. RTA-OS allows them to be used in any AUTOSAR version. This option has to be used to cause them to be generated because they add a small amount of extra run-time overhead. These APIs use the same mechanism that is used for the 'Asynchronous TASK activation' option, but allow you to have both synchronous and asynchronous versions at the same time. Defaults to FALSE.

12.4.12 AsyncQ

XML name async_qlen

Description

This option can be used to specify the size of the queue used when the 'Asynchronous TASK activation' or 'Support ActivateTaskAsyn and SetEventAsyn' options are in effect. The default queue size is 10 for each core. Specify one decimal number per core, using comma separators. A value 0 may only be used for cores that have no cross-core activation.

12.4.13 Timing Protection Interrupt

XML name timing_interrupt

Description

TRUE to use a Category 1 Interrupt to enforce Timing Protection. When this is TRUE and timing protection is configured, you must respond to the Os_Cbk_SetTimeLimit and Os_Cbk_SuspendTimeLimit callbacks and call Os_TimingFaultDetected if the interrupt fires without being suspended. You must create and configure the interrupt. Each OS core should have its own Category 1 Timing Protection Interrupt. Defaults to FALSE.

12.4.14 Omit Timing Protection

XML name no_timing_protection

Description

Timing protection features are included in the OS if any Task or ISR is configured with Timing Protection values. This option allows you to omit all timing protection checks. Defaults to FALSE.

12.4.15 Add Function Protection

XML name function_protection

Description

Adds the ability to police execution limits and recover from memory protection exceptions in OS Application functions. The API CallAndProtectFunction extends CallTrustedFunction by adding an execution time limit, and the ability to terminate a function. Defaults to FALSE.

12.4.16 Omit Memory Protection

XML name no_memory_protection

Description

Memory protection features are included in the OS if there are any OS applications that are Untrusted or have TrustedApplicationWithProtection set true. This option allows you to omit all memory protection features. Defaults to FALSE.

12.4.17 Untrusted code can read OS data

XML name protection_allows_os_reads

Description

When memory protection is enabled, it is assumed that untrusted code may not read OS variables. All OS service calls therefore switch the processor into protected mode. If, however, your memory protection strategy allows read-access to OS data then certain APIs can be optimized to avoid this mode switch because they only read OS data. (Also applies to OS Application that have TrustedApplicationWithProtection set to true. Defaults to FALSE.)

12.4.18 Single Memory Protection Zone

XML name single_zone_protection

Description

If all code that is subject to memory protection can run with a single set of MPU values, then the MPU can be set up before starting the OS and there is no need to call Os_Cbk_SetMemoryAccess. This option allows you to omit the Os_Cbk_SetMemoryAccess call. Defaults to FALSE.

12.4.19 Stack Only Memory Protection

XML name stack_only_protection

Description

If you want to run all untrusted code with the same basic memory protection settings but apply protection to the stack, then the OS only has to pass the stack-related fields (Address and Size) plus Application to Os_Cbk_SetMemoryAccess. This option removes the Task, ISR and Function values from the callback. (Also applies to OS Application that have TrustedApplicationWithProtection set to true.) Defaults to FALSE.

12.4.20 Omit Service Protection in SC3/SC4

XML name no_service_protection

Description

AUTOSAR expects extra service protection checks to be applied in SC3 and SC4. This option allows you to omit these checks. Defaults to FALSE.

12.4.21 Omit TerminateApplication

XML name no_terminate_application

Description

The TerminateApplication API is normally supported in the OS if there are any OS applications. This option allows you to omit support for TerminateApplication() and reduce system overheads. Defaults to FALSE.

12.4.22 Only Terminate Untrusted Applications

XML name only_terminate_untrusted

Description

By default, Trusted OS Applications can be terminated if there is a Timing Protection violation or TerminateApplication() is called. Setting this option tells the OS not to terminate trusted code. This can result in run-time savings. Defaults to FALSE.

12.4.23 Enable Time Monitoring

XML name meter_execution

Description

Record the execution times for Tasks and Category 2 ISRs. Defaults to FALSE.

12.4.24 Enable Elapsed Time Recording

XML name meter_elapsed_time

Description

Accumulate the total execution time for Tasks, Category 2 ISRs and Idle. Defaults to FALSE. (Requires Time Recording to be enabled)

12.4.25 Enable Activation Monitoring

XML name monitor_activations

Description

Record the activation times for Tasks. Defaults to FALSE.

12.4.26 Support Delayed Task Execution

XML name delayed_tasks

Description

Adds support for the APIs Os_SetDelayedTasks, Os_AddDelayedTasks and Os_RemoveDelayedTasks. Defaults to FALSE.

12.4.27 Collect OS usage metrics

XML name metrics

Description

Collects run-time information about how often OS features get used. Refer to Os_Metrics.h to see what data gets collected. Defaults to FALSE.

12.4.28 Additional Task Hooks

XML name task_lifetime_hooks

Description

Adds support for Os_Cbk_TaskStart and Os_Cbk_TaskEnd hooks. These act like Pre/Post Task hooks but don't get called when the task is pre-empted. Defaults to FALSE.

12.4.29 Task Activation Hook

XML name task_activation_hooks

Description

Adds support for Os_Cbk_TaskActivated hook. This gets called each time a Task is successfully activated. Defaults to FALSE.

12.4.30 Task Termination Hook

XML name task_termination_hooks

Description

Adds support for Os_Cbk_TaskTerminated hook. This gets called each time a Task is successfully terminated. Defaults to FALSE.

12.4.31 Additional ISR Hooks

XML name isr_lifetime_hooks

Description

Adds support for Os_Cbk_ISRStart, Os_Cbk_ISREnd, Os_Cbk_CrosscoreISRStart and Os_Cbk_CrosscoreISREnd hooks. These get called when an Category 2 or Crosscore ISR starts/ends. They don't get called again if the ISR is pre-empted. Defaults to FALSE.

12.4.32 Provide spinlock statistics

XML name spinlock_info

Description

Adds support for the GetSpinlockInfo API. Additional run-time statistics are recorded for spinlocks and the GetSpinlockInfo is supported to provide access to them. Defaults to FALSE.

12.4.33 Force spinlock error checks

XML name spinlock_std_check

Description

The spinlock APIs do not check for errors in STANDARD status builds. This option can be used to force the spinlock checks in both EXTENDED and STANDARD status builds. Defaults to FALSE.

12.4.34 Add Spinlock APIs for CAT1 ISRs

XML name unchecked_spinlocks

Description

The AUTOSAR spinlock APIs may only be called from TASKs or Category 2 ISRs. This option can be used to cause 3 new APIs UncheckedGetSpinlock, UncheckedTryToGetSpinlock and UncheckedReleaseSpinlock to be generated. These can be called from Category 1 ISRs, but the error checking that can be applied is much restricted so they should be used with care. Defaults to FALSE.

12.4.35 Stack Sampling

XML name stack_sampling

Description

Adds support for Os_Cbk_CheckStackDepth hook. It gets called when a TASK or Category 2 ISR starts. It also gets called from within the Os_GetStackUsage API. Defaults to FALSE.

12.4.36 MemMap level

XML name memmap_level

Description

Select how the AUTOSAR MemMap inclusion mechanism applied to OS code.

Settings

Value	Description
NONE	Omit MemMap includes
DECLARATION	MemMap inclusions are used around declarations
FULL	MemMap inclusions are used for declarations and definitions/externs (default)

12.4.37 IOC Data

XML name ioc_data

Description

Allows control over the MemMap sections used for some internal IOC data buffers.

Settings

Value	Description
Global	Use OS VAR section
Core	Use VAR_COREn section for communications that run on a single core (default)
Writer	Use _SEC_app_VAR section, where app is the OS Application of the sender so that writes do not need to change the processor protection mode

12.4.38 IOC Code

XML name ioc_code

Description

Allows control over the MemMap sections used for IOC API code.

Settings

Value	Description
OS	Use standard OS code section (default)
Caller	Use ..._SEC_app_LIB section, where app is the OS Application of the API caller

12.4.39 Disable IOC optimizations

XML name no_ioc_optimization

Description

RTA-RTE is able to provide RTA-OS with additional call-pattern information so that the IOC code can be better optimized. Set this option to TRUE to disable these optimizations. Defaults to FALSE.

12.4.40 IOC blocking threshold

XML name ioc_blocking_threshold

Description

IOC code optimizations for RTA-RTE can result in data transfers for many IOC communications being performed in a single batch operation. Interrupts may be disabled during this transfer, which may block higher priority code from running. Setting a non-zero value here controls the number of bytes that will be transferred in a batch. Note that because transfers for each communication must not be split, a batch might be slightly bigger than the value specified.

12.4.41 Omit Default Implementations

XML name omit_default_implementations

Description

RTA-OS normally provides default implementations for Os_Cbk_Idle, Os_Cbk_InShutdown and Os_Cbk_StackOverrunHook in the generated library. Set this option to TRUE to omit these functions from the library. Defaults to FALSE.

12.4.42 Allow forced TASK terminations

XML name allow_forced_task_terminations

Description

Many real-world AUTOSAR systems do not safely support termination of tasks part way through execution. If the OS sees that the configuration relates to a full-stack AUTOSAR system it will remove the ability to force-terminate tasks from the OS library unless this option is set to TRUE. This means that errors such as timing budget overruns must result in shutdown or reset of the ECU. Defaults to FALSE when full-stack AUTOSAR systems are detected in the configuration.

12.4.43 Always clear OS data

XML name avoid_noinit

Description

Some OS data is placed in NO_INIT memory sections. Set this option to TRUE to clear all OS data on resets. Defaults to FALSE.

13 Output File Formats

13.1 RTA-TRACE Configuration files

RTA-OS generates an RTA-TRACE configuration file when RTA-TRACE is enabled. The format of this file is similar to the ORTI format and is described in detail in the *RTA-TRACE OS Instrumenting Kit Manual*.

13.2 ORTI Files

This section describes the ORTI objects output by RTA-OS.

When ORTI output is supported by a port and ORTI generation is configured then a file called `RTA0S.orti` is generated when the kernel is built using **rtaosgen**.

An ORTI object encapsulates information about OS objects in RTA-OS, for example tasks, ISRs, alarms etc. An application may contain zero or more instances of each ORTI objects, each of which has a unique name. Each ORTI object has a number of attributes and each attribute has a value.

For example, the OS has a `RUNNINGTASK` attribute that shows the task that is currently running.

The following sections present the ORTI objects generated. Each section has the following structure:

Object

Name of the ORTI object

Description

A description of the ORTI object.

Attributes

The attributes for the ORTI object.

Attribute	Description
Attribute Name	*Attribute ORTI file description* - Description of attribute

Each row of the table names the attribute being described and gives a brief explanation of it. The name of each attribute is given in the Attribute column. Attributes that are prefix with vs_ have been added for RTA-OS support and are not standard ORTI attributes. Your debugger may or may not be able to display these attributes depending on how well it conforms to the ORTI standard.

Many debuggers display the attribute name. However, some debuggers choose to display the attribute description that is present in the ORTI file instead. The descriptions used in RTA-OS appear in quotation marks at the start of the Description column.

13.2.1 OS

Object

OS

Description

There is one OS object per AUTOSAR core. The name is derived from the RTA-OS project name.

Attributes

Attribute	Description
RUNNINGTASK	*Running task* - The name of the TASK that is currently running. If an ISR interrupts a task this attribute will continue to display the name of the task that was interrupted while the ISR is executing.
RUNNINGTASKPRIORITY	*Running task priority* - The current priority of the running task, using the same terms as in the OIL file. RUNNINGTASKPRIORITY does not show the effect of locking a resource shared by tasks and ISRs.
RUNNINGISR2	*Running cat 2 ISR* - The value of this attribute is the name of the Category 2 ISR that is currently running (if there is one). NO_ISR is displayed if no Category 2 ISR is currently running.
SERVICETRACE	*OS Services Watch*- Indicates the entry or exit of a service routine (an RTA-OS Component API call) and the name of this routine. Some debuggers recognize this attribute as a special trace attribute and can provide additional diagnostic support. On other debuggers, you will be shown which API call was most recently entered or completed.
LASTERROR	*Last OSEK error* - Gives the name of the last error that has occurred. Initially set to E_OK.
CURRENTAPPMODE	*Current AppMode* - Current application mode using the names stated in the XML file. The value *unknown AppMode* is reported if the application mode does not conform to a value in the XML file.

13.2.2 Task

Object

TASK

Description

Generated in response to task declarations in the configuration file.

Attributes

Attribute	Description
STATE	*State* - The task state. One of SUSPENDED, RUNNING, READY and WAITING.
vs_BASEPRIORITY	*Base priority* - Gives the base priority of the task. The base priority is the priority of the task as defined in the OIL file.
PRIORITY	*Dispatch priority* - Gives the dispatch priority of the task. The dispatch priority is the priority that the task starts running at. This can be higher than the base priority if internal resources are used or if the task is non-preemptable.
CURRENTACTIVATIONS	*Activations* - Gives the maximum number of activations allowed.

13.2.3 Category 1 ISR

There are no ORTI objects generated for Category 1 ISRs.

13.2.4 Category 2 ISR

There are no ORTI objects generated for Category 2 ISRs.

13.2.5 Resource

Object

RESOURCE

Description

Generated in response to resource declarations in the configuration file.

Attributes

Attribute	Description
PRIORITY	*Ceiling Priority* - Gives the ceiling priority of the resource in terms of the task priority that defines the ceiling.

Attribute	Description
LOCKER	*Resource locker* - Shows the current holder of the resource.
STATE	*Resource State* - Shows the state of the resource as locked or not locked.

13.2.6 Events

There are no ORTI objects generated for events.

13.2.7 Counter

There are no ORTI objects generated for counters.

13.2.8 Alarm

Object

ALARM

Description

Only generated in response to alarm declarations in the configuration file.

Attributes

Attribute	Description
ALARMTIME	*Alarm Time* - Shows when the alarm expires next. Refer to the Count value in the COUNTER object to establish the current count value.
CYCLETIME	*Cycle Time* - Gives the period of the cycle for a cyclic alarm. CYCLETIME will be zero for a single-shot alarm.
ACTION	*Action* - The action to perform when the alarm expires. This can include the following: • Activate a task. • Set an event. • Execute a callback function.
STATE	*Alarm state* - Indicates whether the alarm is running. Takes the value RUNNING or STOPPED.
COUNTER	*Counter* - Gives the name of the counter to which this alarm is attached.

13.2.9 Schedule Table

Object

SCHEDULETABLE

Description

Only generated in response to schedule table declarations in the configuration file.

Attributes

Attribute	Description
COUNTER	*Counter* - Gives the name of the counter to which this schedule table is attached.
STATE	*State* - Indicates the state of the schedule table.
EXPIRYTIME	*Expiry Time* - The tick at which the next expiry point is due to be processed.
NEXT	*Next table* - The next schedule table (if set).

14 Compatibility and Migration

This chapter provides compatibility information for RTA-OS with other ETAS tooling and outlines the major changes between RTA-OS and the earlier RTA-OSEK series of operating systems to assist users migrating to RTA-OS.

14.1 ETAS Tools

The following table outlines the compatibility between RTA-OS and other ETAS software tools. Compatibility is split into two parts - the configuration language and the use of the API. The following indications are given:

- ✓ Fully compatible
- ✓ Partially compatible, see the notes for more details
- ✗ Not compatible

For a more detailed discussion of specific cases, please contact ETAS.

Product	Version	Compatibility		Notes
		Config	API	
ERCOSEK	4.x	✗	✓	1
RTA-OSEK	4.x	✗	✓	2
	5.x	✗	✓	2
RTA-TRACE	2.x	✓	✓	3
RTA-RTE2.x	1.x	✗	✓	4
ASCET	4.x	✗	✗	??
	5.x	✗	✓	??
	6.x	✗	✓	??

Notes on ETAS Tool compatibility

1. OSEK API calls are portable with the exceptions which have slightly modified behavior in AUTOSAR OS:
   - `StartOS()` does not return.
   - `SetRelAlarm()` cannot use zero as the offset parameter.
2. See Section 14.2 for specific details
3. RTA-OS adopts the AUTOSAR guidelines for namespaces in basic software modules for all internal names. However, the external names of all AUTOSAR OS API calls, macros and type definitions are provided in the external API for compatibility with the AUTOSAR standard.

   Any RTA-OS functionality which is not part of the AUTOSAR OS adopts the AUTOSAR naming convention for both internal and external names. For the OS this means:

   - Variable, API call names and constants are prefixed 0s_
   - Macros are prefixed 0S_

The AUTOSAR naming convention has also been applied to RTA-TRACE instrumentation code. While this does not change the behavior of RTA-TRACE and it transparent to users, it does mean that the documentation that ships with RTA-TRACE does not accurately reflect the names of API calls and types used in RTA-OS. However, conversion between the documented names and those generated is trivial:

- All API calls, types and variables are prefixed `Os_`. For example:

  ```
  LogTracepoint(MyTracepoint)
  ```

  becomes

  ```
  Os_LogTracepoint(MyTracepoint).
  ```

- All macros are prefixed `OS_`. For example:

  ```
  TRACE_ERRORS_CLASS
  ```

  becomes

  ```
  OS_TRACE_ERRORS_CLASS.
  ```

4. RTA-RTE2.x generates an OS configuration in OIL that is compatible with AUTOSAR OS R2.x but this is not compatible with AUTOSAR OS R3.x. Most OS calls generated by RTA-RTE2.x are compatible with AUTOSAR OS. The API call `StartScheduleTable()` is used by the RTE library when integrating with an AUTOSAR OS R1.0. This call needs to be replaced by the `StartScheduleTableRel()` call to work with AUTOSAR OS. Full compatibility with the RTE is possible if the `OSENV_UNSUPPORTED` is defined instead. You should consult the *RTA-RTE2x Toolchain Integration Guide* for further details.

14.2 API Call Compatibility

The following table shows the compatibility between RTA-OS, AUTOSAR, RTA-OSEK and the OSEK family of operating systems. It does not cover the APIs added for AUTOSAR multicore operation, because none of these exist in the earlier standards.

API call	OSEK OS v2.2.x	RTA-OSEK v4.x	AUTOSAR R1.0 SC1	RTA-OSEK v5.x	AUTOSAR R3.0 SC1	AUTOSAR R3.0 SC2	AUTOSAR R3.0 SC3	AUTOSAR R3.0 SC4	RTA-OS	See Section
ActivateTask	✓	✓	✓	✓	✓	✓	✓	✓	✓	
ActivateTaskset		✓		✓						14.2.1
AdvanceSchedule		✓		✓						14.2.3
AssignTaskset		✓		✓						14.2.1
CallTrustedFunction							✓	✓	✓	
CancelAlarm	✓	✓	✓	✓	✓	✓	✓	✓	✓	
ChainTask	✓	✓	✓	✓	✓	✓	✓	✓	✓	
ChainTaskset		✓		✓						14.2.1

API call	OSEK OS v2.2.x	RTA-OSEK v4.x	AUTOSAR R1.0 SC1	RTA-OSEK v5.x	AUTOSAR R3.0 SC1	AUTOSAR R3.0 SC2	AUTOSAR R3.0 SC3	AUTOSAR R3.0 SC4	RTA-OS	See Section
CheckISRMemoryAccess							✓	✓	✓	
CheckObjectAccess							✓	✓	✓	
CheckObjectOwnership							✓	✓	✓	
CheckTaskMemoryAccess							✓	✓	✓	
ClearEvent	✓	✓	✓	✓	✓	✓	✓	✓	✓	
CloseCOM	✓	✓	✓	✓						14.2.4
DisableAllInterrupts	✓	✓	✓	✓	✓	✓	✓	✓	✓	
EnableAllInterrupts	✓	✓	✓	✓	✓	✓	✓	✓	✓	
GetActiveApplicationMode	✓	✓	✓	✓	✓	✓	✓	✓	✓	
GetAlarm	✓	✓	✓	✓	✓	✓	✓	✓	✓	
GetAlarmBase	✓	✓	✓	✓	✓	✓	✓	✓	✓	
GetApplicationID							✓	✓	✓	
GetArrivalpointDelay		✓		✓						14.2.3
GetArrivalpointNext		✓		✓						14.2.3
GetArrivalpointTasksetRef		✓		✓						14.2.3
GetCounterValue		✓		✓	✓	✓	✓	✓	✓	
GetElapsedCounterValue					✓	✓	✓	✓	✓	
GetEvent	✓	✓	✓	✓	✓	✓	✓	✓	✓	
GetExecutionTime		✓		✓						14.2.2
GetISRID			✓	✓	✓	✓	✓	✓	✓	
GetLargestExecutionTime		✓		✓						14.2.2
GetMessageResource	✓	✓	✓	✓						14.2.4
GetMessageStatus	✓	✓	✓	✓						14.2.4
GetResource	✓	✓	✓	✓	✓	✓	✓	✓	✓	
GetScheduleNext		✓		✓						14.2.3
GetScheduleStatus		✓		✓						14.2.3
GetScheduleTableStatus		✓	✓	✓	✓	✓	✓	✓	✓	
GetScheduleValue		✓		✓						14.2.3
GetStackOffset		✓		✓						14.2.11
GetTaskID	✓	✓	✓	✓	✓	✓	✓	✓	✓	
GetTasksetRef		✓		✓						14.2.1
GetTaskState	✓	✓	✓	✓	✓	✓	✓	✓	✓	
IncrementCounter			✓	✓	✓	✓	✓	✓	✓	
InitCOM	✓	✓	✓	✓						14.2.4
InitCounter				✓						
MergeTaskset		✓		✓						14.2.1
NextScheduleTable			✓	✓	✓	✓	✓	✓	✓	
Os_AdvanceCounter									✓	
Os_GetExecutionTime									✓	14.2.2
Os_GetISRMaxExecutionTime									✓	14.2.2
Os_GetISRMaxStackUsage									✓	14.2.11
Os_GetStackUsage									✓	14.2.11
Os_GetStackValue									✓	14.2.11
Os_GetTaskMaxExecutionTime									✓	14.2.2

API call	OSEK OS v2.2.x	RTA-OSEK v4.x	AUTOSAR R1.0 SC1	RTA-OSEK v5.x	AUTOSAR R3.0 SC1	AUTOSAR R3.0 SC2	AUTOSAR R3.0 SC3	AUTOSAR R3.0 SC4	RTA-OS	See Section
Os_GetTaskMaxStackUsage									✓	14.2.11
Os_ResetISRMaxExecutionTime									✓	14.2.2
Os_ResetISRMaxStackUsage									✓	14.2.11
Os_ResetTaskMaxExecutionTime									✓	14.2.2
Os_ResetTaskMaxStackUsage									✓	14.2.11
Os_Restart									✓	
Os_SetRestartPoint									✓	
osAdvanceCounter				✓						14.2.7
osResetOS				✓						14.2.12
ReadFlag	✓	✓	✓	✓						14.2.4
ReceiveMessage	✓	✓	✓	✓						14.2.4
ReleaseMessageResource	✓	✓	✓	✓						14.2.4
ReleaseResource	✓	✓	✓	✓	✓	✓	✓	✓	✓	
RemoveTaskset		✓		✓						14.2.1
ResetFlag	✓	✓	✓	✓						14.2.4
ResetLargestExecutionTime		✓		✓						14.2.2
ResumeAllInterrupts	✓	✓	✓	✓	✓	✓	✓	✓	✓	
ResumeOSInterrupts	✓	✓	✓	✓	✓	✓	✓	✓	✓	
Schedule	✓	✓	✓	✓	✓	✓	✓	✓	✓	
SendMessage	✓	✓	✓	✓						14.2.4
SetAbsAlarm	✓	✓	✓	✓	✓	✓	✓	✓	✓	
SetArrivalpointDelay		✓		✓						14.2.3
SetArrivalpointNext		✓		✓						14.2.3
SetEvent	✓	✓	✓	✓	✓	✓	✓	✓	✓	
SetRelAlarm	✓	✓	✓	✓	✓	✓	✓	✓	✓	14.2.8
SetScheduleNext		✓		✓						
SetScheduleTableAsync						✓		✓	✓	
ShutdownOS	✓	✓	✓	✓	✓	✓	✓	✓	✓	14.2.6
StartCOM	✓	✓	✓	✓						14.2.4
StartOS	✓	✓	✓	✓	✓	✓	✓	✓	✓	14.2.5
StartSchedule		✓		✓						14.2.3
StartScheduleTable			✓	✓						14.2.9
StartScheduleTableAbs					✓	✓	✓	✓	✓	
StartScheduleTableRel					✓	✓	✓	✓	✓	
StartScheduleTableSynchron						✓		✓	✓	
StopCOM	✓	✓	✓	✓						14.2.4
StopSchedule		✓		✓						14.2.3
StopScheduleTable			✓	✓	✓	✓	✓	✓	✓	
SuspendAllInterrupts	✓	✓	✓	✓	✓	✓	✓	✓	✓	
SuspendOSInterrupts	✓	✓	✓	✓	✓	✓	✓	✓	✓	
SyncScheduleTable						✓		✓	✓	
TerminateApplication							✓	✓	✓	
TerminateTask	✓	✓	✓	✓	✓	✓	✓	✓	✓	
TestArrivalpointWriteable		✓		✓						14.2.3

API call	OSEK OS v2.2.x	RTA-OSEK v4.x	AUTOSAR R1.0 SC1	RTA-OSEK v5.x	AUTOSAR R3.0 SC1	AUTOSAR R3.0 SC2	AUTOSAR R3.0 SC3	AUTOSAR R3.0 SC4	RTA-OS	See Section
`TestEquivalentTaskset`		✓		✓						14.2.1
`TestSubTaskset`		✓		✓						14.2.1
`Tick_<CounterID>`		✓		✓						14.2.10
`TickSchedule`		✓		✓						14.2.3
`WaitEvent`	✓	✓	✓	✓	✓	✓	✓	✓	✓	

14.2.1 Tasksets

RTA-OSEK taskset API. Tasksets have been removed from RTA-OS. The functionality can be implemented by executing multiple `ActivateTask()` calls in sequence. However, note that this only provides the same run-time behavior when the set of tasks that are activated are all of equal or lower priority than the task making the calls.

14.2.2 Time Monitoring

The RTA-OSEK timing build is replaced by the configuration option 'Time Monitoring'. This means that it is possible to use `EXTENDED` status without including the code for time monitoring.

In RTA-OS the API calls are modified to take the `Os_` prefix but have identical behavior as the old RTA-OSEK calls. However, there are now specific calls for tasks and ISRs that replace the `GetLargestExecutionTime` and `GetLargestExecutionTime` calls.

- `Os_GetLargestExecutionTime` is replaced by `Os_Get[Task|ISR]MaxExecutionTime`
- `Os_ResetLargestExecutionTime` is replaced by `Os_Reset[Task|ISR]MaxExecutionTime`

14.2.3 Schedules

The RTA-OSEK schedule mechanism is replaced by AUTOSAR's ScheduleTable Mechanism. Note that it is not possible in RTA-OS to modify the schedule at runtime (this functionality is not supported by AUTOSAR OS). If runtime modification is required then alarms should be used instead.

14.2.4 OSEK COM

In OSEK OS, OSEK COM features may be provided by the OS (when OSEK COM is not used). This feature is removed in AUTOSAR OS as internal communication for applications is provided by the AUTOSAR RTE.

14.2.5 Behavior of StartOS()

The StartOS() call may return in OSEK OS. This is the behavior provided in RTA-OSEK. In AUTOSAR OS this behavior is prohibited - the call *must not* return. This is the behavior provided by RTA-OS. This means that it is no longer possible to use an idle loop placed after StartOS() as the idle mechanism. In RTA-OS the kernel will busy wait by default when there are no tasks and ISRs to run. You can replace this default behavior by providing a function called Os_Cbk_Idle() that implements your own idle (background task) functionality.

14.2.6 Behavior of ShutdownOS()

OSEK OS allows implementations to return from ShutdownOS(). In AUTOSAR OS ShutdownOS() must not return. RTA-OSEK always had the behavior specific by AUTOSAR OS, but it you are migrating from another OS then you may need to modify your application to reflect this change.

14.2.7 Hardware Counter Driver

The RTA-OSEK hardware counter driver call, osAdvanceCounter() is renamed Os_AdvanceCounter in RTA-OS. The behavior of the call is also modified. In RTA-OSEK the call returned the status of the counter so the user could set up the next expiry. In RTA-OS this operation is performed internally (via a call to the user-provided Os_Cbk_Set_CounterID API callback) for the first setup and application code must then call the Os_Cbk_Status_CounterID to check for multiple expiries.

14.2.8 Forbidding of Zero for SetRelAlarm()

SetRelAlarm(_, 0, _) is allowed in OSEK OS but forbidden in AUTOSAR OS.

14.2.9 Changes to Schedule Table API

The AUTOSAR OS standard has modified the call to start a schedule table so that the mechanism has the same concepts of absolute and relative start that are found with OSEK OS alarms. The API call StartScheduleTable() has been removed from the standard and is replaced by StartScheduleTable[Rel|Abs] in AUTOSAR R3.0. If you need to replicate the behavior of the StartScheduleTable(Tbl,At) call then you should used StartScheduleTableRel(Tbl,At).

14.2.10 Software Counter Driver

The RTA-OSEK Tick_CounterID() calls have been replaced by AUTOSAR standard IncrementCounter() counter call which takes the CounterID as a parameter. However, to replicate the performance improvements that the 'static' version of the call provides, RTA-OS includes a 'static' version of the AUTOSAR call - IncrementCounter_CounterID() - which has identical behavior to the Tick_CounterID() call.

14.2.11 Stack Monitoring

The behavior of stack measurement is modified between RTA-OSEK and RTA-OS. In RTA-OSEK stack measurements are made from the base address of the stack using the `GetStackOffset()`. Typically the base address of the stack was given to RTA-OSEK at link time by defining label called `SP_INIT`.

In RTA-OS the `GetStackOffset()` call is replaced by `Os_GetStackValue()`.

RTA-OSEK required you to calculate the amount of stack used by each task or ISR. You can still do this with RTA-OS, but an additional API call, `Os_GetStackUsage()`, has been provided that returns the stack consumed by the calling task/ISR alone at the point of the call. The avoids the need to do any stack calculations yourself.

RTA-OS also logs the worst-case observed stack usage for each task/ISR when a context switch (or a call to `Os_GetStackUsage()` is made. Additional API calls are provided to get the largest observed stack usage for each task/ISR and to reset the largest observed value.

This model parallels the time monitoring functionality provided by RTA-OS.

14.2.12 Restarting the OS

Neither OSEK OS or AUTOSAR OS provide facilities to re-start the OS at runtime. As this is commonly required functionality, RTA-OSEK provided the `osReset0S()` API call that allowed a restart to be performed.

In RTA-OS this is replaced by a general-purpose restart mechanism. The API call `Os_SetRestartPoint` is provided that can be made anywhere before you call `StartOS()` to place a marker from where the restart should happen. This means you can re-initialize any hardware required before the call to `StartOS()`. A restart is then achieved by calling `Os_Restart` which jumps to the marker you have set.

15 Contacting ETAS

15.1 Technical Support

Technical support is available to all users with a valid support contract. If you do not have a valid support contract, please contact your regional sales office (see below).

The best way to get technical support is by email. Any problems or questions about the use of the product should be sent to:

rta.hotline@etas.com

If you prefer to discuss your problem with the technical support team, you call the support hotline on:

+44 (0)1904 562624.

The hotline is available during normal office hours (0900-1730 GMT/BST).

In either case, it is helpful if you can provide technical support with the following information:

- Your support contract number
- The version of the ETAS tools you are using
- The version of the compiler tool chain you are using
- The command line (or reproduction of steps) that result in an error message
- The error messages or return codes you received (if any)
- Your `.xml`, `.arxml` and `.rtaos` files
- The file `Diagnostic.dmp` if it was generated

15.2 General Enquiries

15.2.1 ETAS Global Headquarters

ETAS GmbH

Borsigstrasse 24	Phone:	+49 711 3423-0
70469 Stuttgart	Fax:	+49 711 3423-2106
Germany	WWW:	www.etas.com

15.2.2 ETAS Local Sales & Support Offices

Contact details for your local sales office and local technical support team (where available) can be found on the ETAS web site:

ETAS subsidiaries	www.etas.com/en/contact.php
ETAS technical support	www.etas.com/en/hotlines.php