Document Title	Specification of Operating System
Document Owner	AUTOSAR
Document Responsibility	AUTOSAR
Document Identification No	034
Document Status	Final
Part of AUTOSAR Standard	Classic Platform
Part of Standard Release	4.4.0

Document Change History

Date	Release	Changed by	Change Description
2018-10-31	4.4.0	AUTOSAR Release Management	• New asynchronous services • ARTI support (DRAFT) • Editorial changes / clarifications
2017-12-08	4.3.1	AUTOSAR Release Management	• minor corrections / clarifications / editorial changes; For details please refer to the ChangeDocumentation
2016-11-30	4.3.0	AUTOSAR Release Management	• Added new API for peripheral access • Added new API for interrupt handling • Minor updates/clarification of descriptions • Editorial changes
2015-07-31	4.2.2	AUTOSAR Release Management	• Allow calls to ControlIdle from all cores • Minor updates/clarification of descriptions • Editorial changes
2014-10-31	4.2.1	AUTOSAR Release Management	• Add support for AsilQmProtection • Minor updates/clarification of descriptions • Editorial changes
2014-03-31	4.1.3	AUTOSAR Release Management	• Changed multiplicity of attributes in IocSender/ReceiverProperties • Minor updates/clarification of descriptions • Editorial changes

Document Change History

Date	Release	Changed by	Change Description
2013-10-31	4.1.2	AUTOSAR Release Management	• Clarification on E_OS_NESTING_DEADLOCK • Update of table 2 • Corrected multiplicity of ECUC_Os_00393 • Minor updates/clarification of descriptions • Editorial changes • Removed chapter(s) on change documentation
2013-03-15	4.1.1	AUTOSAR Administration	• Add support for ECU degradation • Changed service interface description to a formal format • Several minor changes and clarifications
2011-12-22	4.0.3	AUTOSAR Administration	• Included MultiCore support from former “Specification of Multi-Core OS Architecture”
2010-09-30	3.1.5	AUTOSAR Administration	• Clarification in 7.8.1 (meaning of "do nothing") and 7.1.2.1 ("OSEK declarations") • Minor changes as typos and rewording
2010-02-02	3.1.4	AUTOSAR Administration	• Extension of services (Chapter 12) • States in OS- Applications • Active termination of other OS-Applications in possible (Chapter8) • Legal disclaimer revised • Chapter 10.4 revised
2009-02-04	3.1.2	AUTOSAR Administration	• Changes in OS configuration: • removed "OsAppModeId" Parameter from OsAppModeContainer • added optional references from OsAppModeContainer to OsAlarm, OsTask and OsScheduleTable
2008-08-13	3.1.1	AUTOSAR Administration	• Legal Disclaimer revised

Document Change History

Date	Release	Changed by	Change Description
2008-02-01	3.0.2	AUTOSAR Administration	• Added "OsScheduleTableDuration" parameter to configuration specification chapter
2007-12-21	3.0.1	AUTOSAR Administration	• Changed methods for timing protection • Moved configuration from OIL to AUTOSAR XML • Clarrified description for synchronization and schedule tables • Document meta information extended • Small layout adaptations made
2007-01-24	2.1.15	AUTOSAR Administration	• Added support for SoftwareFreeRunningTimer (SWFRT) incl. 2 new APIs • Added API to start a schedule table synchron • Misc. Corrections, Clarification and further explanations • Legal disclaimer revised • Release Notes added • "Advice for users" revised • "Revision Information" added
2006-05-16	2.0	AUTOSAR Administration	• Document structure adapted to common Release 2.0 SWS Template. • Major changes in chapter 10 • Structure of document changed partly • Other changes see chapter 14
2005-05-31	1.0	AUTOSAR Administration	• Initial Release

Table of Content

1 Introduction and functional overview

This document describes the essential requirements on the AUTOSAR Operating System to satisfy the top-level requirements presented in the AUTOSAR SRS [2].

In general, operating systems can be split up in different groups according to their characteristics, e.g. statically configured vs. dynamically managed. To classify the AUTOSAR OS, here are the basic features: the OS

- is configured and scaled statically
- is amenable to reasoning of real-time performance
- provides a priority-based scheduling policy
- provides protective functions (memory, timing etc.) at run-time
- is hostable on low-end controllers and without external resources

This feature set defines the type of OS commonly used in the current generation of automotive ECUs, with the exception of Telematic/Infotainment systems. It is assumed that Telematic/Infotainment systems will continue to use proprietary Oss under the AUTOSAR framework (e.g. Windows CE, VxWorks, QNX, etc.). In the case where AUTOSAR components are needed to run on these proprietary Oss, the interfaces defined in this document should be provided as an Operating System Abstraction Layer (OSAL).

This document uses the industry standard [15] (ISO 17356-3) as the basis for the AUTOSAR OS. The reader should be familiar with this standard before reading this document.

This document describes extensions to, and restrictions of [15].

2 Acronyms and abbreviations

Abbreviation	***Description***
API	Application Programming Interface
AR	AUTOSAR
BSW	Basic Software
BSWMD	Basic Software Module Description
CDD	Complex Driver
COM	Communication
ECC	Extended Conformance Class
ECU	Electronic Control Unit
HW	Hardware
ID	Identifier
IOC	Inter OS-Application communicator
ISR	Interrupt Service Routine
LE	A locatable entity is a distinct piece of software that has the same effect regardless of which core it is located.
MC	Multi-Core
MCU	Microcontroller Unit
ME	Mutual exclusion
MPU	Memory Protection Unit
NMI	Mutual exclusion
OIL	OSEK Implementation Language
OS	Operating System
OSEK/VDX	Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug
RTE	Run-Time Environment
RTOS	Real Time Operating System
SC	Single-Core
SLA	Software Layered Architecture
SW	Software
SWC	Software Component
SWFRT	Software FreeRunningTimer

2.1 Glossary of Terms

Term:	***Definition***	
Access Right	An indication that an object (e.g. Task, ISR, hook function) of an OS-Application has the permission of access or manipulation with respect to memory, OS services or (set of) OS objects.	
Cardinality	The number of items in a set.	
Counter	An operating system object that registers a count in ticks. There are two types of counters:	
	Hardware Counter	A counter that is advanced by hardware (e.g. timer). The count value is maintained by the peripheral "in hardware".
	Software Counter	A counter which is incremented by making the `IncrementCounter()` API call (see SWS_Os_00399). The count value is maintained by the operating system "in software".
Deadline	The time at which a Task/Category 2 ISR must reach a certain point during its execution defined by system design relative to the stimulus that triggered activation. See Figure 2.1	
Delay	The number of ticks between two adjacent expiry points on a schedule table.	

	A pair of expiry points X and Y are said to be adjacent when: • There is no expiry point Z such that X.Offset < Z.Offset < Y.Offset. In this case the Delay = Y.Offset-X.Offset • X and Y are the Final Expiry Point and the Initial Expiry Point respectively. In this case Delay = (Duration-X.Offset)+Y.Offset When used in the text, Delay is a relative number of ticks measured from a specified expiry point. For example: X.Delay is the delay from X to the next expiry point.	
Deviation	The minimum number of ticks between the current position on an explicitly synchronized schedule table and the value of the synchronization count modulo the duration of the schedule table.	
Duration	The number of ticks from a notional zero at which a schedule table wraps.	
Execution Time	Tasks: The net time a task spends in the `RUNNING` state without entering the `SUSPENDED` or `WAITING` state excluding all preemptions due to ISRs which preempt the task. An extended task executing the `WaitEvent()` API call to wait on an event which is already set notionally enters the `WAITING` state. For multiple activated basic tasks the net time is per activation of a task. ISRs: The net time from the first to the last instruction of the user provided Category 2 interrupt handler excluding all preemptions due to higher priority ISRs executing in preference. Execution time includes the time spent in the error, pretask and posttask hooks and the time spent making OS service calls.	
Execution Budget	Maximum permitted execution time for a Task/ISR.	
Expiry Point	The offset on a Schedule Table, measured from zero, at which the OS activates tasks and/or sets events.	
	Initial Expiry Point	The expiry point with the smallest offset
	Final Expiry Point	The expiry point with the largest offset
Hook Function	A Hook function is implemented by the user and invoked by the operating system in the case of certain incidents. In order to react to these on system or application level, there are two kinds of hook functions	
	Application-specific	Hook functions within the scope of an individual OS-Application.
	System-specific	Hook functions within the scope of the complete system (in general provided by the integrator).
Initial Offset	The smallest expiry point offset on a schedule table. This can be zero.	
Interarrival Time	Basic Tasks The time between successively entering the `READY` state from the `SUSPENDED` state. Activation of a task always represents a new arrival. This applies in the case of multiple activations, even if an existing instance of the task is in the `RUNNING` or `READY` state. Extended Tasks: The time between successively entering the `READY` state from the `SUSPENDED` or `WAITING` states. Setting an event for a task in the `WAITING` state represents a new arrival if the task is waiting on the event. Waiting for an event in the `RUNNING` state which is already set represents a new arrival. ISRs: The time between successive occurrences of an interrupt. See Figure 2.1.	

Interrupt Lock Time	The time for which a Task/ISR executes with Category 1 interrupts disabled/suspended and/or Category 2 interrupts disabled/suspended .	
Interrupt Source Enable	The switch which enables a specific interrupt source in the hardware.	
Interrupt Vector Table	Conceptually, the interrupt vector table contains the mapping from hardware interrupt requests to (software) interrupt service routines. The real content of the Interrupt Vector Table is very hardware specific, e.g. it can contain the start addresses of the interrupt service routines.	
Final Delay	The difference between the Final Expiry Point offset and the duration on a schedule table in ticks. This value defines the delay from the Final Expiry Point to the logical end of the schedule table for single-shot and "nexted" schedule tables.	
Forced OS-Application Termination	The operating system frees all system objects, e.g. forcibly terminates Tasks, disables interrupts, etc., which are associated to the OS-Application. OS-Application and internal variables are potentially left in an undefined state.	
Forced Termination	The OS terminates the Task/Category 2 ISR and does "unlock" its held resources. For details see SWS_Os_00108 and SWS_Os_00109.	
Linker File	File containing linking settings for the linker. The syntax of the linker file depends on the specific linker and, consequently, definitions are stored "linker-specific" in the linker file.	
Lock Budget	Maximum permitted Interrupt Lock Time or Resource Lock Time.	
Master core	A master core is a core from which the AUTOSAR system is bootstrapped.	
Memory Protection Unit	A Memory Protection Unit (MPU) enables memory partitioning with individual protection attributes. This is distinct from a Memory Management Unit (MMU) that provides a mapping between virtual addresses and physical memory locations at runtime. Note that some devices may realise the functionality of an MPU in an MMU.	
Mode	Describes the permissions available on a processor.	
	Privileged	In general, in »privileged mode« unrestricted access is available to memory as well as the underlying hardware.
	Non-privileged	In »non-privileged mode« access is restricted.
Modulus	The number of ticks required to complete a full wrap of an OSEK counter. This is equal to `OsCounterMaxAllowedValue` +1 ticks of the counter.	
OS-Application	A collection of OS objects	
	Trusted	An OS-Application that may be executed in privileged mode and may have unrestricted access to the API and hardware resources. Only trusted applications can provide trusted functions.
	Non-trusted	An OS-Application that is executed in non-privileged mode has restricted access to the API and hardware resources.
OS object	Object that belongs to a single OS-Application: Task, ISR, Alarm, Event, Schedule Table, Resource, Trusted Function, Counter, Applicaton-specific hook.	
OS Service	OS services are the API of the operating system.	
Protection Error	Systematic error in the software of an OS-Application.	
	Memory access violation	A protection error caused by access to an address in a manner for which no access right exists.
	Timing fault	A protection error that violates the timing protection.
	Illegal service	A protection error that violates the service protection, e.g. unauthorized call to OS service.
	Hardware exception	division by zero, illegal instruction etc.
Resource Lock Time	The time an OSEK resource is held by a Task/ISR (excluding the preemptions of the Task/ISR by higher prior Tasks/ISRs).	
Response Time	The time between a Task/ISR being made ready to execute and generating a specified response. The time includes all preemptions. See Figure 2.1	
Restart an OS-Application	An OS-Application can be restarted after self-termination or being forcibly terminated because of a protection error. When an OS-Application is restarted, the OS activates the configured `OsRestartTask`.	

Scalability Class	The features of the OS (e.g. Memory Protection or Timing Protection), described by this document, can be grouped together to customize the operating system to the needs of the application. There are 4 defined groups of features which are named scalability classes. For details see Chapter 7.11	
Schedule Table	Encapsulation of a statically defined set of expiry points.	
Section	Part of an object file in which instructions or data are combined to form a unit (contiguous address space in memory allocated for data or code). A section in an object file (object file format) has a name and a size. From the linker perspective, two different sides can be distinguished:	
	Input section	memory section in an input object file of the linker.
	Output section	memory section in an output object file of the linker.
Set (of OS objects)	This document uses the term set, indicating a collection of the same type of OS objects, in the strict mathematical sense, i.e.: - a set contains zero or more OS objects (this means a set can be empty) - the OS objects in the set are unique (this means there cannot be duplicate OS objects in the set)	
Spinlock	A spinlock is a locking mechanism where the TASK waits in a loop ("spins") repeatedly checking for a shared variable to become a certain value. The value indicates whether the lock is free or not. In Multi-Core systems the comparison and changing of the variable typically requires an atomic operation. As the TASK remains active but is not doing anything useful, a spinlock is a busy waiting mechanism	
Spinlock variable	A spinlock variable is a shared variable used by a spinlock to indicate whether a spinlock is free or occupied.	
Symbol	Address label that can be imported/used by software modules and resolved by the linker. The precise syntax of the labels is linker-specific. Here, these address labels are used to identify the start and end of memory sections.	
	Start symbol	Tags the start of a memory section
	End symbol	Tags the end of a memory section
Synchronization of schedule tables with a synchronization counter	Synchronization with a synchronization counter is achieved, if the expiry points of the schedule table are processed within an absolute deviation from the synchronization counter that is smaller than or equal to a precision threshold.	
Synchronization Counter	The “Synchronization Counter”, distinct from an OS counter object, is an external counter, external to the OS, against which expiry points of a schedule table are synchronized	
Task	A Task is the object which executes (user) code and which is managed by the OS. E.g. the OS switches between different Tasks (“schedules”). There are 2 types of Tasks; for more details see [15].	
	Basic Task	A Task which can not block by itself. This means that it can not wait for (OS) event(s).
	Extended Task	A Task which can block by itself and wait for (OS) event(s).
Time Frame	The minmum inter-arrival time for a Task/ISR.	
Trusted Function	A service provided by a trusted OS-Application that can be used by other OS-Applications (trusted or non-trusted).	
Worst case execution time (WCET)	The longest possible execution time.	
Write access	Storing a value in a register or memory location. All memory accesses that have the consequence of writing (e.g. reads that have the side effect of writing to a memory location) are treated as write accesses.	

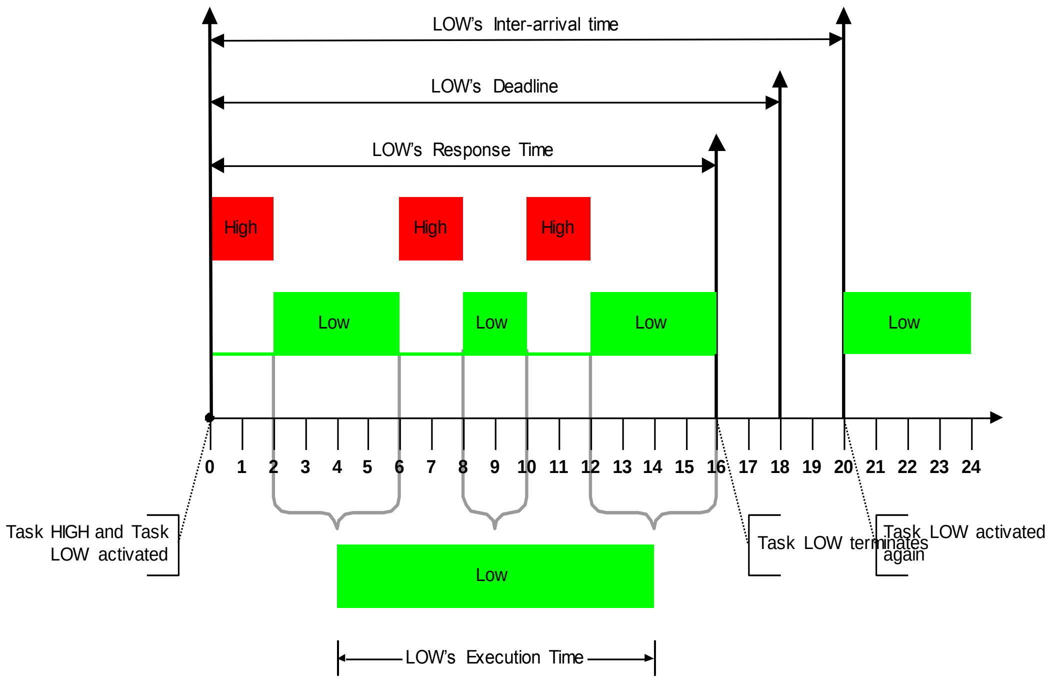

Figure 2.1: Definition of Timing Terminology

3 Related documentation

3.1 Input documents

[1] Layered Software Architecture
AUTOSAR_EXP_LayeredSoftwareArchitecture.pdf

[2] Requirements on Operating System
AUTOSAR_SRS_OS.pdf

[3] General Requirements on Basic Software Modules
AUTOSAR_SRS_BSWGeneral.pdf

[4] Specification of the Virtual Functional Bus
AUTOSAR_EXP_VFB.pdf

[5] Requirements on Software FreeRunningTimer
AUTOSAR_SRS_FreeRunningTimer.pdf

[6] Specification of GPT Driver
AUTOSAR_SWS_GPTDriver.pdf

[7] Specification of Standard Types
AUTOSAR_SWS_StandardTypes.pdf

[8] Specification of Memory Mapping
AUTOSAR_SWS_MemoryMapping.pdf

[9] Specification of RTE
AUTOSAR_SWS_RTE.pdf

[10] Specification of ECU Configuration
AUTOSAR_TPS_ECUConfiguration.pdf

[11] Basic Software Module Description Template
AUTOSAR_TPS_BSWModuleDescriptionTemplate.pdf

[12] List of Basic Software Modules
AUTOSAR_TR_BSWModuleList.pdf

[13] Specification of RTE
AUTOSAR_SWS_RTE.pdf

[14] General Specification of Basic Software Modules
AUTOSAR_SWS_BSWGeneral.pdf

3.2 Related standards and norms

3.2.1 ISO 17356

The ISO 17356 (“Road vehicles -- Open interface for embedded automotive applications”) is a standard which was previously published by the OSEK/VDX organization.

[15] ISO 17356-3: 2005: Road vehicles -- Open interface for embedded automotive applications -- Part 3: OSEK/VDX Operating System (OS)

[16] ISO 17356-6:2006: Road vehicles -- Open interface for embedded automotive applications -- Part 6: OSEK/VDX Implementation Language (OIL)

3.3 Company Reports, Academic Work, etc.

[17] Extensions of OSEK OS for Protected Applications
OSEK Support Project DC058_02
DaimlerChrysler AG

3.4 Related specification

AUTOSAR provides a General Specification on Basic Software modules [14] (SWS BSW General), which is also valid for Operating System.

Thus, the specification SWS BSW General shall be considered as additional and required specification for Operating System.

4 Constraints and assumptions

4.1 Existing Standards

This document makes the following assumptions about the referenced related standards and norms:

- [15] provides a sufficiently flexible scheduling policy to schedule AUTOSAR systems.
- [15] is a mature specification and implementations are used in millions of ECUs worldwide.
- [15] does not provide sufficient support for isolating multi-source software components at runtime.
- [15] does not provide sufficient runtime support for demonstrating the absence of some classes of fault propagation in a safety-case.

4.2 Terminology

The specification uses the following operators when requirements specify multiple terms:

NOT : negation of a single term e.g. NOT Weekend
AND : conjunction of two terms e.g. Weekend AND Saturday
OR : disjunction of two terms e.g. Monday OR Tuesday

A requirement comprising multiple terms is evaluated left to right.

The precedence rules are:

Highest Precedence	**NOT**
Lowest Precedence	**AND OR**

The expression NOT X AND Y means (NOT X) AND (Y)

Where operators of the same precedence are used in the same sentence, commas are used to disambiguate. The expression X AND Y, OR Z means (X AND Y) OR Z.

4.3 Interaction with the RTE

The configuration of an AUTOSAR system [4] maps the »runnables« of a »software component« to (one or more) tasks that are scheduled by the operating system. All runnables in a task share the same protection boundary. In AUTOSAR, a software component must not include an interrupt handler. A software component is therefore implemented as runnables executing within the body of a task, or set of tasks, only.

Runnables get access to hardware-sourced data through the AUTOSAR RTE. The RTE provides the runtime interface between runnables and the basic software modules. The basic software modules also comprise a number of tasks and ISRs that are scheduled by the operating system.

It is assumed that the software component templates and the description of the basic software modules provide sufficient information about the required runtime behavior to be able to specify the attributes of tasks required to configure the OS.

4.4 Operating System Abstraction Layer (OSAL)

Systems that do not use the OS defined in AUTOSAR can provide a platform for the execution of AUTOSAR software components using an Operating System Abstraction Layer. The interface to the OSAL is exactly that defined for the AUTOSAR OS.

4.5 Multi-Core Hardware assumptions

There are currently several existing and suggested HW-architectures[1] for Multi-Core microprocessors. There is considerable variation in the features offered by these architectures. Therefore this section attempts to capture a common set of architectural features required for Multi-Core.
Hardware assumptions shall remain assumptions and shall not become official Autosar requirements.

4.5.1 CPU Core features

1. More than one core on the same piece of silicon.

2. The HW offers a method that can be used by the SW to identify a core.

3. The hardware supports atomic read and atomic write operations for a fixed word length depending on the hardware.

4. The hardware supports some atomic Test-And-Set functionality or similar functionalities that can be used to built a critical section shared between cores. Additional atomic operations may exist.

5. The cores may have the same instruction set; at least a common basic instruction set is available on all cores. Core specific add-ons may exist but they are not taken into account.

6. The cores have the same data representation. For example, the same size of integer, same byte and bit order, etc.

7. If per-core caches exist, AUTOSAR requires support for RAM - cache coherency in HW or in SW. In software means that the cache-controller can be

[1] In this context “architecture” encompasses: the connections between cores and memory, and to peripherals and how interrupts work.

programmed by the SW in a way that it invalidates cache lines or excludes certain memory regions from caching.

8. In case of an exception (such as an illegal memory reference or divide by zero) the exception occurs on the core that introduced the exception.

9. For notification purposes, it is possible to trigger an interrupt/trap on any core.

4.5.2 Memory features

Shared RAM is available to all cores; at least all cores can share a substantial part of the memory.

Flash shall be shared between all cores at least. However, performance can be improved if Flash/RAM can be partitioned so that there are separate pathways from cores to Flash.

A single address space is assumed, at least in the shared parts of the memory address space.

The AUTOSAR Multi-Core architecture shall be capable to run on systems that do and do not support memory protection. If memory protection exists, all cores are covered by a hardware based memory protection.

4.5.3 Multi-Core Limitations

- In AUTOSAR R4.0, it is not supported to activate additional cores under control of AUTOSAR after the Operating System was started.
- The scheduling algorithm does not assign TASKs dynamically to cores.
- The AUTOSAR OS RESOURCE algorithm is not supported across cores. RESOURCES can be used locally, between TASKs that are bound to the same core but not between TASKs/ISRs which are bound to different cores.

4.6 Limitations

4.6.1 Hardware

The core AUTOSAR operating system assumes free access to hardware resources, which are managed by the OS itself. This includes, but is not limited to, the following hardware:

- interrupt control registers
- processor status words
- stack pointer(s)

Specific (extended) features of the core operating system extend the requirements on hardware resource. The following list outlines the features that have requirements on the hardware. Systems that do not use these OS features do not have these hardware requirements.

- Memory Protection: A hardware memory protection unit is required. All memory accesses that have the consequence of writing (e.g. reads that have the side effect of writing to a memory location) shall be treated as writes.

- Time Protection: Timer Hardware for monitoring execution times and arrival rates.

- »Privileged« and »non-privileged« modes on the MCU: to protect the OS against internal corruption caused by writes to OS controlled registers. This mode must not allow OS-Applications to circumvent protection (e.g. write registers which govern memory protection, write to processor status word etc.). The privileged mode must be under full control of the protected OS which uses the mode internally and to transfer control back and forth from a non-trusted OS-Application to a trusted OS-Application. The microprocessor must support a controlled means which moves a processor into this privileged mode.

- Local/Global Time Synchronization: A global time source is needed.

In general hardware failures in the processor are not detected by the operating system. In the event of hardware failure, correct operation of the OS cannot be guaranteed.

The resources managed by a specific OS implementation have to be defined within the appropriate configuration file of the OS.

4.6.2 Programming Language

The API of the operating system is defined as C function calls or macros. If other languages are used they must adapt to the C interface.

4.6.3 Miscellaneous

The operating system does not provide services for dynamic memory management.

4.7 Applicability to car domains

The operating system has the same design constraints regarding size and scalability under which [15] was designed. The immediate domain of applicability is therefore currently body, chassis and power train ECUs. However, there is no reason that the OS cannot be used to implement ECUs for infotainment applications.

5 Dependencies to other modules

There are no forced dependencies on other modules, however:

- It is assumed that the operating system may use timer units directly to drive counters.
- If the user needs to drive scheduling directly from global time, then a global time interrupt is required.
- If the user needs to synchronize the processing of a schedule table to a global time, the operating system needs to be told the global time using the `SyncScheduleTable()` service.
- The IOC described in this document provides communication between OS-Applications. The IOC generation is based on configuration information which is generated by the RTE generator. On the other hand the RTE uses functions generated by the IOC to transmit data.

5.1 File structure

5.1.1 Code file structure

The code file structure of the Operating system module is not fixed, besides the requirements in the General SRS.

5.1.2 Header file structure

The IOC generator generates an additional header file Ioc.h. Users of the Ioc.h shall include the Ioc.h file. If an implementation of the IOC requires additional header files, it is free to include them. The header files are self-contained, that means they will include all other header files, which they require.

5.1.3 ARTI File Structure (DRAFT)

To support ARTI based debugging and tracing, all source files with ARTI hook macros shall include an “arti.h” file. This file (along with the corresponding arti.c file) will be provided by the ARTI hook implementer, i.e. the tracing tool. When building the final executable, the linker will pull in the compiled arti.c file, too.
The usage of the ARTI hook macros is configurable. If the OS is configured to not use ARTI, the inclusion of “arti.h” may be omitted, and the ARTI hooks macros may be expanded to empty macros (“nothing”).

6 Requirements Traceability

This chapter contains references to requirements of other AUTOSAR documents.

Requirement	Description	Satisfied by
SRS_BSW_00003	All software modules shall provide version and identification information	SWS_Os_00767
SRS_BSW_00006	The source code of software modules above the µC Abstraction Layer (MCAL) shall not be processor and compiler dependent.	SWS_Os_00767
SRS_BSW_00007	All Basic SW Modules written in C language shall conform to the MISRA C 2012 Standard.	SWS_Os_00767
SRS_BSW_00009	All Basic SW Modules shall be documented according to a common standard.	SWS_Os_00767
SRS_BSW_00010	The memory consumption of all Basic SW Modules shall be documented for a defined configuration for all supported platforms.	SWS_Os_00767
SRS_BSW_00161	The AUTOSAR Basic Software shall provide a microcontroller abstraction layer which provides a standardized interface to higher software layers	SWS_Os_00767
SRS_BSW_00162	The AUTOSAR Basic Software shall provide a hardware abstraction layer	SWS_Os_00767
SRS_BSW_00168	SW components shall be tested by a function defined in a common API in the Basis-SW	SWS_Os_00767
SRS_BSW_00170	The AUTOSAR SW Components shall provide information about their dependency from faults, signal qualities, driver demands	SWS_Os_00767
SRS_BSW_00172	The scheduling strategy that is built inside the Basic Software Modules shall be compatible with the strategy used in the system	SWS_Os_00767
SRS_BSW_00301	All AUTOSAR Basic Software Modules shall only import the necessary information	SWS_Os_00767
SRS_BSW_00302	All AUTOSAR Basic Software	SWS_Os_00767

	Modules shall only export information needed by other modules	
SRS_BSW_00305	Data types naming convention	SWS_Os_00767
SRS_BSW_00306	AUTOSAR Basic Software Modules shall be compiler and platform independent	SWS_Os_00767
SRS_BSW_00307	Global variables naming convention	SWS_Os_00767
SRS_BSW_00308	AUTOSAR Basic Software Modules shall not define global data in their header files, but in the C file	SWS_Os_00767
SRS_BSW_00309	All AUTOSAR Basic Software Modules shall indicate all global data with read-only purposes by explicitly assigning the const keyword	SWS_Os_00767
SRS_BSW_00310	API naming convention	SWS_Os_00767
SRS_BSW_00312	Shared code shall be reentrant	SWS_Os_00767
SRS_BSW_00314	All internal driver modules shall separate the interrupt frame definition from the service routine	SWS_Os_00767
SRS_BSW_00318	Each AUTOSAR Basic Software Module file shall provide version numbers in the header file	SWS_Os_00767
SRS_BSW_00321	The version numbers of AUTOSAR Basic Software Modules shall be enumerated according specific rules	SWS_Os_00767
SRS_BSW_00325	The runtime of interrupt service routines and functions that are running in interrupt context shall be kept short	SWS_Os_00767
SRS_BSW_00327	Error values naming convention	SWS_Os_00767
SRS_BSW_00328	All AUTOSAR Basic Software Modules shall avoid the duplication of code	SWS_Os_00767
SRS_BSW_00330	It shall be allowed to use macros instead of functions where source code is used and runtime is critical	SWS_Os_00767
SRS_BSW_00333	For each callback function it shall be specified if it is called from interrupt context or not	SWS_Os_00767
SRS_BSW_00334	All Basic Software Modules shall provide an XML file that	SWS_Os_00767

	contains the meta data	
SRS_BSW_00335	Status values naming convention	SWS_Os_00767
SRS_BSW_00337	Classification of development errors	SWS_Os_00767
SRS_BSW_00339	Reporting of production relevant error status	SWS_Os_00767
SRS_BSW_00342	It shall be possible to create an AUTOSAR ECU out of modules provided as source code and modules provided as object code, even mixed	SWS_Os_00767
SRS_BSW_00344	BSW Modules shall support link-time configuration	SWS_Os_00767
SRS_BSW_00347	A Naming seperation of different instances of BSW drivers shall be in place	SWS_Os_00767
SRS_BSW_00350	All AUTOSAR Basic Software Modules shall allow the enabling/disabling of detection and reporting of development errors.	SWS_Os_00767
SRS_BSW_00351	Encapsulation of compiler specific methods to map objects	SWS_Os_00815
SRS_BSW_00357	For success/failure of an API call a standard return type shall be defined	SWS_Os_00767
SRS_BSW_00358	The return type of init() functions implemented by AUTOSAR Basic Software Modules shall be void	SWS_Os_00767
SRS_BSW_00361	All mappings of not standardized keywords of compiler specific scope shall be placed and organized in a compiler specific type and keyword header	SWS_Os_00767
SRS_BSW_00369	All AUTOSAR Basic Software Modules shall not return specific development error codes via the API	SWS_Os_00767
SRS_BSW_00373	The main processing function of each AUTOSAR Basic Software Module shall be named according the defined convention	SWS_Os_00767
SRS_BSW_00374	All Basic Software Modules shall provide a readable module vendor identification	SWS_Os_00767
SRS_BSW_00375	Basic Software Modules shall	SWS_Os_00767

	report wake-up reasons	
SRS_BSW_00377	A Basic Software Module can return a module specific types	SWS_Os_00767
SRS_BSW_00378	AUTOSAR shall provide a boolean type	SWS_Os_00767
SRS_BSW_00379	All software modules shall provide a module identifier in the header file and in the module XML description file.	SWS_Os_00767
SRS_BSW_00383	The Basic Software Module specifications shall specify which other configuration files from other modules they use at least in the description	SWS_Os_00767
SRS_BSW_00384	The Basic Software Module specifications shall specify at least in the description which other modules they require	SWS_Os_00767
SRS_BSW_00385	List possible error notifications	SWS_Os_00767
SRS_BSW_00386	The BSW shall specify the configuration for detecting an error	SWS_Os_00767
SRS_BSW_00401	Documentation of multiple instances of configuration parameters shall be available	SWS_Os_00767
SRS_BSW_00404	BSW Modules shall support post-build configuration	SWS_Os_00767
SRS_BSW_00405	BSW Modules shall support multiple configuration sets	SWS_Os_00767
SRS_BSW_00406	A static status variable denoting if a BSW module is initialized shall be initialized with value 0 before any APIs of the BSW module is called	SWS_Os_00767
SRS_BSW_00407	Each BSW module shall provide a function to read out the version information of a dedicated module implementation	SWS_Os_00767
SRS_BSW_00409	All production code error ID symbols are defined by the Dem module and shall be retrieved by the other BSW modules from Dem configuration	SWS_Os_00767
SRS_BSW_00410	Compiler switches shall have defined values	SWS_Os_00767
SRS_BSW_00411	All AUTOSAR Basic Software Modules shall apply a naming rule for enabling/disabling the existence of the API	SWS_Os_00767

SRS_BSW_00413	An index-based accessing of the instances of BSW modules shall be done	SWS_Os_00767
SRS_BSW_00414	Init functions shall have a pointer to a configuration structure as single parameter	SWS_Os_00767
SRS_BSW_00415	Interfaces which are provided exclusively for one module shall be separated into a dedicated header file	SWS_Os_00767
SRS_BSW_00417	Software which is not part of the SW-C shall report error events only after the DEM is fully operational.	SWS_Os_00767
SRS_BSW_00419	If a pre-compile time configuration parameter is implemented as "const" it should be placed into a separate c-file	SWS_Os_00767
SRS_BSW_00422	Pre-de-bouncing of error status information is done within the DEM	SWS_Os_00767
SRS_BSW_00423	BSW modules with AUTOSAR interfaces shall be describable with the means of the SW-C Template	SWS_Os_00767
SRS_BSW_00437	Memory mapping shall provide the possibility to define RAM segments which are not to be initialized during startup	SWS_Os_00767
SRS_BSW_00439	Enable BSW modules to handle interrupts	SWS_Os_00767
SRS_BSW_00440	The callback function invocation by the BSW module shall follow the signature provided by RTE to invoke servers via Rte_Call API	SWS_Os_00767
SRS_BSW_00441	Naming convention for type, macro and function	SWS_Os_00767
SRS_Frt_00020	The configuration and initialization shall be performed by the module providing the SWFRT functionality (OS) if the GPT Timer is not used .	SWS_Os_00374
SRS_Frt_00022	It shall be possible to state which HW Timer is used	SWS_Os_00370
SRS_Frt_00025	Access methods to time information shall be provided for different users.	SWS_Os_00383, SWS_Os_00392
SRS_Frt_00030	The read - out value shall start with Zero	SWS_Os_00384

SRS_Frt_00031	The SWFRT shall increment i.e.	SWS_Os_00384
SRS_Frt_00032	Wrap around shall work without software interaction.	SWS_Os_00767
SRS_Frt_00033	There shall be a function to achieve an atomic read the of the timer's value.	SWS_Os_00377
SRS_Frt_00034	The module shall provide functionality to calculate the ticks elapsed between a previously stored value (passed as a parameter) and the current timer value.	SWS_Os_00382
SRS_Frt_00047	The SWFRT shall provide a "user" dependent API (function / macro) to convert ticks to time.	SWS_Os_00393
SRS_Os_00097	The OS shall provide an API that is backward compatible to the API of OSEK OS	SWS_Os_00001
SRS_Os_00098	The Operating System shall provide statically configurable schedule tables based on time tables as an optional service	SWS_Os_00002, SWS_Os_00007
SRS_Os_00099	The Operating System shall provide a mechanism which allows switching between different schedule tables	SWS_Os_00191
SRS_Os_11000	The OS may offer support to protect the memory sections of an OS-Application against read accesses by all other OS-Applications	SWS_Os_00026
SRS_Os_11001	The OS shall provide partitions which allow for fault isolation and fault recovery capabilities	SWS_Os_00056
SRS_Os_11002	The operating system shall provide the ability to synchronize the processing of schedule tables with a global system time base	SWS_Os_00013, SWS_Os_00199, SWS_Os_00201, SWS_Os_00206, SWS_Os_00227
SRS_Os_11003	The operating system shall be able to monitor stack usage and check for a stack overflow on a per executable object basis	SWS_Os_00067, SWS_Os_00068
SRS_Os_11005	The operating system shall prevent an OS-Application from modifying the memory of other OS-Applications	SWS_Os_00195, SWS_Os_00207, SWS_Os_00208, SWS_Os_00795, SWS_Os_00806, SWS_Os_00807, SWS_Os_91010, SWS_Os_91011, SWS_Os_91012, SWS_Os_91013, SWS_Os_91014, SWS_Os_91015,

		SWS_Os_91016, SWS_Os_91017, SWS_Os_91018
SRS_Os_11006	The operating system shall allow tasks and ISRs within an OS-Application to exchange data	SWS_Os_00086, SWS_Os_00087, SWS_Os_00196
SRS_Os_11007	The operating system shall allow OS-Applications to execute shared code	SWS_Os_00081
SRS_Os_11008	The OS shall not allow a timing fault in any OS-Application to propagate	SWS_Os_00028, SWS_Os_00033, SWS_Os_00037, SWS_Os_00048, SWS_Os_00064, SWS_Os_00089, SWS_Os_00465, SWS_Os_00469, SWS_Os_00470, SWS_Os_00471, SWS_Os_00472, SWS_Os_00473, SWS_Os_00474
SRS_Os_11009	The operating system shall prevent the corruption of the OS by any call of a system service	SWS_Os_00051, SWS_Os_00052, SWS_Os_00069, SWS_Os_00070, SWS_Os_00088, SWS_Os_00092, SWS_Os_00093
SRS_Os_11010	The operating system shall prevent an OS-Application modifying OS objects that are not owned by that OS-Application	SWS_Os_00056
SRS_Os_11011	The OS shall protect itself against OS-Applications attempting to modify control registers directly which are managed by the OS	SWS_Os_00096, SWS_Os_00245, SWS_Os_00808, SWS_Os_00809, SWS_Os_00810, SWS_Os_00811, SWS_Os_00812, SWS_Os_00813, SWS_Os_00814, SWS_Os_91019, SWS_Os_91020, SWS_Os_91021
SRS_Os_11012	The OS shall provide scalability for its protection features	SWS_Os_00240, SWS_Os_00241
SRS_Os_11013	The OS shall be capable of notifying the occurrence of a protection error at runtime	SWS_Os_00033, SWS_Os_00037, SWS_Os_00044, SWS_Os_00051, SWS_Os_00056, SWS_Os_00064, SWS_Os_00068, SWS_Os_00070, SWS_Os_00088, SWS_Os_00093, SWS_Os_00210, SWS_Os_00246
SRS_Os_11014	In case of a protection error, the OS shall provide an action for recovery on OS-, OS-Application and task/ISR-level	SWS_Os_00033, SWS_Os_00037, SWS_Os_00106, SWS_Os_00107, SWS_Os_00108, SWS_Os_00109, SWS_Os_00110, SWS_Os_00243, SWS_Os_00244
SRS_Os_11016	The OS implementation shall offer scalability which is configurable by a generation tool	SWS_Os_00240, SWS_Os_00241
SRS_Os_11018	The OS shall provide interrupt mask functions	SWS_Os_00299
SRS_Os_11019	The AUTOSAR OS generation	SWS_Os_00336

	tool shall create the interrupt vector table	
SRS_Os_11020	The OS shall provide a standard interface to tick a software counter	SWS_Os_00286
SRS_Os_11021	The OS shall provide a mechanism to cascade multiple software counters from a single hardware counter.	SWS_Os_00301
SRS_Os_80001	The OS shall be able to manage multiple closely coupled CPU Cores	SWS_Os_00568, SWS_Os_00569, SWS_Os_00579, SWS_Os_00583, SWS_Os_00596, SWS_Os_00600, SWS_Os_00606, SWS_Os_00616, SWS_Os_00626, SWS_Os_00627, SWS_Os_00628, SWS_Os_00672, SWS_Os_00673, SWS_Os_00674, SWS_Os_00675
SRS_Os_80003	The multi core extension shall provide the same degree of predictability as the single core	SWS_Os_00570, SWS_Os_00571, SWS_Os_00573
SRS_Os_80005	OsApplications and as a result TASKS and OsISRs shall be assigned statically to cores	SWS_Os_00570, SWS_Os_00571, SWS_Os_00572, SWS_Os_00573, SWS_Os_00667
SRS_Os_80006	Initialization/Start-up of the system shall be synchronized	SWS_Os_00572, SWS_Os_00574, SWS_Os_00575, SWS_Os_00576, SWS_Os_00577, SWS_Os_00578, SWS_Os_00579, SWS_Os_00580, SWS_Os_00581, SWS_Os_00582, SWS_Os_00584, SWS_Os_00585, SWS_Os_00607, SWS_Os_00608, SWS_Os_00609, SWS_Os_00610, SWS_Os_00625, SWS_Os_00668, SWS_Os_00669, SWS_Os_00670, SWS_Os_00676, SWS_Os_00677, SWS_Os_00678, SWS_Os_00679, SWS_Os_00680, SWS_Os_00681, SWS_Os_00682, SWS_Os_00683, SWS_Os_00684, SWS_Os_00685
SRS_Os_80007	Shutdown procedure shall be triggered by any core	SWS_Os_00586, SWS_Os_00587, SWS_Os_00588, SWS_Os_00616, SWS_Os_00617, SWS_Os_00621, SWS_Os_00713, SWS_Os_00714, SWS_Os_00715, SWS_Os_00716
SRS_Os_80008	It shall be a common OS configuration across multiple cores	SWS_Os_00567, SWS_Os_00582
SRS_Os_80011	The number of cores that the operating system manages shall be configurable offline	SWS_Os_00583
SRS_Os_80013	The behaviour of services shall be identical to single core systems	SWS_Os_00569, SWS_Os_00589, SWS_Os_00590, SWS_Os_00591, SWS_Os_00592, SWS_Os_00593,

		SWS_Os_00594, SWS_Os_00595, SWS_Os_00607, SWS_Os_00618, SWS_Os_00619, SWS_Os_00623, SWS_Os_00629, SWS_Os_00630, SWS_Os_00631, SWS_Os_00635, SWS_Os_00636, SWS_Os_00637, SWS_Os_00638, SWS_Os_00639, SWS_Os_00640, SWS_Os_00643, SWS_Os_00645, SWS_Os_00646, SWS_Os_00647, SWS_Os_00663, SWS_Os_00664, SWS_Os_00665
SRS_Os_80015	The MC extensions shall provide a mechanism to activate tasks on different cores	SWS_Os_00596, SWS_Os_00598, SWS_Os_00599, SWS_Os_00600, SWS_Os_00816, SWS_Os_00818, SWS_Os_00819, SWS_Os_91022, SWS_Os_91023
SRS_Os_80016	Event mechanism shall work across cores	SWS_Os_00602, SWS_Os_00604, SWS_Os_00605, SWS_Os_00817
SRS_Os_80018	A method to synchronize tasks on more than one core shall be provided	SWS_Os_00632, SWS_Os_00633, SWS_Os_00634, SWS_Os_00641, SWS_Os_00642, SWS_Os_00644, SWS_Os_00648, SWS_Os_00649, SWS_Os_00650, SWS_Os_00652, SWS_Os_00653, SWS_Os_00654, SWS_Os_00655, SWS_Os_00656, SWS_Os_00657, SWS_Os_00658, SWS_Os_00659, SWS_Os_00660, SWS_Os_00661
SRS_Os_80020	A data exchange mechanism shall be provided	SWS_Os_00611, SWS_Os_00671, SWS_Os_00718, SWS_Os_00719, SWS_Os_00720, SWS_Os_00721, SWS_Os_00722, SWS_Os_00723, SWS_Os_00724, SWS_Os_00725, SWS_Os_00726, SWS_Os_00727, SWS_Os_00728, SWS_Os_00729, SWS_Os_00730, SWS_Os_00731, SWS_Os_00732, SWS_Os_00733, SWS_Os_00734, SWS_Os_00735, SWS_Os_00736, SWS_Os_00737, SWS_Os_00738, SWS_Os_00739, SWS_Os_00740, SWS_Os_00741, SWS_Os_00742, SWS_Os_00743, SWS_Os_00744, SWS_Os_00745, SWS_Os_00746, SWS_Os_00747, SWS_Os_00748, SWS_Os_00749, SWS_Os_00750, SWS_Os_00751, SWS_Os_00752, SWS_Os_00753, SWS_Os_00754, SWS_Os_00755, SWS_Os_00756, SWS_Os_00757, SWS_Os_00758, SWS_Os_00759, SWS_Os_00760, SWS_Os_00761, SWS_Os_00803, SWS_Os_00805
SRS_Os_80021	The MC extension of the AUTOSAR environment shall support a mutual exclusion mechanism between cores	SWS_Os_00612, SWS_Os_00613, SWS_Os_00614, SWS_Os_00615, SWS_Os_00620, SWS_Os_00622, SWS_Os_00624, SWS_Os_00648,

	that shall not cause deadlocks	SWS_Os_00649, SWS_Os_00650, SWS_Os_00651, SWS_Os_00652, SWS_Os_00653, SWS_Os_00654, SWS_Os_00655, SWS_Os_00656, SWS_Os_00657, SWS_Os_00658, SWS_Os_00659, SWS_Os_00660, SWS_Os_00661, SWS_Os_00666, SWS_Os_00686, SWS_Os_00687, SWS_Os_00688, SWS_Os_00689, SWS_Os_00690, SWS_Os_00691, SWS_Os_00692, SWS_Os_00693, SWS_Os_00694, SWS_Os_00695, SWS_Os_00696, SWS_Os_00697, SWS_Os_00698, SWS_Os_00699, SWS_Os_00700, SWS_Os_00701, SWS_Os_00703, SWS_Os_00704, SWS_Os_00705, SWS_Os_00706, SWS_Os_00707, SWS_Os_00708, SWS_Os_00709, SWS_Os_00710, SWS_Os_00711, SWS_Os_00712, SWS_Os_00792, SWS_Os_00801
SRS_Os_80023	The OS shall execute an operation which can be selected at runtime, in case no task is going to be scheduled on a specific core	SWS_Os_00770, SWS_Os_00771, SWS_Os_00802
SRS_Os_80026	It shall be possible to start any of the cores in a multi core system	SWS_Os_00574, SWS_Os_00575, SWS_Os_00576, SWS_Os_00577, SWS_Os_00584, SWS_Os_00585, SWS_Os_00676, SWS_Os_00677, SWS_Os_00678, SWS_Os_00679, SWS_Os_00680, SWS_Os_00681, SWS_Os_00682, SWS_Os_00683, SWS_Os_00684, SWS_Os_00685
SRS_Os_80027	It shall be possible to initialize any of the cores in a multi core system	SWS_Os_00574, SWS_Os_00575, SWS_Os_00576, SWS_Os_00577, SWS_Os_00584, SWS_Os_00585, SWS_Os_00676, SWS_Os_00677, SWS_Os_00678, SWS_Os_00679, SWS_Os_00680, SWS_Os_00681, SWS_Os_00682, SWS_Os_00683, SWS_Os_00684, SWS_Os_00685

7 Functional specification

7.1 Core OS

7.1.1 Background & Rationale

The OSEK/VDX Operating System [15] is widely used in the automotive industry and has been proven in use in all classes of ECUs found in modern vehicles. The concepts that OSEK OS has introduced are widely understood and the automotive industry has many years of collective experience in engineering OSEK OS based systems.

OSEK OS is an event-triggered operating system. This provides high flexibility in the design and maintenance of AUTOSAR based systems. Event triggering gives freedom for the selection of the events to drive scheduling at runtime, for example angular rotation, local time source, global time source, error occurrence etc.

For these reasons the core functionality of the AUTOSAR OS shall be based upon the OSEK OS. In particular OSEK OS provides the following features to support concepts in AUTOSAR:

fixed priority-based scheduling
facilities for handling interrupts
only interrupts with higher priority than tasks
some protection against incorrect use of OS services
a startup interface through `StartOS()` and the `StartupHook()`
a shutdown interface through `ShutdownOS()` and the `ShutdownHook()`

OSEK OS provides many features in addition to these. Readers should consult the specification [15] for details.

Basing AUTOSAR OS on OSEK OS means that legacy applications will be backward compatible – i.e. applications written for OSEK OS will run on AUTOSAR OS. However, some of the features introduced by AUTOSAR OS require restrictions on the use of existing OSEK OS features or extend existing OSEK OS features.

7.1.2 Requirements

[SWS_Os_00001] ⌈The Operating System module shall provide an API that is backward compatible with the OSEK OS API [15]. ⌋ (SRS_Os_00097)

7.1.2.1 Restrictions on OSEK OS

It is too inefficient to achieve timing and memory protection for alarm callbacks. They are therefore not allowed in specific scalability classes (SWS_Os_00242)

[SWS_Os_00242] ⌈The Operating System module shall only allow Alarm Callbacks in Scalability Class 1. ⌋ ( )

OSEK OS is required to provide functionality to handle inter-task (internal) communication according to the OSEK COM specification when internal communication only is required in the system. In AUTOSAR, internal communication is provided by the AUTOSAR RTE or by AUTOSAR COM at least one of which will be present for all AUTOSAR ECUs.

AUTOSAR OS, when used in an AUTOSAR system, therefore does not need to support internal communication.

An OSEK OS must implement internal communication if the symbol `LOCALMESSAGESONLY` is defined. AUTOSAR OS can deprecate the need to implement OSEK COM functionality and maintain compatibility with OSEK suite of specifications by ensuring that AUTOSAR OS always exists in an environment where `LOCALMESSAGESONLY` is undefined.

OSEK OS has one special resource called `RES_SCHEDULER`. This resource has 2 specific aspects:

1. It is always present in the system, even if it is not configured. This means that the `RES_SCHEDULER` is always known by the OS.
2. It has always the highest Task priority. This means a Task which allocates this resource can not be preempted by other Tasks.

Since special cases are always hard to handle (e.g. in this case with respect to timing protection) AUTOSAR OS handles `RES_SCHEDULER` as any other resource. This means that the `RES_SCHEDULER` is not automatically created. However, a configuration attribute allows that a resource in AUTOSAR OS can optionally be assigned the priority of the highest priority task in the system.

For backwards compatibility with OSEK OS systems, see Chapter 12.7 on how to configure a standard resource called `RES_SCHEDULER` in a way that make it compatible with the resource of the same name which is declared automatically in OSEK OS.

In OSEK OS users must declare Operating System objects with specific macros (e.g. DeclareTask(), …) An AUTOSAR OS implementation shall not depend on such declarations and shall (for backwards compatibility) supply macros without functionality.

7.1.2.2 Undefined Behaviour in OSEK OS

There are a number of cases where the behaviour of OSEK OS is undefined. These cases represent a barrier to portability. AUTOSAR OS tightens the OSEK OS specification by defining the required behaviour.

[SWS_Os_00304] ⌈If in a call to `SetRelAlarm()` the parameter “increment” is set to zero, the service shall return `E_OS_VALUE` in standard and extended status . ⌋ ( )

[SWS_Os_00424] ⌈The first call to `StartOS()` (for starting the Operating System) shall not return. ⌋ ( )

[SWS_Os_00425] ⌈If `ShutdownOS()` is called and `ShutdownHook()` returns then the Operating System module shall disable all interrupts and enter an endless loop. ⌋ ( )

7.1.2.3 Extensions to OSEK OS

[SWS_Os_00299] ⌈The Operating System module shall provide the services `DisableAllInterrupts()`, `EnableAllInterrupts()`, `SuspendAllInterrupts()`, `ResumeAllInterrupts()` prior to calling `StartOS()` and after calling `ShutdownOS()`.⌋ (SRS_Os_11018)

It is assumed that the static variables of the functions mentioned in SWS_Os_00299 are initialized.

[SWS_Os_00301] ⌈The Operating System module shall provide the ability to increment a software counter as an alternative action on alarm expiry. ⌋ (SRS_Os_11021)

The Operating System module provides API service `IncrementCounter()` (see SWS_Os_00399) to increment a software counter.

[SWS_Os_00476] ⌈The Operating System module shall allow to automatically start preconfigured absolute alarms during the start of the Operating System. ⌋ ( )

SWS_Os_00476 is an extension to OSEK OS which allows this only for relative alarms.

[SWS_Os_00566] ⌈The Operating System API shall check in extended mode all pointer arguments for a `NULL` pointer and return `E_OS_PARAM_POINTER` in extended status if such an argument is `NULL`. ⌋ ( )

7.2 Software Free Running Timer

Due to the fact that the number of timers is often very limited, some functionality and configuration is added to extend the reuse of timers. E.g. this allows timer measurements. For more details see also [5] (SWFRT).

[SWS_Os_00374] ⌈The Operating System module shall handle all the initialization and configuration of timers used directly by the Operating System module and not handled by the GPT driver. ⌋ (SRS_Frt_00020)

The Operating System module provides API service `GetCounterValue()` (see SWS_Os_00383) to read the current count value of a counter (returning either the hardware timer ticks if counter is driven by hardware or the software ticks when user drives counter).

The Operating System module provides API service `GetElapsedValue()` (see SWS_Os_00392) to get the number of ticks between the current tick value and a previously read tick value.

[SWS_Os_00384] ⌈The Operating System module shall adjust the read out values of hardware timers (which drive counters) in such that the lowest value is zero and consecutive reads return an increasing count value until the timer wraps at its modulus. ⌋ (SRS_Frt_00030, SRS_Frt_00031)

7.3 Schedule Tables

7.3.1 Background & Rationale

It is possible to implement a statically defined task activation mechanism using an OSEK counter and a series of auto started alarms. In the simple case, this can be achieved by specifying that the alarms are not modified once started. Run-time modifications can only be made if relative synchronization between alarms can be guaranteed. This typically means modifying the alarms while associated counter tick interrupts are disabled.

Schedule Tables address the synchronization issue by providing an encapsulation of a statically defined set of expiry points. Each expiry point defines:

- one or more actions that must occur when it is processed where an action is the activation of a task or the setting of an event.
- An offset in ticks from the start of the schedule table

Each schedule table has a duration in ticks. The duration is measured from zero and defines the modulus of the schedule table.

At runtime, the Operating System module will iterate over the schedule table, processing each expiry point in turn. The iteration is driven by an OSEK counter. It therefore follows that the properties of the counter have an impact on what is possible to configure on the schedule table.

7.3.2 Requirements

7.3.2.1 Structure of a Schedule Table

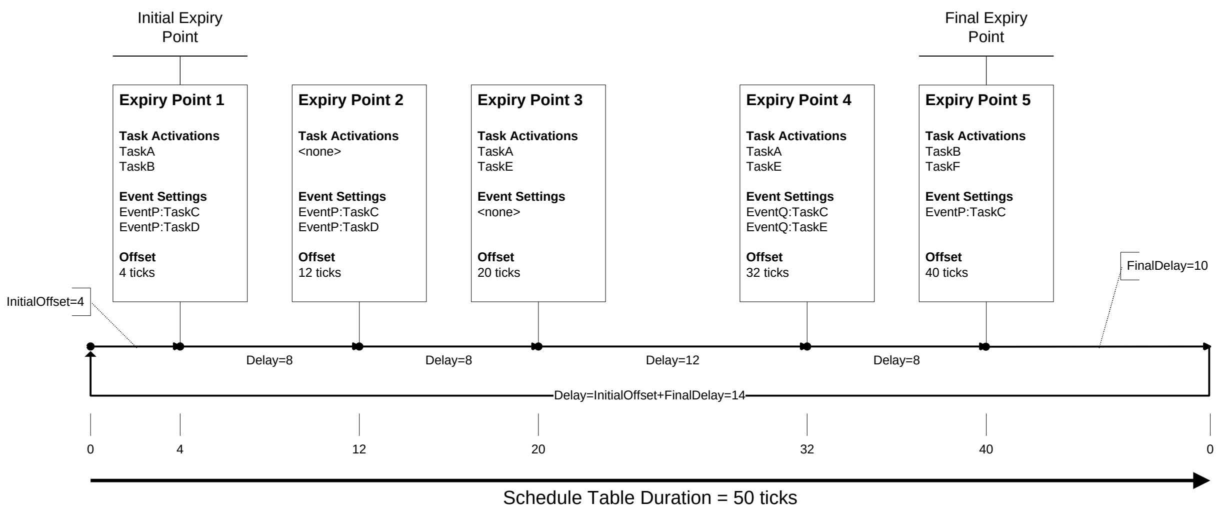

Figure 7.1: Anatomy of a Schedule Table

[SWS_Os_00401] ⌈A schedule table shall have at least one expiry point. ⌋ ( )

[SWS_Os_00402] ⌈An expiry point shall contain a (possibly empty) set of tasks to activate. ⌋ ( )

[SWS_Os_00403] ⌈An expiry point shall contain a (possibly empty) set of events to set. ⌋ ( )

[SWS_Os_00404] ⌈An expiry point shall contain an offset in ticks from the start of the schedule table. ⌋ ( )

7.3.2.2 Constraints on Expiry Points

There is no use case for an empty expiry point, so each one must define at least one action.

[SWS_Os_00407] ⌈An expiry point shall activate at least one task OR set at least one event. ⌋ ( )

The OS needs to know the order in which expiry points are processed. It is therefore necessary to ensure that the expiry points on a schedule table can be totally ordered. This is guaranteed by forcing each expiry point on a schedule table to have a unique offset.

[SWS_Os_00442] : ⌈Each expiry point on a given schedule table shall have a unique offset. ⌋ ( )

Iteration over expiry points on a schedule table is driven by an OSEK counter. The characteristics of the counter – `OsCounterMinCycle` and `OsCounterMaxAllowedValue` – place constraints on expiry point offsets.

[SWS_Os_00443] ⌈The Initial Offset shall be zero OR in the range `OsCounterMinCycle` .. `OsCounterMaxAllowedValue` of the underlying counter. ⌋ ( )

Simlarly, constraints apply to the delays between of adjacent expiry points and the delay to the logical end of the schedule table.

[SWS_Os_00408] ⌈The delay between adjacent expiry points shall be in the range `OsCounterMinCycle` .. `OsCounterMaxAllowedValue` of the underlying counter. ⌋ ( )

7.3.2.3 Processing Schedule Tables

[SWS_Os_00002] ⌈The Operating System module shall process each expiry point on a schedule table from the Initial Expiry Point to the Final Expiry Point in order of increasing offset. ⌋ (SRS_Os_00098)

[SWS_Os_00007] ⌈The Operating System module shall permit multiple schedule tables to be processed concurrently. ⌋ (SRS_Os_00098)

[SWS_Os_00409] ⌈A schedule table of the Operating System module shall be driven by exactly one counter. ⌋ ( )

[SWS_Os_00410] ⌈The Operating System module shall be able to process at least one schedule table per counter at any given time. ⌋ ( )

[SWS_Os_00411] ⌈The Operating System module shall make use of ticks so that one tick on the counter corresponds to one tick on the schedule table. ⌋ ( )

It is possible to activate a task and set (one or more unique) events for the same task at the same expiry point. The ordering of task activations and event settings performed from the expiry point could lead to different implementations exhibiting different behaviour (for example, activating a suspended task and then setting and event on the task would succeed but if the ordering was reversed then the event setting would fail). To prevent such non-determinism, it is necessary to enforce a strict ordering of actions on the expiry point.

[SWS_Os_00412] ⌈The Operating System module shall process all task activations on an expiry point first and then set events. ⌋ ( )

A schedule table always has a defined state and the following figure illustrates the different states (for a non-synchronized schedule table) and the transitions between them.

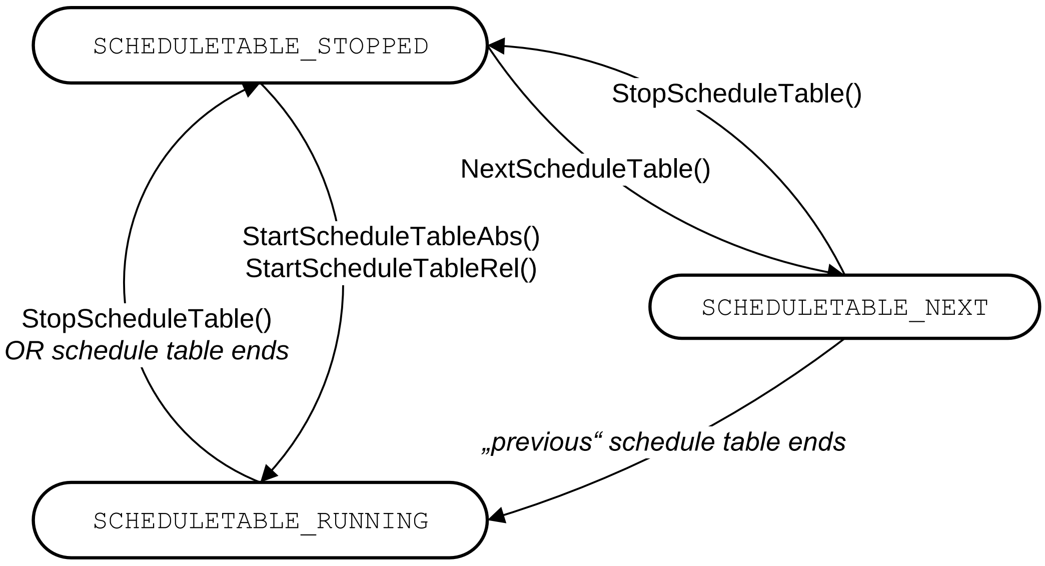

Figure 7.2: States of a schedule table

If a schedule table is not active – this means that is not processed by the Operating System – the state is `SCHEDULETABLE_STOPPED`. After starting a schedule tables enters the `SCHEDULETABLE_RUNNING` state where the OS processes the expiry points. If the service to switch a schedule table is called a schedule table enters the the `SCHEDULETABLE_NEXT` state and waits until the “current” schedule table ends.

7.3.2.4 Repeated Schedule Table Processing

A schedule table may or may not repeat after the final expiry point is processed. This allows two types of behaviour:

1. single-shot – the schedule table processes each expiry point in sequence and then stops at the end. This is useful for triggering a phased sequence of actions in response to some trigger

2. repeating – the schedule table processes each expiry point in turn, After processing the final expiry point, it loops back to the initial expirt point. This is useful for building applications that perform repeated processing or system which need to synchronise processing to a driver source.

A repeating schedule table means that each expiry point is repeated at a period equal to the schedule table duration.

[SWS_Os_00413] ⌈The schedule table shall be configurable as either single-shot or repeating. ⌋ ( )

[SWS_Os_00009] ⌈If the schedule table is single-shot, the Operating System module shall stop the processing of the schedule table Final Delay ticks after the Final Expiry Point is processed. ⌋ ( )

[SWS_Os_00427] ⌈If the schedule table is single-shot, the Operating System module shall allow a Final Delay between 0 .. `OsCounterMaxAllowedValue` of the underlying counter. ⌋ ( )

[SWS_Os_00444] ⌈For periodic schedule tables the value of Final Delay shall be in the range `OsCounterMinCycle` .. `OsCounterMaxAllowedValue` of the underlying counter. ⌋ ( )

[SWS_Os_00194] ⌈After processing the Final Expiry Point, and if the schedule table is repeating, the Operating System shall process the next Initial Expiry Point, after Final Delay plus Initial Offset ticks have elapsed. ⌋ ( )

7.3.2.5 Controlling Schedule Table Processing

The application is responsible for starting and stopping the processing of a schedule table.

The Operating System module provides the service `StartScheduleTableAbs()` (see SWS_Os_00358) to start the processing of a schedule table at an absolute value “Start” on the underlying counter. (The Initial Expiry Point has to be processed when the value of the underlying counter equals Start + InitialOffset).

The Operating System module provides the service `StartScheduleTableRel()` (see SWS_Os_00347) to start the processing of a schedule table at “Offset” relative to the “Now” value on the underlying counter (The Initial Expiry Point shall be processed when the value of the underlying counter equals Now + Offset + InitialOffset).

The figure below illustrates the two different methods for a schedule table driven by a counter with a modulus of 65536 (i.e. an `OsCounterMaxAllowedValue = 65535`).

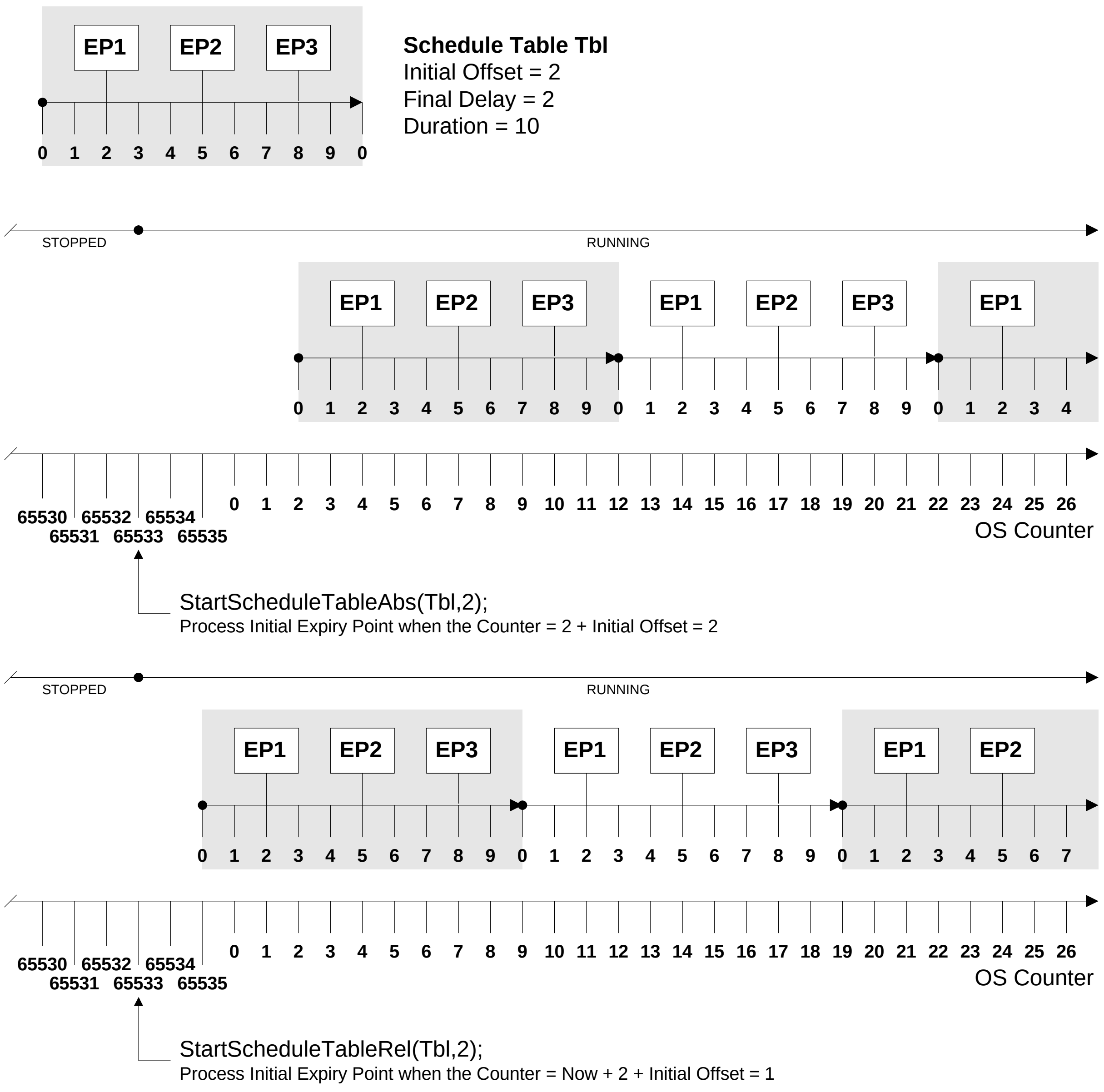
EP1
EP2
EP3
Schedule Table Tbl
Initial Offset = 2
Final Delay = 2
Duration = 10
STOPPED
RUNNING
OS Counter
StartScheduleTableAbs(Tbl,2);
Process Initial Expiry Point when the Counter = 2 + Initial Offset = 2
StartScheduleTableRel(Tbl,2);
Process Initial Expiry Point when the Counter = Now + 2 + Initial Offset = 1

Figure 7.3: Starting a Schedule Table at an Absolute and a Relative Count

The Operating System module provides the service `StopScheduleTable()` (see SWS_Os_00006) to cancel the processing of a schedule table immediately at any point while the schedule table is running.

[SWS_Os_00428] ⌈If schedule table processing has been cancelled before reaching the Final Expiry Point and is subsequently restarted then SWS_Os_00358/SWS_Os_00347 means that the re-start occurs from the start of the schedule table. ⌋ ( )

The Operating System module provides the service `NextScheduleTable()` (see SWS_Os_00191) to switch the processing from one schedule table to another schedule table.

[SWS_Os_00414] ⌈When a schedule table switch is requested, the OS shall continue to process expiry points on the current schedule table. After the Final Expiry Point there will be a delay equivalent to Final Delay ticks before processing the switched-to schedule table. The initial expiry point will be processed after initial offset. ⌋ ( )

The Operating System module provides the service `GetScheduleTableStatus()` (see SWS_Os_00227) to query the state of a schedule table.

Schedule tables can be configured (see chapter 10) to start automatically during start of the Operating System module (like Tasks and Alarms in OSEK OS). OSEK OS defines a specific order: Autostart of Tasks is performed before autostart of alarms. AUTOSAR OS extends this with schedule tables.

[SWS_Os_00510] ⌈The Operating System module shall perform the autostart of schedule tables during startup after the autostart of Tasks and Alarms. ⌋ ( )

7.4 Schedule Table Synchronization

7.4.1 Background & Rationale

The absolute time at which the Initial Expiry Point on a schedule table is processed is under user control. However, if the schedule table repeats then it is not guaranteed that the absolute count value at which the initial expiry point was first processed is the same count value at which it is subsequently processed. This is because the duration of the schedule table need not be equal to the counter modulus.

In many cases it may be important that schedule table expiry points are processed at specific absolute values of the underlying counter. This is called **synchronization**. Typical use-cases include:

- Synchronization of expiry points to degrees of angular rotation for motor management

- Synchronizing the computation to a global (network) time base. Note that in AUTOSAR, the Operating System does not provide a global (network) time source because

  1. a global time may not be needed in many cases
  2. other AUTOSAR modules, most notably FlexRay, provide this independently to the Operating System
  3. if the Operating System is required to synchronize to multiple global (network) time sources (for example when building a gateway between two time-triggered networks) the Operating System cannot be the source of a unique global time.

AUTOSAR OS provides support for synchronization in two ways:

1. implicit synchronization – the counter driving the schedule table is the counter with which synchronization is required. This is typically how synchronization with time-triggered networking technologies (e.g. FlexRay, TTP) is achieved – the underlying hardware manages network time synchronization and simply presents time as an output/compare timer interface to the Operating System. The following figure shows the possible states for schedule tables with implicit synchronization.

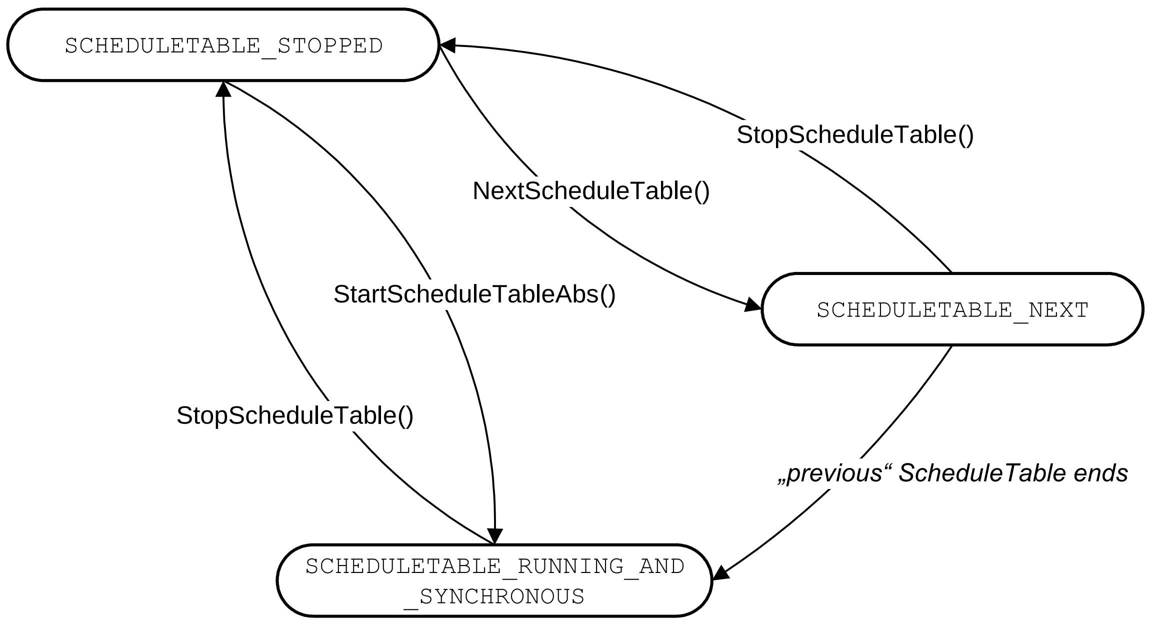

Figure 7.4: States of an implicit synchronized schedule table

2. explicit synchronization – the schedule table is driven by an Operating System counter which is **not** the counter with which synchronization is required. The Operating System provides additional functionality to keep schedule table processing driven by the Operating System counter synchronized with the synchronization counter. This is typically how synchronization with periodically broadcast global times works. The next figure shows the states of such schedule tables.

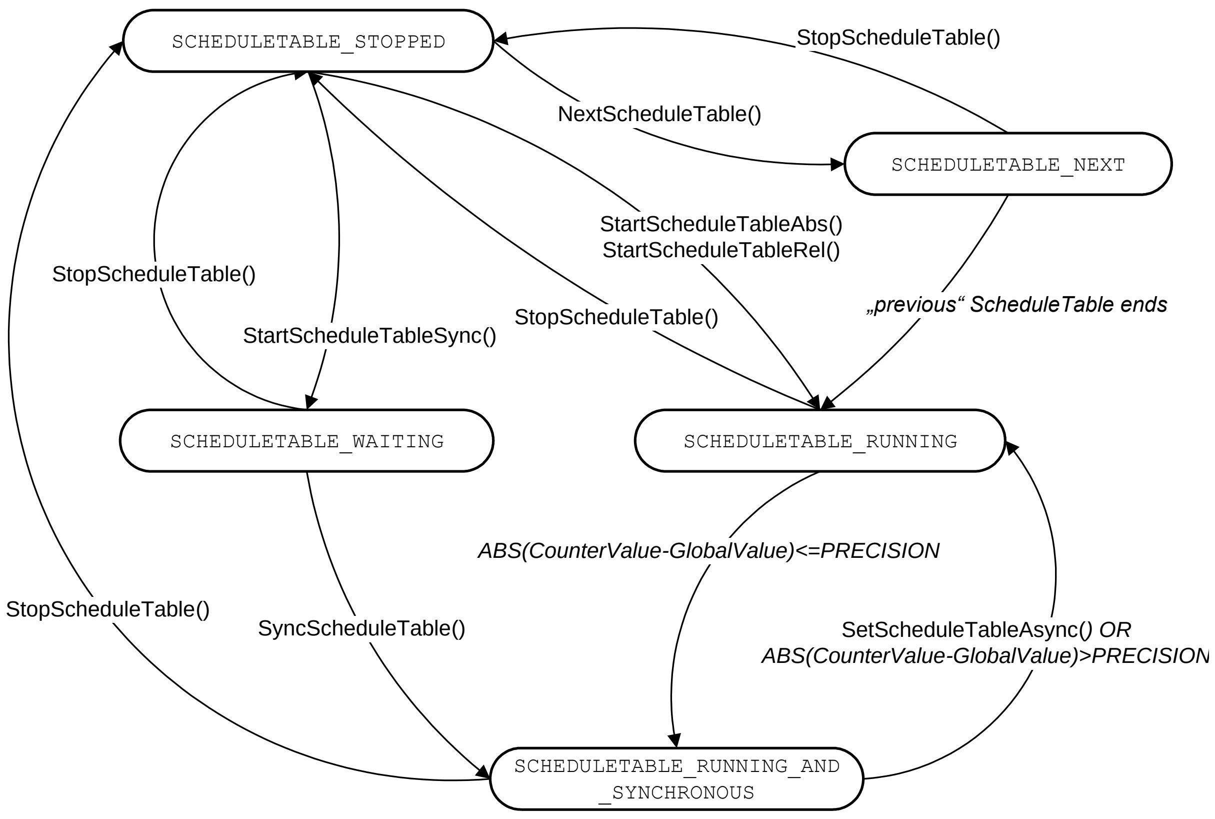

Figure 7.5: States of an explicit synchronized schedule table (not all conditions for transitions are shown in the picture)

7.4.2 Requirements

[SWS_Os_00013] ⌈The Operating System module shall provide the ability to synchronize the processing of schedule table to known counter values. ⌋ (SRS_Os_11002)

7.4.2.1 Implicit Synchronization

The Operating System module does not need to provide any additional support for implicit synchronization of schedule tables. However, it is necessary to constrain configuration and runtime control of the schedule table so that ticks on the configured schedule table can be aligned with ticks on the counter. This requires the range of the schedule table to be identical to the range of the counter (the equality of tick resolution of each is guaranteed by the requirements on the schedule table / counter interaction):

[SWS_Os_00429] ⌈A schedule table of the Operating System module that is implicitly synchronized shall have a Duration equal to `OsCounterMaxAllowedValue + 1` of its associated OSEK OS counter. ⌋ ( )

To synchronize the processing of the schedule table it must be started at a known counter value. The implication of this is that a schedule table requiring implicit synchronization must only be started at an absolute counter value and cannot be started at a relative count value.

[SWS_Os_00430] ⌈The Operating System module shall prevent a schedule table that is implicitly synchronized from being started at a relative count value. ⌋ ( )

When the schedule table is started at an absolute counter value each expiry point will be processed when the counter equals the value specified in the service call plus expiry point's offset. The common use-case is to ensure that the offsets specified in the schedule table configuration correspond to absolute values of the underlying counter. This is achieved trivially using `StartScheduleTableAbs(Tbl,0)` as shown below.

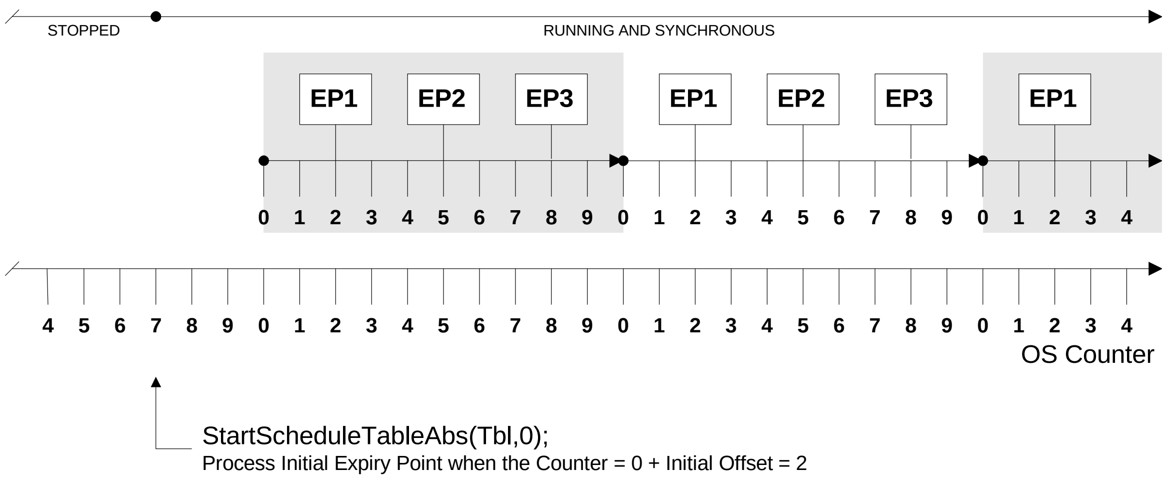

Figure 7.6: Example for implicit synchronized schedule table

7.4.2.2 Explicit Synchonization

An explicitly synchronized schedule table requires additional support from the Operating System module. The schedule table is driven by an Operating System module's counter as normal (termed the "drive counter") but processing needs to be synchronized with a different counter (termed the "synchronization counter") which is not an Operating System module's counter object.

The following constraints must be enforced between the schedule table, the Operating System module's counter and the synchronization counter:

Constraint1:

[SWS_Os_00431] ⌈A schedule table that is explicitly synchronized shall have a duration no greater than modulus of the drive counter. ⌋ ( )

Constraint2:

[SWS_Os_00462] ⌈A schedule table that is explicitly synchronized shall have a duration equal to the modulus of the synchronization counter. ⌋ ( )

Constraint3:

[SWS_Os_00463] ⌈The synchronization counter shall have the same resolution as the drive counter associated with the schedule table. This means that a tick on the schedule table has the same duration as a tick on the synchronization counter. ⌋ ( )

Note that it is in the responsibility of the Operating System module user to verify that Constraints 2 and 3 are satisfied by their system.

The function of explicit synchronization is for the Operating System module to keep processing each expiry point at absolute value of the synchronization counter equal to the expiry point's offset. This means that explicit synchronization always assumes that the notional zero of the schedule table has to be synchronized with absolute value zero on the synchronization counter.

To achieve this, the Operating System module must be told the value of the synchronization counter by the user. As the modulus of the synchronization counter and the schedule table are identical, the Operating System module can use this information to calculate drift. The Operating System module then automatically adjusts the delay between specially configured expiry points, retarding them or advancing them as appropriate, to ensure that synchronization is maintained.

1.1.1 Startup

There are two options for starting an explicitly synchronized schedule table:

Asynchronous start: Start the schedule table at an arbitrary value of the synchronization counter.
Synchronous start: Start the schedule table at absolute value zero of the synchronization counter only after a synchronization count has been provided. This may mean waiting for first synchronization indefinitely.

Asynchronous start is provided by the existing absolute and relative schedule table start services. Both of these services set the point at which the initial expiry point is processed with respect to the driver counter not the synchronization counter. This allows the schedule table to start running before the value of the synchronization counter is known.

Synchronous start requires an additional service that starts the schedule table only after the Operating System module is told the value of the synchronization counter.

The Operating System module provides the service `StartScheduleTableSynchron()` (see SWS_Os_00201) to start an explicitly synchronized schedule table synchronously. The Initial Expiry Point will be processed after (Duration – Value) + Initial Offset ticks of the driver counter have elapsed where Value is the absolute value of the synchronization counter provided to the schedule table.

[SWS_Os_00435] ⌈If an explicitly synchronized schedule table was started synchronously, then the Operating System module shall guarantee that it has state "waiting" when the call of service `StartScheduleTableSynchron()` returns. ⌋ ( )

1.1.2 Providing a Synchronization Count

The Operating System module must be told the value of the synchronization counter. Since the schedule table duration is equal to the modulus of the synchronization counter, the Operating System module can use this to determine the drift between the current count value on the schedule table time and the synchronization count and decide whether (or not) any action to achieve synchronization is required.

The Operating System module provides the service `SyncScheduleTable()` (see SWS_Os_00199) to provide the schedule table with a synchronization count and start synchronization.

1.1.3 Specifying Synchronization Bounds

A schedule table defaults to denying adjustment at all expiry points. Adjustment is allowed only when explicitly configured. The range of adjustment that the Operating System module can make at an adjustable expiry point is controlled by specifying:

- `OsScheduleTableMaxShorten` : the maximum value that can be subtracted from the expiry offset
- `OsScheduleTableMaxLengthen`: the maximum value that can be added to the expiry point offset

The following figure illustrates the behaviour depending on `OsScheduleTableMaxShorten` and `OsScheduleTableMaxLengthen`:

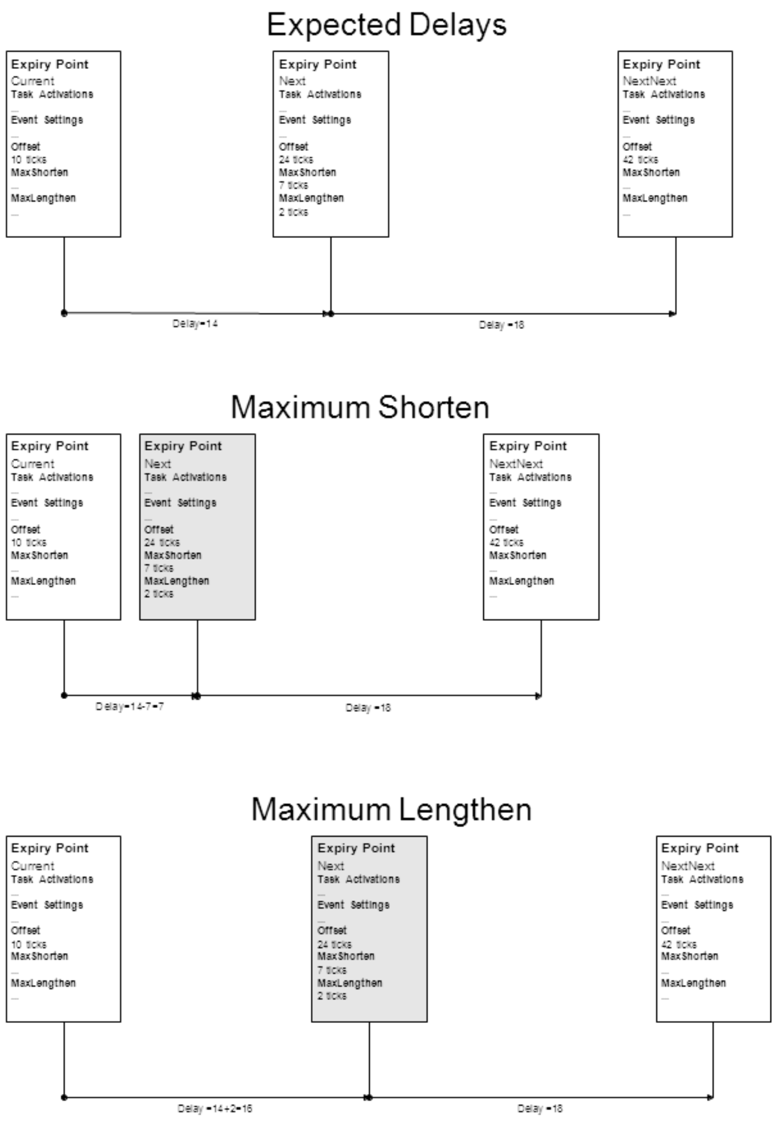

Figure 7.7: Adjustment of Exipry Points

[SWS_Os_00415] ⌈An expiry point shall permit the configuration of a `OsScheduleTableMaxShorten` that defines the maximum number of ticks that can be subtracted from expiry point offset. ⌋ ( )

[SWS_Os_00416] ⌈An expiry point shall permit the configuration of a `OsScheduleTableMaxLengthen` that defines the maximum number of ticks that can be added to expiry point offset. ⌋ ( )

When performing synchrioniszation it is important that the expiry points on the schedule table are processed according to the total ordering defined by their offsets. This means that the range of permitted values for `OsScheduleTableMaxShorten` and `OsScheduleTableMaxLengthen` must ensure that the next expiry point is not retarded into the past or advanced beyond more than one iteration of the schedule table.

[SWS_Os_00436] ⌈The value of (Offset – `OsScheduleTableMaxShorten`) of an expiry point shall be greater than (Offset + `OsCounterMinCycle`) of the pervious expiry point. ⌋ ( )

[SWS_Os_00559] ⌈The value of `OsScheduleTableMaxLengthen` shall be smaller than the duration of the schedule table. ⌋ ( )

[SWS_Os_00437] ⌈The value of (`OsScheduleTableMaxLengthen + delay_from_previous_EP`) of an expiry point shall be less than the `OsCounterMaxAllowedValue` of the underlying counter. ⌋ ( )

Explicitly synchronized schedule tables allow the tolerance of some drift between the schedule table value and the synchronization counter value. This tolerance can be zero, indicating that the schedule table is not considered synchronized unless the values are indentical..

[SWS_Os_00438] ⌈A schedule table shall define a precision bound with a value in the range 0 to duration. ⌋ ( )

7.4.2.3 Performing Synchronization

The Operating System module uses the synchronization count to support (re-)synchronization of a schedule table at each expiry point by calculating an adjustment to the delay to the next expiry point. This provides faster re-synchronization of the schedule table than doing the action on the final expiry point.

[SWS_Os_00206] ⌈When a new synchronization count is provided, the Operating System module shall calculate the current deviation between the explicitly synchronized scheduled table and the synchronization count. ⌋ (SRS_Os_11002)

It is meaningless to try and synchronise an explicitly synchronized schedule table before a synchronization count is provided.

[SWS_Os_00417] ⌈The Operating System module shall start to synchronise an explicitly synchronized schedule table after a synchronization count is provided AND shall continue to adjust expiry points until synchronized. ⌋ ( )

[SWS_Os_00418] ⌈The Operating System module shall set the state of an explicitly synchronized schedule table to “running and synchronous” if the deviation is less than or equal to the configured `OsScheduleTblExplicitPrecision` threshold. ⌋ ( )

[SWS_Os_00419] ⌈The Operating System module shall set the state of an explicitly synchronized schedule table to "running" if the deviation is greater than the configured `OsScheduleTblExplicitPrecision` threshold.⌋ ( )

[SWS_Os_00420] ⌈IF the deviation is non-zero AND the next expiry point is adjustable AND the table is behind the sync counter (TableTicksAheadOfSyncCounter <= TableTicksBehindOfSyncCounter) THEN the OS shall set the next EP to expire delay - min(MaxShorten, Deviation) ticks from the current expiry.⌋ ( )

[SWS_Os_00421] ⌈IF the deviation is non-zero AND the next expiry point is adjustable AND the table is ahead of the sync counter (TableTicksAheadOfSyncCounter > TableTicksBehindOfSyncCounter) THEN the OS shall set the next EP to expire delay + min(MaxLengthen, Deviation) ticks from the current expiry.⌋ ( )

Figure 7.8: shows explicit synchronization of a schedule table. It assumes the following:

- EP1-3 have `OsScheduleTableMaxLengthen`=2
- EP1-3 have `OsScheduleTableMaxShorten` =1

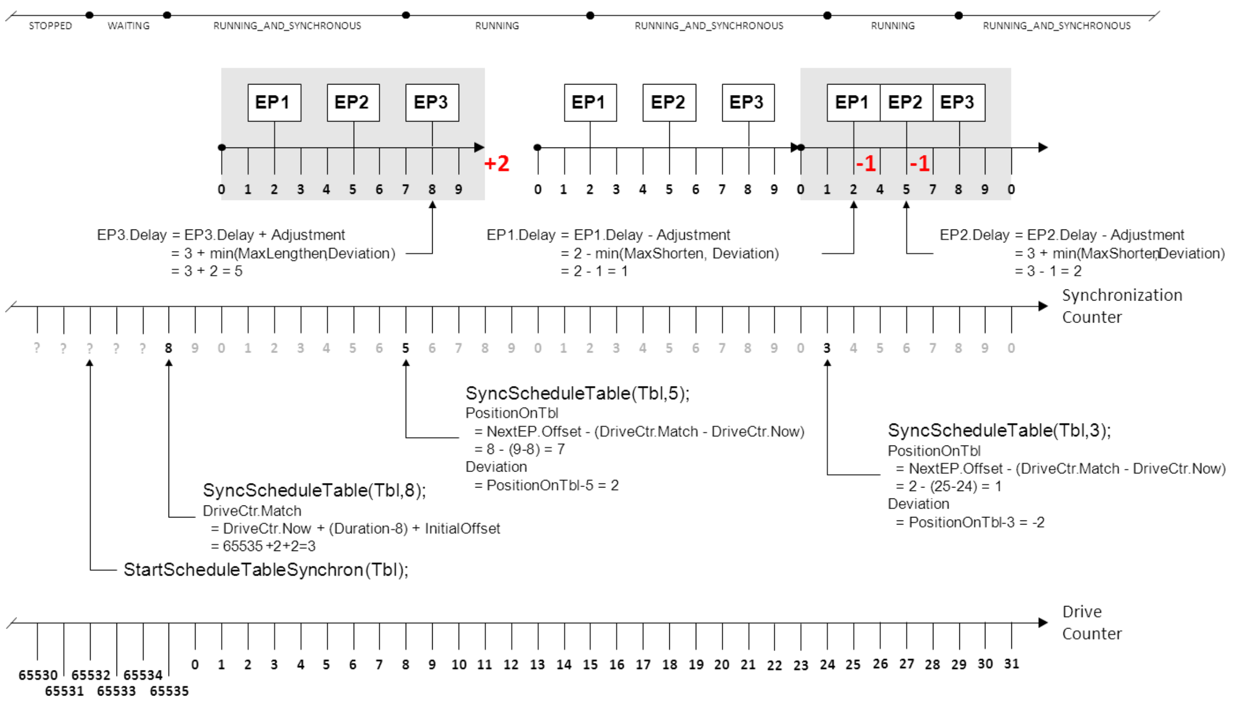
STOPPED
WAITING
RUNNING_AND_SYNCHRONOUS
RUNNING
RUNNING_AND_SYNCHRONOUS
RUNNING
RUNNING_AND_SYNCHRONOUS
EP1
EP2
EP3
+2
-1
-1
EP3.Delay = EP3.Delay + Adjustment
= 3 + min(MaxLengthen,Deviation)
= 3 + 2 = 5
EP1.Delay = EP1.Delay - Adjustment
= 2 - min(MaxShorten, Deviation)
= 2 - 1 = 1
EP2.Delay = EP2.Delay - Adjustment
= 3 + min(MaxShorten,Deviation)
= 3 - 1 = 2
Synchronization Counter
SyncScheduleTable(Tbl,5);
PositionOnTbl
= NextEP.Offset - (DriveCtr.Match - DriveCtr.Now)
= 8 - (9-8) = 7
Deviation
= PositionOnTbl-5 = 2
SyncScheduleTable(Tbl,3);
PositionOnTbl
= NextEP.Offset - (DriveCtr.Match - DriveCtr.Now)
= 2 - (25-24) = 1
Deviation
= PositionOnTbl-3 = -2
SyncScheduleTable(Tbl,8);
DriveCtr.Match
= DriveCtr.Now + (Duration-8) + InitialOffset
= 65535 +2+2=3
StartScheduleTableSynchron(Tbl);
Drive Counter

Figure 7.8: Explict Schedule Table Synchronization

The Operating System module provides the service `SetScheduleTableAsync()` (see SWS_Os_00422) to cancel synchronization being performed at adjustable expiry points on a schedule table.

The Operating System module provides the service `GetScheduleTableStatus()` (see SWS_Os_00227) to query the state of a schedule table also with respect to synchronization.

7.5 Stack Monitoring Facilities

7.5.1 Background & Rationale

On processors that do not provide any memory protection hardware it may still be necessary to provide a “best effort with available resources” scheme for detectable classes of memory faults. Stack monitoring will identify where a task or ISR has exceeded a specified stack usage at context switch time. This may mean that there is considerable time between the system being in error and that fault being detected. Similarly, the error may have been cleared at the point the fault is notified (the stack may be less than the specified size when the context switch occurs).

It is not usually sufficient to simply monitor the entire stack space for the system because it is not necessarily the Task/ISR that was executing that used more than stack space than required – it could be a lower priority object that was pre-empted.

Significant debugging time can be saved by letting the Operating System correctly identify the Task/Category 2 ISR in error.

Note that for systems using a MPU and scalability class 3 or 4 a stack overflow may cause a memory exception before the stack monitoring is able to detect the fault.

7.5.2 Requirements

[SWS_Os_00067] ⌈The Operating System module shall provide a stack monitoring which detects possible stack faults of Task(s)/Category 2 ISR(s). ⌋ (SRS_Os_11003)

[SWS_Os_00068] ⌈If a stack fault is detected by stack monitoring AND no `ProtectionHook()` is configured, the Operating System module shall call the `ShutdownOS()` service with the status `E_OS_STACKFAULT`. ⌋ (SRS_Os_11003, SRS_Os_11013)

[SWS_Os_00396] ⌈If a stack fault is detected by stack monitoring AND a `ProtectionHook()` is configured the Operating System module shall call the `ProtectionHook()` with the status `E_OS_STACKFAULT`. ⌋ ( )

7.6 OS-Application

7.6.1 Background & Rationale

An AUTOSAR OS must be capable of supporting a collection of Operating System objects (Tasks, ISRs, Alarms, Schedule tables, Counters) that form a cohesive functional unit. This collection of objects is termed an *OS-Application*.

The Operating System module is responsible for scheduling the available processing resource between the OS-Applications that share the processor. If OS-Application(s) are used, all Tasks, ISRs, Counters, Alarms and Schedule tables must belong to an OS-Application. All objects which belong to the same OS-Application have access to each other. The right to access objects from other OS-Applications may be granted during configuration. An event is accessible if the task for which the event can be set is accessible. Access means that these Operating System objects are allowed as parameters to API services.

There are two classes of OS-Application:

(1) Trusted OS-Applications are allowed to run with monitoring or protection features disabled at runtime. They may have unrestricted access to memory, the Operating System module's API, and need not have their timing behaviour enforced at runtime. They are allowed to run in privileged mode when supported by the processor. The Operating System module assumes that trusted OS-Applications (and trusted functions) do not cause an memory related protection fault. If such a fault happens the system stability is likely gone and a shutdown may be the only option.

(2) Non-Trusted OS-Applications are not allowed to run with monitoring or protection features disabled at runtime. They have restricted access to memory, restricted access to the Operating System module's API and have their timing behaviour enforced at runtime. They are not allowed to run in privileged mode when supported by the processor.

It is assumed that the Operating System module itself is trusted.

There are services offered by the AUTOSAR OS which give the caller information about the access rights and the membership of objects. These services are intended to be used in case of an inter-OS-Application call for checking access rights and arguments.

Note that Resource obejcts do not belong to any OS-Application, but access to them must be explicitely granted. (The same principle applies to spinlocks in Multi-Core systems)

The running OS-Application is defined as the OS-Application to which the currently running Task or ISR belongs. In case of a hook routine the Task or ISR which caused the call of the hook routine defines the running OS-Application.

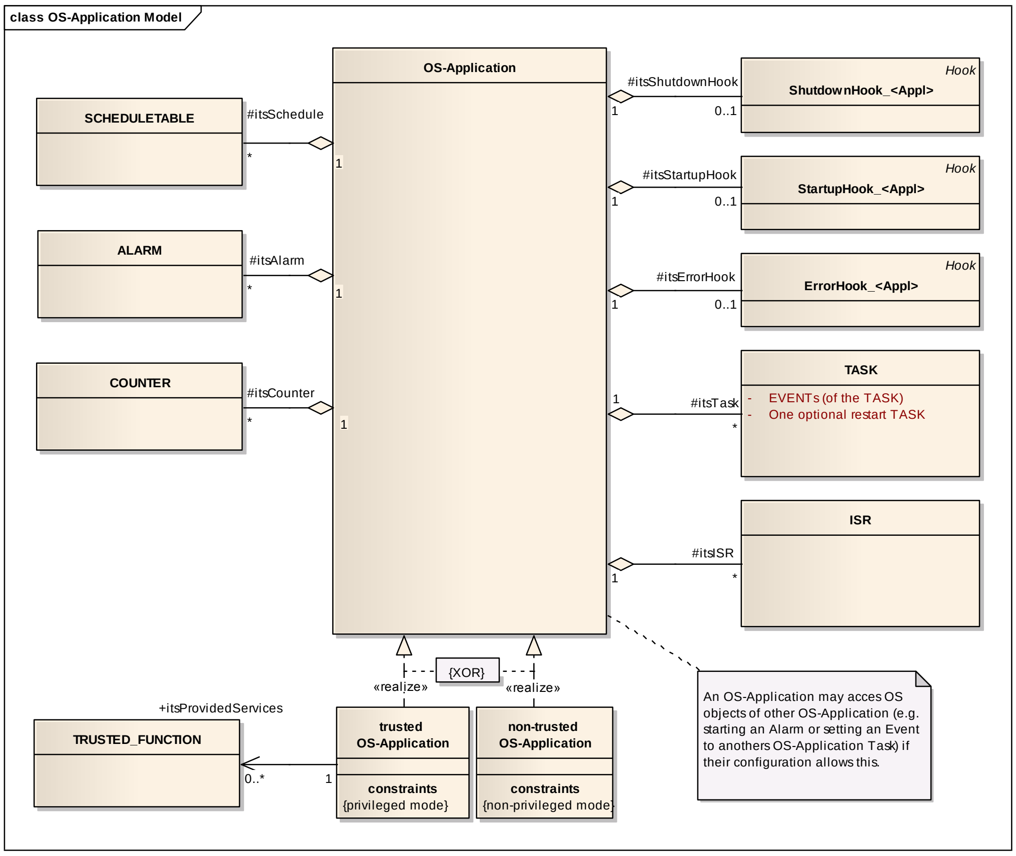
class OS-Application Model
OS-Application
SCHEDULETABLE
#itsSchedule
ALARM
#itsAlarm
COUNTER
#itsCounter
Hook
ShutdownHook_<Appl>
#itsShutdownHook
Hook
StartupHook_<Appl>
#itsStartupHook
Hook
ErrorHook_<Appl>
#itsErrorHook
TASK
- EVENTs (of the TASK)
- One optional restart TASK
#itsTask
ISR
#itsISR
{XOR}
«realize»
«realize»
TRUSTED_FUNCTION
+itsProvidedServices
trusted OS-Application
constraints {privileged mode}
non-trusted OS-Application
constraints {non-privileged mode}
An OS-Application may acces OS objects of other OS-Application (e.g. starting an Alarm or setting an Event to anothers OS-Application Task) if their configuration allows this.

Figure 7.9: UML-model of OS-Application

OS-Applications have a state which defines the scope of accessability of its Operating System objects from other OS-Applications. Each OS-Application is always in one of the following states:

- Active and accessible (APPLICATION_ACCESSIBLE): Operating System objects may be accessed from other OS-Applications. This is the default state at startup.
- Currently in restart phase (APPLICATION_RESTART). Operating System objects can not be accessed from other OS-Applications. State is valid until the OS-Application calls AllowAccess().
- Terminated and not accessible (APPLICATION_TERMINATED): Operating System objects can not be accessed from other OS-Applications. State will not change.

The following figure shows the states and the possible transitions:

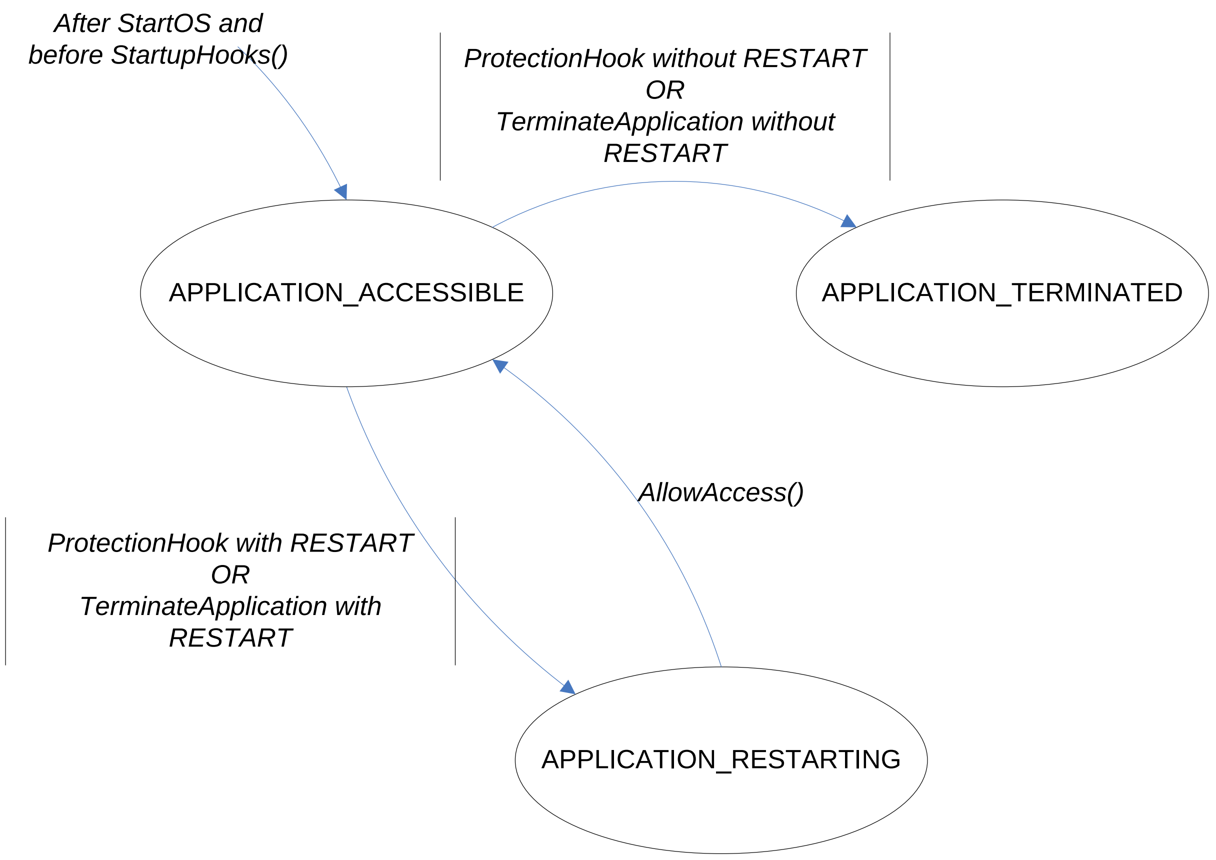
After StartOS and before StartupHooks()
ProtectionHook without RESTART OR TerminateApplication without RESTART
APPLICATION_ACCESSIBLE
APPLICATION_TERMINATED
AllowAccess()
ProtectionHook with RESTART OR TerminateApplication with RESTART
APPLICATION_RESTARTING

Figure 7.13: States of OS-Applications

7.6.2 Requirements

[SWS_Os_00445] ⌈The Operating System module shall support OS-Applications which are a configurable selection of Trusted Functions, Tasks, ISRs, Alarms, Schedule tables, Counters, hooks (for startup, error and shutdown). ⌋ ( )

[SWS_Os_00446] ⌈The Operating System module shall support the notion of trusted and non-trusted OS-Applications. ⌋ ( )

[SWS_Os_00464] ⌈Trusted OS-Applications may offer services ("trusted services") to other (even non-trusted) OS-Applications. ⌋ ( )

The Operating System module provides the services `GetApplicationID()` and `GetCurrentApplicationID()` (see SWS_Os_00016) to determine the configured resp. currently executing OS-Application (a unique identifier shall be allocated to each application).

The Operating System module provides the service `CheckObjectOwnership()` (see SWS_Os_00017) to determine to which OS-Application a given Task, ISR, Counter, Alarm or Schedule Table belongs.

The Operating System module provides the service `CheckObjectAccess()` (see SWS_Os_00256) to determine which OS-Applications are allowed to use the IDs of a Task, Resource, Counter, Alarm or Schedule Table in API calls.

The Operating System module provides the service `TerminateApplication()` (see SWS_Os_00258) to terminate the OS-Application to which the calling Task/Category 2 ISR/application specific error hook belongs. (This is an OS-Application level variant of the `TerminateTask()` service)

The Operating System provides the service `TerminateApplication()` (see SWS_Os_00258) to terminate another OS-Application AND calls to this service shall be ignored if the caller does not belong to a trusted OS-Application.

[SWS_Os_00447] ⌈If the Operating System module terminates an OS-Application, then it shall:

terminate all running, ready and waiting Tasks/ISRs of the OS-Application AND

disable all interrupts of the OS-Application AND

stop all active alarms of the OS-Applications AND

stop all schedule tables of the OS-Application. ⌋ ( )

[SWS_Os_00448] ⌈The Operating System module shall prevent access of OS-Applications, trusted or non-trusted, to objects not belonging to this OS-Application, except access rights for such objects are explicitly granted by configuration. ⌋ ( )

The Operating System provides the service `GetApplicationState()` (see SWS_Os_00499) to request the current state of an OS-Application.

[SWS_Os_00500] ⌈The Operating System module shall set the state of all OS-Applications after the call of `StartOS()` and before any StartupHook is called to `APPLICATION_ACCESSIBE`. ⌋ ( )

The Operating System module provides the service `AllowAccess()` (see SWS_Os_00501) to set the own state of an OS-Application from `APPLICATION_RESTARTING` to `APPLICATION_ACCESSIBLE`.

[SWS_Os_00502] ⌈If an OS-Application is terminated (e.g. through a service call or via protection hook) and no restart is requested, then the Operating System module shall set the state of this OS-Application to `APPLICATION_TERMINATED`. ⌋ ( )

[SWS_Os_00503] ⌈If an OS-Application is terminated (e.g. through a service call or via protection hook) and a restart is requested, then the Operating System module shall set the state of this OS-Application to `APPLICATION_RESTARTING`. ⌋ ( )

[SWS_Os_00504] ⌈The Operating System module shall deny access to Operating System objects from other OS-Applications to an OS-Application which is not in state `APPLICATION_ACCESSIBLE`. ⌋ ( )

[SWS_Os_00509] ⌈If a service call is made on an Operating System object that is owned by another OS-Application without state `APPLICATION_ACCESSIBLE`, then the Operating System module shall return `E_OS_ACCESS`. ⌋ ( )

An example for SWS_Os_00509 is a call to ActivateTask() for a task in an OS-Application that is restarting.

7.7 Protection Facilities

Protection is only possible for Operating System managed objects. This means that:

- It is not possible to provide protection during runtime of Category 1 ISRs, because the operating system is not aware of any Category 1 ISRs being invoked. Therefore, if any protection is required, Category 1 ISRs have to be avoided. If Category 1 interrupts AND OS-Applications are used together then all Category 1 ISR must belong to a trusted OS-Application.

- It is not possible to provide protection between functions called from the body of the same Task/Category 2 ISR.

7.7.1 Memory Protection

7.7.1.1 Background & Rationale

Memory protection will only be possible on processors that provide hardware support for memory protection.

The memory protection scheme is based on the (data, code and stack) sections of the executable program.

Stack: An OS-Application comprises a number of Tasks and ISRs. The stack for these objects, by definition, belongs only to the owner object and there is therefore no need to share stack data between objects, even if those objects belong to the same OS-Application.
Memory protection for the stacks of Tasks and ISRs is useful mainly for two reasons:

(1) Provide a more immediate detection of stack overflow and underflow for the Task or ISR than can be achieved with stack monitoring

(2) Provide protection between constituent parts of and OS-Application, for example to satisfy some safety constraints.

Data: OS-Applications can have private data sections and Tasks/ISRs can have private data sections. OS-Application's private data sections are shared by all Tasks/ISRs belonging to that OS-Application.

Code: Code sections are either private to an OS-Application or can be shared between all OS-Applications (to use shared libraries). In the case where code protection is not used, executing incorrect code will eventually result in a memory, timing or service violation.

7.7.1.2 Requirements

Data Sections and Stack

[SWS_Os_00198] ⌈The Operating System module shall prevent write access to its own data sections and its own stack from non-trusted OS-Applications. ⌋ ( )

[SWS_Os_00795] ⌈The OS shall offer the possibility to restrict write access of trusted OS-Applications in the same way as it is done for non-trusted OS-Applications." ⌋(SRS_Os_11005**)**

This can be configured with the `OsTrustedApplicationWithProtection`.

Private data of an OS-Application

[SWS_Os_00026] ⌈The Operating System module may prevent read access to an OS-Application's data section attempted by other non-trusted OS-Applications. ⌋ (SRS_Os_11000)

[SWS_Os_00086] ⌈The Operating System module shall permit an OS-Application read and write access to that OS-Application's own private data sections. ⌋ (SRS_Os_11006)

[SWS_Os_00207] ⌈The Operating System module shall prevent write access to the OS-Application's private data sections from other non-trusted OS-Applications. ⌋ (SRS_Os_11005)

Private Stack of Task/ISR

[SWS_Os_00196] ⌈The Operating System module shall permit a Task/Category 2 ISR read and write access to that Task's/Category 2 ISR's own private stack. ⌋ (SRS_Os_11006)

[SWS_Os_00208] ⌈The Operating System module may prevent write access to the private stack of Tasks/Category 2 ISRs of a non-trusted application from all other Tasks/ISRs in the same OS-Application. ⌋ (SRS_Os_11005)

[SWS_Os_00355] ⌈The Operating System module shall prevent write access to all private stacks of Tasks/Category 2 ISRs of an OS-Application from other non-trusted OS-Applications. ⌋ ( )

Private data of a Task/ISR

[SWS_Os_00087] ⌈The Operating System module shall permit a Task/Category 2 ISR read and write access to that Task's/Category 2 ISR's own private data sections. ⌋ (SRS_Os_11006)

[SWS_Os_00195] ⌈The Operating System module may prevent write access to the private data sections of a Task/Category 2 ISR of a non-trusted application from all other Tasks/ISRs in the same OS-Application. ⌋ (SRS_Os_11005)

[SWS_Os_00356] ⌈The Operating System module shall prevent write access to all private data sections of a Task/Category 2 ISR of an OS-Application from other non-trusted OS-Applications. ⌋ ( )

Code Sections

[SWS_Os_00027] ⌈The Operating System module may provide an OS-Application the ability to protect its code sections against executing by non-trusted OS-Applications. ⌋ ( )

[SWS_Os_00081] ⌈The Operating System module shall provide the ability to provide shared library code in sections that are executable by all OS-Applications. ⌋ (SRS_Os_11007)

Peripherals

[SWS_Os_00209] ⌈If `OsTrustedApplicationWithProtection == FALSE` then the Operating System module shall permit trusted OS-Applications read and write access to peripherals. ⌋ ( )

[SWS_Os_00083] ⌈The Operating System module shall allow non-trusted OS-Applications to write to their assigned peripherals only (incl. reads that have the side effect of writing to a memory location). ⌋ ( )

Memory Access Violation

[SWS_Os_00044] ⌈If a memory access violation is detected, the Operating System module shall call the Protection Hook with status code `E_OS_PROTECTION_MEMORY`. ⌋ (SRS_Os_11013)

7.7.2 Timing Protection

7.7.2.1 Background & Rationale

A timing fault in a real-time system occurs when a task or interrupt misses its deadline at runtime.

AUTOSAR OS does not offer deadline monitoring for timing protection. Deadline monitoring is insufficient to correctly identify the Task/ISR causing a timing fault in an AUTOSAR system. When a deadline is violated this may be due to a timing fault introduced by an unrelated Task/ISR that interferes/blocks for too long. The fault in this case lies with the unrelated Task/ISR and this will propagate through the system until a Task/ISR misses its deadline. The Task/ISR that misses a deadline is therefore not necessarily the Task/ISR that has failed at runtime, it is simply the earliest point that a timing fault is detected.

If action is taken based on a missed deadline identified with deadline monitoring this would potentially use false evidence of error to terminate a correct OS-Application in favour of allowing an incorrect OS-Application to continue running. The problem is best illustrated by example. Consider a system with the following configuration:

TaskID	Priority	Execution Time	Deadline (=Period)
A	High	1	5
B	Medium	3	10
C	Low	5	15

Assuming that all tasks are ready to run at time zero, the following execution trace would be expected and all tasks would meet their respective deadlines.

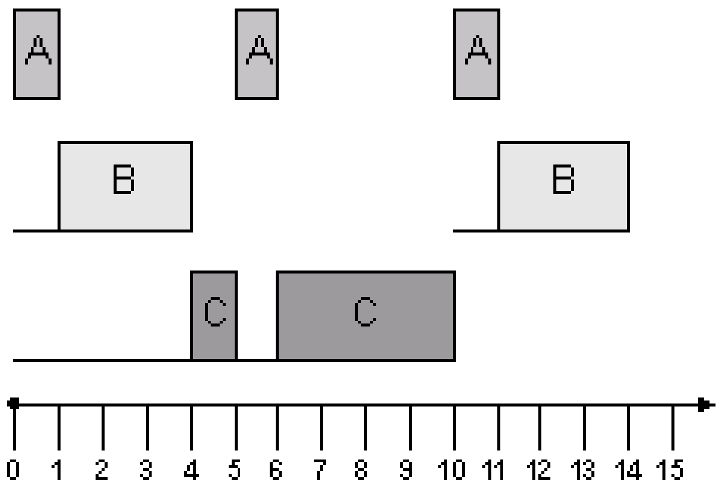

Figure 7.10: Example execution trace

Now consider the case when tasks A and B behave incorrectly. The figure below shows both task A and task B executing for longer than specified and task B arriving 2 ticks earlier than specified. Both tasks A and B meet their deadlines. Task C however, behaves correctly but it fails to meet its deadline because of the incorrect execution of Tasks A and B. This is fault propagation – a fault in an unrelated part of the system is causing a correctly functioning part of the system to fail.

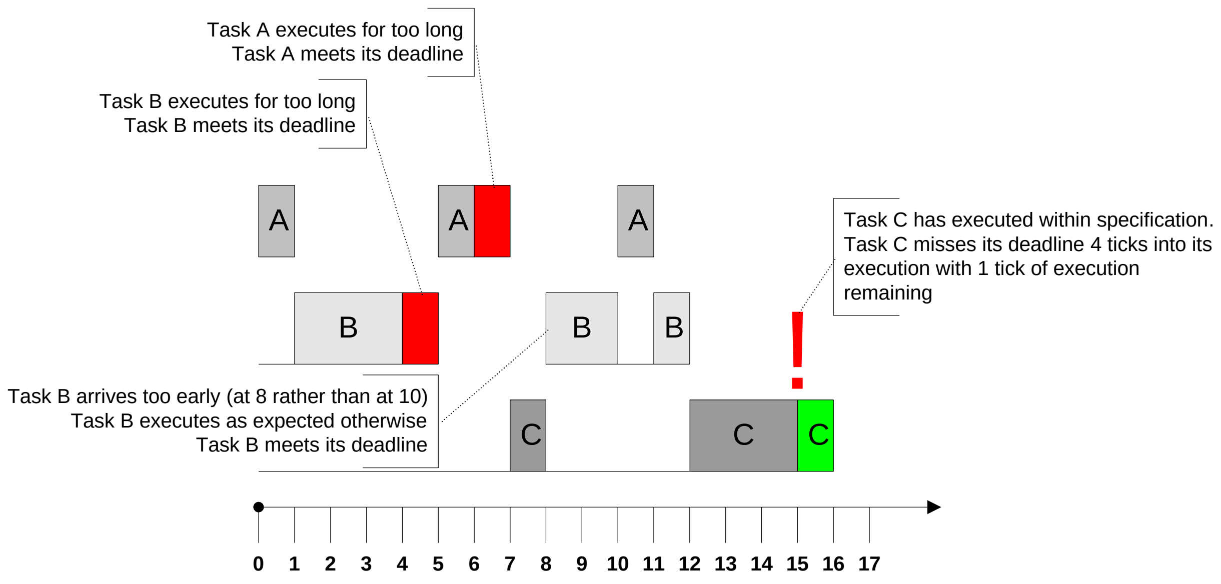
Task A executes for too long
Task A meets its deadline
Task B executes for too long
Task B meets its deadline
A
A
A
Task C has executed within specification.
Task C misses its deadline 4 ticks into its execution with 1 tick of execution remaining
B
B
B
!
Task B arrives too early (at 8 rather than at 10)
Task B executes as expected otherwise
Task B meets its deadline
C
C
C
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17

Figure 7.11: Insufficiency of Deadline Monitoring

Whether a task or ISR meets its deadline in a fixed priority preemptive operating system like AUTOSAR OS is determined by the following factors:

the execution time of Task/ISRs in the system the blocking time that Task/ISRs suffers from lower priority Tasks/ISRs locking shared resources or disabling interrupts

the interarrival rate of Task/ISRs in the system

For safe and accurate timing protection it is necessary for the operating system to control these factors at runtime to ensure that Tasks/ISRs can meet their respective deadlines.

AUTOSAR OS prevents timing errors from (1) by using *execution time protection* to guarantee a statically configured upper bound, called the Execution Budget, on the execution time of:

Tasks
Category 2 ISRs

AUTOSAR OS prevents timing errors from (2) by using *locking time protection* to guarantee a statically configured upper bound, called the Lock Budget, on the time that:

Resources are held by Tasks/Category 2 ISRs
OS interrupts are suspended by Tasks/Category 2 ISRs
ALL interrupts are suspended/disabled by Tasks/Category 2 ISRs

AUTOSAR OS prevents timing errors from (3) by using *inter-arrival time protection* to guarantee a statically configured lower bound, called the Time Frame, on the time between:

- A task being permitted to transition into the `READY` state due to:
  - Activation (the transition from the `SUSPENDED` to the `READY` state)
  - Release (the transition from the `WAITING` to the `READY` state)
- A Category 2 ISR arriving
  An arrival occurs when the Category 2 ISR is recognized by the OS

Inter-arrival time protection for basic tasks controls the time between successive activations, irrespective of whether activations are queued or not. In the case of queued activations, activating a basic task which is in the `READY` or `RUNNING` state is a new activation because it represents the activation of a new instance of the task. Inter-arrival time protection therefore interacts with queued activation to control the rate at which the queue is filled.

Inter-arrival time protection for extended tasks controls the time between successive activations *and* releases. When a task is in the `WAITING` state and multiple events are set with a single call to `SetEvent()` this represents a single release. When a task waits for one or more events which are already set this represents a notional Wait/Release/Start transition and therefore is considered as a new release.

The following figure shows how execution time protection and inter-arrival time protection interact with the task state transition model for AUTOSAR OS.

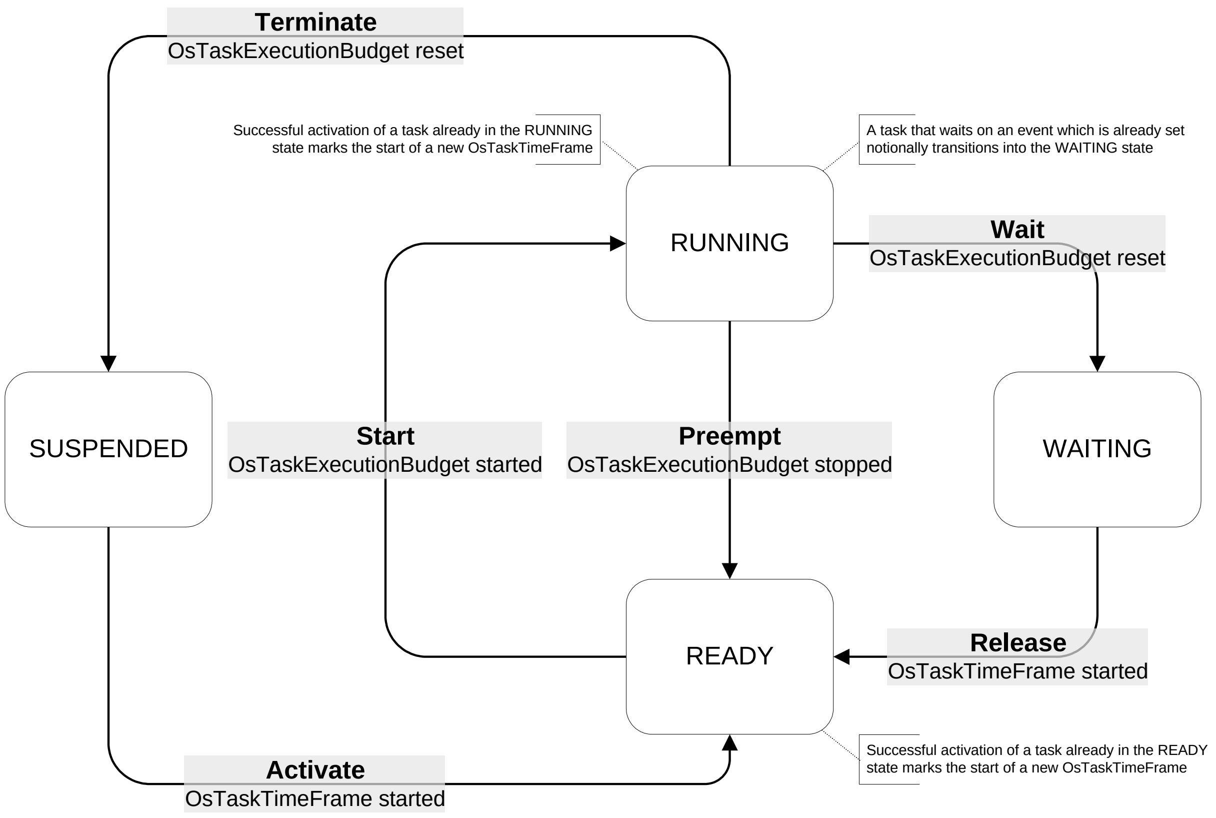
Terminate
OsTaskExecutionBudget reset
Successful activation of a task already in the RUNNING state marks the start of a new OsTaskTimeFrame
A task that waits on an event which is already set notionally transitions into the WAITING state
RUNNING
Wait
OsTaskExecutionBudget reset
SUSPENDED
Start
OsTaskExecutionBudget started
Preempt
OsTaskExecutionBudget stopped
WAITING
READY
Release
OsTaskTimeFrame started
Successful activation of a task already in the READY state marks the start of a new OsTaskTimeFrame
Activate
OsTaskTimeFrame started

Figure 7.12: Time protection interaction with the task state transition model

Notes:

1. Inter-arrival time enforcement on Category 2 ISRs can be used to protect an ECU from a "babbling idiot" source of interrupts (e.g. a CAN controller taking an interrupt each time a frame is received from another ECU on the network).
2. Timing protection only applies to Tasks or Category 2 ISRs. There is no protection for Category 1 ISRs. If timing protection error occurs during a category 1 ISR, consistency of the Operating System module can not be guaranteed. Therefore we discourage timing protection in systems with category 1 interrupts.
3. Timing protection does not apply before the Operating System module is started.
4. In the case of trusted OS-Applications it is essential that all timing information is correct, otherwise the system may fail at run-time. For a non-trusted OS-Application, timing protection can be used to enforce timing boundaries between executable objects.

7.7.2.2 Requirements

[SWS_Os_00028] ⌈In a non-trusted OS-Application, the Operating System module shall apply timing protection to every Task/Category 2 ISR of this non-trusted OS-Application. ⌋ (SRS_Os_11008)

[SWS_Os_00089] ⌈In a trusted OS-Application, the Operating System module shall provide the ability to apply timing protection to Tasks/Category 2 ISRs of this OS-Application. ⌋ (SRS_Os_11008)

[SWS_Os_00397] ⌈If no OS-Application is configured, the Operating System module shall be able to apply timing protection to Tasks/Category 2 ISRs. ⌋ ( )

Timing Protection: Tasks

[SWS_Os_00064] ⌈If a task's OsTaskExecutionBudget is reached then the Operating System module shall call the `ProtectionHook()` with `E_OS_PROTECTION_TIME`. ⌋ (SRS_Os_11008, SRS_Os_11013)

[SWS_Os_00473] ⌈The Operating System module shall reset a task's OsTaskExecutionBudget on a transition to the `SUSPENDED` or `WAITING` states. ⌋ (SRS_Os_11008)

[SWS_Os_00465] ⌈The Operating System module shall limit the inter-arrival time of tasks to one per OsTaskTimeFrame. ⌋ (SRS_Os_11008)

[SWS_Os_00469] ⌈The Operating System module shall start an OsTaskTimeFrame when a task is activated successfully. ⌋ (SRS_Os_11008)

[SWS_Os_00472] ⌈The Operating System module shall start an OsTaskTimeFrame when a task is released successfully. ⌋ (SRS_Os_11008)

[SWS_Os_00466] ⌈If an attempt is made to activate a task before the end of an OsTaskTimeFrame then the Operating System module shall not perform the activation AND shall call the `ProtectionHook()` with `E_OS_PROTECTION_ARRIVAL`. ⌋ ( )

[SWS_Os_00467] ⌈If an attempt is made to release a task before the end of an OsTaskTimeFrame then the Operating System module shall not perform the release AND shall call the `ProtectionHook()` with `E_OS_PROTECTION_ARRIVAL` AND the event shall be set. ⌋ ( )

Timing Protection: ISRs

[SWS_Os_00210] ⌈If a Category 2 ISR's OsIsrExecutionBudget is reached then the Operating System module shall call the `ProtectionHook()` with `E_OS_PROTECTION_TIME`. ⌋ (SRS_Os_11013)

[SWS_Os_00474] ⌈The Operating System module shall reset an ISR's OsIsrExecutionBudget when the ISR returns control to the OS or terminates. ⌋ (SRS_Os_11008)

[SWS_Os_00470] ⌈The Operating System module shall limit the inter-arrival time of Category 2 ISRs to one per OsIsrTimeFrame. ⌋ (SRS_Os_11008)

[SWS_Os_00471] ⌈The Operating System module shall measure the start of an OsIsrTimeFrame from the point at which it recognises the interrupt (i.e. in the Operating System interrupt wrapper). ⌋ (SRS_Os_11008)

[SWS_Os_00048] ⌈If Category 2 interrupt occurs before the end of the OsIsrTimeFrame then the Operating System module shall not execute the user provided ISR AND shall call the `ProtectionHook()` with `E_OS_PROTECTION_ARRIVAL`. ⌋ (SRS_Os_11008)

Timing Protection: Resource Locking and Interrupt Disabling

[SWS_Os_00033] ⌈If a Task/Category 2 ISR holds an OSEK Resource and exceeds the Os[Task|Isr]ResourceLockBudget, the Operating System module shall call the `ProtectionHook()` with `E_OS_PROTECTION_LOCKED`. ⌋ (SRS_Os_11008, SRS_Os_11013, SRS_Os_11014)

[SWS_Os_00037] ⌈If a Task/Category 2 ISR disables interrupts (via `Suspend/Disable|All/OS|Interrupts()`) and exceeds the configured Os[Task|Isr][All|OS]InterruptLockBudget, the Operating System module shall call the `ProtectionHook()` with `E_OS_PROTECTION_LOCKED.` ⌋ (SRS_Os_11008, SRS_Os_11013, SRS_Os_11014)

7.7.2.3 Implementation Notes

Execution time enforcement requires hardware support, e.g. a timing enforcement interrupt. If an interrupt is used to implement the time enforcement, the priority of this interrupt has to be high enough to "interrupt" the supervised tasks or ISRs.

Depending on the real hardware support this could mean that DisableAllInterrupts and SuspendAllInterrupts disable not all interrupts (e.g. all interrupts except of the interrupt used for timing protection) or that the usage of Category 1 ISRs – which bypass the Operating System (and also the timing protection) – is limited somehow.

The implementation has to document such implementation specific behaviour (e.g. the limitations when timing protection is used).

7.7.3 Service Protection

Background & Rationale

As OS-Applications can interact with the Operating System module through services, it is essential that the service calls will not corrupt the Operating System module itself. Service Protection guards against such corruption at runtime.

There are a number of cases to consider with Service Protection: An OS-Application makes an API call

(1) with an invalid handle or out of range value.

(2) in the wrong context, e.g. calling `ActivateTask()` in the `StartupHook()`.

(3) or fails to make an API call that results in the OSEK OS being left in an undefined state, e.g. it terminates without a `ReleaseResource()` call

(4) that impacts on the behaviour of every other OS-Application in the system, e.g. `ShutdownOS()`

(5) to manipulate Operating System objects that belong to another OS-Application (to which it does not have the necessary permissions), e.g. an OS-Application tries to execute `ActivateTask()` on a task it does not own.

The OSEK OS already provides some service protection through the status codes returned from service calls and this will provide the basis for service protection. This means that service protection will only apply for the extended status of OSEK OS.

However, OSEK OS does not cover all the cases outlined above. The following sections describe – besides the mandatory extended status – the additional protection requirements to be applied in each of these cases.

7.7.3.1 Invalid Object Parameter or Out of Range Value

1.1.4 Background & Rationale

The current OSEK OS' service calls already return `E_OS_ID` on invalid objects (i.e. objects not defined in the OIL file) and `E_OS_VALUE` for out of range values (e.g. setting an alarm cycle time less than `OsCounterMinCycle`).

1.1.5 Requirements

[SWS_Os_00051] ⌈If an invalid address (address is not writable by this OS-Application) is passed as an out-parameter to an Operating System service, the Operating System module shall return the status code `E_OS_ILLEGAL_ADDRESS`. ⌋ (SRS_Os_11009, SRS_Os_11013)

7.7.3.2 Service Calls Made from Wrong Context

1.1.6 Background & Rationale

The current OSEK OS defines the valid calling context for service calls (see [15]), however protects against only a small set of these invalid calls, e.g. calling `TerminateTask()` from a Category 2 ISR.

Service	**Task**	**Cat1 ISR**	**Cat2 ISR**	**Error Hook**	**PreTask Hook**	**PostTask Hook**	**Startup Hook**	**Shutdown Hook**	**Alarm Callback**	**Protection Hook**
ActivateTask	✓		✓							
ActivateTaskAsyn	✓		✓							
TerminateTask	✓		C							
ChainTask	✓		C							
Schedule	✓		C							
GetTaskID	✓		✓	✓	✓	✓				✓
GetTaskState	✓		✓	✓	✓	✓				
DisableAllInterrupts	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
EnableAllInterrupts	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
SuspendAllInterrupts	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
ResumeAllInterrupts	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
SuspendOSInterrupts	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
ResumeOSInterrupts	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
GetResource	✓		✓							
ReleaseResource	✓		✓							
SetEvent	✓		✓							
SetEventAsyn	✓		✓							
ClearEvent	✓		C							
GetEvent	✓		✓	✓	✓	✓				
WaitEvent	✓		C							
GetAlarmBase	✓		✓	✓	✓	✓				
GetAlarm	✓		✓	✓	✓	✓				
SetRelAlarm	✓		✓							
SetAbsAlarm	✓		✓							
CancelAlarm	✓		✓							
GetActiveApplicationMode	✓		✓	✓	✓	✓	✓	✓		
StartOS										
ShutdownOS	✓		✓	✓			✓			
GetApplicationID	✓		✓	✓	✓	✓	✓	✓		✓

Service	**Task**	**Cat1 ISR**	**Cat2 ISR**	**Error Hook**	**PreTask Hook**	**PostTask Hook**	**Startup Hook**	**Shutdown Hook**	**Alarm Callback**	**Protection Hook**
GetISRID	✓		✓	✓						✓
CallTrustedFunction	✓		✓							
CheckISRMemoryAccess	✓		✓	✓						✓
CheckTaskMemoryAccess	✓		✓	✓						✓
CheckObjectAccess	✓		✓	✓						✓
CheckObjectOwnership	✓		✓	✓						✓
StartScheduleTableRel	✓		✓							
StartScheduleTableAbs	✓		✓							
StopScheduleTable	✓		✓							
NextScheduleTable	✓		✓							
StartScheduleTableSynchron	✓		✓							
SyncScheduleTable	✓		✓							
GetScheduleTableStatus	✓		✓							
SetScheduleTableAsync	✓		✓							
IncrementCounter	✓		✓							
GetCounterValue	✓		✓							
GetElapsedValue	✓		✓							
TerminateApplication	✓		✓	✓[2]						
AllowAccess	✓		✓							
GetApplicationState	✓		✓	✓	✓	✓	✓	✓		✓
ControlIdle	✓		✓							
GetCurrentApplicationID	✓		✓	✓	✓	✓	✓	✓		✓
ReadPeripheral8	✓		✓							
ReadPeripheral16	✓		✓							
ReadPeripheral32	✓		✓							
WritePeripheral8	✓		✓							
WritePeripheral16	✓		✓							
WritePeripheral32	✓		✓							
ModifyPeripheral8	✓		✓							
ModifyPeripheral16	✓		✓							
ModifyPeripheral32	✓		✓							
DisableInterruptSource	✓		✓							
EnableInterruptSource	✓		✓							
ClearPendingInterrupt	✓		✓							

Tab. 1: Allowed Calling Context for OS Service Calls

In the table above “C” indicates that validity is only “Checked in Extended status by `E_OS_CALLEVEL`”.

[2] Only in case of self termination.

1.1.7 Requirements

[SWS_Os_00088] ⌈If an OS-Application makes a service call from the wrong context AND is currently not inside a Category 1 ISR the Operating System module shall not perform the requested action (the service call shall have no effect), and return `E_OS_CALLEVEL` or the “invalid value” of the service. ⌋ (SRS_Os_11009, SRS_Os_11013)

7.7.3.3 Services with Undefined Behaviour

1.1.8 Background & Rationale

There are a number of situations where the behaviour of OSEK OS is undefined in extended status. This is unacceptable when protection is required as it would allow the Operating System module to be corrupted through its own service calls. The implementation of service protection for the Operating System module must therefore describe and implement a behaviour that does not jeopardise the integrity of the system or of any OS-Application which did not cause the specific error.

1.1.9 Requirements

Tasks ends without calling a `TerminateTask()` or `ChainTask()`

[SWS_Os_00052] ⌈If a task returns from its entry function without making a `TerminateTask()` or `ChainTask()` call, the Operating System module shall terminate the task (and call the `PostTaskHook()` if configured). ⌋ (SRS_Os_11009)

[SWS_Os_00069] ⌈If a task returns from its entry function without making a `TerminateTask()` or `ChainTask()` call AND the error hook is configured, the Operating System module shall call the `ErrorHook()` (this is done regardless of whether the task causes other errors, e.g. `E_OS_RESOURCE`) with status `E_OS_MISSINGEND` before the task leaves the `RUNNING` state. ⌋ (SRS_Os_11009)

[SWS_Os_00070] ⌈If a task returns from the entry function without making a `TerminateTask()` or `ChainTask()` call and still holds OSEK Resources, the Operating System module shall release them. ⌋ (SRS_Os_11009, SRS_Os_11013)

[SWS_Os_00239] ⌈If a task returns from the entry function without making a `TerminateTask()` or `ChainTask()` call and interrupts are still disabled, the Operating System module shall enable them. ⌋ ( )

Category 2 ISR ends with locked interrupts or allocated resources

[SWS_Os_00368] ⌈If a Category 2 ISR calls `DisableAllInterrupts()` / `SuspendAllInterrupts()` / `SuspendOSInterrupts()` and ends (returns) without calling the corresponding `EnableAllInterrupts()` / `ResumeAllInterrupts()` / `ResumeOSInterrupts()`, the Operating System module shall perform the missing service and shall call the `ErrorHook()` (if configured) with the status `E_OS_DISABLEDINT`. ⌋ ( )

[SWS_Os_00369] ⌈If a Category 2 ISR calls `GetResource()` and ends (returns) without calling the corresponding `ReleaseResource()`, the Operating System module shall perform the `ReleaseResource()` call and shall call the `ErrorHook()` (if configured) with the status `E_OS_RESOURCE` (see [12], section 13.1). ⌋ ( )

PostTaskHook called during ShutdownOS()

[SWS_Os_00071] ⌈If the `PostTaskHook()` is configured, the Operating System module shall not call the hook if `ShutdownOS()` is called. ⌋ ( )

Tasks/ISRs calls EnableAllInterrupts/ResumeAllInterrupts/ResumeOSInterrupts without a corresponding disable

[SWS_Os_00092] ⌈If `EnableAllInterrupts()` / `ResumeAllInterrupts()` / `ResumeOSInterrupts()` are called and no corresponding `DisableAllInterupts()` / `SuspendAllInterrupts()` / `SuspendOSInterrupts()` was done before, the Operating System module shall not perform this Operating System service. ⌋ (SRS_Os_11009)

Tasks/ISRs calling OS services when DisableAllInterupts/SuspendAllInterrupts/SuspendOSInterrupts called

[SWS_Os_00093] ⌈If interrupts are disabled/suspended by a Task/ISR/Hook and the Task/ISR/Hook calls any Operating System service (excluding the interrupt services) then the Operating System module shall ignore the service AND shall return `E_OS_DISABLEDINT` if the service returns a `StatusType` value. ⌋ (SRS_Os_11009, SRS_Os_11013)

7.7.3.4 Service Restrictions for Non-Trusted OS-Applications

1.1.10 Background & Rationale

The Operating System service calls available are restricted according to the calling context (see Section 7.7.3.2). In a protected system, additional constraints need to be placed to prevent non-trusted OS-Applications executing API calls that can have a global effect on the system. Each level of restriction is a proper subset of the previous level as shown in the figure below.

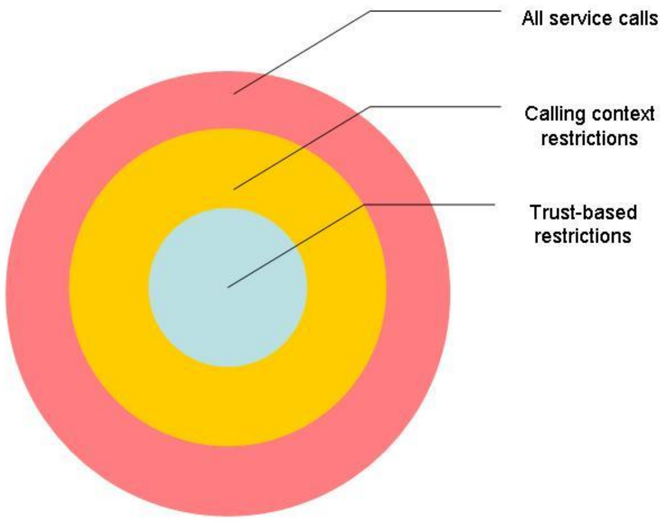

Figure 7.13: API Restrictions

There are two defined integrity levels:

1. Trusted
2. Non-Trusted

that correspond exactly with trusted and non-trusted OS-Applications.

1.1.11 Requirements

[SWS_Os_00054] ⌈The Operating System module shall ignore calls to `ShutdownOS()` from non-trusted OS-Applications. ⌋ ( )

7.7.3.5 Service Calls on Objects in Different OS-Applications

1.1.12 Background

Section 7.7.3.1 stated that `E_OS_ID` is returned by OSEK OS service calls when the object is invalid. Under the protection scheme a service call can be invalid because

the caller does not have valid permissions for the object (a new meaning for multi-OS-Application systems).
This is a similar case to an object not being accessible in OSEK OS (for example, when a task tries to get a resource which exists in the system but has not been configured as used by the task).

1.1.13 Requirements

[SWS_Os_00056] ⌈If an OS-object identifier is the parameter of an Operating System module's system service, and no sufficient access rights have been assigned to this OS-object at configuration time (Parameter `Os[...]AccessingApplication`) to the calling Task/Category 2 ISR, the Operating System module's system service shall return `E_OS_ACCESS`.⌋ `(SRS_Os_11001, SRS_Os_11010, SRS_Os_11013)`

[SWS_Os_00449] ⌈CheckTaskMemoryAccess and CheckIsrMemoryAccess check the memory access. Memory access checking is possible for all OS-Applications and from all OS-Applications and does not need granted rights.⌋ `( )`

SWS_Os_00449 is an exception to SWS_Os_00056.

[SWS_Os_00450] ⌈CheckObjectAccess checks the access rights for Operating System objects. Checking object access is possible for all OS-Applications and from all OS-Applications and does not need granted rights.⌋ `( )`

SWS_Os_00450 is an exception to SWS_Os_00056.

7.7.4 Protecting the Hardware used by the OS

7.7.4.1 Background & Rationale

Where a processor supports privileged and non-privileged mode it is usually the case that certain registers, and the instructions to modify those registers, are inaccessible outside the privileged mode.

On such hardware, executing the Operating System module in privileged mode and Tasks/ISRs in non-privileged mode protects the registers fundamental to Operating System module operation from inadvertent corruption by the objects executing in non-privileged mode. The Operating System module's services will need to execute in privileged mode as they will need to modify the registers that are protected outside this mode.

The Operating System module can use the control registers of the MPU, timer unit(s), interrupt controller, etc. and therefore it is necessary to protect those registers against non-trusted OS-Applications.

7.7.4.2 Requirements

[SWS_Os_00058] ⌈If supported by hardware, the Operating System module shall execute non-trusted OS-Applications in non-privileged mode. ⌋ ( )

[SWS_Os_00096] ⌈As far as supported by hardware, the Operating System module shall not allow non-trusted OS-Applications to access control registers managed by the Operating System module. ⌋ (SRS_Os_11011)

[SWS_Os_00245] ⌈If an instruction exception occurs (e.g. division by zero) the Operating System module shall call the protection hook with `E_OS_PROTECTION_EXCEPTION`. ⌋ (SRS_Os_11011)

7.7.4.3 Implementation Notes

When the Operating System module is running non-trusted OS-Applications, the Operating System module's treatment of interrupt entry and hook routines must be carefully managed.

Interrupt handling: Where the MCU supports different modes (as discussed in this section) ISRs will require the Operating System module to do extra work in the `ISR()` wrapper. ISRs will typically be entered in privileged mode. If the handler is part of a non-trusted OS-Application then the `ISR()` wrapper must make sure that a switch to non-privileged mode occurs before the handler executes.

7.7.5 Providing »Trusted Functions«

7.7.5.1 Background & Rationale

An OS-Application can invoke a Trusted Function provided by (another) trusted OS-Application. That can require a switch from non-privileged to privileged mode. This is typically achieved by these operations:

Each trusted OS-Application may export services which are callable from other OS-Applications.
During configuration these trusted services must be configured to be called from a non-trusted OS-Application.
The call from the non-trusted OS-Application to the trusted service is using a mechanism (e.g. trap/software interrupt) provided by the Operating System. The

service is passed as an identifier that is used to determine, in the trusted environment, if the service can be called.

The Operating System offers services to check if a memory region is write/read/execute accessible from an OS-Application. It also returns information if the memory region is part of the stack space.

The Operating System software specification does not provide support for »non-trusted services«.

7.7.5.2 Requirements

[SWS_Os_00451] ⌈The Operating System module shall allow exporting services from trusted OS-Applications. ⌋ ( )

The Operating System module provides the service `CallTrustedFunction()` (see SWS_Os_00097) to call a trusted function from a (trusted or non-trusted) OS-Application.

[SWS_Os_00100] ⌈If `CallTrustedFunction()` is called and the called trusted function is not configured the Operating System module shall call the ErrorHook with `E_OS_SERVICEID`. ⌋ ( )

The Operating System module provides the services `CheckISRMemoryAccess()` and `CheckTaskMemoryAccess()` (see SWS_Os_00512 and SWS_Os_00513) for OS-Applications to check if a memory region is write/read/execute accessible from a Task/Category 2 ISR and also return information if the memory region is part of the stack space.

7.8 Protection Error Handling

7.8.1 Background & Rationale

The Operating System can detect protection errors based on statically configured information on what the constituent parts of an OS-Application can do at runtime. See Section 7.7.

Unlike monitoring, protection facilities will trap the erroneous state at the point the error occurs, resulting in the shortest possible time between transition into an erroneous state and detection of the fault. The different kinds of protection errors are described in the glossary. If a protection error occurs before the Operating System module is started the behaviour is not defined. If a protection error happens during shutdown, e.g. in the application-specific shutdown hook, an endless loop between the shutdown service and the protection hook may occur.

In the case of a protection error, the Operating System module calls a user provided Protection Hook for the notification of protection errors at runtime. The Protection Hook runs in the context of the Operating System module and must therefore be trusted code.

The Operating System module itself needs only to detect an error and provide the ability to act. The Protection Hook can select one out of four options the Operating System module provides, which will be performed after returning from the Protection Hook, depending on the return value of the Protection Hook. The options are:

1. do nothing
2. forcibly terminate the faulty Task/Category 2 ISR
3. forcibly terminate all tasks and ISRs in the faulty OS-Application
    a. without restart of the OS-Application
    b. with restart of the OS-Application
4. shutdown the Operating System module.

Requirements SWS_Os_00243 and SWS_Os_00244 define the order of the default reaction if no faulty Task/Category 2 ISR or OS-Application can be found, e.g. in the system specific hook routines. Also OS-Applications are only mandatory in Scalability Classes 3 and 4, therefore in other Scalability Classes OS-Applications need not be defined.

Note that forcibly terminating interrupts is handled differently in “forcibly terminate the faulty ISR” and “forcibly terminate the OS-Application”. If a faulty ISR is forcibly terminated, the current invocation of the ISR is terminated. A subsequent invocation is allowed. If the OS-Application is forcibly terminated, then the interrupt source is also disabled, preventing subsequent interrupts.

Notes regarding the return value `PRO_IGNORE`

The meaning of "do nothing" (`PRO_IGNORE`) means that the error reaction is ignored. The `PRO_IGNORE` is only allowed in specific situations (currently: arrival rate errors). After the error is detected (e.g. as specified in SWS_Os_00466 or SWS_Os_00467) the protection hook is called. If the hook returns with `PRO_IGNORE` the OS does continue its normal operation. If a service call was the root cause of the violation (e.g. an `ActivateTask`()) and protection hook returns `PRO_IGNORE` the service call shall continue its operation (e.g. to activate a Task) and return `E_OK` (if successful and possible).

Example 1: A task calls `ActivateTask`(B) and causes a arrival rate violation. The activation is not performed (SWS_Os_00466) and protection hook is called. When returning `PRO_IGNORE` the OS continues and the `ActivateTask`() service activates B and returns `E_OK`.

Example 2: A task A calls `SetEvent`() for task B (which currently waits for the event). The OS detects (SWS_Os_00467) an arrival rate violation and performs a call of the protection hook. When the call returns with PRO_IGNORE, the SetEvent() service continues and sets the event. Task B changes to READY state and a rescheduling might happen. The SetEvent() service call will return E_OK to task A.

7.8.2 Requirements

[SWS_Os_00211] ⌈The Operating System module shall execute the ProtectionHook() with the same permissions as the Operating System module. ⌋ ( )

[SWS_Os_00107] ⌈If no ProtectionHook() is configured and a protection error occurs, the Operating System module shall call ShutdownOS().⌋ (SRS_Os_11014)

[SWS_Os_00106] ⌈If the ProtectionHook() returns PRO_IGNORE and was called with E_OS_PROTECTION_ARRIVAL the Operating System module shall return control to the user application. ⌋ (SRS_Os_11014)

[SWS_Os_00553] ⌈If the ProtectionHook() returns PRO_TERMINATETASKISR the Operating System module shall forcibly terminate the faulty Task/Category 2 ISR. ⌋ ( )

[SWS_Os_00554] ⌈If the ProtectionHook() returns PRO_TERMINATEAPPL the Operating System module shall forcibly terminate the faulty OS-Application. ⌋ ( )

[SWS_Os_00555] ⌈If the ProtectionHook() returns PRO_TERMINATEAPPL_RESTART the Operating System module shall forcibly terminate the faulty OS-Application and afterwards restart the OS-Application. ⌋ ( )

[SWS_Os_00556] ⌈If the ProtectionHook() returns PRO_SHUTDOWN the Operating System module shall call the ShutdownOS().⌋ ( )

[SWS_Os_00506] ⌈If the ProtectionHook() is called with E_OS_PROTECTION_ARRIVAL the only valid return values are PRO_IGNORE or PRO_SHUTDOWN [3]. Returning other values will result in a call to ShutdownOS().⌋ ()

[SWS_Os_00475] ⌈If the ProtectionHook() returns PRO_IGNORE and the ProtectionHook() was not called with E_OS_PROTECTION_ARRIVAL then the Operating System module shall call ShutdownOS().⌋ ()

[SWS_Os_00243] ⌈If the ProtectionHook() returns PRO_TERMINATETASKISR and no Task or ISR can be associated with the error, the running OS-Application is

[3] The reason for this case is that the Task which is supervised is not necessary active (and can not be e.g. terminated) and it can be that the caller of the activation is the real problem.

forcibly terminated by the Operating System module. If even no OS-Application can be assigned, `ShutdownOS()` is called. ⌋ (SRS_Os_11014)

[SWS_Os_00244] ⌈If the `ProtectionHook()` returns `PRO_TERMINATEAPPL` or `PRO_TERMINATEAPPL_RESTART` and no OS-Application can be assigned, `ShutdownOS()` is called. ⌋ (SRS_Os_11014)

[SWS_Os_00557] ⌈If the `ProtectionHook()` returns `PRO_TERMINATEAPPL_RESTART` and no `OsRestartTask` was configured for the faulty OS-Application, `ShutdownOS()` is called. ⌋ ()

[SWS_Os_00108] ⌈If the Operating System module forcibly terminates a task, it terminates the task, releases all allocated OSEK resources and calls `EnableAllInterrupts()`/ `ResumeOSInterrupts()` / `ResumeAllInterrupts()` if the Task called `DisableAllInterrupts()` / `SuspendOSInterrupts()` / `SuspendAllInterrupts()` before without the corresponding `EnableAllInterrupts()`/ `ResumeOSInterrupts()` / `ResumeAllInterrupts()` call. ⌋ (SRS_Os_11014)

[SWS_Os_00109] ⌈If the Operating System module forcibly terminates an interrupt service routine, it clears the interrupt request, aborts the interrupt service routine (The interrupt source stays in the current state.) and releases all OSEK resources the interrupt service routine has allocated and calls `EnableAllInterrupts()` / `ResumeOSInterrupts()` / `ResumeAllInterrupts()` if the interrupt called `DisableAllInterrupts()` / `SuspendOSInterrupts()` / `SuspendAllInterrupts()` before without the corresponding `EnableAllInterrupts()`/ `ResumeOSInterrupts()` / `ResumeAllInterrupts()` call. ⌋ (SRS_Os_11014)

[SWS_Os_00110] ⌈If the Operating System module shall forcibly terminates an OS-Application, it:shall

- forcibly terminate all Tasks/ISRs of the OS-Application AND
- cancel all alarms of the OS-Application AND
- stop schedule tables of the OS-Application AND
- disable interrupt sources of Category 2 ISRs belonging to the OS-Application⌋ (SRS_Os_11014)

[SWS_Os_00111] ⌈When the Operating System module restarts an OS-Application, it shall activate the configured `OsRestartTask.` ⌋ ()

7.9 Operating System for Multi-Core

This chapter specifies some extensions that allow to use an AUTOSAR system on Multi-Core micro-processors. It describes the main philosophy as well as additional extensions to the existing OS functionality regarding Multi-Core. The following chapter contains a specification of a new mechanism within the OS called IOC (Inter OS-Application Communicator) that supports the communication between OS-Applications located on the same or on different cores

7.9.1 Background & Rationale

The existing AUTOSAR-OS is based on the OSEK/VDX Operating system which is widely used in the automotive industry. The AUTOSAR Multi-Core OS is derived from the existing AUTOSAR OS.
The Multi-Core OS in AUTOSAR is not a virtual ECU concept, instead it shall be understood as an OS that shares the same configuration and most of the code, but operates on different data structures for each core.
To reduce the memory footprint all cores should use the same code base. Sometimes it can be beneficial to spend some more ROM/Flash, e.g. to use a local ROM, and "double" parts of the code to get faster ROM/Flash access.

7.9.1.1 Requirements

[SWS_Os_00567] ⌈The generated part of the OS is derived from a single configuration that contains the relevant information for all cores. This implies, that IDs (e.g. TASKID, RESOURCEID, …) are unique across cores. Every ID shall refer exactly to one entity independent from the core on which the entity is accessed. This applies also to objects that cannot be shared between cores. ⌋ (SRS_Os_80008)

7.9.2 Scheduling

The priority of the TASKs drives the scheduling. Since multiple cores run truly parallel, several TASKs can execute at the same time.

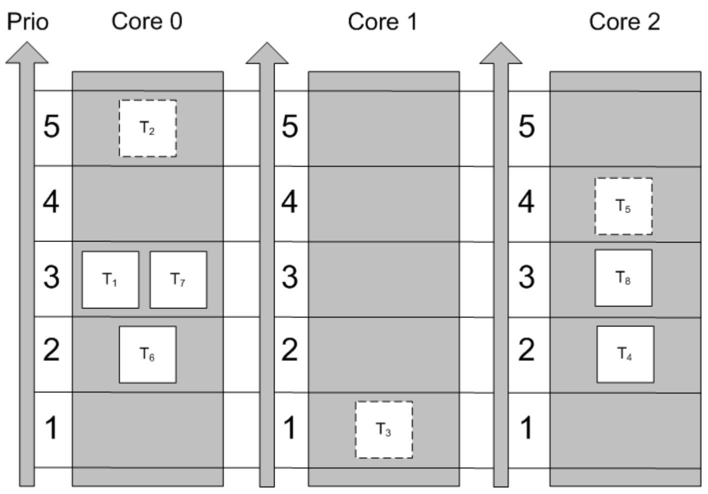

Figure 2: Priorities are assigned to TASKS. The cores schedule independently from each other. The TASKS T2, T3 and T5 are executed in true parallelism. TASKs with the same priority on the same core will be executed in order of activation; TASKs with the same priority on different cores may not be executed in the order of activation, since the cores schedule independent from each other.

The OS can be entered on each core in parallel. This optimizes scalability towards multiple cores. The cores schedule independently. This implies that the schedule on one core does not consider the scheduling on the other cores[4]. A low priority TASK on one core may run in parallel with a high priority TASK on another core.
TASKs and ISRs cannot dynamically change cores by means of the scheduling algorithm.

7.9.2.1 Requirements

[SWS_Os_00568] ⌈Implementations shall be able to independently execute a TASK or an ISR on each started AUTOSAR OS core in parallel. ⌋ (SRS_Os_80001)

[SWS_Os_00569] ⌈The scheduling strategy as defined in AUTOSAR OS shall apply for each individual core in a Multi-Core system, for the TASKs and ISR assigned to the core. ⌋ (SRS_Os_80001, SRS_Os_80013)

7.9.3 Locatable entities (LE)

A locatable entity is an entity that has to be located entirely on one core. The assignment of LEs to cores is defined at configuration time (`OsApplicationCoreRef`).

In this release of the AUTOSAR standard OS-Applications shall be the LEs. Because every TASK has to run on some core, the usage of OS-Applications becomes obligatory in AUTOSAR R4.0 for Multi-Core systems. BSW modules are not allowed to ignore OS-Applications, even if they do not use any protection mechanisms. This is independent from the SC class.

[4] This also applies to TASKs with the same priority, bound to different cores. It also means that non-preemptive tasks cannot be preempted on the core they are running, but tasks on other cores can run in parallel.

As is stated in the AUTOSAR Specification of the Operating System, if OS-Applications are used, all Tasks, ISR etc. must belong to an OS-Application. This implies, that no AUTOSAR software exists outside of an OS-Application in Multi-Core systems.

On single-core systems OS-Applications are available only for SC3 and SC4 because the mechanism is used to support memory protection and implies the usage of extended mode. In Multi-core systems OS-Applications are always available independend of memory protection and on SC1 standard mode shall be possible.

7.9.3.1 Requirements

[SWS_Os_00570] ⌈All TASKs that are assigned to the same OS-Application shall execute on the same core. ⌋ (SRS_Os_80003, SRS_Os_80005)

[SWS_Os_00571] ⌈All ISRs that are assigned to the same OS-Application shall execute on the same core. ⌋ (SRS_Os_80003, SRS_Os_80005)

[SWS_Os_00572] ⌈ISR balancing (if supported by the HW) shall be switched off at boot time by the OS. ⌋ (SRS_Os_80005, SRS_Os_80006)

[SWS_Os_00764] ⌈The OS module shall support OS-Applications in case of Multi-Core also for SC1 and SC2. ⌋ ( )

[SWS_Os_00763] ⌈In an SC1 system standard mode shall be possible. ⌋ ( )

[SWS_Os_00573] ⌈The binding of OS-Applications to cores shall be configured within the OS-Application container. ⌋ (SRS_Os_80003, SRS_Os_80005)

A new configuration item: `OsApplicationCoreRef` within the OS-Application container shall be used to define the core to which the OS-Application is bound. The OS generator will map the configuration parameter “CORE” to a certain core, so that all OS-Applications with the same configuration parameter reside on the same core.

7.9.4 Multi-Core start-up concept

The way cores are started depends heavily on the hardware. Typically the hardware only starts one core, referred as the master core, while the other cores (slaves) remain in halt state until they are activated by the software.

In contrast to such a master-slave system other boot concepts with cores that start independently from each other are conceivable. However it is possible to emulate master-slave behavior on such systems by software.

The AUTOSAR Multi-Core OS specification requires a system with master-slave start-up behavior, either supported directly by the hardware or emulated in software. The master core is defined to be the core that requires no software activation, whereas a slave core requires activation by software.

In Multi-Core configurations, each slave core that is used by AUTOSAR must be activated before `StartOS` is entered on the core. Depending on the hardware, it may be possible to only activate a subset of the available cores from the master. The slave cores might activate additional cores before calling `StartOS`. All cores that belong to the AUTOSAR system have to be activated by the designated AUTOSAR API function. Additionally, the `StartOS` function has to be called on all these cores.

If a core is activated it executes some HW and compiler specific operations, before the "`main`" function is called. In case the same "`main`" function is executed on each core, the cores have to be differentiated by their specific core Id within the function.

Example:
```
void main ()
{
 StatusType rv;
 […]
 switch (GetCoreID())
 {
  case OS_CORE_ID_MASTER:
   […]
   StartCore(OS_CORE_ID_0, &rv);
   StartOS(OSDEFAULTAPPMODE);
   break;
  case OS_CORE_ID_0:
   […]
   StartCore(OS_CORE_ID_1, &rv);
   StartOS(DONOTCARE);
   break;
  otherwise:
   StartOS(DONOTCARE);
 }
}
```

`StartOS` synchronizes all cores twice. The first synchronization point is located before the StartupHooks are executed, the second after the OS-Application specific StartupHooks have finished and before the scheduler is started. The exact point where the second synchronization occurs depends on the implementation, but it shall be before the scheduling is started. This release of the AUTOSAR specification does not support timeouts during the synchronization phase. Cores that are activated with `StartCore` but do not call `StartOS` may cause the system to hang. It is in the responsibility of the integrator to avoid such behavior.

As shown in Figure 3, the `StartUpHook` is called on every core right after the first synchronization. However, there is only one `StartUpHook` in the system. If, for example, core-individual functionality must be executed during `StartupHook` the `GetCoreID` function can be used to discriminate the individual cores. After the global StartUpHook has finished each core performs the StartUpHooks of its OS-Applications . Since OS-Applications are bound to cores the OS-Application specific StartUpHooks are executed only on the core to which the corresponding OS-Application is bound.

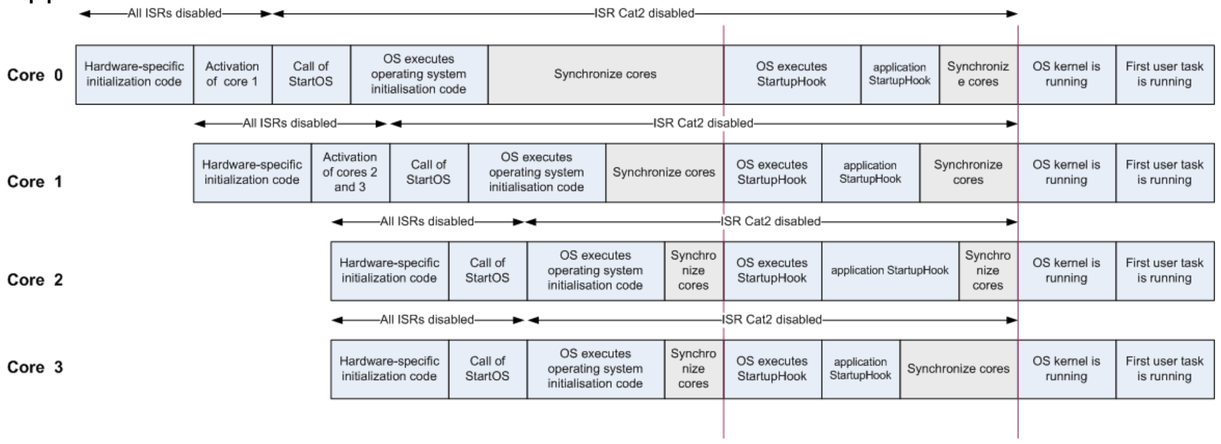

Figure 3: This figure shows an example of an initialization process with 4 cores.

7.9.4.1 Requirements

[SWS_Os_00574] ⌈The master core shall be able to activate cores. ⌋ (SRS_Os_80006, SRS_Os_80026, SRS_Os_80027)

[SWS_Os_00575] ⌈Any slave core shall be able to activate cores. ⌋ (SRS_Os_80006, SRS_Os_80026, SRS_Os_80027)

[SWS_Os_00576] ⌈It shall be allowed to use only a subset of the cores available on a µC for the AUTOSAR system. ⌋ (SRS_Os_80006, SRS_Os_80026, SRS_Os_80027)

[SWS_Os_00577] ⌈The cores shall boot in master-slave mode. If this is not supported by the hardware, it shall be that the cores boot in parallel and emulate the behavior of a master-slave system. ⌋ (SRS_Os_80006, SRS_Os_80026, SRS_Os_80027)

[SWS_Os_00578] ⌈In case of an emulation a slave core (CoreS), which is controlled by the AUTOSAR OS (AUTOSAR core), shall not enter the main function before another core has activated the slave core by means of `StartCore(CoreS)`. ⌋ (SRS_Os_80006)

[SWS_Os_00579] ⌈All cores that belong to the AUTOSAR system shall be synchronized within the `StartOS` function before the scheduling is started and after the global `StartupHook` is called. ⌋ (SRS_Os_80001, SRS_Os_80006)

[SWS_Os_00580] ⌈All cores that belong to the AUTOSAR system shall be synchronized within the `StartOS` before the global `StartupHook` is called. ⌋ (SRS_Os_80006)

[SWS_Os_00581] ⌈The global `StartupHook` shall be called on all cores immediately after the first synchronisation point. ⌋ (SRS_Os_80006)

[SWS_Os_00582] ⌈The OS-Application-specific StartupHooks shall be called after the global `StartupHook` but only on the cores to which the OS-Application is bound. ⌋ (SRS_Os_80006, SRS_Os_80008)

7.9.5 Cores under control of the AUTOSAR OS

The AUTOSAR OS controls several cores as stated above. It need not control all cores of a µC, however. The maximum number of controlled cores shall be configured within the "OsOS" section of the configuration.

The AUTOSAR OS API provides a `StartCore` function to start the cores under its control. The `StartCore` function takes a scalar value parameter of type `CoreIdType,` specifying the core that shall be started. `StartCore` can be called more than once on the master core and also on slave cores. Each core can only be started once, however. For example:

```
StartusType rv1, rv2;

StartCore(OS_CORE_ID_1, &rv1);
StartCore(OS_CORE_ID_2, &rv2);

if (rv1 != E_OK) || (rv2 != E_OK)
 EnterPanicMode();

StartOS(OSDEFAULTAPPMODE);
```

The `StartOS` function shall be called on all cores that have been activated by `StartCore`. It is not allowed to call `StartCore` from a core that has already called `StartOS`.

Cores that belong to the AUTOSAR system shall be started by the designated AUTOSAR OS API service `StartCore`.

7.9.5.1 Requirements

[SWS_Os_00583] ⌈The number of cores that can be controlled by the AUTOSAR OS shall be configured offline.
A new configuration item (`OsNumberOfCores`) within the “**OsOS**” container is used to specify the maximum number of cores that are controlled by the AUTOSAR OS. If no value for (`OsNumberOfCores`) has been specified the number of cores shall be one. ⌋ (SRS_Os_80001, SRS_Os_80011)

7.9.6 Cores which are not controlled by the AUTOSAR OS

The function `StartNonAutosarCore` can be used both before and after `StartOS`. It is provided to activate cores that are controlled by another OS or no OS at all, AUTOSAR functions shall not be called on these cores, otherwise the behavior is unspecified.

7.9.6.1 Requirements

[SWS_Os_00584] ⌈The AUTOSAR OS shall provide a function called `StartNonAutosarCore` that can be used to start cores, which are not controlled by the AUTOSAR OS. ⌋ (SRS_Os_80006, SRS_Os_80026, SRS_Os_80027)

[SWS_Os_00585] ⌈It shall be possible to activate cores that are not controlled by the AUTOSAR OS before and after calling `StartOS`. ⌋ (SRS_Os_80006, SRS_Os_80026, SRS_Os_80027)

7.9.7 Multi-Core shutdown concept

AUTOSAR supports two shutdown concepts, the synchronized shutdown and the individual shutdown concept. While the synchronized shutdown is triggered by the new API function `ShutdownAllCores()`, the individual shutdown is invoked by the existing API function `ShutdownOS()`.

7.9.7.1 Synchronized shutdown concept

If a TASK with the proper rights calls “`ShutdownAllCores`”, a signal is sent to all other cores to induce the shutdown procedure. Once the shutdown procedure has started on a core, interrupts and TASKs are not further processed, and no scheduling will take place, therefore it makes no sense to activate any TASK, however no error will be generated. It is in the responsibility of the application developer/system integrator to make sure that any preparations for shutdown on application and basic software level are completed before calling “`ShutdownAllCores`”. (e.g. by means of the ECU state manager).

During the shutdown procedure every core executes its OS-Application specific `ShutdownHook` functions, followed by a synchronization point. After all cores have reached the synchronization point the global `ShutdownHook` function is executed by all cores in parallel.

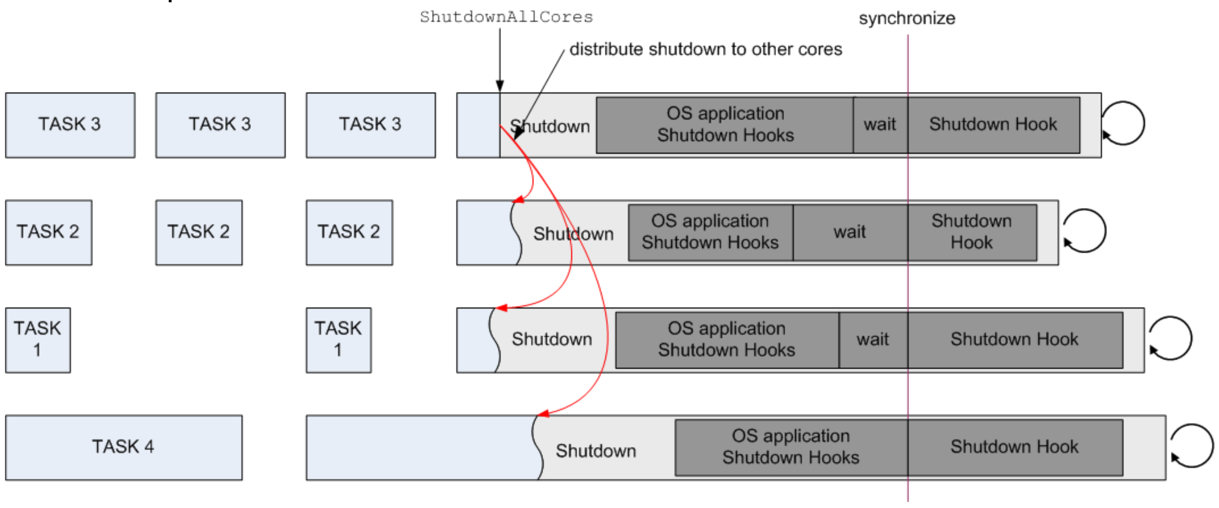

Figure 4: Example of a shutdown procedure.

[SWS_Os_00586] ⌈During the shutdown, the OS-Application specific `ShutdownHook` shall be called on the core on which the corresponding OS-Application is bound. ⌋ (SRS_Os_80007)

[SWS_Os_00587] ⌈Before calling the global `ShutdownHook`, all cores shall be synchronized. ⌋ (SRS_Os_80007)

[SWS_Os_00588] ⌈The global `ShutdownHook` shall be called on all cores. ⌋ (SRS_Os_80007)

7.9.7.2 Individual shutdown concept

If a TASK calls `ShutdownOS` the OS will be shut down on the core on which `ShutdownOS` has been called. Every core shall be able to invoke `ShutdownOS`. Similar to `StartOS` this function will shutdown the individual core. To shutdown the whole ECU `ShutdownOS` has to be called on every core. The function will not return. Individual shutdown is not supported in AUTOSAR R4.x (AUTOSAR mode management will not use it).

7.9.7.3 Shutdown in case of fatal internal errors

In multicore systems it can happen that a fatal internal OS error is detected only on one core. In such cases a local shutdown of that core does not make sense.

[SWS_Os_00762] ⌈In cases where the OS detects a fatal internal error all cores shall be shut down. ⌋ ( )

7.9.8 OS service functionality (overview)

Within this chapter we describe which existing single core AUTOSAR OS functionality has been extended. The following table gives an overview of all standard OS API functions. The column “Multi-Core support” contains one of the following values:

- **Extended:** The function that has been extended substantially to support special Multi-Core functionality.
- **Adapted**: the function required some minor changes but basically remains unchanged.
- **Unchanged:** the behavior of the function has not changed.
- **New:** the function is a new AUTOSAR OS API-function.

Service	Multi-Core support	Annotation
ActivateTask	Extended	Cross core use shall be supported.
AllowAccess	Unchanged	Works only on the same core
CallTrustedFunction	Adapted	Function must be bound to the same core
CancelAlarm	Extended	Cross core use shall be supported
ChainTask	Extended	Cross core use shall be supported.
CheckISRMemoryAccess	Unchanged	
CheckObjectAccess	Unchanged	
CheckObjectOwnership	Unchanged	
CheckTASKMemoryAccess	Unchanged	
ClearEvent	Unchanged	
ControlIdle	Unchanged	Is allowed to be called from any core
DisableAllInterrupts	Unchanged	Works only on the same core
EnableAllInterrupts	Unchanged	Works only on the same core
GetActiveApplicationMode	Unchanged	
GetAlarm	Extended	Cross core use shall be supported
GetAlarmBase	Extended	Cross core use shall be supported
GetApplicationID	Unchanged	
GetApplicationState	Extended	Cross core use shall be supported
GetCoreID	New	ID of the current core
GetCounterValue	Adapted	Cross core is not allowed.
GetElapsedValue	Adapted	Cross core is not allowed.
GetEvent	Unchanged	
GetISRID	Unchanged	
GetNumberOfActivatedCores	New	Number of cores activated during startup.
GetResource	Adapted	Nestable with spinlocks
GetScheduleTableStatus	Extended	Cross core use shall be supported.
GetSpinlock	New	Occupy a spinlock
GetTaskID	Unchanged	Works only on the same core
GetTaskState	Extended	Cross core use shall be supported
IncrementCounter	Adapted	Cross core is not allowed.
NextScheduleTable	Unchanged	

ReleaseResource	Adapted	Nestable with spinlocks
ReleaseSpinlock	New	Release a spinlock
ResumeAllInterrupts	Unchanged	Works only on the same core
ResumeOSInterrupts	Unchanged	Works only on the same core
Schedule	Adapted	Check for unreleased spinlocks
SetAbsAlarm	Extended	Cross core use shall be supported
SetEvent	Extended	Cross core use shall be supported.
SetRelAlarm	Extended	Cross core use shall be supported
SetScheduleTableAsync	Unchanged	
ShutdownAllCores	New	Synchronized shutdown.
ShutdownOS	Extended	Support for MC systems
StartCore	New	Start additional core
StartOS	Extended	Support for MC systems
StartNonAutosarCore	New	Start additional core
StartScheduleTableAbs	Extended	Cross core use shall be supported.
StartScheduleTableRel	Extended	Cross core use shall be supported.
StartScheduleTableSynchron	Unchanged	
StopScheduleTable	Extended	Cross core use shall be supported.
SuspendAllInterrupts	Unchanged	Works only on the same core
SuspendOSInterrupts	Unchanged	Works only on the same core
SyncScheduleTable	Unchanged	
TerminateApplication	Extended	Check for unreleased spinlocks. Cross core use shall be supported.
TerminateTask	Adapted	Check for unreleased spinlocks
TryToGetSpinlock	New	Try to occupy a spinlock
WaitEvent	Adapted	Check for unreleased spinlocks

Tab. 2: gives an overview of changes to the OS Service Calles

Service	Task	Cat1 ISR	Cat2 ISR	Error Hook	PreTask Hook	PostTask Hook	Startup Hook	Shutdown Hook	Alarm Callback	Protection Hook
GetNumberOfActivatedCores	✓		✓							
GetCoreID	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
StartCore										
StartNonAutosarCore	✓									
GetSpinlock	✓		✓							
ReleaseSpinlock	✓		✓							
TryToGetSpinlock	✓		✓							

Service	Task	Cat1 ISR	Cat2 ISR	Error Hook	PreTask Hook	PostTask Hook	Startup Hook	Shutdown Hook	Alarm Callback	Protection Hook
`GetNumberOfActivatedCores`	✓		✓							
`ShutdownAllCores`	✓		✓	✓			✓			

Tab. 3: Allowed Calling Context for OS Service Calls

[SWS_Os_00589] ⌈All functions that are not allowed to operate cross core shall return E_OS_CORE in extended status if called with parameters that require a cross core operation. ⌋ (SRS_Os_80013)

7.9.9 GetTaskID

`GetTaskID` can be called both from TASK and ISR2 level. When called from an interrupt routine, on Single-Core systems, `GetTaskID` returns either the interrupted TASK or indicates that no TASK is running. On Multi-Core systems it

1. indicates that no TASK is running on the core or,
2. returns the ID of the interrupted TASK on the core.

7.9.10 Interrupt disabling

Note: All types of interrupts can only be disabled on the local core. This implies that the interrupt flags on other cores remain in their current state. Scheduling continues on the other cores. Running ISRs on other cores continue executing.

7.9.10.1 Requirements

[SWS_Os_00590] ⌈The OS service "`DisableAllInterrupts`" shall only affect the core on which it is called. ⌋ (SRS_Os_80013)

[SWS_Os_00591] ⌈The OS service "`EnableAllInterrupts`" shall only affect the core on which it is called. ⌋ (SRS_Os_80013)

[SWS_Os_00592] ⌈The OS service "`SuspendAllInterrupts`" shall only affect the core on which it is called. ⌋ (SRS_Os_80013)

[SWS_Os_00593] ⌈The OS service "`ResumeAllInterrupts`" shall only affect the core on which it is called. ⌋ (SRS_Os_80013)

[SWS_Os_00594] ⌈The OS service "`SuspendOSInterrupts`" shall only affect the core on which it is called. ⌋ (SRS_Os_80013)

[SWS_Os_00595] ⌈The OS service "`ResumeOSInterrupts`" shall only affect the core on which it is called. ⌋ (SRS_Os_80013)

7.9.11 TASK activation

TASK activation shall be extended to work across cores. This document will not specify any implementation details. This functions timing behavior can be slower when working across cores. If a TASK has to be activated on another core, a scheduling decision is necessary on that core. If the core has not been started an error is generated.

7.9.11.1 Requirements

[SWS_Os_00596] ⌈It shall be possible to activate a TASK that is part of an OS-Application located on another core, as long as the assigned access rights allow it. ⌋ (SRS_Os_80001, SRS_Os_80015)

[SWS_Os_00598] ⌈The call of `ActivateTask` across cores shall behave synchronously, i.e. a call returns after the task has been activated or an error has been detected. It shall not be possible to continue execution on the calling core before `ActivateTask` is accomplished on the remote core. ⌋ (SRS_Os_80015)

[SWS_Os_00599] ⌈In case of an error when calling `ActivateTask` across cores, the error handler shall be called on the core on which `ActivateTask` was originally called. ⌋ (SRS_Os_80015)

[SWS_Os_00816]⌈ The operating system shall provide a asynchronous version of `ActivateTask` which does not return errors to the caller, but only calls the (global) error hook (if configured). The function name shall be `ActivateTaskAsyn`.
⌋ (SRS_Os_80015)

7.9.12 TASK Chaining

TASK chaining shall be extended to work across cores. This document will not specify any implementation details. This function's timing behavior can be slower when working across cores. If a TASK has to be activated on another core, a scheduling decision is necessary on that core. If the core has not been activated, an error is generated.

7.9.12.1 Requirements

[SWS_Os_00600] ⌈It shall be possible to chain a TASK that is part of an OS-Application located on another core, as long as the assigned access rights allow it. ⌋ (SRS_Os_80001, SRS_Os_80015)

7.9.13 EVENT setting

`SetEvent` shall be extended to work across cores. This document will not specify any implementation details. This function's timing behavior can be slower when working across cores. If the core has not been activated, an error is generated.

7.9.13.1 Requirements

[SWS_Os_00602] ⌈It shall be possible to set an EVENT that is part of an OS-Application located on another core, as long as the assigned access rights allow it. ⌋ (SRS_Os_80016)

[SWS_Os_00604] ⌈The call of `SetEvent` across cores shall behave synchronously, i.e. a call returns after the Event has been set or an error has been detected. It shall not be possible to continue execution on the calling core before `SetEvent` is accomplished on the remote core. ⌋ (SRS_Os_80016)

[SWS_Os_00605] ⌈In case of an error when calling `SetEvent` across cores, the error handler shall be called on the core on which `SetEvent` was originally called. ⌋ (SRS_Os_80016)

[SWS_Os_00817]⌈ The operating system shall provide a asynchronous version of `SetEvent` which does not return errors to the caller, but only calls the (global) error hook (if configured). The function name shall be `SetEventAsyn`.
⌋ (SRS_Os_80016)

7.9.14 Activating additional cores

The mechanism by which additional cores can be activated as described in section 7.9.5

7.9.15 Start of the OS

It is necessary to extend the functionality of `StartOS`. This is because `StartOS` is called once on each core. The user provides the so called application mode[5] to the Operating System through the call parameter of `StartOS(AppMode)`. The

[5] This is the application mode of the Operating System and shall not be confused by other application modes defined in the AUTOSAR mode management.

application mode defines which of the configured (startup) objects (Tasks, Alarms, ScheduleTables) the OS automatically starts.

On a Multi-Core system all cores shall run in the same application mode. If `StartOS` is called with the Appmode `DONOTCARE,` the AppMode of the other cores is used. At least one core has to define an AppMode other than `DONOTCARE.`

If the application mode is the same on all cores, `StartOS` will proceed its task. More details can be found in chapter 7.9.4.

7.9.15.1 Requirements

[SWS_Os_00606] ⌈The AUTOSAR specification does not support the activation of AUTOSAR cores after calling `StartOS` on that core. If `StartCore` is called after `StartOS` it shall return with `E_OS_ACCESS` in extended status. ⌋ (SRS_Os_80001)

[SWS_Os_00607] ⌈`StartOS` shall start the OS on the core on which it is called. ⌋ (SRS_Os_80006, SRS_Os_80013)

[SWS_Os_00608] ⌈If more than one core calls `StartOS` with an AppMode other than “`DONOTCARE`”, the AppModes shall be the same. `StartOS` shall check this at the first synchronisation point. In case of violation, `StartOS` shall not start the scheduling, shall not call any `StartupHooks`, and shall enter an endless loop on every core. ⌋ (SRS_Os_80006)

[SWS_Os_00609] ⌈If `StartOS` is called with the AppMode “`DONOTCARE`” the application mode of the other core(s) (differing from “`DONOTCARE`”) shall be used. ⌋ (SRS_Os_80006)

[SWS_Os_00610] ⌈At least one core shall define an AppMode other than “`DONOTCARE`”. ⌋ (SRS_Os_80006)

[SWS_Os_00611] ⌈If the IOC is configured, `StartOS` shall initialize the data structures of the IOC. ⌋ (SRS_Os_80020)

7.9.16 TASK termination

The termination of TASKs requires an additional check: It is not allowed to terminate a TASK while a spinlock is occupied. If `TerminateTask` / ChainTask is called with an occupied spinlock an error is returned.

7.9.16.1 Requirements

If TerminateTask (or ChainTask) is called while the calling TASK holds a spinlock, the behavior is undefined in standard status.

[SWS_Os_00612] ⌈In extended status TerminateTask / ChainTask shall return with an error (E_OS_SPINLOCK), which can be evaluated in the application. ⌋ (SRS_Os_80021)

[SWS_Os_00613] ⌈Spinlocks occupied by TASKS that are terminated in response to a protection hook shall be automatically released. This applies also to the case in which an OS-Application is terminated. ⌋ (SRS_Os_80021)

7.9.17 Termination of OS-Applications

Similar to TASKs an OS-Application cannot be terminated while any of its TASKs occupy a spinlock. In such cases, the lock is automatically released. To avoid an avalanche of error handling, no calls to the ErrorHook are made.

It might be possible that TerminateApplication(A) is called in parallel from different cores. The implementation has to support such a call pattern by executing the first arriving call of TerminateApplication(A) and ignoring any subsequent calls until the termination is completed.

7.9.17.1 Requirements

[SWS_Os_00614] ⌈TerminateApplication shall check if any of the TASKs in theOS-Application have occupied a spinlock. If so, the spinlocks shall be released. ⌋ (SRS_Os_80021)

[SWS_Os_00615] ⌈If TerminateApplication(A) is called in parallel from different cores, the OsApplication “A” is terminated by the first call, any subsequent calls will return with 'E_OK' in standard and extended status without doing anything, until the termination is completed. ⌋ (SRS_Os_80021)

7.9.18 Shutdown of the OS

Every core shall be able to invoke shutdown by using the ShutdownOS function. By calling ShutdownOS only the calling core will enter the shutdown procedure.
If the user wants to shutdown all cores (more or less in parallel) ShutdownAllCores shall be used.

ShutdownOS and ShutdownAllCores will not return.

The OS service `ShutdownOS` is not used by the AUTOSAR mode management in AUTOSAR R4.0. The function is offered for users that run the OS on cores without RTE and without mode management.

7.9.18.1 Requirements

[SWS_Os_00616] ⌈`ShutdownOS` shall be callable from each core running an AUTOSAR OS. ⌋ (SRS_Os_80001, SRS_Os_80007)

[SWS_Os_00617] ⌈`ShutdownOS` shall shutdown the core on which it was called. ⌋ (SRS_Os_80007)

[SWS_Os_00618] ⌈The OS shall not start TASKs of an OS-Application once the shutdown procedure has been entered on a particular core. ⌋ (SRS_Os_80013)

[SWS_Os_00619] ⌈The AUTOSAR OS function `ShutdownOS` shall be callable in parallel on multiple cores. ⌋ (SRS_Os_80013)

[SWS_Os_00620] ⌈`ShutdownOS` shall release all spinlocks which are occupied by the calling core. ⌋ (SRS_Os_80021)

[SWS_Os_00621] ⌈`ShutdownAllCores` shall be callable from each core running an AUTOSAR OS. ⌋ (SRS_Os_80007)

7.9.19 Waiting for EVENTs

The EVENT waiting mechanism must be adapted to the new Multi-Core spinlock functionality:

A TASK might be de-scheduled when calling `WaitEvent`, in which case it would not be able to release the spinlock. `WaitEvent` must therefore check if the calling TASK holds a spinlock. As with RESOURCES, spinlocks cannot be occupied by TASKs in wait state.

7.9.19.1 Requirements

[SWS_Os_00622] ⌈The AUTOSAR Operating System `WaitEvent` API service shall check if it has been called while the calling TASK has occupied a spinlock. In extended status an error `E_OS_SPINLOCK` shall be returned and the TASK shall not enter the wait state. ⌋ (SRS_Os_80021)

7.9.20 Calling trusted functions

Functions can be declared as trusted as part of an OS-Application. They can then only be executed through the `CallTrustedFunction` API function. Assuming that the access rights are configured accordingly, a TASK from OS-Application A can call a trusted function from OS-Application B.

On a Multi-Core system, these trusted function calls from one OS-Application to another are limited to the same core.

7.9.20.1 Requirements

[SWS_Os_00623] ⌈The OS API function `CallTrustedFunction` shall return `E_OS_ACCESS` in extended status if the target trusted function is part of an OS-Application on another core. ⌋ (SRS_Os_80013)

7.9.21 Invoking reschedule

The `Schedule` API service must be adapted to the new Multi-Core spinlock functionality in the same manner as `WaitEvent`.

A TASK shall not actively force a de-scheduling while it occupies spinlocks.

7.9.21.1 Requirements

[SWS_Os_00624] ⌈The AUTOSAR Operating System `Schedule` API service shall check if it has been called while the calling TASK has occupied a spinlock. In extended status an error `E_OS_SPINLOCK` shall be returned and the scheduler shall not be called. ⌋ (SRS_Os_80021)

7.9.22 RESOURCE occupation

The `GetResource` function allows mutual exclusion between TASKs on the same core. The OS generator shall check offline that the TASKs are not on different cores.(see 7.9.30) and the `GetResource` function will check this requirement online.

The priority ceiling protocol (used by `GetResource`) temporarily changes the priority of a TASK. Such an approach fails on Multi-Core systems as the priorities are local to each core. Therefore the ceiling protocol is not sufficient to protect a critical section against access from different cores.

[**SWS_Os_00801**]⌈ If Spinlocks and Resources are locked by a Task/ISR they have to be unlocked in strict LIFO order. ReleaseResource() shall return E_OS_NOFUNC if the unlock order is violated. No other functionality shall be performed.⌋ (SRS_Os_80021)

7.9.23 The CoreID

Every HW assigns a unique physical Id to a core. The physical core Id is the only way to distinguish between cores. The physical core Ids of a µC are not necessarily consecutive and do not necessarily start with zero.

The SW requires a mechanism to identify a core, e.g. to use core specific variables. Because the physical core Id usually can not be used as a direct array index for core specific variables, a logical CoreID is necessary to map physical core Ids to array indexes. In the SW it is not necessary to know the physical core Id, the logical CoreID is sufficient.

The mapping of OSApplications and other SW objects to cores is specified in the configuration files. All such mappings shall be HW independent and therefore shall not be based on the physical core Id but on the logical CoreID.

The function GetCoreID internally maps the physical core Id to the logical CoreID. The mapping is implementation specific. GetCoreID can be either a C function or a macro.

7.9.23.1 Requirements

[SWS_Os_00625] ⌈The AUTOSAR Operating System API function `GetCoreID` shall be callable before `StartOS`. ⌋ (SRS_Os_80006)

[SWS_Os_00626] ⌈An implementation shall offer a function `GetNumberOfActivatedCores` that returns the number of cores running the AUTOSAR OS. ⌋ (SRS_Os_80001)

[SWS_Os_00627] ⌈An implementation shall define a set of constants `OS_CORE_ID_<No>` of the type `CoreIdType` with `<No>` a value from `0` to "`OsNumberOfCores -1`. ⌋ (SRS_Os_80001)

[SWS_Os_00628] ⌈An implementation shall offer a constant `OS_CORE_ID_MASTER` of the type `CoreIdType` that refers to the master core. ⌋ (SRS_Os_80001)

7.9.24 COUNTERs, background & rationale

A COUNTER is represented by a COUNTER value, measured in "ticks", and some COUNTER-specific constants.

Similarly to Single-Core situation, each operating system (on each core) offers at least one COUNTER that is derived from a timer. Therefore, it is possible to define several COUNTERs which belong to different OS-Applications and either resides on the same or different cores.

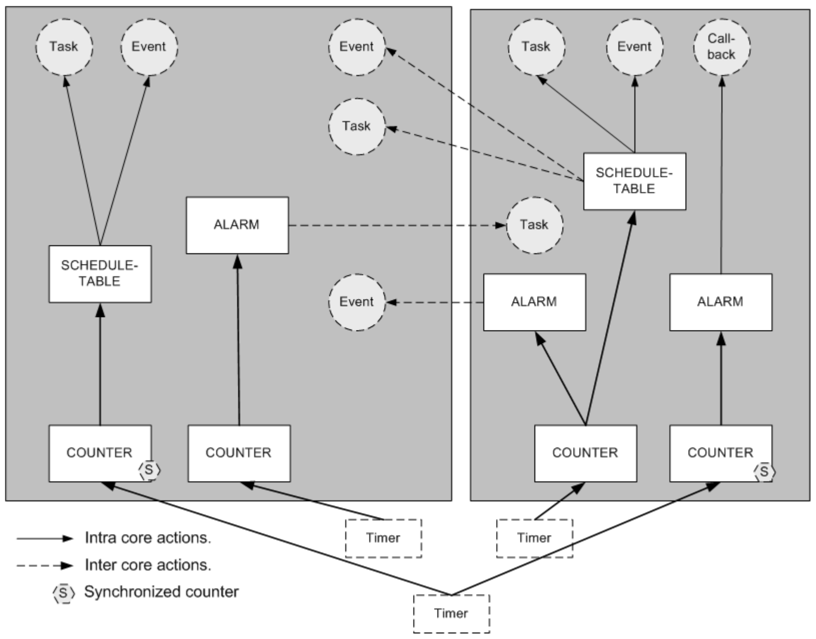

Figure 5: Examples of allowed configurations for COUNTERs, ALARMs, Schedule-tables and ISRs.

7.9.25 Multi-Core restrictions on COUNTERs

The AUTOSAR OS can only increment COUNTERSs on the core on which it resides. A COUNTER which is assigned to an OS-Application *X* cannot be incremented by an OS-Application *Y* if *X* and *Y* are assigned to different cores.

7.9.25.1 Requirements

[SWS_Os_00629] ⌈A COUNTER belonging to an OS-Application shall be incremented by the core on which the OS-Application resides. The COUNTER shall not be incremented by other cores. ⌋ (SRS_Os_80013)

[SWS_Os_00630] ⌈It shall not be allowed to drive a schedule table from a COUNTER, which is assigned to a different core. ⌋ (SRS_Os_80013)

[SWS_Os_00631] ⌈It shall not be allowed to drive an ALARM from a COUNTER, which is assigned to a different core. ⌋ (SRS_Os_80013)

There are two different reasons for these restrictions:

1. Race conditions can occur when cross-core modification of COUNTER is allowed (one core waits for a COUNTER to be modified by another core).
2. The core which is incrementing the COUNTER has to check if ALARMs which are based on the COUNTER have expired. Handling of expired ALARMs is more complex when different cores manipulate the same ALARMs, because mutual exclusion becomes necessary.

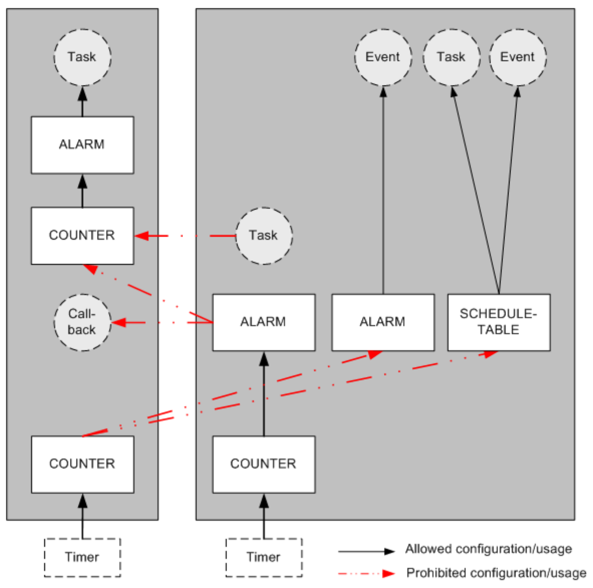

Figure 6: Example of disallowed configurations for COUNTERs, ALARMs, Schedule-tables and Call-backs.

7.9.26 Synchronization of COUNTERs

COUNTERs are used to drive ALARMs and schedule tables. To synchronize ALARMs and schedule tables that reside on different cores, the corresponding COUNTERs have to be synchronized.

For example, if the hardware supports this, it is possible that corresponding free running hardware counters on different cores use the same timer (same counter value maintained by the periperial) and therefor provide the same timebase on different cores. Software COUNTERs can then get advanced by alarms attached to these core local corresponding hardware counters, e.g to drive synchronized schedule tables on different cores.

The quality of the synchronicity depends on the hardware architecture and on the system configuration. .

7.9.27 ALARMs

The ALARM mechanism of the AUTOSAR Operating System provides services to activate TASKs, set EVENTs, increment COUNTERs, or call an ALARM-call-back.

As stated above, ALARMS can only be bound to a COUNTER which resides on the same core. TASKs can be activated and EVENTs can be set with an ALARM action regardless of the core to which the TASK is bound. The access rights defined by OS-Applications have to be respected, however. Additionaly it shall be allowed to manipulate ALARMS when they are bound to other cores. The API-services `SetRelAlarm, SetAbsAlarm, and CancelAlarm` can be used to manipulate parameters of ALARMs on other cores.

7.9.27.1 Requirements

[SWS_Os_00632] ⌈If an ALARM expires, it shall be allowed to activate a TASK on a different core. ⌋ (SRS_Os_80018)

[SWS_Os_00633] ⌈If an ALARM expires, it shall be allowed to set an EVENT on a different core. ⌋ (SRS_Os_80018)

[SWS_Os_00634] ⌈The AUTOSAR Operating System shall process an ALARM on the core on which its corresponding OS-Application resides. ⌋ (SRS_Os_80018)

[SWS_Os_00635] ⌈ALARM callbacks shall be executed on the core to which the ALARM is bound. This is only applicable to SC1 systems, because otherwise Alarm Callback are not allowed (SWS_Os_00242). ⌋ (SRS_Os_80013)

[SWS_Os_00636] ⌈`SetRelAlarm` shall also work on an ALARM that is bound to another core. ⌋ (SRS_Os_80013)

[SWS_Os_00637] ⌈`SetAbsAlarm` shall also work on an ALARM that is bound to another core. ⌋ (SRS_Os_80013)

[SWS_Os_00638] ⌈`CancelAlarm` shall also work on an ALARM that is bound to another core. ⌋ (SRS_Os_80013)

[SWS_Os_00639] ⌈`GetAlarmBase` shall also work on an ALARM that is bound to another core. ⌋ (SRS_Os_80013)

[SWS_Os_00640] ⌈`GetAlarm` shall also work on an ALARM that is bound to another core. ⌋ (SRS_Os_80013)

7.9.28 Schedule tables

Similarly to ALARMs, schedule tables can be used to activate TASKs and set EVENTs. As with ALARMs, a schedule table can only be bound to a COUNTER which resides on the same core.

To simplify system startup, it should be possible to start schedule tables on other cores. The system designer is responsible for the correct handling of schedule tables. For example, schedule tables can be controlled from one core.

7.9.28.1 Requirements

[SWS_Os_00641] ⌈A schedule table shall be able to activate a TASK bound on a core other than the one upon which the schedule tables resides. ⌋ (SRS_Os_80018)

[SWS_Os_00642] ⌈A schedule table shall be able to set an EVENT on a core other than the one upon which the schedule tables resides⌋ (SRS_Os_80018)

[SWS_Os_00643] ⌈The AUTOSAR Operating System shall process a schedule table on the core on which its corresponding OS-Application resides. ⌋ (SRS_Os_80013)

[SWS_Os_00644] ⌈The API call "`StartScheduleTableAbs`" shall be able to start schedule tables of OS-Applications residing on other cores. ⌋ (SRS_Os_80018)

[SWS_Os_00645] ⌈The API call "`StartScheduleTableRel`" shall be able to start schedule tables of OS-Applications residing on other cores. ⌋ (SRS_Os_80013)

[SWS_Os_00646] ⌈The API call "`StopScheduleTable`" shall be able to stop schedule tables of OS-Applications residing on other cores. ⌋ (SRS_Os_80013)

[SWS_Os_00647] ⌈The API service "`GetScheduleTableStatus`" shall be able to get the status of a schedule table that is part of an OS-Application residing on a different core. ⌋ (SRS_Os_80013)

7.9.29 The spinlock mechanism

With the Multi-Core concept, a new mechanism is needed to support mutual exclusion for TASKS on different cores. This new mechanism shall not be used between TASKs on the same core because it makes no sense. In such cases the AUTOSAR Operating System returns an error.

A "`SpinlockType`", which is similar to OSEK's "`ResourceType`", shall be used. Spinlocks are configured offline.

A spinlock is a busy waiting mechanism that polls a (lock) variable until it becomes available. Typically, this requires an atomic "test and set" functionality, the details of which are implementation specific.

Once a lock variable is occupied by a TASK/ISR2, other TASKs/ISR2s on other cores shall be unable to occupy the lock variable. The spinlock mechanism will not de-schedule these other TASKs while they poll the lock variable. However it might happen that a TASK/ISR with a higher priority becomes ready while the lock variable is being polled. In such cases the spinning TASK will be interfered. This is illustrated in Figure 7.

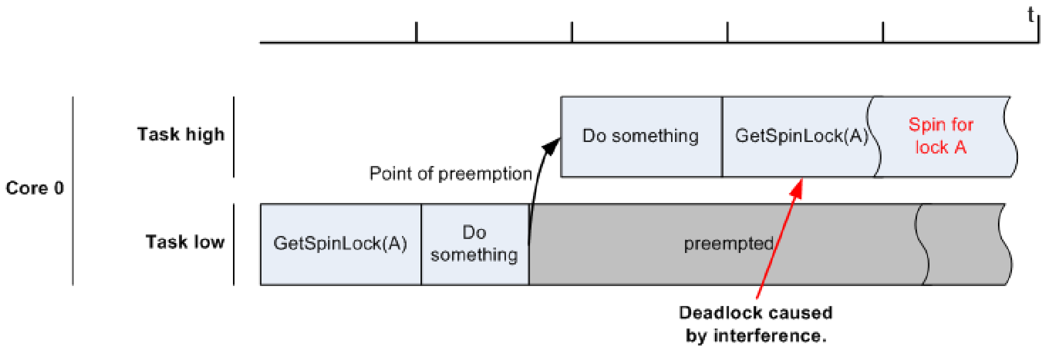

Figure 7: A deadlock situation caused by interference, the high priority TASK spins indefinitely because the low priority TASK has occupied the spinlock. In such cases the second `GetSpinlock` call will return with an error

A user can protect a TASK against such a situation by, for example, rapping the spinlock with `SuspendAllInterrupts`, so that it cannot be interfered by other TASKS. The OS can do this automatically for the caller see configuration parameter OsSpinlockLockMethod (on page 104).
A second deadlock situation can be created by nested spinlocks calls, as illustrated in Figure 8.

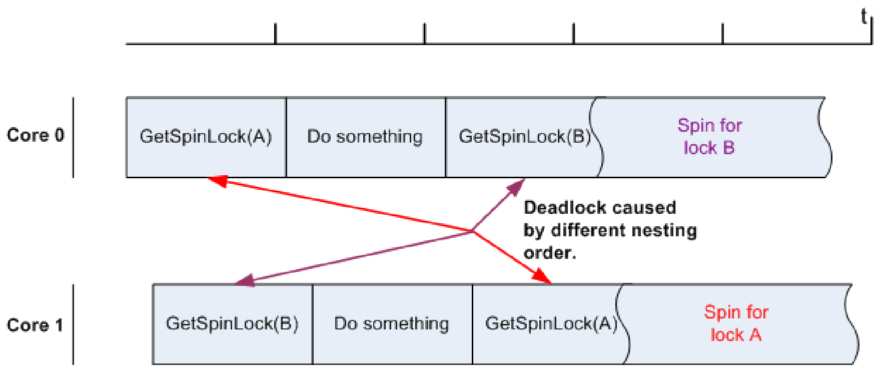

Figure 8: This figure shows a typical deadlock caused by two spinlocks taken in different order by TASKS on two different cores.

To avoid deadlocks it is not allowed to nest different spinlocks. Optionally if spinlocks shall be nested, a unique order has to be defined. Spinlocks can only be taken in this order whereas it is allowed to skip individual spinlocks. Cycles are not allowed within the defined order. This is illustrated in Figure 9.

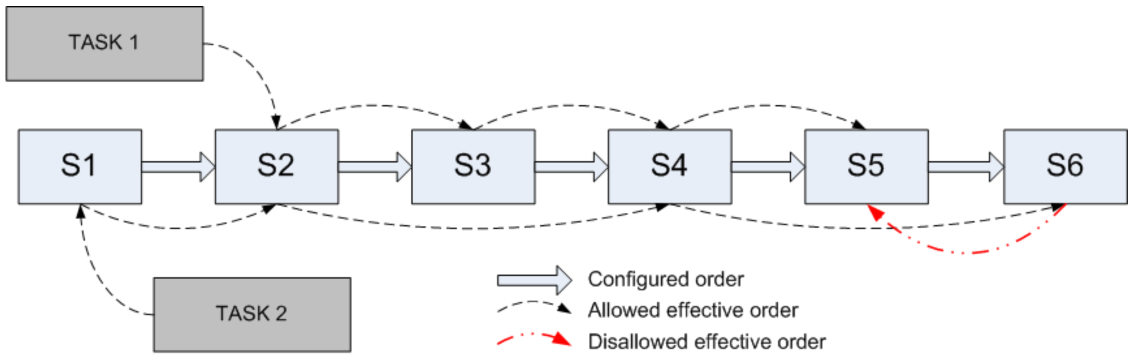

Figure 9: This figure shows an example in which two TASKS have access to a set of spinlocks S1 -- S6. It is allowed to occupy the spinlocks in the predefined order and it is allowed to skip spinlocks. If multiple spinlocks are occupied at the same time, locking and unlocking has to occur in strict LIFO order.

The spinlock mechanism is not deadlock free by itself. The order in which spinlocks from Tasks/ISRs are requested has to be mentioned in the configuration description. If a task occupies a spinlock, scheduling shall be restricted.

Note: AUTOSAR does not prescribe which algorithms are used to implement spinlocks. Since users may want to analyze the timing behavior (e.g. lock times) an implementation shall document the real behavior.

7.9.29.1Requirements

[SWS_Os_00648] ⌈The AUTOSAR Operating System shall provide a spinlock mechanism that works across cores. ⌋ (SRS_Os_80018, SRS_Os_80021)

[SWS_Os_00649] ⌈The AUTOSAR Operating System shall provide a `GetSpinlock` function which occupies a spinlock. If the spinlock is already occupied, `GetSpinlock` shall keep on trying to occupy the spinlock until it succeeds. ⌋ (SRS_Os_80018, SRS_Os_80021)

[SWS_Os_00650] ⌈`GetSpinlock` shall be callable from TASK level. ⌋ (SRS_Os_80018, SRS_Os_80021)

[SWS_Os_00651] ⌈`GetSpinlock` shall be callable from ISR2 level. ⌋ (SRS_Os_80021)

The behavior of `GetSpinlock` is undefined if called from a category 1 ISR

[SWS_Os_00652] ⌈The AUTOSAR Operating System shall provide a `TryToGetSpinlock` function which occupies a spinlock. If the spinlock is already occupied by a TASK, `TryToGetSpinlock` shall return. ⌋ (SRS_Os_80018, SRS_Os_80021)

[SWS_Os_00653] ⌈`TryToGetSpinlock` shall be callable from TASK level. ⌋ (SRS_Os_80018, SRS_Os_80021)

[SWS_Os_00654] ⌈ `TryToGetSpinlock` shall be callable from ISR2 level. ⌋ (SRS_Os_80018, SRS_Os_80021)

[SWS_Os_00655] ⌈The AUTOSAR Operating System shall provide a `ReleaseSpinlock` function which releases an occupied spinlock. If the spinlock is not occupied an error shall be returned. ⌋ (SRS_Os_80018, SRS_Os_80021)

[SWS_Os_00656] ⌈`ReleaseSpinlock` shall be callable from TASK level. ⌋ (SRS_Os_80018, SRS_Os_80021)

[SWS_Os_00657] ⌈`ReleaseSpinlock` shall be callable from ISR2 level. ⌋ (SRS_Os_80018, SRS_Os_80021)

[SWS_Os_00658] ⌈The AUTOSAR Operating System shall generate an error if a TASK tries to occupy a spinlock that is assigned to a TASK/ISR2 on the same core (including itself). ⌋ (SRS_Os_80018, SRS_Os_80021)

[SWS_Os_00659] ⌈The AUTOSAR Operating System shall generate an error if an ISR2 tries to occupy a spinlock that is assigned to a TASK/ISR2 on the same core. ⌋ (SRS_Os_80018, SRS_Os_80021)

[SWS_Os_00660] ⌈A unique order in which multiple spinlocks can be occupied by a TASK/ISR2 on one core should be configurable in the AUTOSAR Operating System. This might be realized by the configuration item (`OsSpinlockSuccessor{NEXT_SPINLOCK}`) where "`NEXT_SPINLOCK`" refers to the consecutive spinlock. (See page 202) ⌋ (SRS_Os_80018, SRS_Os_80021)

[SWS_Os_00661] ⌈The AUTOSAR Operating System shall generate an error if a TASK/ISR2 on a core, where the same or a different TASK/ISR already holds a spinlock, tries to seize another spinlock that has not been configured as a direct or indirect successor of the latest acquired spinlock (by means of the `OsSpinlockSuccessor` configuration parameter) or if no successor is configured. ⌋ (SRS_Os_80018, SRS_Os_80021)

7.9.30 Offline checks

AUTOSAR RESOURCES cannot be shared between TASKs/ISRs on different cores. The OS generator has to check if a user tries to assign a RESOURCE to TASKs on different cores and stop the generation process with an error.

COUNTERS cannot be accessed from OS-Applications on different cores. The OS generator has to reject configurations that violate this rule.

The linked list of spinlocks must be free of cycles to allow correct nesting of spinlocks in order to prevent deadlocks.

The OS generator tool must check that an OSApplication does not get assigned to a non existing core. Additional checks at configuration time, e.g. by an AUTOSAR description editor are recommended.

7.9.30.1Requirements

[OS662] The OS generator tool shall return with an error if it detects a RESOURCE referred to by any TASKs or ISRs assigned to different cores. ⌋ (SRS_Os_80021)

[SWS_Os_00663] ⌈The OS generator tool shall return with an error if an ALARM is assigned to a COUNTER on a different core. ⌋ (SRS_Os_80013)

[SWS_Os_00664] ⌈The OS generator tool shall return with an error if a COUNTER on a different core shall be incremented as an ALARM action. ⌋ (SRS_Os_80013)

[SWS_Os_00665] ⌈The OS generator tool shall return with an error if a schedule table is assigned to a COUNTER on a different core. ⌋ (SRS_Os_80013)

[SWS_Os_00666] ⌈The OS generator tool shall return with an error if the linked list of spinlocks is not free of cycles. ⌋ (SRS_Os_80021)

[SWS_Os_00667] ⌈The OS generator tool shall check the assignement of OsApplications (including the tasks assigned to the OsApplication) to cores and return an error in case any of these cores does not exist. ⌋ (SRS_Os_80005)

7.9.31 Auto start Objects

Before scheduling starts the AUTOSAR Operating System[6] activates all auto-start objects that are configured. This mechanism shall work similar on a Multi-Core system. Before scheduling starts, the Multi-Core OS shall activate all configured auto-start objects on the respective core. Due to the fact that OS-Applications are defined as the locatable entity no further configuration container is required. Auto-start objects are already configured as part of an OS-Application.

[6] StartOS

7.9.31.1 Requirements

[SWS_Os_00668] ⌈The AUTOSAR Operating System shall automatically activate all auto-start TASKs configured for the current AppMode, with respect to the core, before the initial start of the scheduling. ⌋ (SRS_Os_80006)

[SWS_Os_00669] ⌈The AUTOSAR Operating System shall automatically activate all auto-start ALARMs configured for the current AppMode, with respect to the core, before the initial start of the scheduling. ⌋ (SRS_Os_80006)

[SWS_Os_00670] ⌈The AUTOSAR Operating System shall automatically activate all auto-start schedule tables configured for the current AppMode, with respect to the core, before the initial start of the scheduling. ⌋ (SRS_Os_80006)

7.10 Inter-OS-Application Communicator (IOC)

7.10.1 Background & Rationale

IOC stands for Inter OS-Application Communicator.

The "IOC" is responsible for the communication between OS-Applications and in particular for the communication crossing core or memory protection boundaries. Its internal functionality is closely connected to the Operating System.

[SWS_Os_00671] ⌈The IOC implementation shall be part of the Operating System

The IOC is a third type of communication, in addition to
Intra OS-Application communication: Always handled within the RTE
Inter ECU communication: Already available via well defined interfaces to the communication stack (COM) ⌋ (SRS_Os_80020)

Memory protection boundaries are a characteristic of OS-Applications and special communication mechanisms are needed to cross them. Multi-Core systems may also need additional measures to make communication between cores safe.

All AUTOSAR software, both BSW and software components, must belong to an OS-Application (s. 7.9.3), but not necessarily to the same one. It is expected that the BSW will be trusted code, but it shall be defined as one or more OS-Applications.

The IOC provides communication services between OS-Applications and in particular over core boundaries in Multi-Core systems. Because the cross-core communication is always an inter-OS-Application communication, the two mechanisms are combined. An inter OS-Application communication may not necessarily require a cross core communication, however.

Communication between OS-Applications is expected to be more frequent than inter ECU communication. This would be the case when existing; closely related Software Components and their runnable entities are distributed to two or more cores to increase system performance. Meeting timing constraints is expected to become more difficult, when runnables which have been designed to run on a single core are distributed over several cores.

In systems with only one core, the IOC can be omitted completely, if just one OS-Application is available, or if no OS-Application uses memory protection mechanisms.

The IOC does not provide standardized support for measurement of IOC channels.

7.10.2 IOC - General purpose

The IOC provides communication services which can be accessed by clients which need to communicate across OS-Application boundaries on the same ECU.
The RTE uses IOC services to communicate across such boundaries. All communication must be routed through the RTE on sender (or client) and on receiver (or server) side.

Direct access to IOC services by clients other than the RTE is currently not supported, but possible, if the client (e.g. a CDD) provides a hand written or generated IOC Configuration Description as specified and specific callback functions if necessary. Only sender/receiver communication is supported however by the IOC.

Software Components and/or BSW modules located in the same OS-Application (and hence on the same core) should not communicate by invoking IOC services. This would be less efficient than communication via RTE only. However, in case of IOC supported N:1 communication, if not all of the senders and the receiver are in the same OS-Application the IOC must be used.

To keep the RTE as hardware independent as possible, all inter OS-Application and inter core communication mechanisms and implementation variants are encapsulated in the IOC. The IOC internal functionality is dependent on hardware architecture properties, in particular on the memory architecture.

The IOC has to guarantee data consistency in inter OS-Application and inter core (Multi-Core systems) communication, this means in particular:

- In queued communication the sequential order of communication operations shall remain unchanged. In the N:1 communication case, the order of the messages from the different sources is a property of the implementation.

- The content of all data sent in one communication operation shall remain unchanged, i.e. each communication operation shall be treated as atomic operation.
- The lock mechanism (interrupt locks; spinlocks; lock free implementation; ...) which is used by the IOC to guarantee the data consistency is not standardized.

7.10.3 IOC functionality

7.10.3.1 Communication

The IOC provides sender-receiver (signal passing) communication only. The RTE (or adapted BSW modules in a future release of this specification) translates Client-Server invocations and response transmissions into Sender-Receiver communication.

1:1, N:1 and N:M (unqueued only) communication are supported by the IOC.

The IOC allows the transfer of one data item per atomic communication operation. A data item can either be a value for atomic basic data types or a reference for complex data structures. The data structure must be implemented as a single memory block, however. This way the data item can be transmitted in one piece. The IOC does not need to know the internal data structure. The basic memory address and length (which can be calculated from the type of the data item) is sufficient. The IOC does, e.g., not support a conversion of endianness between cores.

Transferring more than one data item in one operation is also supported for 1:1 communication only. In this case several types and memory addresses have to be used by the IOC function. The advantage compared to sequential IOC calls is that mechanisms to open memory protection boundaries and to notify the receiver have to be executed just once. Additionally, all data items are guaranteed to be consistent, because they are transferred in one atomic operation.

The IOC provides both, unqueued (Last-is-Best, data semantics) or queued (First-In-First-Out, event semantics) communication operations. If present, the IOC internal queue has a configurable length.

Each atomic communication operation gets specified individually by its own description block in a Configuration Description with regard to sender, receiver, data type(s), notification, and queuing.

7.10.3.2 Notification

The IOC optionally notifies the receiver as soon as the transferred data is available for access on the receiver side, by calling a configured callback function which gets provided by the user of the communication.

A possible implementation is to trigger an interrupt (Cat. 2) mechanism to invoke the callback function from the ISR on receiver side, or to use a microcontroller supplied trap. The callback function shall be efficient and compact, because it is called from within the ISR.

In certain cases, it might not be necessary to trigger an ISR to notify the receiver. The IOC generator can then select the appropriate IOC internal notification method based on the hardware architecture and other constraints. This might be more efficient than an ISR for communication between OsApplications on the same core.

The notification might be handled completely by the client of the IOC, e.g. when the RTE calls the IOC send function, and then notifies the receiver side RTE that new data are available from the IOC. In this case, the IOC is not affected at all by the details of the notification mechanism.

In case such alternative solutions prove to be more efficient, the IOC internal notification might get removed in future AUTOSAR releases.

7.10.4 IOC interface

The interface between RTE and IOC shall be similar to the interface between Software Components and the RTE, i.e. by generating specific interfaces for each communication operation instead of providing a generic API.

This supports optimization methods (like function inlining or replacing function calls by macros) much better than standardized interfaces. Most of the optimization can be performed offline at code generation time instead of consuming valuable real-time resources.

There is a unique set of IOC service APIs (at least to send and receive data) for each data communication specified in the IOC Configuration Description. Each service API gets generated and can be identified by a unique Id for each data communication. In case of N:1 communication, each sender must use its own API.

The same IOC service API and hence the same 1:1 communication can get used by more than one runnable inside the same SWC both on sender and on receiver side. However, the IOC functions are not reentrant, because otherwise e.g. spinlock errors could occur in case the IOC uses spinlocks in Multi-Core systems. The same IOC API must therefore only be called sequentially. This is no problem, if all runnable entities are scheduled within the same TASK, otherwise the caller is responsible to guarantee that the same IOC API is not called again before it returns from a different invocation.

Software Components may access the IOC only via RTE. Only the RTE decides which communication services to use to support the communication needs of Software Components.

Direct access to IOC services by BSW modules is not supported, but allowed for CDDs and other modules, if unavoidable. The clients have to provides a hand written or generated IOC Configuration Description as specified. In case of notification of the receiver, a specific callback function has to be specified and provided by the client. Only sender/receiver communication is supported however by the IOC.

7.10.5 IOC internal structure

This section gives some hints on possible IOC implementation options.

The IOC may enter the privileged mode to cross the protection boundaries between OS-Applications. The IOC therefore has to be part of the OS. Note that functionality that is placed in the kernel context might be non-interruptible by TASKs or ISR2. The functionality can be interrupted by Cat1 ISRs, however.

The IOC send service writes data into a buffer located in a memory area which is shared with the receiving communication partners (This is one possible implementation example using shared memory). Depending on the hardware architecture and other constraints, different implementation options might be available within the IOC. These options shall be transparent to the client (RTE), however.

The IOC ensures data consistency, i.e. there is a protection against concurrent access to the same data from all senders and the receiver for protection against inconsistent behavior and data corruption. The implementation can be hardware dependent.

In systems with shared memory, there can be a specific communication buffer for each data item in a memory section which is shared between the sending and receiving OS-Applications.

If an IOC communication with event semantics (queued) is configured the length of the queue shall be defined.

7.10.6 IOC configuration and generation

Data element specific interfaces between RTE and IOC require extensive code generation. Instead of generating the IOC together with the RTE, a sequential code generation process is used, to separate generic RTE code generation and hardware dependent IOC code generation as much as possible. The following steps shall be performed:

- Step 1: Specify all information about the allocation of Software Components to OS-Applications and cores in the ECU Configuration Description file.
- Step 2: Generate the RTE. The RTE generator creates data element specific IOC services calls and the corresponding IOC Configuration Description blocks (XML format) to specify the communication relations for each data element.
- Step 3: Generate the IOC code, according to the IOC Configuration Description (Step 2) while considering the hardware description files. Additionally, generate a header file (Ioc.h) for inclusion in RTE.c to provide definitions, function prototypes and macros.

Each atomic communication has to be specified in the IOC Configuration Description in a standardized XML format. There is one description block per communication operation specifying:

- Unique identifier
- Data type(s)
- Sender properties
- Receiver properties
- Name of callback function on receiver side in case of notification.
- Whether communication is queued or unqueued (last is best)
- In case of queued communication: Length of the queue

For details see Chapter 10.3

For each inter-OS-Application communication, the RTE generator creates one or more calls to an IOC function to send or receive data, and adds a corresponding description block to the IOC Configuration Description.

There are possibly multiple sources which contribute to the IOC configuration (e.g., RTE, CDD). The main input will come from the RTE generator. Other sources for the IOC Configuration Description (not supported in this specification revision) might be BSW module configuration tools or non-AUTOSAR components, which are allowed to use BSW services.

In ECUs with only one OS-Application, the IOC Configuration Description can be omitted.

7.10.7 IOC integration examples

This section describes two typical use cases that show how the IOC can support communication between OS-Applications. In both examples the OS-Applications are located on different cores of a Multi-Core system.

7.10.7.1 Example 1 - 1:1 sender/receiver communication without notification

One Software Component sends data items in "EVENT" semantics (queued) to another Software Component located on a different core. A runnable entity on the receiver side is invoked periodically (e.g. by an ALARM) and receives the data via RTE (see Figure 10).

Because the communication crosses core boundaries, the RTE invokes the IOC to transfer the data from core 0 to core 1.

On the sending side, the
```
    Rte_Send_<port>_<item> (..., <data>)
```

call is mapped to an
```
    IocSend_<Id> (<data>)
```

call.

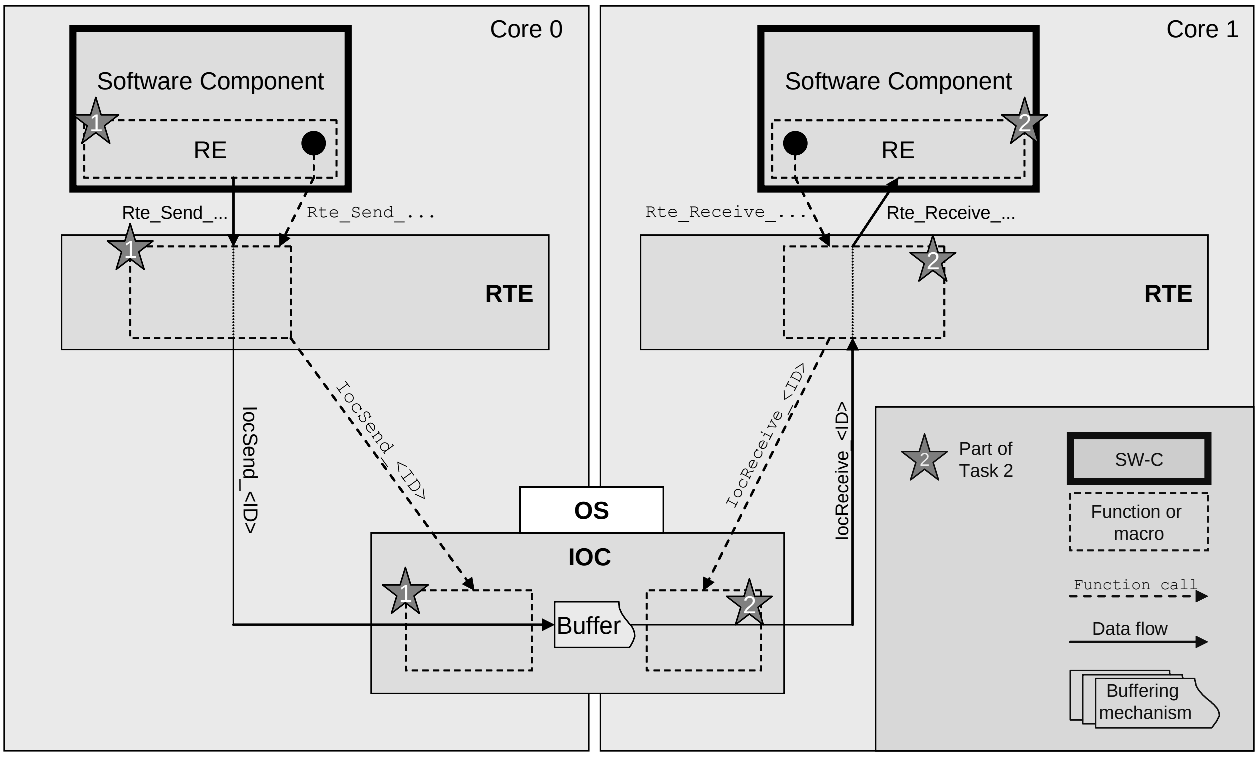

Figure 10: IOC without notification

In this example, the IocSend service writes the data into a buffer, located in a shared memory area which can get read by the receiver via the IOC.

On the receiving side, the receiving runnable gets invoked periodically. The

```
Rte_Receive_<port>_<item> (..., <data>)
```

call is mapped to an

```
IocReceive_<Id> (<data>)
```

call to read data from the IOC internal queue. An additional queue within the RTE is not necessary for 1:1 communication.

The IOC generator generates all the send and receive functions. The functions might be defined as macros for optimization purposes.

This kind of port to port communication without notification is suitable for:

- Sender/receiver communication
- Queued or unqueued communication
- 1:1 communication.

7.10.7.2 Example 2 - N:1 client/server communication with receiver notification by RTE

One Software Component invokes a service operation that is provided by another Software Component located on a different core. A runnable entity on the receiver side is activated to calculate the result (see **Figure 11**).

The RTE realizes the service on client side by mapping the client/server call to a sender/receiver communication. Because the communication crosses core boundaries, the RTE uses the IOC to transfer the data from Core 0 to Core 1.

On the sending side, the

```
    Rte_Call_<port>_<op> (..., <data>)
```

call is mapped to a

```
    IocSend_<Id> (<data>)
```

call to transmit the parameters over the IOC to the core hosting the server runnable.

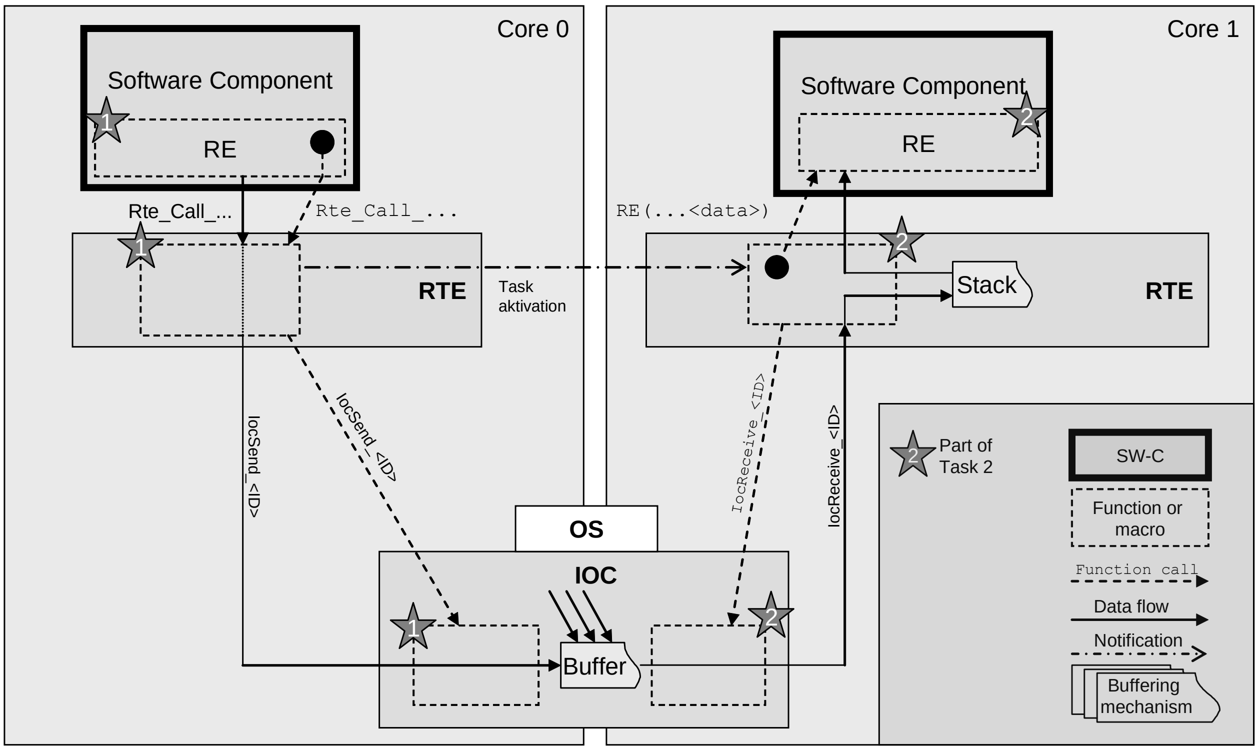

Figure 11: IOC with notification by RTE

After writing the data into the IOC internal queue buffer, the Rte_Call function uses an OS call to notify the receiver by activating the server TASK on the receiving core. This TASK is provided by the RTE. This TASK body is responsible for reading the data from the IOC buffer by calling IocReceive function and for forwarding the data to the server runnable. Depending on the return value of the IOC function, the IocReceive and server runnable calls might be repeated several times to empty the IOC internal queued buffer (if specified).

The result of the service on Core 1 is transferred back to the client on Core 0 in a similar way. The communication path of the result is not displayed in **Figure 11**.

This kind of port to port communication with notification by the RTE is suitable for:

- Sender/receiver communication with notification
- Client/server communication. In this case the RTE has to provide services to map the server call into 1:1 sender/receiver communication for the server call and another sender/receiver communication to return the result to the client
- Queued or unqueued communication
- 1:1 communication, if the receiver does not poll for data periodically (In this case, the solution in example 1 might have been more suitable)
- N:1 communication.

7.10.8 Future extensions

Some features are not supported by the first release of this specification, but might get added in a later release:

- In the future, the IOC will handle direct and efficient communication among BSW modules or between BSW modules and Software Components (via the RTE) located in different OS applications. Additional support of direct access from BSW modules to IOC services will be added.
- Other notification options (like activation of a specified TASK on receiver side) might be added later to the IOC.

7.11 System Scalability

7.11.1 Background & Rationale

In order to customize the operating system to the needs of the user and to take full advantage of the processor features the operating system can be scaled according to the following scalability classes

Feature	Described in Section	Scalability Class 1	Scalability Class 2	Scalability Class 3	Scalability Class 4	Hardware requirements
OSEK OS (all conformance classes)	7.1	✓	✓	✓	✓	
Counter Interface	8.4.17	✓	✓	✓	✓	
SWFRT Interface	8.4.18, 8.4.19	✓	✓	✓	✓	
Schedule Tables	7.3	✓	✓	✓	✓	
Stack Monitoring	7.5	✓	✓	✓	✓	
ProtectionHook	7.8		✓	✓	✓	
Timing Protection	7.7.2		✓		✓	Timer(s) with high priority interrupt
Global Time /Synchronization Support	7.4		✓		✓	Global time source
Memory Protection	7.7.1, 7.7.4			✓	✓	MPU
OS-Applications	7.6, 7.12			✓	✓	
Service Protection	7.7.3			✓	✓	
CallTrustedFunction	7.7.5			✓	✓	(Non-)privileged Modes

Tab. 4: Scalability classes

Feature	Scalability Class 1	Scalability Class 2	Scalability Class 3	Scalability Class 4
Minimum number of Schedule Tables supported	2	8	2	8
Minimum number of OS-Applications supported	0	0	2	2
Minimum number of software Counters supported	8	8	8	8

Tab. 5: Minimum requirements of scalability classes

7.11.2 Requirements

[SWS_Os_00240]⌈If an implementation of a lower scalability class supports features of higher classes then the interfaces for the features must comply with this Operating System software specification. ⌋(SRS_Os_11012, SRS_Os_11016)

[SWS_Os_00241] ⌈The Operating System module shall support the features according to the configured scalability class. (See **Tab. 4**) ⌋ (SRS_Os_11012, SRS_Os_11016)

[SWS_Os_00327]⌈The Operating System module shall always use extended status in Scalability Class 3 and 4. ⌋ ( )

7.12 Hook Functions

7.12.1 Background & Rationale

Hook routines as defined in OSEK OS run at the level of the Operating System module and therefore can only belong to the trusted environment. Furthermore, these hook routines are global to the system (system-specific) and will probably be supplied by the ECU integrator.

In AUTOSAR however, each OS-Application may have the need to execute application specific code e.g. initialize some hardware in its own additional (application-specific) startup hook. These are called application specific hook routines. In general the application specific hooks have the same properties as the hook routines described in the OSEK OS specification. Differences are described below.

7.12.2 Requirements

[SWS_Os_00439] ⌈The Operating System module shall provide the OSEK error macros (`OSError…()`) to all configured error hooks AND there shall be two (like in OIL) global configuration parameters to switch these macros on or off. ⌋ ( )

StartupHook

[SWS_Os_00060] ⌈If an application-specific startup hook is configured for an OS-Application `<App>`, the Operating System module shall call `StartupHook_<App>` on startup of the Operating System module. ⌋ ( )

[SWS_Os_00226] ⌈The Operating System module shall execute an application-specific startup hook with the access rights of the associated OS-Application. ⌋ ( )

[SWS_Os_00236] ⌈If both a system-specific and one (or more) application specific startup hook(s) are configured, the Operating System module shall call the system-specific startup hook before the application-specific startup hook(s). ⌋ ( )

ShutdownHook

[SWS_Os_00112] ⌈If an application-specific shutdown hook is configured for an OS-Application `<App>`, the Operating System module shall call `ShutdownHook_<App>` on shutdown of the OS. ⌋ ( )

[SWS_Os_00225] ⌈The Operating System module shall execute an application-specific shutdown hook with the access rights of the associated OS-Application. ⌋ ( )

[SWS_Os_00237] ⌈If both a system-specific and one (or more) application specific shutdown hook(s) are configured, the Operating System module shall call the system-specific shutdown hook after the application-specific shutdown hook(s). ⌋ ( )

Error Hook

[SWS_Os_00246] ⌈When an error occurs AND an application-specific error hook is configured for the faulty OS-Application `<App>`, the Operating System module shall call that application-specific error hook `ErrorHook_<App>` after the system specific error hook is called (if configured). ⌋ (SRS_Os_11013)

[SWS_Os_00085] ⌈The Operating System module shall execute an application-specific error hook with the access rights of the associated OS-Application. ⌋ ( )

[SWS_Os_00367] ⌈Operating System module's services which do not return a `StatusType` - except `ActivateTaskAsyn` and `SetEventAsyn` - shall not raise the error hook(s). ⌋ ( )

7.13 Hardware peripheral access

7.13.1 Background & Rationale

On some MCU architectures, there are memory mapped hardware registers (peripheral area), which are only accessible in specific modes (e.g. in privileged mode). As long as a Tasks/ISRs is running with full hardware access they can directly access these registers. If memory protection is used by the Operating System, Task/ISRs of non-trusted Os-Applications cannot access such registers directly because this would be recognized as a memory violation by the Operating System.

To allow access to such registers even from non-trusted applications the Operating Systems offers the following APIs to read, write and modify registers:

- `StatusType ReadPeripheral8 (AreaIdType Area, const uint8 * Address, uint8  * ReadValue)`
- `StatusType ReadPeripheral16(AreaIdType Area, const uint16 * Address, uint16 * ReadValue)`
- `StatusType ReadPeripheral32(AreaIdType Area, const uint32 * Address, uint32 * ReadValue)`

- `StatusType WritePeripheral8 (AreaIdType Area, uint8 * Address, uint8 WriteValue)`
- `StatusType WritePeripheral16(AreaIdType Area, uint16 * Address, uint16 WriteValue)`
- `StatusType WritePeripheral32(AreaIdType Area, uint32 * Address, uint32 WriteValue)`

- `StatusType ModifyPeripheral8 (AreaIdType Area, uint8 * Address, uint8 Clearmask, uint8 Setmask)`
- `StatusType ModifyPeripheral16(AreaIdType Area, uint16 * Address, uint16 Clearmask, uint16 Setmask)`
- `StatusType ModifyPeripheral32(AreaIdType Area, uint32 * Address, uint32 Clearmask, uint32 Setmask)`

In order to control the access to the registers the access has to be configured for each OsApplication. By this the Os can check during run-time if a caller has sufficient rights.

7.13.2 Requirements

[SWS_Os_00806]⌈ Check access to peripheral registers
The Operating System shall only execute access to peripheral registers via APIs ReadPeripheralX, WritePeripheralX and ModifyPeripheralX if :

1. parameter Address is in range of `OsPeripheralAreaStartAddress` and `OsPeripheralAreaEndAddress`
2. parameter Area is valid
3. the caller is configured to have sufficient rights (`OsPeripheralAreaAccessingApplication`).

⌋ (SRS_Os_11005)

[SWS_Os_00807]⌈ Error handling of peripheral access API
If the Operating System detects an error (see [SWS_Os_00806]) while executing a ReadPeripheralX, WritePeripheralX and ModifyPeripheralX the OS shall return the appropriate StatusType and call the `ErrorHook()`. Otherwise E_OK shall be returned.⌋ (SRS_Os_11005)

7.14 Interrupt source API

7.14.1 Background & Rationale

The Operating System needs to guarantee the scheduling, wherefore it needs to be the only component which accesses the interrupt controller. Therefore it provides to other BSW/CDD components the interfaces DisableInterruptSource, EnableInterruptSource and ClearPendingInterrupt to give access to the interrupt control registers of category 2 ISRs.

The pair of DisableInterruptSource/EnableInterruptSource may be used for two different purposes:
1. A specific interrupt should be masked for a short time (potentially to avoid data consistency problems). A masked request shall be served afterwards, once the interrupt source gets enabled again.
2. Interrupt requests of a specific source should be ignored for a specific time (potentially a longer time e.g. while the CAN driver sleeps). After enabling the source, only new requests should be considered.

7.14.2 Requirements

[SWS_Os_00808]⌈ The Operating System shall provide for each category 2 interrupt source (`OsIsrCategory == CATEGORY_2`) the APIs DisableInterruptSource, EnableInterruptSource and ClearPendingInterrupt. ⌋ (SRS_Os_11011)

DisableInterruptSource/EnableInterruptSource does not support nested calls.

[SWS_Os_00809]⌈ Nested calls of interrupt source control API
The the Operating System shall return E_OS_NOFUNC (in EXTENDED status) in case DisableInterruptSource is called for an interrupt source which is already disabled or EnableInterruptSource is called for an interrupt source which is already enabled. ⌋ (SRS_Os_11011)

[SWS_Os_00810]⌈ Error handling of interrupt source control API
If the Operating System detects an error while executing a DisableInterruptSource, EnableInterruptSource and ClearPendingInterrupt the OS shall return the appropriate StatusType and call the `ErrorHook()`. Otherwise E_OK shall be returned. ⌋ (SRS_Os_11011)

[SWS_Os_00811]⌈ A call of EnableInterruptSource shall enable the requested interrupt source by modifying the interrupt controller registers. Additionally it shall clear the interrupt pending flag.⌋ (SRS_Os_11011)

[SWS_Os_00812]⌈ A call of DisableInterruptSource shall disable the requested interrupt source by modifying the interrupt controller registers.⌋ (SRS_Os_11011)

[SWS_Os_00813]⌈ A call of ClearPendingInterrupt shall clear the interrupt pending flag by modifying the respective interrupt controller registers.⌋ (SRS_Os_11011)

[SWS_Os_00814]⌈ Clearing of pending interrupts shall be restricted to clearing the pending flag in the interrupt controller.⌋ (SRS_Os_11011)

Note: This does not necessarily guarantee that the interrupt request is cleared successfully, i.e. the ISR may still be serviced afterwards. (This may happen due to racing conditions or as the request needs to be cleared in the requesting hardware unit also.)

7.15 Error classification

AUTOSAR BSW modules normaly report their errors to Det (development errors) or Dem (production errors). The OS handles errors differently (see also [15]) and does not report its errors to Dem/Det. If a reporting of errors to Dem/Det is needed the user can perform these actions in the `ErrorHook()`.
The following table contains all error codes which might be reported from the OS (besides those already defined in [15])

Type or error	***Related error code***	***Value***
Service can not be called.	`E_OS_SERVICEID`	`Assigned by implementation`
An invalid address is given as a parameter to a service.	`E_OS_ILLEGAL_ADDRESS`	`Assigned by implementation`
Tasks terminates without a `TerminateTask()` or `ChainTask()` call.	`E_OS_MISSINGEND`	`Assigned by implementation`
A service of the OS is called inside an interrupt disable/enable pair.	`E_OS_DISABLEDINT`	`Assigned by implementation`
A stack fault detected via stack monitoring by the OS	`E_OS_STACKFAULT`	`Assigned by implementation`
A memory access violation occurred	`E_OS_PROTECTION_MEMORY`	`Assigned by implementation`
A Task exceeds its execution time budget	`E_OS_PROTECTION_TIME`	`Assigned by implementation`
A Category 2 ISR exceeds its execution time budget		
A Task/Category 2 ISR arrives before its timeframe has expired	`E_OS_PROTECTION_ARRIVAL`	`Assigned by implementation`
A Task/Category 2 ISR blocks for too long	`E_OS_PROTECTION_LOCKED`	`Assigned by implementation`
A trap occurred	`E_OS_PROTECTION_EXCEPTION`	`Assigned by implementation`
Core is not available	`E_OS_CORE`	`Assigned by implementation`
De-scheduling with occupied spinlock	`E_OS_SPINLOCK`	`Assigned by implementation`
Deadlock situation due to interference	`E_OS_INTERFERENCE_DEADLOCK`	`Assigned by implementation`
Potential deadlock due to wrong nesting	`E_OS_NESTING_DEADLOCK`	`Assigned by implementation`
A null pointer was given as argument	`E_OS_PARAM_POINTER`	`Assigned by implementation`

7.16 ARTI Hook Macros (DRAFT)

The OS shall incorporate special macros that can be used by an ARTI trace tool to insert tracing functionality of any kind.
The hooks for an AUTOSAR CP OS do follow the general structure of ARTI macros:

```
ARTI_TRACE(_contextName, _className, _instanceName, instanceParameter,
_eventName, eventParameter);
```

Some of the parameters are using literal text (Token) rather than a symbolic identifier. This allows a macro definition concatenating these parameters to more specific macros. Passing and evaluating all parameters at run-time would be very costly especially by means of run-time consumption.
Here is a possible implementation of the generic ARTI_TRACE macro as it could be defined by a ARTI trace tool vendor to match the interface of his trace tool:

```
#define ARTI_TRACE(_contextName, _className, _instanceName,
instanceParameter, _eventName, eventParameter) \
    ARTI_TRACE ## _ ## _className ## _ ## _eventName ## _ ## _instanceName
## _ ## _contextName ( (instanceParameter), (eventParameter) )
```

Such an implementation will generate one hook for all the possible combinations of `_className`, `_eventName` and `_contextName` and pass only parameters `instance_id` and `event_value` at run-time.
The parameters' meanings are described in the following.

- _contextName Token, literal text, name of the context. One of the following:
  - NOSUSP indicating that the hook gets called in a context where interrupts are disabled
  - SPRVSR indicating that the called hook may disable interrupts
  - USER indicating the called hook cannot disable interrupts
- _className Token, literal text, name of the class of macros. Predefined classes for an AUTOSAR OS are:
  - AR_CP_OS_APPLICATION starts and stops the application
  - AR_CP_OS_TASKSCHEDULER schedules tasks
  - AR_CP_OS_CAT2DISPATCHER dispatches CAT2 interrupts
  - AR_CP_OS_SERVICECALLS calls service routines
  - AR_CP_OS_SPINLOCK calls spinlocks
- _instanceName Short name of the OS instance as defined in the ARXML.
- instanceParameter Index [uint32] 0..4294967295 of the CPU core as seen by the OS (<Core Index>). Should always start with 0 and count up consecutively. This might be equal to the index of the physical core, but doesn't have to be.
- _eventName Token, literal text, name of the event as defined for a particular class.
- eventParameter A [uint32] 0..4294967295 value as an argument to an event.

Therefore all ARTI macros for an AUTOSAR OS do compile the following template:

```
ARTI_TRACE(_contextName, <AR OS Class Name>, <OS Short Name>, <Core Index>,
<Event Name>, <Event Parameter>)
```

Example of hook call in OS:

```
ARTI_TRACE( NOSUSP, AR_CP_OS_TASKSCHEDULER, OS1, (uint32)GetCoreID(),
OsTask_Activation, (uint32)GetTaskID() );
```

Example of preprocessed output:

```
ARTI_TRACE_NOSUSP_AR_CP_OS_TASKSCHEDULER_OS1_OsTask_Activation(
(uint32)GetCoreID(), (uint32)GetTaskID() );
```

7.16.1 Class AR_CP_OS_APPLICATION

Event description	Application state transition	_eventName	eventParameter
Start	StartOS → ACCESSIBLE	OsApplication_Start	OS Application Index
Termination	ACCESSIBLE → TERMINATED	OsApplication_Terminate	OS Application Index
Restart	ACCESSIBLE → RESTARTING	OsApplication_Restart	OS Application Index
Allow Access	RESTARTING → ACCESSIBLE	OsApplication_AllowAccess	OS Application Index

7.16.2 Class AR_CP_OS_TASKSCHEDULER

The class AR_CP_OS_TASKSCHEDULER contains events allowing the tracing of OS tasks as defined for the AUTOSAR Classic Platform.
ARTI macros of the class AR_CP_OS_TASKSCHEDULER do compile the following template:

```
ARTI_TRACE(_contextName, AR_CP_OS_TASKSCHEDULER, <OS Short Name>, <Core
Index>, <Event Name>, <Task Index>)
```

The <Core Index> for any event in the following table shall represent core id where the corresponding tasks is scheduled on.
The following events are part of the class AR_CP_OS_TASKSCHEDULER:

Event description	Task state transition	_eventName	eventParameter
Termination of a task	Running → Suspended	OsTask_Stop	Task Index
Successful task activation	Suspended → activated (Ready)	OsTask_Activation	Task Index
Start of a task	Activated (Ready) → Running	OsTask_Start	Task Index
Continuation of a terminating task which previously was in the Waiting state	Released (Ready) → Running	OsTask_Resuming	Task Index
Suspension of a terminating task	Running → Waiting	OsTask_Waiting	Task Index

Release of a terminating task	Waiting → Released (Ready)	OsTask_Release	Task Index

7.16.3 Class AR_CP_OS_CAT2DISPATCHER

The class AR_CP_OS_CAT2DISPATCHER contains events allowing the tracing of CAT2 interrupts as defined for the AUTOSAR Classic Platform.
ARTI macros of the class AR_CP_OS_CAT2DISPATCHER do compile the following template:

```
ARTI_TRACE(_contextName, AR_CP_OS_CAT2DISPATCHER, <OS Short Name>, <Core
Index>, <Event Name>, <Isr Index>)
```

The <Core Index> for any event in the following table shall represent core id where the corresponding CAT2 interrupt is scheduled on.
The following events are part of the class AR_CP_OS_CAT2DISPATCHER:

Event description	ISR state transition	_eventName	eventParameter
Start of an CAT2 ISR	→ Running	OsIsr2_Start	CAT2 ISR Index
End of an CAT2 ISR	Running → Terminated	OsIsr2_Stop	CAT2 ISR Index

7.16.4 Class AR_CP_OS_SERVICECALLS

The class AR_CP_OS_SERVICECALLS contains events allowing the tracing of service calls that are relevant for the tracing of schedule events and interrupts.

OS Service Call	_eventName	eventParameter
DisableAllInterrupts()	OsServiceCall_DisableAllInterrupts	0
EnableAllInterrupts()	OsServiceCall_EnableAllInterrupts	0
SuspendAllInterrupts()	OsServiceCall_SuspendAllInterrupts	nesting depth
ResumeAllInterrupts()	OsServiceCall_ResumeAllInterrupts	nesting depth
SuspendOSInterrupts()	OsServiceCall_SuspendOSInterrupts	nesting depth
ResumeOSInterrupts()	OsServiceCall_ResumeOSInterrupts	nesting depth
GetSpinlock()	OsServiceCall_GetSpinlock	SpinlockIdType
TryToGetSpinlock()	OsServiceCall_TryToGetSpinlock	SpinlockIdType
ReleaseSpinlock()	OsServiceCall_ReleaseSpinlock	SpinlockIdType

7.16.5 Class AR_CP_OS_SPINLOCK

Description	_eventName	eventParameter
AUTOSAR Spinlock lock attempt	OsSpinlock_Try	SpinlockId Type
AUTOSAR Spinlock start	OsSpinlock_Start	SpinlockId Type
AUTOSAR Spinlock completion	OsSpinlock_Stop	SpinlockId Type

8 API specification

This chapter contains the APIs offered by the operating system. Note that not all services are available in all scalability classes, and that the behavior of some services is extended for specific scalability classes. For example, API to relatively start a schedule table has an additional check if the schedule table allows implicit synchronization. This check is only performed in SC2 and SC4 where synchronization of schedule tables is supported.

8.1 Constants

8.1.1 Error codes of type StatusType

The following constants are available in a multi-core environment.

[SWS_Os_91007] ⌈

Name:	`AppModeType`		
Type:	`Enumeration`		
Range:	`DONOTCARE`	--	--
Description:	AppMode of the core shall be inherited from another core.		
Available via:	`Os.h`		

⌋ ()

[SWS_Os_91002] ⌈

Name:	`TotalNumberOfCores`		
Type:	`scalar`		
Range:	`1..65535`	`--`	--
Description:	The total number of cores		
Available via:	`Os.h`		

⌋ ()

Additional constants are in section 7.15 and [15].

8.2 Macros

```
OSMEMORY_IS_READABLE(<AccessType>)
OSMEMORY_IS_WRITEABLE(<AccessType>)
OSMEMORY_IS_EXECUTABLE(<AccessType>)
OSMEMORY_IS_STACKSPACE(<AccessType>)
```

These macros return a value not equal to zero if the memory is readable / writable / executable or stack space. The argument of the macros must be of type AccessType. Typically the return value of the service `Check[Task|ISR]MemoryAccess()` is used as argument for these macros.

8.3 Type definitions

8.3.1 ApplicationType (for OS-Applications)

[SWS_Os_00772] ⌈

Name:	ApplicationType		
Type:	uint32		
Range:	INVALID_OSAPPLICATION	--	--
Description:	This data type identifies the OS-Application.		
Available via:	Os.h		

⌋ ()

8.3.2 ApplicationStateType

[SWS_Os_00773] ⌈

Name:	ApplicationStateType		
Type:	scalar		
Range:	APPLICATION_ACCESSIBLE	--	--
	APPLICATION_RESTARTING	--	--
	APPLICATION_TERMINATED	--	--
Description:	This data type identifies the state of an OS-Application.		
Available via:	Os.h		

⌋ ()

8.3.3 ApplicationStateRefType

[SWS_Os_00774] ⌈

Name:	ApplicationStateRefType
Type:	pointer
Description:	This data type points to location where a ApplicationStateType can be stored.
Available via:	Os.h

⌋ ()

8.3.4 TrustedFunctionIndexType

[SWS_Os_00775] ⌈

Name:	TrustedFunctionIndexType
Type:	scalar
Description:	This data type identifies a trusted function.
Available via:	Os.h

⌋ ()

8.3.5 TrustedFunctionParameterRefType

[SWS_Os_00776] ⌈

Name:	TrustedFunctionParameterRefType
Type:	pointer
Description:	This data type points to a structure which holds the arguments for a call to a trusted function.
Available via:	Os.h

⌋ ()

8.3.6 AccessType

[SWS_Os_00777] ⌈

Name:	AccessType
Type:	integral
Description:	This type holds information how a specific memory region can be accessed.
Available via:	Os.h

⌋ ()

8.3.7 ObjectAccessType

[SWS_Os_00778] ⌈

Name:	ObjectAccessType		
Type:	--		
Range:	ACCESS	--	--
	NO_ACCESS	--	--
Description:	This data type identifies if an OS-Application has access to an object.		
Available via:	Os.h		

⌋ ()

8.3.8 ObjectTypeType

[SWS_Os_00779] ⌈

Name:	ObjectTypeType		
Type:	--		
Range:	OBJECT_TASK	--	--
	OBJECT_ISR	--	--
	OBJECT_ALARM	--	--

	OBJECT_RESOURCE	--	--
	OBJECT_COUNTER	--	--
	OBJECT_SCHEDULETABLE	--	--
Description:	This data type identifies an object.		
Available via:	Os.h		

⌋ ()

8.3.9 MemoryStartAddressType

[SWS_Os_00780] ⌈

Name:	MemoryStartAddressType
Type:	--
Description:	This data type is a pointer which is able to point to any location in the MCU address space.
Available via:	Os.h

⌋ ()

8.3.10 MemorySizeType

[SWS_Os_00781] ⌈

Name:	MemorySizeType
Type:	--
Description:	This data type holds the size (in bytes) of a memory region.
Available via:	Os.h

⌋ ()

8.3.11 ISRType

[SWS_Os_00782] ⌈

Name:	ISRType		
Type:	--		
Range:	INVALID_ISR	--	--
Description:	This data type identifies an interrupt service routine (ISR).		
Available via:	Os.h		

⌋ ()

8.3.12 ScheduleTableType

[SWS_Os_00783] ⌈

Name:	ScheduleTableType
Type:	--
Description:	This data type identifies a schedule table.
Available via:	Os.h

⌋ ()

8.3.13 ScheduleTableStatusType

[SWS_Os_00784] ⌈

Name:	ScheduleTableStatusType		
Type:	--		
Range:	SCHEDULETABLE_STOPPED	--	--
	SCHEDULETABLE_NEXT	--	--
	SCHEDULETABLE_WAITING	--	--
	SCHEDULETABLE_RUNNING	--	--
	SCHEDULETABLE_RUNNING_AND_SYNCHRONOUS	--	--
Description:	This type describes the status of a schedule. The status can be one of the following: o The schedule table is not started (SCHEDULETABLE_STOPPED) o The schedule table will be started after the end of currently running schedule table (schedule table was used in NextScheduleTable() service) (SCHEDULETABLE_NEXT) o The schedule table uses explicit synchronization, has been started and is waiting for the global time. (SCHEDULETABLE_WAITING) o The schedule table is running, but is currently not synchronous to a global time source (SCHEDULETABLE_RUNNING) o The schedule table is running and is synchronous to a global time source (SCHEDULETABLE_RUNNING_AND_SYNCHRONOUS)		
Available via:	Os.h		

⌋ ()

8.3.14 ScheduleTableStatusRefType

[SWS_Os_00785] ⌈

Name:	ScheduleTableStatusRefType
Type:	--
Description:	This data type points to a variable of the data type ScheduleTableStatusType.
Available via:	Os.h

⌋ ()

8.3.15 ProtectionReturnType

[SWS_Os_00787] ⌈

Name:	ProtectionReturnType		
Type:	--		
Range:	PRO_IGNORE	--	--

	PRO_TERMINATETASKISR	--	--
	PRO_TERMINATEAPPL	--	--
	PRO_TERMINATEAPPL_RESTART	--	--
	PRO_SHUTDOWN	--	--
Description:	This data type identifies a value which controls further actions of the OS on return from the protection hook.		
Available via:	Os.h		

⌋ ()

8.3.16 RestartType

[SWS_Os_00788] ⌈

Name:	RestartType		
Type:	--		
Range:	RESTART	--	--
	NO_RESTART	--	--
Description:	This data type defines the use of a Restart Task after terminating an OS-Application.		
Available via:	Os.h		

⌋ ()

8.3.17 PhysicalTimeType

[SWS_Os_00789] ⌈

Name:	PhysicalTimeType
Type:	--
Description:	This data type is used for values returned by the conversion macro (see OS393()) OS_TICKS2<Unit>_<Counter>().
Available via:	Os.h

⌋ ()

8.3.18 CoreIdType

[SWS_Os_00790] ⌈

Name:	CoreIdType		
Type:	scalar		
Range:	OS_CORE_ID_MASTER	--	refers to the master core, may be an alias for OS_CORE_ID_<x>
	OS_CORE_ID_0..OS_CORE_ID_65533	--	refers to logical core 0, core 1 etc.
Description:	CoreIdType is a scalar that allows identifying a single core. The CoreIdType shall represent the logical CoreID		
Available via:	Os.h		

⌋ ()

8.3.19 SpinlockIdType

[SWS_Os_00791] ⌈

Name:	`SpinlockIdType`		
Type:	`scalar`		
Range:	`1..65535`	`--`	0x01, 0x02, ...: identifies a spinlock instance
	`INVALID_SPINLOCK`	`--`	represents an invalid spinlock instance
Description:	SpinlockIdType identifies a spinlock instance and is used by the API functions: GetSpinlock, ReleaseSpinlock and TryToGetSpinlock.		
Available via:	`Os.h`		

⌋ ()

8.3.20 TryToGetSpinlockType

[SWS_Os_00792] ⌈

Name:	`TryToGetSpinlockType`		
Type:	`Enumeration`		
Range:	`TRYTOGETSPINLOCK_SUCCESS`	--	Spinlock successfully occupied
	`TRYTOGETSPINLOCK_NOSUCCESS`	--	Unable to occupy the spinlock
Description:	The TryToGetSpinlockType indicates if the spinlock has been occupied or not.		
Available via:	`Os.h`		

⌋ (SRS_Os_80021)

8.3.21 IdleModeType

[SWS_Os_00793] ⌈

Name:	`IdleModeType`		
Type:	`scalar`		
Range:	`IDLE_NO_HALT`	`--`	the core does not perform any specific actions during idle time
Description:	This data type identifies the idle mode behavior.		
Available via:	`Os.h`		

⌋ ()

8.3.22 AreaIdType

[SWS_Os_91000] ⌈

Name:	`AreaIdType`		
Type:	`scalar`		
Range:	`0..65534`	`--`	identifies a peripheral area
Description:	AreaIdType identifies a peripheral area and is used by the API functions: ReadPeripheralX, WritePeripheralX and ModifyPeripheralX		
Available via:	`Os.h`		

⌋ ()

8.4 Function definitions

The availability of the following services is defined in **Tab. 4**. The use of these services may be restricted depending on the context they are called from. See
Tab. 1 for details.

8.4.1 GetApplicationID

[SWS_Os_00016] ⌈

Service name:	GetApplicationID	
Syntax:	`ApplicationType GetApplicationID( void )`	
Service ID[hex]:	0x00	
Sync/Async:	Synchronous	
Reentrancy:	Reentrant	
Parameters (in):	None	
Parameters (inout):	None	
Parameters (out):	None	
Return value:	ApplicationType	<identifier of running OS-Application> or INVALID_OSAPPLICATION
Description:	This service determines the OS-Application (a unique identifier has to be allocated to each application) where the caller originally belongs to (was configured to).	
Available via:	`Os.h`	

⌋ ()

[SWS_Os_00261] ⌈ `GetApplicationID()` shall return the application identifier to which the executing Task/ISR/hook was configured. ⌋ ( )

[SWS_Os_00262] ⌈If no OS-Application is running, `GetApplicationID()` shall return `INVALID_OSAPPLICATION`. ⌋ ()

[SWS_Os_00514] ⌈Availability of `GetApplicationID()`: Available in Scalability Classes 3 and 4 and in multi-core systems. ⌋ ()

8.4.2 GetCurrentApplicationID

[SWS_Os_00797] ⌈

Service name:	GetCurrentApplicationID
Syntax:	`ApplicationType GetCurrentApplicationID( void )`
Service ID[hex]:	0x27
Sync/Async:	Synchronous
Reentrancy:	Reentrant
Parameters (in):	None

Parameters (inout):	None	
Parameters (out):	None	
Return value:	ApplicationType	<identifier of the OS-Application> or INVALID_OSAPPLICATION
Description:	This service determines the OS-Application where the caller of the service is currently executing. Note that if the caller is not within a CallTrustedFunction() call the value is equal to the result of GetApplicationID().	
Available via:	`Os.h`	

⌋ ()

[SWS_Os_00798]⌈ `GetCurrentApplicationID()` shall return the application identifier in which the current Task/ISR/hook is executed. ⌋ ( )

[SWS_Os_00799] ⌈If no OS-Application is running, `GetCurrentApplicationID()` shall return `INVALID_OSAPPLICATION`. ⌋ ( )

[SWS_Os_00800] ⌈Availability of `GetCurrentApplicationID()`: Available in Scalability Classes 3 and 4. ⌋ ( )

8.4.3 GetISRID

[SWS_Os_00511] ⌈

Service name:	GetISRID	

Syntax:	`ISRType GetISRID(`
	`    void`
	`)`	

Service ID[hex]:	0x01	
Sync/Async:	Synchronous	
Reentrancy:	Reentrant	
Parameters (in):	None	
Parameters (inout):	None	
Parameters (out):	None	
Return value:	ISRType	<Identifier of running ISR> or INVALID_ISR
Description:	This service returns the identifier of the currently executing ISR.	
Available via:	`Os.h`	

⌋ ()

[SWS_Os_00263] ⌈If called from category 2 ISR (or Hook routines called inside a category 2 ISR), `GetISRID()` shall return the identifier of the currently executing ISR. ⌋ ( )

[SWS_Os_00264] ⌈If its caller is not a category 2 ISR (or Hook routines called inside a category 2 ISR), `GetISRID()` shall return `INVALID_ISR`. ⌋ ( )

[SWS_Os_00515] ⌈Availability of `GetISRID()`: Available in all Scalability Classes. ⌋ ( )

8.4.4 CallTrustedFunction

[SWS_Os_00097] ⌈

Service name:	CallTrustedFunction	

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

Service ID[hex]:	0x02	
Sync/Async:	Depends on called function. If called function is synchronous then service is synchronous. May cause rescheduling.	
Reentrancy:	Reentrant	
Parameters (in):	FunctionIndex	Index of the function to be called.
	FunctionParams	Pointer to the parameters for the function - specified by the FunctionIndex - to be called. If no parameters are provided, a NULL pointer has to be passed.
Parameters (inout):	None	
Parameters (out):	None	
Return value:	StatusType	E_OK: No Error E_OS_SERVICEID: No function defined for this index
Description:	A (trusted or non-trusted) OS-Application uses this service to call a trusted function	
Available via:	`Os.h`	

⌋ ()

[SWS_Os_00265] ⌈ If <FunctionIndex> is a defined function index, `CallTrustedFunction()` shall call the function <FunctionIndex> out of a list of implementation specific trusted functions with the protection settings of the OS-Application which provides the trusted function AND shall return `E_OK` after completion. ⌋ ( )

[SWS_Os_00312] ⌈Caveats of `CallTrustedFunction()`:

- The called trusted function must conform to the following C prototype: `void TRUSTED_<name_of_the_trusted_service>( TrustedFunctionIndexType,TrustedFunctionParameterRefType);` (The arguments are the same as the arguments of CallTrustedFunction).
- Normally, a user will not directly call this service, but it will be part of some standard interface, e.g. a standard I/O interface.
- It is the duty of the called trusted function to check rights of passed parameters, especially if parameters are interpreted as out parameters.
- It should be noted that the `CallTrustedFunction()` does not disable timing protection for the task which called the service. This may lead to timing faults (calls of the `ProtectionHook()`) even inside of a trusted OS-Application. It is

therefore recommended to use `CallTrustedFunction()` only for stateless functions (e.g. functions which do not write or do not have internal states) ⌋ ( )

[**SWS_Os_00266**] ⌈When CallTrustedFunction() calls the function <FunctionIndex>, that function shall be executed with the same processor mode, memory protection boundaries and the service protection limitations of the OS-Application to which it belongs. The notion of "current application" shall remain that of the calling Task or Category 2 ISR. ⌋ ( )

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.

[SWS_Os_00565] ⌈ When CallTrustedFunction() is called and the caller of CallTrustedFunction() is supervised with timing protection, the Operating System shall delay any timing protection errors until the CallTrustedFunction() returns to a OsApplication with `OsTrustedApplicationDelayTimingViolationCall == FALSE`.⌋ ( )

[SWS_Os_00564] ⌈ If such a violation is detected inside a nested call sequence of CallTrustedFunction() of a task, the delay shall last until the return of CallTrustedFunction() to an OsApplication with `OsTrustedApplicationDelayTimingViolationCall == FALSE`.⌋ ( )

[SWS_Os_00563] ⌈The OperatingSystem shall not schedule any other Tasks which belong to the same OS-Application as the non-trusted caller of the service. It shall be done by priority ceiling. Also interrupts of Category 2 which belong to the same OS-Application shall be disabled during the execution of the service.⌋ ( )

[SWS_Os_00364] ⌈If `CallTrustedFunction()` calls the trusted function from a Category 2 ISR context, that function shall continue to run on the same interrupt priority and be allowed to call all system services defined for Category 2 ISR (see table in chapter 7.7.3.2). ⌋ ( )

[SWS_Os_00365] ⌈If `CallTrustedFunction()` calls the trusted function from a task context, that function shall continue to run on the same priority and be allowed to call all system services defined for tasks (see table in chapter 7.7.3.2). ⌋ ( )

[SWS_Os_00292] ⌈If the function index <FunctionIndex> in a call of `CallTrustedFunction()` is undefined, `CallTrustedFunction()` shall return `E_OS_SERVICEID`. ⌋ ( )

[SWS_Os_00516] ⌈Availability of `CallTrustedFunction()`: Available in Scalability Classes 3 and 4. ⌋ ( )

8.4.5 CheckISRMemoryAccess

[SWS_Os_00512] ⌈

Service name:	CheckISRMemoryAccess	
Syntax:	`AccessType CheckISRMemoryAccess( ISRType ISRID, MemoryStartAddressType Address, MemorySizeType Size )`	
Service ID[hex]:	0x03	
Sync/Async:	Synchronous	
Reentrancy:	Reentrant	
Parameters (in):	ISRID	ISR reference
	Address	Start of memory area
	Size	Size of memory area
Parameters (inout):	None	
Parameters (out):	None	
Return value:	AccessType	Value which contains the access rights to the memory area.
Description:	This service checks if a memory region is write/read/execute accessible and also returns information if the memory region is part of the stack space.	
Available via:	`Os.h`	

⌋ ()

[SWS_Os_00267] ⌈If the ISR reference <ISRID> in a call of `CheckISRMemoryAccess()` is valid, `CheckISRMemoryAccess()` shall return the access rights of the ISR on the specified memory area. ⌋ ( )

[SWS_Os_00313] ⌈If an access right (e.g. “read”) is not valid for the whole memory area specified in a call of `CheckISRMemoryAccess()`, `CheckISRMemoryAccess()` shall yield no access regarding this right. ⌋ ( )

[SWS_Os_00268] ⌈If the ISR reference <ISRID> is not valid, `CheckISRMemoryAccess()` shall yield no access rights. ⌋ ( )

[SWS_Os_00517] ⌈Availability of `CheckISRMemoryAccess()`: Available in Scalability Classes 3 and 4. ⌋ ( )

8.4.6 CheckTaskMemoryAccess

[SWS_Os_00513] ⌈

Service name:	CheckTaskMemoryAccess	

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

Service ID[hex]:	0x04	
Sync/Async:	Synchronous	
Reentrancy:	Reentrant	
Parameters (in):	TaskID	Task reference
	Address	Start of memory area
	Size	Size of memory area
Parameters (inout):	None	
Parameters (out):	None	
Return value:	AccessType	Value which contains the access rights to the memory area.
Description:	This service checks if a memory region is write/read/execute accessible and also returns information if the memory region is part of the stack space.	
Available via:	`Os.h`	

⌋ ()

[SWS_Os_00269] ⌈If the Task reference <TaskID> in a call of `CheckTaskMemoryAccess()` is valid, `CheckTaskMemoryAccess()` shall return the access rights of the task on the specified memory area. ⌋ ( )

[SWS_Os_00314] ⌈If an access right (e.g. "read") is not valid for the whole memory area specified in a call of `CheckTaskMemoryAccess()`, `CheckTaskMemoryAccess()` shall yield no access regarding this right. ⌋ ( )

[SWS_Os_00270] ⌈If the Task reference <TaskID> in a call of `CheckTaskMemoryAccess()` is not valid, `CheckTaskMemoryAccess()` shall yield no access rights. ⌋ ( )

[SWS_Os_00518] ⌈Availability of `CheckTaskMemoryAccess()`: Available in Scalability Classes 3 and 4⌋ ( )

8.4.7 CheckObjectAccess

[SWS_Os_00256] ⌈

Service name:	CheckObjectAccess

Syntax:	`ObjectAccessType CheckObjectAccess(`
	`    ApplicationType ApplID,`

	`ObjectTypeType ObjectType,`
	`void ...`
	`)`	

Service ID[hex]:	0x05	
Sync/Async:	Synchronous	
Reentrancy:	Reentrant	
Parameters (in):	ApplID	OS-Application identifier
	ObjectType	Type of the following parameter
	...	The object to be examined
Parameters (inout):	None	
Parameters (out):	None	
Return value:	ObjectAccessType	ACCESS if the ApplID has access to the object NO_ACCESS otherwise
Description:	This service determines if the OS-Applications, given by ApplID, is allowed to use the IDs of a Task, Resource, Counter, Alarm or Schedule Table in API calls.	
Available via:	`Os.h`	

⌋ ()

[SWS_Os_00271] ⌈If the OS-Application <ApplID> in a call of `CheckObjectAccess()` has access to the queried object, `CheckObjectAccess()` shall return `ACCESS`. ⌋ ( )

[SWS_Os_00272] ⌈If the OS-Application <ApplID> in a call of `CheckObjectAccess()` has no access to the queried object, `CheckObjectAccess()` shall return `NO_ACCESS`. ⌋ ( )

[SWS_Os_00423] ⌈If in a call of `CheckObjectAccess()` the object to be examined is not a valid object OR <ApplID> is invalid OR <ObjectType> is invalid THEN `CheckObjectAccess()` shall return `NO_ACCESS`. ⌋ ( )

[SWS_Os_00519] ⌈Availability of `CheckObjectAccess()`: Available in Scalability Classes 3 and 4. ⌋ ( )

8.4.8 CheckObjectOwnership

[SWS_Os_00017] ⌈

Service name:	CheckObjectOwnership	

Syntax:	`ApplicationType CheckObjectOwnership(`
	`ObjectTypeType ObjectType,`
	`void ...`
	`)`	

Service ID[hex]:	0x06	
Sync/Async:	Synchronous	
Reentrancy:	Reentrant	
Parameters (in):	ObjectType	Type of the following parameter
	...	The object to be examined

Parameters (inout):	None	
Parameters (out):	None	
Return value:	ApplicationType	<OS-Application>: the OS-Application to which the object ObjectType belongs or INVALID_OSAPPLICATION if the object does not exists
Description:	This service determines to which OS-Application a given Task, ISR, Counter, Alarm or Schedule Table belongs	
Available via:	`Os.h`	

⌋ ()

[SWS_Os_00273] ⌈If the object ObjectType specified in a call of `CheckObjectOwnership()` exists, `CheckObjectOwnership()` shall return the identifier of the OS-Application to which the object belongs. ⌋ ( )

[SWS_Os_00274] ⌈If in a call of `CheckObjectOwnership()` the specified object ObjectType is invalid OR the argument of the type (the "…") is invalid OR the object does not belong to any OS-Application, `CheckObjectOwnership()` shall return `INVALID_OSAPPLICATION`. ⌋ ( )

[SWS_Os_00520] ⌈Availability of `CheckObjectOwnership()`:Available in Scalability Classes 3 and 4 and in multi-core systems. ⌋ ( )

8.4.9 StartScheduleTableRel

[SWS_Os_00347] ⌈

Service name:	StartScheduleTableRel	
Syntax:	`StatusType StartScheduleTableRel( ScheduleTableType ScheduleTableID, TickType Offset )`	
Service ID[hex]:	0x07	
Sync/Async:	Synchronous	
Reentrancy:	Reentrant	
Parameters (in):	ScheduleTableID	Schedule table to be started
	Offset	Number of ticks on the counter before the the schedule table processing is started
Parameters (inout):	None	
Parameters (out):	None	
Return value:	StatusType	E_OK: No Error E_OS_ID (only in EXTENDED status): ScheduleTableID not valid. E_OS_VALUE (only in EXTENDED status): Offset is greater than (OsCounterMaxAllowedValue - InitialOffset) or is equal to 0. E_OS_STATE: Schedule table was already started.
Description:	This service starts the processing of a schedule table at "Offset" relative to the "Now" value on the underlying counter.	
Available via:	`Os.h`	

⌋()

[SWS_Os_00275] ⌈If the schedule table <ScheduleTableID> in a call of `StartScheduleTableRel()` is not valid, `StartScheduleTableRel()` shall return `E_OS_ID`. ⌋ ( )

[SWS_Os_00452] ⌈If the schedule table <ScheduleTableID> in a call of `StartScheduleTableRel()` is implicitely synchronized (`OsScheduleTblSyncStrategy = IMPLICIT`), `StartScheduleTableRel()` shall return `E_OS_ID`. ⌋ ( )

[SWS_Os_00332] ⌈If <Offset> in a call of `StartScheduleTableRel()` is zero `StartScheduleTableRel()` shall return `E_OS_VALUE`. ⌋ ( )

[SWS_Os_00276] ⌈If the offset <Offset>) is greater than `OsCounterMaxAllowedValue` of the underlying counter minus the Initial Offset, `StartScheduleTableRel()` shall return `E_OS_VALUE`. ⌋ ( )

[SWS_Os_00277] ⌈If the schedule table <ScheduleTableID> in a call of `StartScheduleTableRel()` is not in the state `SCHEDULETABLE_STOPPED`, `StartScheduleTableRel()` shall return `E_OS_STATE`. ⌋ ( )

[SWS_Os_00278] ⌈If the input parameters of `StartScheduleTableRel()` are valid and the state of schedule table <ScheduleTableID> is `SCHEDULETABLE_STOPPED`, then `StartScheduleTableRel()` shall start the processing of a schedule table <ScheduleTableID>. The Initial Expiry Point shall be processed after <Offset> + Initial Offset ticks have elapsed on the underlying counter. The state of <ScheduleTableID> is set to `SCHEDULETABLE_RUNNING` before the service returns to the caller. ⌋ ( )

[SWS_Os_00521] ⌈Availability of `StartScheduleTableRel()`: Available in all Scalability Classes. ⌋ ( )

8.4.10 StartScheduleTableAbs

[SWS_Os_00358] ⌈

Service name:	StartScheduleTableAbs

Syntax:	`StatusType StartScheduleTableAbs(`
	`    ScheduleTableType ScheduleTableID,`
	`    TickType Start`
	`)`

Service ID[hex]:	0x08
Sync/Async:	Synchronous

Reentrancy:	Reentrant	
Parameters (in):	ScheduleTableID	Schedule table to be started
	Start	Absolute counter tick value at which the schedule table is started
Parameters (inout):	None	
Parameters (out):	None	
Return value:	StatusType	E_OK: No Error E_OS_ID (only in EXTENDED status): ScheduleTableID not valid E_OS_VALUE (only in EXTENDED status): "Start" is greater than OsCounterMaxAllowedValue E_OS_STATE: Schedule table was already started
Description:	This service starts the processing of a schedule table at an absolute value "Start" on the underlying counter.	
Available via:	`Os.h`	

⌋ ()

[SWS_Os_00348] ⌈If the schedule table <ScheduleTableID> in a call of `StartScheduleTableAbs()` is not valid, `StartScheduleTableAbs()` shall return `E_OS_ID`. ⌋ ( )

[SWS_Os_00349] ⌈If the <Start> in a call of `StartScheduleTableAbs()` is greater than the `OsCounterMaxAllowedValue` of the underlying counter, `StartScheduleTableAbs()` shall return `E_OS_VALUE`. ⌋ ( )

[SWS_Os_00350] ⌈If the schedule table <ScheduleTableID> in a call of `StartScheduleTableAbs()` is not in the state `SCHEDULETABLE_STOPPED`, `StartScheduleTableAbs()` shall return `E_OS_STATE`. ⌋ ( )

[SWS_Os_00351] ⌈If the input parameters of `StartScheduleTableAbs()` are valid and <ScheduleTableID> is in the state `SCHEDULETABLE_STOPPED`, `StartScheduleTableAbs()` shall start the processing of schedule table <ScheduleTableID> when the underlying counter next equals <Start> and shall set the state of <ScheduleTableID> to
- `SCHEDULETABLE_RUNNING` (for a non-synchronized / Explicitly synchronized schedule table) OR
- `SCHEDULETABLE_RUNNING_AND_SYNCHRONOUS` (for implicitly synchronized schedule table)
before returning to the user. (The Initial Expiry Point will be processed when the underlying counter next equals <Start>+Initial Offset). ⌋ ( )

[SWS_Os_00522] ⌈Availability of `StartScheduleTableAbs()`: Available in all Scalability Classes. ⌋ ( )

8.4.11 StopScheduleTable

[SWS_Os_00006] ⌈

Service name:	StopScheduleTable	

Syntax:	`StatusType StopScheduleTable(`
	`    ScheduleTableType ScheduleTableID`
	`)`	

Service ID[hex]:	0x09	
Sync/Async:	Synchronous	
Reentrancy:	Reentrant	
Parameters (in):	ScheduleTableID	Schedule table to be stopped
Parameters (inout):	None	
Parameters (out):	None	
Return value:	StatusType	E_OK: No Error E_OS_ID (only in EXTENDED status): ScheduleTableID not valid. E_OS_NOFUNC: Schedule table was already stopped
Description:	This service cancels the processing of a schedule table immediately at any point while the schedule table is running.	
Available via:	`Os.h`	

⌋ ()

[SWS_Os_00279] ⌈If the schedule table identifier <ScheduleTableID> in a call of `StopScheduleTable()` is not valid, `StopScheduleTable()` shall return `E_OS_ID`. ⌋ ()

[SWS_Os_00280] ⌈If the schedule table with identifier <ScheduleTableID> is in state `SCHEDULETABLE_STOPPED` when calling `StopScheduleTable()`, `StopScheduleTable()` shall return `E_OS_NOFUNC`. ⌋ ()

[SWS_Os_00281] ⌈If the input parameters of `StopScheduleTable()` are valid, `StopScheduleTable()` shall set the state of <ScheduleTableID> to `SCHEDULETABLE_STOPPED` and (stop the schedule table <ScheduleTableID> from processing any further expiry points and) shall return `E_OK`. ⌋ ()

[SWS_Os_00523] ⌈Availability of `StopScheduleTable()`: Available in all Scalability Classes. ⌋ ()

8.4.12 NextScheduleTable

[SWS_Os_00191] ⌈

Service name:	NextScheduleTable

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

Service ID[hex]:	0x0a	
Sync/Async:	Synchronous	
Reentrancy:	Reentrant	
Parameters (in):	ScheduleTableID_From	Currently processed schedule table
	ScheduleTableID_To	Schedule table that provides its series of expiry points
Parameters (inout):	None	
Parameters (out):	None	
Return value:	StatusType	E_OK: No error E_OS_ID (only in EXTENDED status): ScheduleTableID_From or ScheduleTableID_To not valid E_OS_NOFUNC: ScheduleTableID_From not started E_OS_STATE: ScheduleTableID_To is started or next
Description:	This service switches the processing from one schedule table to another schedule table.	
Available via:	`Os.h`	

⌋ (SRS_Os_00099)

[SWS_Os_00282] ⌈If the input parameter <ScheduleTableID_From> or <ScheduleTableID_To> in a call of `NextScheduleTable()` is not valid, `NextScheduleTable()` shall return `E_OS_ID`. ⌋ ( )

[SWS_Os_00330] ⌈If in a call of `NextScheduleTable()` schedule table <ScheduleTableID_To> is driven by different counter than schedule table <ScheduleTableID_From> then `NextScheduleTable()` shall return an error `E_OS_ID`. ⌋ ( )

[SWS_Os_00283] ⌈If the schedule table <ScheduleTableID_From> in a call of `NextScheduleTable()` is in state `SCHEDULETABLE_STOPPED` OR in state `SCHEDULETABLE_NEXT`, `NextScheduleTable()` shall leave the state of <ScheduleTable_From> and <ScheduleTable_To> unchanged and return `E_OS_NOFUNC`. ⌋ ( )

[SWS_Os_00309] ⌈If the schedule table <ScheduleTableID_To> in a call of `NextScheduleTable()` is not in state `SCHEDULETABLE_STOPPED`, `NextScheduleTable()` shall leave the state of <ScheduleTable_From> and <ScheduleTable_To> unchanged and return `E_OS_STATE`. ⌋ ( )

[SWS_Os_00484] ⌈If `OsScheduleTblSyncStrategy` of <ScheduleTableID_To> in a call of `NextScheduleTable()` is not equal to the `OsScheduleTblSyncStrategy` of <ScheduleTableID_From> then `NextScheduleTable()` shall return `E_OS_ID`. ⌋ ( )

[SWS_Os_00284] ⌈If the input parameters of `NextScheduleTable()` are valid then `NextScheduleTable()` shall start the processing of schedule table <ScheduleTableID_To> <ScheduleTableID_From>.FinalDelay ticks after the Final

Expiry Point on <ScheduleTableID_From> is processed and shall return E_OK. NextScheduleTable() shall process the Initial Expiry Point on <ScheduleTableID_To> at <ScheduleTableID_From>.Final Delay + <ScheduleTable_To>.Initial Offset ticks after the Final Expiry Point on <ScheduleTableID_From> is processed . ⌋ ( )

[SWS_Os_00324] ⌈If the input parameters of NextScheduleTable() are valid AND the <ScheduleTableID_From> already has a “next” schedule table then NextScheduleTable() shall replace the previous “next” schedule table with <ScheduleTableID_To> and shall change the old “next” schedule table state to SCHEDULETABLE_STOPPED. ⌋ ( )

[SWS_Os_00505] ⌈If OsScheduleTblSyncStrategy of the schedule tables <ScheduleTableID_From> and <ScheduleTableID_To> in a call of NextScheduleTable() is EXPLICIT and the Operating System module already synchronizes <ScheduleTableID_From>, NextScheduleTable() shall continue synchonization after the start of processing <ScheduleTableID_To>.⌋ ( )

[SWS_Os_00453] ⌈If the <ScheduleTableID_From> in a call of NextScheduleTable() is stopped, NextScheduleTable() shall not start the “next” schedule table and change its state to SCHEDULETABLE_STOPPED. ⌋ ( )

[SWS_Os_00524] ⌈Availability of NextScheduleTable(): Available in all Scalability Classes. ⌋ ( )

8.4.13 StartScheduleTableSynchron

[SWS_Os_00201] ⌈

Service name:	StartScheduleTableSynchron	
Syntax:	`StatusType StartScheduleTableSynchron( ScheduleTableType ScheduleTableID )`	
Service ID[hex]:	0x0b	
Sync/Async:	Synchronous	
Reentrancy:	Reentrant	
Parameters (in):	ScheduleTableID	Schedule table to be started
Parameters (inout):	None	
Parameters (out):	None	
Return value:	StatusType	E_OK: No Error E_OS_ID (only in EXTENDED status): ScheduleTableID not valid E_OS_STATE: Schedule table was already started
Description:	This service starts an explicitly synchronized schedule table synchronously.	
Available via:	Os.h	

⌋ (SRS_Os_11002)

[SWS_Os_00387] ⌈If in a call of `StartScheduleTableSynchron()` the schedule table <ScheduleTableID> is not valid OR the schedule table <ScheduleTableID> is not explicitly synchronized (`OsScheduleTblSyncStrategy != EXPLICIT`) `StartScheduleTableSynchron()` shall return `E_OS_ID`. ⌋ ()

[SWS_Os_00388] ⌈If the schedule table <ScheduleTableID> in a call of `StartScheduleTableSynchron()` is not in the state `SCHEDULETABLE_STOPPED`, `StartScheduleTableSynchron()` shall return `E_OS_STATE`. ⌋ ()

[SWS_Os_00389] ⌈If <ScheduleTableID> in a call of `StartScheduleTableSynchron()` is valid, `StartScheduleTableSynchron()` shall set the state of <ScheduleTableID> to `SCHEDULETABLE_WAITING` and start the processing of schedule table <ScheduleTableID> after the synchronization count of the schedule table is set via `SyncScheduleTable()`. The Initial Expiry Point shall be processed when (Duration-SyncValue)+InitialOffset ticks have elapsed on the synchronization counter where:

Duration is <ScheduleTableID>.OsScheduleTableDuration
SyncValue is the <Value> parameter passed to the `SyncScheduleTable()`
InitialOffset is the shortest expiry point offset in <ScheduleTableID>⌋ ()

[SWS_Os_00525] ⌈Availability of `StartScheduleTableSynchron()`: Available in Scalability Classes 2 and 4. ⌋ ()

8.4.14 SyncScheduleTable

[SWS_Os_00199] ⌈

Service name:	SyncScheduleTable	

Syntax:	`StatusType SyncScheduleTable(`
	`    ScheduleTableType ScheduleTableID,`
	`    TickType Value`
	`)`	

Service ID[hex]:	0x0c	
Sync/Async:	Synchronous	
Reentrancy:	Reentrant	
Parameters (in):	ScheduleTableID	Schedule table to be synchronized
	Value	The current value of the synchronization counter
Parameters (inout):	None	
Parameters (out):	None	
Return value:	StatusType	E_OK: No errors E_OS_ID (only in EXTENDED status): The ScheduleTableID was not valid or schedule table can not be synchronized (OsScheduleTblSyncStrategy not set or OsScheduleTblSyncStrategy = IMPLICIT) E_OS_VALUE (only in EXETENDED status): The <Value> is out

		of range E_OS_STATE: The state of schedule table <ScheduleTableID> is equal to SCHEDULETABLE_STOPPED
Description:	This service provides the schedule table with a synchronization count and start synchronization.	
Available via:	`Os.h`	

⌋ (SRS_Os_11002)

[SWS_Os_00454] ⌈If the <ScheduleTableID> in a call of `SyncScheduleTable()` is not valid OR schedule table can not be explicitely synchronized (`OsScheduleTblSyncStrategy` is not equal to `EXPLICIT`) `SyncScheduleTable()` shall return `E_OS_ID`. ⌋ ()

[SWS_Os_00455] ⌈If the <Value> in a call of `SyncScheduleTable()` is greater or equal than the `OsScheduleTableDuration`, `SyncScheduleTable()` shall return `E_OS_VALUE`. ⌋ ()

[SWS_Os_00456] ⌈If the state of the schedule table <ScheduleTableID> in a call of `SyncScheduleTable()` is equal to `SCHEDULETABLE_STOPPED` or `SCHEDULETABLE_NEXT` `SyncScheduleTable()` shall return `E_OS_STATE`. ⌋ ()

[SWS_Os_00457] ⌈If the parameters in a call of `SyncScheduleTable()` are valid, `SyncScheduleTable()` shall provide the Operating System module with the current synchronization count for the given schedule table. (It is used to synchronize the processing of the schedule table to the synchronization counter.) ⌋ ()

[SWS_Os_00526] ⌈Availability of `SyncScheduleTable()`: Available in Scalability Classes 2 and 4. ⌋ ()

8.4.15 SetScheduleTableAsync

[SWS_Os_00422] ⌈

Service name:	SetScheduletableAsync	

Syntax:	`StatusType SetScheduletableAsync(`
	`    ScheduleTableType ScheduleTableID`
	`)`	

Service ID[hex]:	0x0d	
Sync/Async:	Synchronous	
Reentrancy:	Reentrant	
Parameters (in):	ScheduleTableID	Schedule table for which status is requested
Parameters (inout):	None	
Parameters (out):	None	
Return value:	StatusType	E_OK: No Error E_OS_ID (only in EXTENDED status): Invalid ScheduleTableID

Description:	This service stops synchronization of a schedule table.
Available via:	`Os.h`

⌋ ()

[SWS_Os_00362] ⌈If `SetScheduleTableAsync`() is called for a running schedule table, the Operating System module shall stop further synchronization until a `SyncScheduleTable`() call is made. ⌋ ( )

[SWS_Os_00323] ⌈If `SetScheduleTableAsync`() is called for a running schedule table the Operating System module shall continue to process expiry points on the schedule table. ⌋ ( )

[SWS_Os_00458] ⌈If OsScheduleTblSyncStrategy of <ScheduleTableID> in a call of `SetScheduleTableAsync`() is not equal to `EXPLICIT OR` if <ScheduleTableID> is invalid then `SetScheduleTableAsync`() shall return `E_OS_ID`. ⌋ ( )

[SWS_Os_00483] ⌈If the current state of the <ScheduleTableID> in a call of `SetScheduleTableAsync`() equals to `SCHEDULETABLE_STOPPED`, `SCHEDULETABLE_NEXT` or `SCHEDULETABLE_WAITING` then `SetScheduleTableAsync()` shall return `E_OS_STATE`. ⌋ ( )

[SWS_Os_00300] ⌈If the current state of <ScheduleTableID> in a call of `SetScheduleTableAsync`() equals `SCHEDULETABLE_RUNNING_AND_SYNCHRONOUS` (or `SCHEDULETABLE_RUNNING)` then `SetScheduleTableAsync()` shall set (or keep in case of `SCHEDULETABLE_RUNNING`) the status of <ScheduleTableID> to `SCHEDULETABLE_RUNNING`. ⌋ ( )

[SWS_Os_00527] ⌈Availability of `SetScheduleTableAsync():` Available in Scalability Classes 2 and 4. ⌋ ( )

8.4.16 GetScheduleTableStatus

[SWS_Os_00227] ⌈

Service name:	GetScheduleTableStatus	

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

Service ID[hex]:	0x0e	
Sync/Async:	Synchronous	
Reentrancy:	Reentrant	
Parameters (in):	ScheduleTableID	Schedule table for which status is requested
Parameters (inout):	None	

Parameters (out):	ScheduleStatus	Reference to ScheduleTableStatusType
Return value:	StatusType	E_OK: No Error E_OS_ID (only in EXTENDED status): Invalid ScheduleTableID
Description:	This service queries the state of a schedule table (also with respect to synchronization).	
Available via:	`Os.h`	

⌋ (SRS_Os_11002)

[SWS_Os_00289] ⌈If the schedule table <ScheduleTableID> in a call of `GetScheduleTableStatus`() is NOT started, `GetScheduleTableStatus()` shall pass back `SCHEDULETABLE_STOPPED` via the reference parameter <ScheduleStatus> AND shall return `E_OK`. ⌋ ()

[SWS_Os_00353] ⌈If the schedule table <ScheduleTableID> in a call of `GetScheduleTableStatus`() was used in a `NextScheduleTable()` call AND waits for the end of the current schedule table, `GetScheduleTableStatus()` shall return `SCHEDULETABLE_NEXT` via the reference parameter <ScheduleStatus> AND shall return `E_OK`. ⌋ ()

[SWS_Os_00354] ⌈If the schedule table <ScheduleTableID> in a call of `GetScheduleTableStatus`() is configured with explicit synchronization AND <ScheduleTableID> was started with `StartScheduleTableSynchron()` AND no synchronization count was provided to the Operating System, `GetScheduleTableStatus()` shall return `SCHEDULETABLE_WAITING` via the reference parameter <ScheduleStatus> AND shall return `E_OK`. ⌋ ()

[SWS_Os_00290] ⌈If the schedule table <ScheduleTableID> in a call of `GetScheduleTableStatus`() is started AND synchronous, `GetScheduleTableStatus()` shall pass back `SCHEDULETABLE_RUNNING_AND_SYNCHRONOUS` via the reference parameter <ScheduleStatus> AND shall return `E_OK`. ⌋ ()

[SWS_Os_00291] ⌈If the schedule table <ScheduleTableID> in a call of `GetScheduleTableStatus`() is started AND NOT synchronous (deviation is not within the precision interval OR the schedule table has been set asynchronous), `GetScheduleTableStatus()` shall pass back `SCHEDULETABLE_RUNNING` via the reference parameter ScheduleStatus AND shall return `E_OK`. ⌋ ()

[SWS_Os_00293] ⌈If the identifier <ScheduleTableID> in a call of `GetScheduleTableStatus`() is NOT valid, `GetScheduleTableStatus()` shall return `E_OS_ID`. ⌋ ()

[SWS_Os_00528] ⌈Availability of `GetScheduleTableStatus()`:Available in all Scalability Classes. ⌋ ()

8.4.17 IncrementCounter

[SWS_Os_00399] ⌈

Service name:	IncrementCounter	

Syntax:	`StatusType IncrementCounter(`
	`    CounterType CounterID`
	`)`	

Service ID[hex]:	0x0f	
Sync/Async:	Synchronous, may cause rescheduling	
Reentrancy:	Reentrant	
Parameters (in):	CounterID	The Counter to be incremented
Parameters (inout):	None	
Parameters (out):	None	
Return value:	StatusType	E_OK: No errors E_OS_ID (only in EXTENDED status): The CounterID was not valid or counter is implemented in hardware and can not be incremented by software
Description:	This service increments a software counter.	
Available via:	`Os.h`	

⌋ ()

[SWS_Os_00285] ⌈If the input parameter <CounterID> in a call of `IncrementCounter`() is not valid OR the counter is a hardware counter, `IncrementCounter()` shall return `E_OS_ID`. ⌋ ( )

[SWS_Os_00286] ⌈If the input parameter of `IncrementCounter`() is valid, `IncrementCounter()` shall increment the counter <CounterID> by one (if any alarm connected to this counter expires, the given action, e.g. task activation, is done) and shall return `E_OK`. ⌋ (SRS_Os_11020)

[SWS_Os_00321] ⌈If in a call of `IncrementCounter`() an error happens during the execution of an alarm action, e.g. `E_OS_LIMIT` caused by a task activation, `IncrementCounter`() shall call the error hook(s), but the `IncrementCounter()` service itself shall return `E_OK`. ⌋ ( )

[SWS_Os_00529] ⌈Caveats of `IncrementCounter`(): If called from a task, rescheduling may take place. ⌋ ( )

[SWS_Os_00530] ⌈Availability of `IncrementCounter`(): Available in all Scalability Classes. ⌋ ( )

8.4.18 GetCounterValue

[SWS_Os_00383] ⌈

Service name:	GetCounterValue	

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

Service ID[hex]:	0x10	
Sync/Async:	Synchronous	
Reentrancy:	Reentrant	
Parameters (in):	CounterID	The Counter which tick value should be read
Parameters (inout):	None	
Parameters (out):	Value	Contains the current tick value of the counter
Return value:	StatusType	E_OK: No errors E_OS_ID (only in EXTENDED status): The <CounterID> was not valid
Description:	This service reads the current count value of a counter (returning either the hardware timer ticks if counter is driven by hardware or the software ticks when user drives counter).	
Available via:	`Os.h`	

⌋ (SRS_Frt_00025)

[SWS_Os_00376] ⌈If the input parameter <CounterID> in a call of `GetCounterValue`() is not valid, `GetCounterValue`() shall return `E_OS_ID`. ⌋ ( )

[SWS_Os_00377] ⌈If the input parameter <CounterID> in a call of `GetCounterValue`() is valid, `GetCounterValue()` shall return the current tick value of the counter via <Value> and return `E_OK`. ⌋ (SRS_Frt_00033)

[SWS_Os_00531] ⌈Caveats of `GetCounterValue`(): Note that for counters of `OsCounterType` = `HARDWARE` the real timer value (the – possibly adjusted – hardware value, see SWS_Os_00384) is returned, whereas for counters of `OsCounterType` = `SOFTWARE` the current "software" tick value is returned. ⌋ ( )

[SWS_Os_00532] ⌈Availability of `GetCounterValue`(): Available in all Scalability Classes. ⌋ ( )

8.4.19 GetElapsedValue

[SWS_Os_00392] ⌈

Service name:	GetElapsedValue

Syntax:	`StatusType GetElapsedValue(`
	`    CounterType CounterID,`
	`    TickRefType Value,`
	`    TickRefType ElapsedValue`

	)	
Service ID[hex]:	0x11	
Sync/Async:	Synchronous	
Reentrancy:	Reentrant	
Parameters (in):	CounterID	The Counter to be read
Parameters (inout):	Value	in: the previously read tick value of the counter out: the current tick value of the counter
Parameters (out):	ElapsedValue	The difference to the previous read value
Return value:	StatusType	E_OK: No errors E_OS_ID (only in EXTENDED status): The CounterID was not valid E_OS_VALUE (only in EXTENDED status): The given Value was not valid
Description:	This service gets the number of ticks between the current tick value and a previously read tick value.	
Available via:	`Os.h`	

⌋ (SRS_Frt_00025)

[SWS_Os_00381] ⌈If the input parameter <CounterID> in a call of `GetElapsedValue`() is not valid `GetElapsedValue`() shall return `E_OS_ID`. ⌋ ( )

[SWS_Os_00391] ⌈If the <Value> in a call of `GetElapsedValue`() is larger than the max allowed value of the <CounterID>, `GetElapsedValue`() shall return `E_OS_VALUE`. ⌋ ( )

[SWS_Os_00382] ⌈If the input parameters in a call of `GetElapsedValue`() are valid, `GetElapsedValue()` shall return the number of elapsed ticks since the given <Value> value via <ElapsedValue> and shall return `E_OK`. ⌋ (SRS_Frt_00034)

[SWS_Os_00460] ⌈`GetElapsedValue`() shall return the current tick value of the counter in the <Value> parameter. ⌋ ( )

[SWS_Os_00533] ⌈Caveats of `GetElapsedValue`():If the timer already passed the <Value> value a second (or multiple) time, the result returned is wrong. The reason is that the service can not detect such a relative overflow. ⌋ ( )

[SWS_Os_00534] ⌈Availability of `GetElapsedValue`(): Available in all Scalability Classes. ⌋ ( )

8.4.20 TerminateApplication

[SWS_Os_00258] ⌈

Service name:	TerminateApplication

Syntax:	`StatusType TerminateApplication(`
	`    ApplicationType Application,`

	`RestartType RestartOption )`	
Service ID[hex]:	0x12	
Sync/Async:	Synchronous	
Reentrancy:	Reentrant	
Parameters (in):	Application	The identifier of the OS-Application to be terminated. If the caller belongs to <Application> the call results in a self termination.
	RestartOption	Either RESTART for doing a restart of the OS-Application or NO_RESTART if OS-Application shall not be restarted.
Parameters (inout):	None	
Parameters (out):	None	
Return value:	StatusType	E_OK: No errors E_OS_ID: <Application> was not valid (only in EXTENDED status) E_OS_VALUE: <RestartOption> was neither RESTART nor NO_RESTART (only in EXTENDED status) E_OS_ACCESS: The caller does not have the right to terminate <Application> (only in EXTENDED status) E_OS_STATE: The state of <Application> does not allow terminating <Application>
Description:	This service terminates the OS-Application to which the calling Task/Category 2 ISR/application specific error hook belongs.	
Available via:	`Os.h`	

⌋ ()

[SWS_Os_00493] ⌈If the input parameter <Application> in a call of `TerminateApplication`() is not valid `TerminateApplication`() shall return `E_OS_ID`. ⌋ ( )

[SWS_Os_00459] ⌈If the <RestartOption> in a call of `TerminateApplication`() is invalid, `TerminateApplication`() shall return `E_OS_VALUE`. ⌋ ( )

[SWS_Os_00494] ⌈If the input parameter <Application> in a call of `TerminateApplication`() is valid AND the caller belongs to a non-trusted OS-Application AND the caller does not belong to <Application> `TerminateApplication`() shall return `E_OS_ACCESS`. ⌋ ( )

[SWS_Os_00507] ⌈If the state of <Application> in a call of `TerminateApplication`() is `APPLICATION_TERMINATED` `TerminateApplication`() shall return `E_OS_STATE`. ⌋ ( )

[SWS_Os_00508] ⌈If the state of <Application> in a call of `TerminateApplication`() is `APPLICATION_RESTARTING` and the caller does not belong to the <Application> then `TerminateApplication`() shall return `E_OS_STATE`. ⌋ ( )

[SWS_Os_00548] ⌈If the state of <Application> in a call of `TerminateApplication`() is `APPLICATION_RESTARTING` AND the caller does belong to the <Application> AND the <RestartOption> is equal `RESTART` then `TerminateApplication`() shall return `E_OS_STATE`. ⌋ ( )

[SWS_Os_00287] ⌈If the parameters in a call of `TerminateApplication`() are valid and the above criteria are met `TerminateApplication`() shall terminate <Application> (i.e. to kill all tasks, disable the interrupt sources of those ISRs which belong to the OS-Application and free all other OS resources associated with the application) AND shall activate the configured `OsRestartTask` of <Application> if <RestartOption> equals `RESTART`. If no `OsRestartTask` is configured, no restart shall happen. If the <Application> is restarted, its state is set to `APPLICATION_RESTARTING` otherwise to `APPLICATION_TERMINATED`. If the caller belongs to <Application> `TerminateApplication`() shall not return, otherwise it shall return `E_OK`.⌋ ( )

[SWS_Os_00535] ⌈Caveats of `TerminateApplication`():
If no applications are configured the implementation shall make sure that this service is not available.
Tasks and interrupts that are owned by a trusted application can terminate any OS-Application. Tasks and interrupts that are owned by a non-trusted application can only terminate their owning OS-Application. ⌋ ( )

Note: Although trusted OS-Application can be forcibly terminated by Tasks/Interrupts of other trusted OS-Applications it is not recommended. This may have further impacts, e.g. to users who are currently part of such an OS-Application via a `CallTrustedFunction()` call.

[SWS_Os_00536] ⌈Availability of `TerminateApplication`(): Available in Scalability Classes 3 and 4. ⌋ ( )

8.4.21 AllowAccess

[SWS_Os_00501] ⌈

Service name:	AllowAccess	

Syntax:	`StatusType AllowAccess(`
	`    void`
	`)`	

Service ID[hex]:	0x13	
Sync/Async:	Synchronous	
Reentrancy:	Reentrant	
Parameters (in):	None	
Parameters (inout):	None	
Parameters (out):	None	
Return value:	StatusType	E_OK: No errors E_OS_STATE:The OS-Application of the caller is in the wrong state
Description:	This service sets the own state of an OS-Application from	

	APPLICATION_RESTARTING to APPLICATION_ACCESSIBLE.
Available via:	`Os.h`

⌋ ()

[SWS_Os_00497] ⌈If the state of the OS-Application of the caller of `AllowAccess()` is not `APPLICATION_RESTARTING` `AllowAccess()` shall return `E_OS_STATE`. ⌋ ( )

[SWS_Os_00498] ⌈If the state of the OS-Application of the caller of `AllowAccess()` is `APPLICATION_RESTARTING`, `AllowAccess()` shall set the state to `APPLICATION_ACCESSIBLE` and allow other OS-Applications to access the configured objects of the callers OS-Application. ⌋ ( )

[SWS_Os_00547] ⌈Availability of `AllowAccess`(): Available in Scalability Classes 3 and 4. ⌋ ( )

8.4.22 GetApplicationState

[SWS_Os_00499] ⌈

Service name:	GetApplicationState	

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

Service ID[hex]:	0x14	
Sync/Async:	Synchronous	
Reentrancy:	Reentrant	
Parameters (in):	Application	The OS-Application from which the state is requested
Parameters (inout):	None	
Parameters (out):	Value	The current state of the application
Return value:	StatusType	E_OK: No errors E_OS_ID: <Application> is not valid (only in EXTENDED status)
Description:	This service returns the current state of an OS-Application.	
Available via:	`Os.h`	

⌋ ()

[SWS_Os_00495] ⌈If the <Application> in a call of `GetApplicationState`() is not valid `GetApplicationState`() shall return `E_OS_ID`. ⌋ ( )

[SWS_Os_00496] ⌈If the parameters in a call of `GetApplicationState`() are valid, `GetApplicationState`() shall return the state of OS-Application <Application> in <Value>.⌋ ( )

[SWS_Os_00537] ⌈Availability of `GetApplicationState`(): Available in Scalability Classes 3 and 4. ⌋ ( )

8.4.23 GetNumberOfActivatedCores

[SWS_Os_00672] ⌈

Service name:	GetNumberOfActivatedCores	

Syntax:	`uint32 GetNumberOfActivatedCores(`
	`    void`
	`)`	

Service ID[hex]:	0x15	
Sync/Async:	Synchronous	
Reentrancy:	Reentrant	
Parameters (in):	None	
Parameters (inout):	None	
Parameters (out):	None	
Return value:	uint32	Number of cores activated by the StartCore function (see below)
Description:	The function returns the number of cores activated by the StartCore function. This function might be a macro.	
Available via:	`Os.h`	

⌋ (SRS_Os_80001)

The function `GetNumberOfActivatedCores` shall be callable from within a TASK and an ISR cat 2. Otherwise the behavior is unspecified.

[SWS_Os_00673] ⌈The return value of `GetNumberOfActivatedCores` shall be less or equal to the configured value of "`OsNumberOfCores`". ⌋ (SRS_Os_80001)

8.4.24 GetCoreID

[SWS_Os_00674] ⌈

Service name:	GetCoreID	

Syntax:	`CoreIdType GetCoreID(`
	`    void`
	`)`	

Service ID[hex]:	0x16	
Sync/Async:	Synchronous	
Reentrancy:	Reentrant	
Parameters (in):	None	
Parameters (inout):	None	
Parameters (out):	None	
Return value:	CoreIdType	The return value is the unique ID of the core.
Description:	The function returns a unique core identifier.	
Available via:	`Os.h`	

⌋ (SRS_Os_80001)

[SWS_Os_00675] ⌈The function GetCoreID shall return the unique logical CoreID of the core on which the function is called. The mapping of physical cores to logical CoreIDs is implementation specific. ⌋ (SRS_Os_80001)

8.4.25 StartCore

[SWS_Os_00676] ⌈

Service name:	StartCore	
Syntax:	`void StartCore( CoreIdType CoreID, StatusType* Status )`	
Service ID[hex]:	0x17	
Sync/Async:	Synchronous	
Reentrancy:	Non Reentrant	
Parameters (in):	CoreID	Core identifier
Parameters (inout):	None	
Parameters (out):	Status	Return value of the function in extended status: E_OK: No Error E_OS_ID: Core ID is invalid. E_OS_ACCESS: The function was called after starting the OS. E_OS_STATE: The Core is already activated. Return value of the function in standard status E_OK: No Error
Return value:	None	
Description:	It is not supported to call this function after StartOS(). The function starts the core specified by the parameter CoreID. The OUT parameter allows the caller to check whether the operation was successful or not. If a core is started by means of this function StartOS shall be called on the core.	
Available via:	`Os.h`	

⌋ (SRS_Os_80006, SRS_Os_80026, SRS_Os_80027)

[SWS_Os_00677] ⌈The function StartCore shall start one core that shall run under the control of the AUTOSAR OS. ⌋ (SRS_Os_80006, SRS_Os_80026, SRS_Os_80027)

[SWS_Os_00678] ⌈Calls to the `StartCore` function after `StartOS() shall return with` E_OS_ACCESS and the core shall not be started. ⌋ (SRS_Os_80006, SRS_Os_80026, SRS_Os_80027)

[SWS_Os_00679] ⌈If the parameter `CoreIDs` refers to a core that was already started by the function `StartCore` the related core is ignored and E_OS_STATE shall be returned. ⌋ (SRS_Os_80006, SRS_Os_80026, SRS_Os_80027)

[SWS_Os_00680] ⌈If the parameter `CoreID` refers to a core that was already started by the function `StartNonAutosarCore` the related core is ignored and E_OS_STATE shall be returned. ⌋ (SRS_Os_80006, SRS_Os_80026, SRS_Os_80027)

[SWS_Os_00681] ⌈There is no call to the ErrorHook() if an error occurs during `StartCore();`⌋ (SRS_Os_80006, SRS_Os_80026, SRS_Os_80027)

8.4.26 StartNonAutosarCore

[SWS_Os_00682] ⌈

Service name:	StartNonAutosarCore	

Syntax:	`void StartNonAutosarCore(`
	`    CoreIdType CoreID,`
	`    StatusType* Status`
	`)`	

Service ID[hex]:	0x18	
Sync/Async:	Synchronous	
Reentrancy:	Non Reentrant	
Parameters (in):	CoreID	Core identifier
Parameters (inout):	None	
Parameters (out):	Status	Return value of the function in standard status: E_OK: No Error E_OS_ID: Core ID is invalid. E_OS_STATE: The Core is already activated. Return value of the function in extended status E_OK: No Error
Return value:	None	
Description:	The function starts the core specified by the parameter CoreID. It is allowed to call this function after StartOS(). The OUT parameter allows the caller to check whether the operation was successful or not. It is not allowed to call StartOS on cores activated by StartNonAutosarCore. Otherwise the behaviour is unspecified.	
Available via:	`Os.h`	

⌋ (SRS_Os_80006, SRS_Os_80026, SRS_Os_80027)

[SWS_Os_00683] ⌈The function `StartNonAutosarCore` shall start a core that is not controlled by the AUTOSAR OS. ⌋ (SRS_Os_80006, SRS_Os_80026, SRS_Os_80027)

[SWS_Os_00684] ⌈If the parameter `CoreID` refers to a core that was already started by the function `StartNonAutosarCore` has no effect and sets “Status” to E_OS_STATE. ⌋ (SRS_Os_80006, SRS_Os_80026, SRS_Os_80027)

[SWS_Os_00685] ⌈If the parameter `CoreID` refers to an unknown core the function `StartNonAutosarCore` has no effect and sets “Status” to E_OS_ID. ⌋ (SRS_Os_80006, SRS_Os_80026, SRS_Os_80027)

8.4.27 GetSpinlock

[SWS_Os_00686] ⌈

Service name:	GetSpinlock	

Syntax:	`StatusType GetSpinlock(`
	`    SpinlockIdType SpinlockId`
	`)`	

Service ID[hex]:	0x19	
Sync/Async:	Synchronous	
Reentrancy:	Reentrant	
Parameters (in):	SpinlockId	The value refers to the spinlock instance that shall be locked.
Parameters (inout):	None	
Parameters (out):	None	
Return value:	StatusType	E_OK - In standard and extended status : No Error E_OS_ID - In extended status: The SpinlockId is invalid E_OS_INTERFERENCE_DEADLOCK - In extended status: A TASK tries to occupy the spinlock while the lock is already occupied by a TASK on the same core. This would cause a deadlock. E_OS_NESTING_DEADLOCK - In extended status: A TASK tries to occupy the spinlock while a TASK on the same core is holding a different spinlock in a way that may cause a deadlock. E_OS_ACCESS - In extended status: The spinlock cannot be accessed.
Description:	GetSpinlock tries to occupy a spin-lock variable. If the function returns, either the lock is successfully taken or an error has occurred. The spinlock mechanism is an active polling mechanism. The function does not cause a de-scheduling.	
Available via:	`Os.h`	

⌋ (SRS_Os_80021)

[SWS_Os_00687] ⌈The function `GetSpinlock` shall occupy a spinlock. If the spinlock is already occupied the function shall busy wait until the spinlock becomes available. ⌋ (SRS_Os_80021)

[SWS_Os_00688] ⌈The function `GetSpinlock` shall return `E_OK` if no error was detected. The spinlock is now occupied by the calling TASK/ISR2 on the calling core. ⌋ (SRS_Os_80021)

[SWS_Os_00689] ⌈The function `GetSpinlock` shall return `E_OS_ID` if the parameter `SpinlockID` refers to a spinlock that does not exist. ⌋ (SRS_Os_80021)

[SWS_Os_00690] ⌈The function `GetSpinlock` shall return `E_OS_INTERFERENCE_DEADLOCK` if the spinlock referred by the parameter `SpinlockID` is already occupied by a TASK/ISR2 on the same core. ⌋ (SRS_Os_80021)

[SWS_Os_00691] ⌈The function `GetSpinlock` shall return `E_OS_NESTING_DEADLOCK` if the sequence by which multiple spinlocks are occupied at the same time on one core do not comply with the configured order. ⌋ (SRS_Os_80021)

[SWS_Os_00692] ⌈The function `GetSpinlock` shall return `E_OS_ACCESS` if the accessing OS-Application was not listed in the configuration (OsSpinlock). ⌋ (SRS_Os_80021)

[SWS_Os_00693] ⌈It shall be allowed to call the function `GetSpinlock` while interrupts are disabled. ⌋ (SRS_Os_80021)

[SWS_Os_00694] ⌈It shall be allowed to call the function `GetSpinlock` while a RESOURCE is occupied. ⌋ (SRS_Os_80021)

8.4.28 ReleaseSpinlock

[SWS_Os_00695] ⌈

Service name:	ReleaseSpinlock	

Syntax:	`StatusType ReleaseSpinlock(`
	`    SpinlockIdType SpinlockId`
	`)`	

Service ID[hex]:	0x1a	
Sync/Async:	Synchronous	
Reentrancy:	Reentrant	
Parameters (in):	SpinlockId	The value refers to the spinlock instance that shall be locked.
Parameters (inout):	None	
Parameters (out):	None	
Return value:	StatusType	E_OK - In standard and extended status: No Error E_OS_ID - In extended status: The SpinlockId is invalid. E_OS_STATE - In extended status: The Spinlock is not occupied by the TASK E_OS_ACCESS - In extended status: The Spinlock cannot be accessed. E_OS_NOFUNC - In extended status: Attempt to release a spinlock while another spinlock (or resource) has to be released before.
Description:	ReleaseSpinlock releases a spinlock variable that was occupied before. Before terminating a TASK all spinlock variables that have been occupied with GetSpinlock() shall be released. Before calling WaitEVENT all Spinlocks shall be released.	
Available via:	`Os.h`	

⌋ (SRS_Os_80021)

[SWS_Os_00696] ⌈The function `ReleaseSpinlock` shall release a spinlock that has been occupied by the same (calling) TASK. If the related `GetSpinlock` call used configured locks (ECUC_Os_01038) the function shall also perform the undo of the used lock.⌋ (SRS_Os_80021)

[SWS_Os_00697] ⌈The function `ReleaseSpinlock` shall return `E_OK` if no error was detected. The spinlock is now free and can be occupied by the same or other TASKs. ⌋ (SRS_Os_80021)

[SWS_Os_00698] ⌈The function `ReleaseSpinlock` shall return `E_OS_ID` if the parameter `SpinlockID` refers to a spinlock that does not exist. ⌋ (SRS_Os_80021)

[SWS_Os_00699] ⌈The function `ReleaseSpinlock` shall return `E_OS_STATE` if the parameter `SpinlockID` refers to a spinlock that is not occupied by the calling TASK. ⌋ (SRS_Os_80021)

[SWS_Os_00700] ⌈The function `ReleaseSpinlock` shall return `E_OS_ACCESS` if the TASK has no access to the spinlock referred by the parameter `SpinlockID`⌋ (SRS_Os_80021)

[SWS_Os_00701] ⌈The function `ReleaseSpinlock` shall return `E_OS_NOFUNC` if the TASK tries to release a spinlock while another spinlock (or resource) has to be released before. No functionality shall be performed. ⌋ (SRS_Os_80021)

8.4.29 TryToGetSpinlock

[SWS_Os_00703] ⌈

Service name:	TryToGetSpinlock	

Syntax:	`StatusType TryToGetSpinlock(`
	`    SpinlockIdType SpinlockId,`
	`    TryToGetSpinlockType* Success`
	`)`	

Service ID[hex]:	0x1b	
Sync/Async:	Synchronous	
Reentrancy:	Reentrant	
Parameters (in):	SpinlockId	The value refers to the spinlock instance that shall be locked.
Parameters (inout):	None	
Parameters (out):	Success	Returns if the lock has been occupied or not
Return value:	StatusType	E_OK - In standard and extended status: No Error E_OS_ID - In extended status: The SpinlockId is invalid. E_OS_INTERFERENCE_DEADLOCK - In extended status: A TASK tries to occupy the spinlock while the lock is already occupied by a TASK on the same core. This would cause a deadlock. E_OS_NESTING_DEADLOCK - In extended status: A TASK tries to occupy a spinlock while holding a different spinlock in a way that may cause a deadlock. E_OS_ACCESS - In extended status: The spinlock cannot be accessed.
Description:	TryToGetSpinlock has the same functionality as GetSpinlock with the difference that if the spinlock is already occupied by a TASK on a different core the function sets the OUT parameter "Success" and returns with E_OK.	
Available via:	`Os.h`	

⌋ (SRS_Os_80021)

[SWS_Os_00704] ⌈The function `TryToGetSpinlock` shall atomically test the availability of the spinlock and if available occupy it. The result of success is returned. ⌋ (SRS_Os_80021)

[SWS_Os_00705] ⌈The function `TryToGetSpinlock` shall set the OUT parameter "`Success`" to `TRYTOGETSPINLOCK_SUCCESS` if the spinlock was successfully occupied, and `TRYTOGETSPINLOCK_NOSUCCESS` if not. In both cases `E_OK` shall be returned. ⌋ (SRS_Os_80021)

[SWS_Os_00706] ⌈If the function `TryToGetSpinlock` does not return `E_OK`, the OUT parameter "`Success`" shall be undefined. ⌋ (SRS_Os_80021)

[SWS_Os_00707] ⌈The function `TryToGetSpinlock` shall return `E_OS_ID` if the parameter `SpinlockID` refers to a spinlock that does not exist. ⌋ (SRS_Os_80021)

[SWS_Os_00708] ⌈The function `TryToGetSpinlock` shall return `E_OS_INTERFERENCE_DEADLOCK` if the spinlock referred by the parameter `SpinlockID` is already occupied by a TASK on the same core. ⌋ (SRS_Os_80021)

[SWS_Os_00709] ⌈The function `TryToGetSpinlock` shall return `E_OS_NESTING_DEADLOCK` if a TASK tries to occupy a spinlock while holding a different spinlock in a way that may cause a deadlock. ⌋ (SRS_Os_80021)

[SWS_Os_00710] ⌈The function `TryToGetSpinlock` shall return `E_OS_ACCESS` if the TASK has no access to the spinlock referred by the parameter `SpinlockID`⌋ (SRS_Os_80021)

[SWS_Os_00711] ⌈It shall be allowed to call the function `TryToGetSpinlock` while interrupts are disabled. ⌋ (SRS_Os_80021)

[SWS_Os_00712] ⌈It shall be allowed to call the function `TryToGetSpinlock` while a RESOURCE is occupied. ⌋ (SRS_Os_80021)

8.4.30 ShutdownAllCores

[SWS_Os_00713] ⌈

Service name:	ShutdownAllCores

Syntax:	`void ShutdownAllCores(`
	`    StatusType Error`
	`)`

Service ID[hex]:	0x1c
Sync/Async:	Synchronous
Reentrancy:	Reentrant

Parameters (in):	Error	<Error> needs to be a valid error code supported by the AUTOSAR OS.
Parameters (inout):	None	
Parameters (out):	None	
Return value:	None	
Description:	After this service the OS on all AUTOSAR cores is shut down. Allowed at TASK level and ISR level and also internally by the OS. The function will never return. The function will force other cores into a shutdown.	
Available via:	`Os.h`	

⌋ (SRS_Os_80007)

[SWS_Os_00714] ⌈A Synchronized shutdown shall be triggered by the API function `ShutdownAllCores`. ⌋ (SRS_Os_80007)

[SWS_Os_00715] ⌈`ShutdownAllCores` shall not return. ⌋ (SRS_Os_80007)

[SWS_Os_00716] ⌈If `ShutdownAllCores is called from non trusted code the call shall be ignored`. ⌋ (SRS_Os_80007)

8.4.31 ControlIdle

[SWS_Os_00769] ⌈

Service name:	ControlIdle	

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

Service ID[hex]:	0x1d	
Sync/Async:	Synchronous	
Reentrancy:	Non Reentrant	
Parameters (in):	CoreID	selects the core which idle mode is set
	IdleMode	the mode which shall be performed during idle time
Parameters (inout):	None	
Parameters (out):	None	
Return value:	StatusType	E_OK: No Error E_OS_ID (only EXTENDED status): Invalid core and/or invalid idleMode
Description:	This API allows the caller to select the idle mode action which is performed during idle time of the OS (e.g. if no Task/ISR is active). It can be used to implement energy savings. The real idle modes are hardware dependent and not standardized. The default idle mode on each core is IDLE_NO_HALT.	
Available via:	`Os.h`	

⌋ ()

[SWS_Os_00770] ⌈The function `ControlIdle` shall return `E_OK` if no error was detected and the parameters are valid⌋ (SRS_Os_80023)

[SWS_Os_00771] ⌈The function `ControlIdle` shall return `E_OS_ID` if the parameter `CoreID` or `IdleMode` is invalid (e.g. refered core does not exist; idlemode is not known). In single core systems the check of CoreID shall be omitted.⌋ (SRS_Os_80023)

[**SWS_Os_00802**]⌈ If the core (given by `CoreID`) is already in another idle mode (different to the given IdleMode) the new `IdleMode` shall become effective the next time that core enters the idle mode.⌋ (SRS_Os_80023)

8.4.32 ReadPeripheralX

[SWS_Os_91013] ⌈

Service name:	ReadPeripheral8	

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

Service ID[hex]:	0x28	
Sync/Async:	Synchronous	
Reentrancy:	Reentrant	
Parameters (in):	Area	hardware peripheral area reference
	Address	memory address
Parameters (inout):	None	
Parameters (out):	ReadValue	content of the given memory location (<Address>)
Return value:	StatusType	E_OK No error E_OS_ID Area id is out of range (EXTENDED status) E_OS_VALUE Address does not belong to given Area (EXTENDED status) E_OS_CALLEVEL Wrong call context of the API function (EXTENDED status) E_OS_ACCESS The calling task or ISR is not allowed to access the given
Description:	This service returns the content of a given memory location (<Address>).	
Available via:	`Os.h`	

⌋ (SRS_Os_11005)

[SWS_Os_91015] ⌈

Service name:	ReadPeripheral16	

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

Service ID[hex]:	0x29	
Sync/Async:	Synchronous	
Reentrancy:	Reentrant	
Parameters (in):	Area	hardware peripheral area reference
	Address	memory address

Parameters (inout):	None	
Parameters (out):	ReadValue	content of the given memory location (<Address>)
Return value:	StatusType	E_OK No error E_OS_ID Area id is out of range (EXTENDED status) E_OS_VALUE Address does not belong to given Area (EXTENDED status) E_OS_CALLEVEL Wrong call context of the API function (EXTENDED status) E_OS_ACCESS The calling task or ISR is not allowed to access the given
Description:	This service returns the content of a given memory location (<Address>).	
Available via:	`Os.h`	

⌋ (SRS_Os_11005)

[SWS_Os_91014] ⌈

Service name:	ReadPeripheral32	

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

Service ID[hex]:	0x2b	
Sync/Async:	Synchronous	
Reentrancy:	Reentrant	
Parameters (in):	Area	hardware peripheral area reference
	Address	memory address
Parameters (inout):	None	
Parameters (out):	ReadValue	content of the given memory location (<Address>)
Return value:	StatusType	E_OK No error E_OS_ID Area id is out of range (EXTENDED status) E_OS_VALUE Address does not belong to given Area (EXTENDED status) E_OS_CALLEVEL Wrong call context of the API function (EXTENDED status) E_OS_ACCESS The calling task or ISR is not allowed to access the given
Description:	This service returns the content of a given memory location (<Address>).	
Available via:	`Os.h`	

⌋ (SRS_Os_11005)

8.4.33 WritePeripheralX

[SWS_Os_91010] ⌈

Service name:	WritePeripheral8

Syntax:	`StatusType WritePeripheral8(`
	`    AreaIdType Area,`
	`    uint8* Address,`
	`    uint8 WriteValue`
	`)`

Service ID[hex]:	0x2b
Sync/Async:	Synchronous
Reentrancy:	Reentrant

Parameters (in):	Area	hardware peripheral area reference
	Address	memory address
Parameters (inout):	None	
Parameters (out):	WriteValue	value to be written at the memory address
Return value:	StatusType	E_OK No error E_OS_ID Area id is out of range (EXTENDED status) E_OS_VALUE Address does not belong to given Area (EXTENDED status) E_OS_CALLEVEL Wrong call context of the API function (EXTENDED status) E_OS_ACCESS The calling task or ISR is not allowed to access the given
Description:	This service writes the <value> to a given memory location (<memory address>).	
Available via:	`Os.h`	

⌋ (SRS_Os_11005)

[SWS_Os_91012] ⌈

Service name:	WritePeripheral16	

Syntax:	`StatusType WritePeripheral16(`
	`    AreaIdType Area,`
	`    uint16* Address,`
	`    uint16 WriteValue`
	`)`	

Service ID[hex]:	0x2c	
Sync/Async:	Synchronous	
Reentrancy:	Reentrant	
Parameters (in):	Area	hardware peripheral area reference
	Address	memory address
Parameters (inout):	None	
Parameters (out):	WriteValue	value to be written at the memory address
Return value:	StatusType	E_OK No error E_OS_ID Area id is out of range (EXTENDED status) E_OS_VALUE Address does not belong to given Area (EXTENDED status) E_OS_CALLEVEL Wrong call context of the API function (EXTENDED status) E_OS_ACCESS The calling task or ISR is not allowed to access the given
Description:	This service writes the <value> to a given memory location (<memory address>).	
Available via:	`Os.h`	

⌋ (SRS_Os_11005)

[SWS_Os_91011] ⌈

Service name:	WritePeripheral32

Syntax:	`StatusType WritePeripheral32(`
	`    AreaIdType Area,`
	`    uint32* Address,`
	`    uint32 WriteValue`
	`)`

Service ID[hex]:	0x2d
Sync/Async:	Synchronous
Reentrancy:	Reentrant

Parameters (in):	Area	hardware peripheral area reference
	Address	memory address
Parameters (inout):	None	
Parameters (out):	WriteValue	content of the given memory location (<Address>)
Return value:	StatusType	E_OK No error E_OS_ID Area id is out of range (EXTENDED status) E_OS_VALUE Address does not belong to given Area (EXTENDED status) E_OS_CALLEVEL Wrong call context of the API function (EXTENDED status) E_OS_ACCESS The calling task or ISR is not allowed to access the given
Description:	This service writes the <value> to a given memory location (<memory address>).	
Available via:	`Os.h`	

⌋ (SRS_Os_11005)

8.4.34 ModifyPeripheralX

[SWS_Os_91016] ⌈

Service name:	ModifyPeripheral8	

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

Service ID[hex]:	0x2e	
Sync/Async:	Synchronous	
Reentrancy:	Reentrant	
Parameters (in):	Area	hardware peripheral area reference
	Address	memory address
	Clearmask	memory address will be modified by an bit-AND
	Setmask	memory address will be modified by an bit-OR
Parameters (inout):	None	
Parameters (out):	None	
Return value:	StatusType	E_OK No error E_OS_ID Area id is out of range (EXTENDED status) E_OS_VALUE Address does not belong to given Area (EXTENDED status) E_OS_CALLEVEL Wrong call context of the API function (EXTENDED status) E_OS_ACCESS The calling task or ISR is not allowed to access the given
Description:	This service modifies a given memory location (<memory address>) with the formula: *<Address> = ((*<Address> & <clearmask>) | <setmask>)	
Available via:	`Os.h`	

⌋ (SRS_Os_11005)

[SWS_Os_91018] ⌈

Service name:	ModifyPeripheral16
Syntax:	`StatusType ModifyPeripheral16(`

	`AreaIdType Area,`
	`uint16* Address,`
	`uint16 Clearmask,`
	`uint16 Setmask`
	`)`	

Service ID[hex]:	0x2d	
Sync/Async:	Synchronous	
Reentrancy:	Reentrant	
Parameters (in):	Area	hardware peripheral area reference
	Address	memory address
	Clearmask	memory address will be modified by an bit-AND
	Setmask	memory address will be modified by an bit-OR
Parameters (inout):	None	
Parameters (out):	None	
Return value:	StatusType	E_OK No error E_OS_ID Area id is out of range (EXTENDED status) E_OS_VALUE Address does not belong to given Area (EXTENDED status) E_OS_CALLEVEL Wrong call context of the API function (EXTENDED status) E_OS_ACCESS The calling task or ISR is not allowed to access the given
Description:	This service modifies a given memory location (<memory address>) with the formula: *<Address> = ((*<Address> & <clearmask>) | <setmask>)	
Available via:	`Os.h`	

⌋ (SRS_Os_11005)

[SWS_Os_91017] ⌈

Service name:	ModifyPeripheral32	

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

Service ID[hex]:	0x2f	
Sync/Async:	Synchronous	
Reentrancy:	Reentrant	
Parameters (in):	Area	hardware peripheral area reference
	Address	memory address
	Clearmask	memory address will be modified by an bit-AND
	Setmask	memory address will be modified by an bit-OR
Parameters (inout):	None	
Parameters (out):	None	
Return value:	StatusType	E_OK No error E_OS_ID Area id is out of range (EXTENDED status) E_OS_VALUE Address does not belong to given Area (EXTENDED status) E_OS_CALLEVEL Wrong call context of the API function (EXTENDED status) E_OS_ACCESS The calling task or ISR is not allowed to access the given
Description:	This service modifies a given memory location (<memory address>) with the	

	formula: *<Address> = ((*<Address> & <clearmask>) | <setmask>)	
Available via:	`Os.h`	

⌋ (SRS_Os_11005)

8.4.35 EnableInterruptSource

[SWS_Os_91020] ⌈

Service name:	EnableInterruptSource	

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

Service ID[hex]:	0x31	
Sync/Async:	Synchronous	
Reentrancy:	Reentrant	
Parameters (in):	ISRID	The ID of a category 2 ISR.
	ClearPending	Defines whether the pending flag shall be cleared (TRUE) or not (FALSE).
Parameters (inout):	None	
Parameters (out):	None	
Return value:	StatusType	E_OK No error. E_OS_ID ISRID is not a valid category 2 ISR identifier (EXTENDED status) E_OS_CALLEVEL Wrong call context of the API function (EXTENDED status) E_OS_ACCESS The calling application is not the owner of the ISR passed in ISRID (Service Protection)
Description:	Enables the interrupt source by modifying the interrupt controller registers. Additionally it may clear the interrupt pending flag	
Available via:	`Os.h`	

⌋ (SRS_Os_11011)

8.4.36 DisableInterruptSource

[SWS_Os_91019] ⌈

Service name:	DisableInterruptSource	

Syntax:	`StatusType DisableInterruptSource(`
	`    ISRType ISRID`
	`)`	

Service ID[hex]:	0x30	
Sync/Async:	Synchronous	
Reentrancy:	Reentrant	
Parameters (in):	ISRID	The ID of a category 2 ISR.
Parameters (inout):	None	
Parameters (out):	None	
Return value:	StatusType	E_OK No error. E_OS_ID ISRID is not a valid category 2 ISR identifier (EXTENDED status) E_OS_CALLEVEL Wrong call context of the API function (EXTENDED status)

		E_OS_ACCESS The calling application is not the owner of the ISR passed in ISRID (Service Protection)
Description:	Disables the interrupt source by modifying the interrupt controller registers.	
Available via:	`Os.h`	

⌋ (SRS_Os_11011)

8.4.37 ClearPendingInterrupt

[SWS_Os_91021] ⌈

Service name:	ClearPendingInterrupt	

Syntax:	`StatusType ClearPendingInterrupt(`
	`    ISRType ISRID`
	`)`	

Service ID[hex]:	0x32	
Sync/Async:	Synchronous	
Reentrancy:	Reentrant	
Parameters (in):	ISRID	The ID of a category 2 ISR.
Parameters (inout):	None	
Parameters (out):	None	
Return value:	StatusType	E_OK No error. E_OS_ID ISRID is not a valid category 2 ISR identifier (EXTENDED status) E_OS_CALLEVEL Wrong call context of the API function (EXTENDED status) E_OS_ACCESS The calling application is not the owner of the ISR passed in ISRID (Service Protection)
Description:	Clears the interrupt pending flag by modifying the interrupt controller registers.	
Available via:	`Os.h`	

⌋ (SRS_Os_11011)

8.4.38 ActivateTaskAsyn

[SWS_Os_91022] ⌈

Service name:	ActivateTaskAsyn	

Syntax:	`void ActivateTaskAsyn(`
	`    TaskType id`
	`)`	

Service ID[hex]:	0x33	
Sync/Async:	Asynchronous	
Reentrancy:	Reentrant	
Parameters (in):	id	The id of the task to be activated
Parameters (inout):	None	
Parameters (out):	None	
Return value:	None	
Description:	Asynchronous version of the ActivateTask() function. Intended to be used for cross core task activation. Possible errors are not returned to the caller, but may be reported via error hooks.	
Available via:	`Os.h`	

⌋ (SRS_Os_80015)

[SWS_Os_00818]⌈ Availability of `ActivateTaskAsyn()`: Available in systems which support OS-Applications. ⌋ (SRS_Os_80015)

Note: If during the task activation an error occurs, and the caller is already gone (e.g. callers OS-Application is already terminated, OR callers core is shutting down OR ...) calls to error hooks are dropped and no reporting is done.

8.4.39 SetEventAsyn

[SWS_Os_91023] ⌈

Service name:	SetEventAsyn	
Syntax:	`void SetEventAsyn( TaskType id, EventMaskType m )`	
Service ID[hex]:	0x34	
Sync/Async:	Asynchronous	
Reentrancy:	Reentrant	
Parameters (in):	id	The id of the task to be activated
	m	Mask of the events to be set
Parameters (inout):	None	
Parameters (out):	None	
Return value:	None	
Description:	Asynchronous version of the SetEvent() function. Intended to be used for cross core event setting. Possible errors are not returned to the caller, but may be reported via error hooks.	
Available via:	`Os.h`	

⌋ (SRS_Os_80015)

[SWS_Os_00819]⌈ Availability of `SetEventAsyn()`: Available in systems which support OS-Applications. ⌋ (SRS_Os_80015)

Note: If during the event setting an error occurs and the caller is already gone (e.g. callers OS-Application is already terminated, OR callers core is shutting down OR ...) calls to error hooks are dropped and no reporting is done.

8.5 IOC

8.5.1 Imported types

In this chapter all types included from the following modules are listed:

Module	***Header File***	***Imported Type***
GENERIC TYPES		<Data1>
		<Data2>

		<Data>
Std_Types	StandardTypes.h	Std_ReturnType
UNDEFINED TYPES		...

8.5.2 Type definitions

None

8.5.3 Constants

Name	Communication	Type	Errorname / Value	Annotation
IOC_E_OK	**All, SND/RCV**	`Std_ReturnType`	`RTE_E_OK / 0`	**No error occurred**
IOC_E_NOK	**All SND/RCV**	`Std_ReturnType`	`RTE_E_NOK / 1`	**Error occurred. Shall be used to identify error cases without error specification.**
IOC_E_LIMIT	**Queued SND**	`Std_ReturnType`	`RTE_E_LIMIT / 130`	**In case of “event” (queued) semantic, the internal buffer within the IOC communication service is full (Case: Receiver slower than sender). This error produces additionally an Overlayed Error on the receiver side at the next data reception.**
IOC_E_LOST_DATA	**Queued RCV**	`Std_ReturnType`	`Overlayed Error RTE_E_LOST_DATA / 64`	**In case of “event” (queued) semantic, this Overlayed Error indicates that the IOC service refuses an IocSend request due to internal buffer overflow.**
IOC_E_NO_DATA	**Queued RCV**	`Std_ReturnType`	`RTE_E_NO_DATA / 131`	**In case of “event” (queued) semantic, no data is available for reception.**

8.5.4 Function definitions

[**SWS_Os_00805**]:⌈ The optional length parameter of the API shall be generated if the VariableDataPrototype is of type dynamic and no size indicator is used in the according ApplicationArrayDataType.⌋ (SRS_Os_80020)

8.5.4.1 IocSend/IocWrite

The `IocWrite` API call is generated for "data" (unqueued) semantics and the `IocSend` API call is generated for "events" (queued) semantics.

[SWS_Os_00718] ⌈

Service name:	IocSend_<IocId>[_<SenderId>]	

Syntax:	`Std_ReturnType IocSend_<IocId>[_<SenderId>](`
	`    <Data> IN,`
	`    [uint16 numberOfBytesIN]`
	`)`	

Service ID[hex]:	0x1e	
Sync/Async:	Asynchronous	
Reentrancy:	This function is generated individually for each sender. The individual function is not reentrant (if called from different runnable entities that belong to the same sender), but different functions can be called in parallel.	

Parameters (in):	IN	Data value to be sent over a communication identified by the <IocId>. The parameter will be passed by value for primitive data elements and by reference for all other types.
		
		Example:
		Std_ReturnType IocSend_RTE_25 (const uint32 UI_Value);
		Std_ReturnType IocSend_RTE_42 (const TASKParams3 *pStr_Value);

	numberOfBytesIN	(optional) number of bytes to be send
Parameters (inout):	None	
Parameters (out):	None	
Return value:	Std_ReturnType	IOC_E_OK: The data has been passed successfully to the communication service. IOC_E_LIMIT: IOC internal communication buffer is full (Case: Receiver is slower than sender). This error produces an IOC_E_LOST_DATA Overlayed Error on the receiver side at the next data reception. IOC_E_LENGTH: The <numberOfBytesIN> exceeds either the internal buffer or is equal zero, so no data is send.
Description:	Performs an "explicit" sender-receiver transmission of data elements with "event" semantic for a unidirectional 1:1 or N:1 communication between OS-Applications located on the same or on different cores. <IocId> is a unique identifier that references a unidirectional 1:1 or N:1 communication. <SenderId> is used only in N:1 communication. Together with <IocId>, it uniquely identifies the sender. It is separated from <IocId> with an underscore. In case of	

	1:1 communication, it shall be omitted.
Available via:	`Ioc.h`

⌋ (SRS_Os_80020)

[SWS_Os_91003] ⌈

Service name:	IocWrite_<IocId>[_<SenderId>]	

Syntax:	`Std_ReturnType IocWrite_<IocId>[_<SenderId>](`
	`    <Data> IN,`
	`    [uint16 numberOfBytesIN]`
	`)`	

Service ID[hex]:	0x1f	
Sync/Async:	Asynhronous	
Reentrancy:	This function is generated individually for each sender. The individual function is not reentrant (if called from different runnable entities that belong to the same sender), but different functions can be called in parallel.	

Parameters (in):	IN	Data value to be sent over a communication identified by the <IocId>. The parameter will be passed by value for primitive data elements and by reference for all other types.
		
		Example:
		Std_ReturnType IocWrite_RTE_25 (const uint32 UI_Value);
		Std_ReturnType IocWrite_RTE_42 (const TASKParams3 *pStr_Value);

	numberOfBytesIN	(optional) number of bytes to be send
Parameters (inout):	None	
Parameters (out):	None	
Return value:	Std_ReturnType	IOC_E_OK: The data has been passed successfully to the communication service. IOC_E_LENGTH: The <numberOfBytesIN> exceeds either the internal buffer or is equal zero, so no data is send.
Description:	Performs an "explicit" sender-receiver transmission of data elements with "data" semantic for a unidirectional 1:1 or N:1 communication between OS-Applications located on the same or on different cores. <IocId> is a unique identifier that references a unidirectional 1:1 or N:1 communication. <SenderId> is used only in N:1 communication. Together with <IocId>, it uniquely identifies the sender. It is separated from <IocId> with an underscore. In case of 1:1 communication, it shall be omitted. <numberOfBytesIN> specifies the size of the data to be transmitted (in bytes).	
Available via:	`Ioc.h`	

⌋ ()

General:

[SWS_Os_00719] ⌈`IocSend/IocWrite` is asynchronous in that way it shall not have to wait for the reception of the data on the receiving side to return from execution. ⌋ (SRS_Os_80020)

[SWS_Os_00720] ⌈The `IocSend/IocWrite` function shall not return until the data given in parameter have been completely physically sent over the communication medium.

For example in case of communication over shared RAM, an `IocSend/IocWrite` shall return when all data have been copied in the target shared RAM. ⌋ (SRS_Os_80020)

[SWS_Os_00721] ⌈In case of "event" (queued) semantic, the `IocSend` function shall guarantee the order of delivery. In case of senders from different cores, the order in which messages are received will be determined by the implementation. ⌋ (SRS_Os_80020)

[SWS_Os_00722] ⌈The `IocSend/IocWrite` function shall support mechanism to guarantee data-Integrity during transmission.

The `IocSend/IocWrite` function shall solve the crossing of the protection boundaries of OS-Applications. It has to be generated in case of intra-core and inter-core communication. ⌋ (SRS_Os_80020)

Parameters:

[SWS_Os_00723] ⌈
The IN <Data> parameter of the `IocSend/IocWrite` function shall be passed by value for primitive data types, as an pointer to the array base type for arrays and by reference for all other types. ⌋ (SRS_Os_80020)

[SWS_Os_00724] ⌈
For data passed as an pointer to the array base type or by reference, the `IocSend/IocWrite` function shall guarantee upon return that the parameter is safe for re-use.⌋ (SRS_Os_80020)

Returned values:

[SWS_Os_00725] ⌈The `IocSend/IocWrite` function shall return `IOC_E_OK` if the data was passed successfully to the communication service. ⌋ (SRS_Os_80020)

[SWS_Os_00726] ⌈In case of "event" semantic the `IocSend` function shall return `IOC_E_LIMIT` if an IOC internal transmission buffer became full (Case: Receiver is slower than sender or/and configured internal IOC buffer size is too small).
If this error occurs the IOC internal buffer could not be filled with the parameter. In that case this error shall produce an IOC_E_LOST_DATA Overlayed Error on the receiver side at the next data reception (s. SWS_Os_00745). ⌋ (SRS_Os_80020)

Internal structures:

[SWS_Os_00727] ⌈In case of "event" semantic the IOC shall configure its internal transmission buffer size with the value of the attribute `OsIocBufferLength`. ⌋ (SRS_Os_80020)

8.5.4.2 IocSendGroup/IocWriteGroup

The `IocWriteGroup` API call is generated for "data" (unqueued) semantics and the `IocSendGroup` API call is generated for "events" (queued) semantics.

[SWS_Os_00728] ⌈

Service name:	IocSendGroup_<IocId>	

Syntax:	`Std_ReturnType IocSendGroup_<IocId>(`
	`    <Data1> IN1,`
	`    [uint16 numberOfBytesIN1,]`
	`    <Data2> IN2,`
	`    [uint16 numberOfBytesIN2,]`
	`    ...`
	`)`	

Service ID[hex]:	0x20	
Sync/Async:	Asynchronous	
Reentrancy:	This function is generated individually for each sender. The individual function is not reentrant (if called from different runnable entities that belong to the same sender), but different functions can be called in parallel.	
Parameters (in):	IN1	List of parameters with data values to be sent over a communication identified by the <IocId>. The parameters will be passed by value for simple data elements and by reference for all other types. Example: Std_ReturnType IocSendGroup_RTE_G1 (const uint32 Ul_Value1, const uint16 Value2, const uint8 Value3, const uint16 Value4);
	numberOfBytesIN1	(optional) number of bytes for parameter IN1 to be send.
	IN2	--
	numberOfBytesIN2	--
	--	--
Parameters (inout):	None	
Parameters (out):	None	
Return value:	Std_ReturnType	IOC_E_OK: The data has been passed successfully to the communication service. IOC_E_LIMIT: IOC internal communication buffer is full (Case: Receiver is slower than sender). This error produces an IOC_E_LOST_DATA Overlayed Error on the receiver side at the next data reception. IOC_E_LENGTH: Al least one of the <numberOfBytesIN<x>> exceeds either the internal buffer or is equal zero, so no data is send.
Description:	Performs an "explicit" sender-receiver transmission of data elements with "event" semantic for a unidirectional 1:1 communication between OS-Applications located on the same or on different cores.	

	This API involves a group of data elements which values are specified in parameter. <IocId> is a unique identifier that references a unidirectional 1:1 communication involving many data elements. The optional parameter <numberOfBytesIN<x>> specifies the size of the data to be transmitted (in bytes) for parameter <IN<x>>.
Available via:	`Ioc.h`

⌋ (SRS_Os_80020)

[SWS_Os_91004] ⌈

Service name:	IocWriteGroup_<IocId>	

Syntax:	`Std_ReturnType IocWriteGroup_<IocId>(`
	`    <Data1> IN1,`
	`    [uint16 numberOfBytesIN1,]`
	`    <Data2> IN2,`
	`    [uint16 numberOfBytesIN2,]`
	`    ...`
	`)`	

Service ID[hex]:	0x21	
Sync/Async:	Asynchronous	
Reentrancy:	This function is generated individually for each sender. The individual function is not reentrant (if called from different runnable entities that belong to the same sender), but different functions can be called in parallel.	
Parameters (in):	IN1	List of parameters with data values to be sent over a communication identified by the <IocId>. The parameters will be passed by value for simple data elements and by reference for all other types. Example: Std_ReturnType IocWriteGroup_RTE_G1 (const uint32 UI_Value1, const uint16 Value2, const uint8 Value3, const uint16 Value4);
	numberOfBytesIN1	(optional) number of bytes for parameter IN1 to be send.
	IN2	--
	numberOfBytesIN2	--
	--	--
Parameters (inout):	None	
Parameters (out):	None	
Return value:	Std_ReturnType	IOC_E_OK: The data has been passed successfully to the communication service. IOC_E_LENGTH: Al least one of the <numberOfBytesIN<x>> exceeds either the internal buffer or is equal zero, so no data is send.
Description:	Performs an "explicit" sender-receiver transmission of data elements with "data" semantic for a unidirectional 1:1 communication between OS-Applications located on the same or on different cores. This API involves a group of data elements which values are specified in parameter.	

	<IocId> is a unique identifier that references a unidirectional 1:1 communication involving many data elements. The optional parameter <numberOfBytesIN<x>> specifies the size of the data to be transmitted (in bytes) for parameter <IN<x>>.
Available via:	`Ioc.h`

⌋ ()

General:

[SWS_Os_00729] ⌈`IocSendGroup/IocWriteGroup` is asynchronous in that way it shall not have to wait for the reception of the data on the receiving side to return from execution. ⌋ (SRS_Os_80020)

[SWS_Os_00730] ⌈The `IocSendGroup/IocWriteGroup` function shall not return until the data given in parameter have been completely physically sent over the communication medium. For example in case of communication over shared RAM, an `IocSendGroup/IocWriteGroup` shall return when all data have been copied in the target shared RAM. ⌋ (SRS_Os_80020)

[SWS_Os_00731] ⌈In case of "event" semantic, the `IocSendGroup` function shall guarantee the order of delivery. ⌋ (SRS_Os_80020)

[SWS_Os_00732] ⌈The `IocSendGroup/IocWriteGroup` function shall support mechanisms to guarantee data-Integrity during transmission.

The `IocSendGroup/IocWriteGroup` function shall solve the crossing of the protection boundaries of OS-Applications. It has to be generated in case of intra-core and inter-core communication. ⌋ (SRS_Os_80020)

Parameters:

[SWS_Os_00733] ⌈
The IN <DataN> parameters of the `IocSendGroup/IocWriteGroup` function shall be passed by values for primitive data types, as pointer to the array base type for arrays and by references for all other types.
⌋ (SRS_Os_80020)

[SWS_Os_00734] ⌈
For data passed as an pointer to the array base type or by reference, the `IocSendGroup/IocWriteGroup` function shall guarantee upon return that the parameter is safe for re-use.
⌋ (SRS_Os_80020)

Returned values:

[SWS_Os_00735] ⌈The `IocSendGroup/IocWriteGroup` function shall return `IOC_E_OK` if the data was passed successfully to the communication service. ⌋ (SRS_Os_80020)

[SWS_Os_00736] ⌈In case of “event” semantic the `IocSendGroup` function shall return `IOC_E_LIMIT` if an IOC internal transmission buffer got full (Case: Receiver is slower than sender or/and configured internal IOC buffer size is too small).

If this error occurs the IOC Internal buffer could not be filled with the parameter. In that case this error produces an IOC_E_LOST_DATA Overlayed Error on the receiver side at the next data reception. ⌋ (SRS_Os_80020)

Internal structures:

[SWS_Os_00737] ⌈In case of “event” semantic the IOC shall configure its internal transmission buffer size with the value of the attribute `OsIocBufferLength`. ⌋ (SRS_Os_80020)

8.5.4.3 IocReceive/IocRead

The `IocRead` API call is generated for "data" and the `IocReceive` API call is generated for "events".

[SWS_Os_00738] ⌈

Service name:	IocReceive_<IocId>	

Syntax:	`Std_ReturnType IocReceive_<IocId>(`
	`    <Data> OUT,`
	`    [uint16* numberOfBytesOUT]`
	`)`	

Service ID[hex]:	0x22	
Sync/Async:	Synchronous	
Reentrancy:	This function is generated individually for each receiver. The individual function is not reentrant (if called from different runnable entities that belong to the same receiver), but different functions can be called in parallel.	
Parameters (in):	None	
Parameters (inout):	None	
Parameters (out):	OUT	Data reference to be filled with the received data element.
	numberOfBytesOUT	(optional) data reference to be filled with the length of the received data element in bytes.
Return value:	Std_ReturnType	IOC_E_OK: Data was received successfully IOC_E_NO_DATA: No data is available for reception. IOC_E_LOST_DATA: This Overlayed Error indicates that the IOC communication service refused an IOCSend request from sender due to an internal buffer overflow. There is no error in the data returned in parameter.
Description:	Performs an "explicit" sender-receiver reception of data elements with "event" semantic for a unidirectional communication between OS-Applications located on the same or on different cores..	

	<IocId> is a unique identifier that references a unidirectional 1:1 or N:1 communication.
Available via:	`Ioc.h`

⌋ (SRS_Os_80020)

[SWS_Os_91005] ⌈

Service name:	IocRead_<IocId>	

Syntax:	`Std_ReturnType IocRead_<IocId>(`
	`    <Data> OUT,`
	`    [uint16* numberOfBytesOUT]`
	`)`	

Service ID[hex]:	0x23	
Sync/Async:	Synchronous	
Reentrancy:	This function is generated individually for each receiver. The individual function is not reentrant (if called from different runnable entities that belong to the same receiver), but different functions can be called in parallel.	
Parameters (in):	None	
Parameters (inout):	None	
Parameters (out):	OUT	Data reference to be filled with the received data element.
	numberOfBytesOUT	(optional) data reference to be filled with the length of the received data element in bytes.
Return value:	Std_ReturnType	IOC_E_OK: Data was received successfully
Description:	Performs an "explicit" sender-receiver reception of data elements with "data" semantic for a unidirectional communication between OS-Applications located on the same or on different cores. <IocId> is a unique identifier that references a unidirectional 1:1 or N:1 communication.	
Available via:	`Ioc.h`	

⌋ ()

General:

[SWS_Os_00739] ⌈A successful call to the `IocReceive/IocRead` function indicates that data has been received successfully in the `OUT <Data>` given in parameter.

The `IocReceive/IocRead` function has to be generated in case of intra-core and inter-core communication. ⌋ (SRS_Os_80020)

[SWS_Os_00740] ⌈If the `OsIocReceiverPullCB` attribute is defined with a callback function name, the IOC shall call this function on the receiving core for each data transmission. ⌋ (SRS_Os_80020)

Parameters:

[SWS_Os_00741] ⌈In case of “data” semantic the `IocRead` function shall always be able to deliver the last available datum. In case of senders from different cores, the precision of the order might be limited by the hardware and implementation. ⌋ (SRS_Os_80020)

[SWS_Os_00742] ⌈The IocReceive/IocRead function shall guarantee upon returning from execution that the reference given in parameter is safe for use. ⌋ (SRS_Os_80020)

[SWS_Os_00803]⌈ The OUT <Data> parameter of the IocReceive/IocRead function shall be passed as an pointer to the array base type for arrays and by reference for all other types.⌋ (SRS_Os_80020)

Returned values:

[SWS_Os_00743] ⌈The IocReceive/IocRead function shall return IOC_E_OK if the data was received successfully in the OUT <Data> parameter. ⌋ (SRS_Os_80020)

[SWS_Os_00744] ⌈In case of “event” semantic and if no data is available the function IocReceive shall return IOC_E_NO_DATA. ⌋ (SRS_Os_80020)

[SWS_Os_00745] ⌈In case of “event” semantic an IOC_E_LOST_DATA Overlayed Error shall be returned by the IocReceive function if the IOC communication service refused an IocSend request from sender due to an internal buffer overflow. There is no error in the data returned in parameter. ⌋ (SRS_Os_80020)

8.5.4.4 IocReceiveGroup/IocReadGroup

The IocReadGroup API call is generated for "data" and the IocReceiveGroup API call is generated for "events".

[SWS_Os_00746] ⌈

Service name:	IocReceiveGroup_<IocId>	

Syntax:	`Std_ReturnType IocReceiveGroup_<IocId>(`
	`    <Data1> OUT1,`
	`    [uint16* numberOfBytesOUT1,]`
	`    <Data2> OUT2,`
	`    [uint16* numberOfBytesOUT2,]`
	`    ...`
	`)`	

Service ID[hex]:	0x24	
Sync/Async:	Synchronous	
Reentrancy:	This function is generated individually for each receiver. The individual function is not reentrant (if called from different runnable entities that belong to the same receiver), but different functions can be called in parallel.	
Parameters (in):	None	
Parameters (inout):	None	
Parameters (out):	OUT1	List of data references to be filled with the received data elements. The specified order of the parameter shall match to the specified order in the corresponding send function.
	numberOfBytesOUT1	(optional) data reference to be filled with the length of the received data element (OUT1) in bytes.
	OUT2	--

	numberOfBytesOUT2	--
	--	--
Return value:	Std_ReturnType	IOC_E_OK: Data was received successfully IOC_E_NO_DATA: No data is available for reception. IOC_E_LOST_DATA: This Overlayed Error indicates that the IOC communication service refused an IOCSend request from sender due to an internal buffer overflow. There is no error in the data returned in parameter.
Description:	Performs an "explicit" sender-receiver transmission of data elements with "event" semantic for a unidirectional 1:1 communication between OS-Applications located on the same or on different cores. This API involves a group of data elements which values are specified in parameter. <IocId> is a unique identifier that references a unidirectional 1:1 communication involving many data elements.	
Available via:	`Ioc.h`	

⌋ (SRS_Os_80020)

[SWS_Os_91006] ⌈

Service name:	IocReadGroup_<IocId>	

Syntax:	`Std_ReturnType IocReadGroup_<IocId>(`
	`    <Data1> OUT1,`
	`    [uint16* numberOfBytesOUT1,]`
	`    <Data2> OUT2,`
	`    [uint16* numberOfBytesOUT2,]`
	`    ...`
	`)`	

Service ID[hex]:	0x25	
Sync/Async:	Synchronous	
Reentrancy:	This function is generated individually for each receiver. The individual function is not reentrant (if called from different runnable entities that belong to the same receiver), but different functions can be called in parallel.	
Parameters (in):	None	
Parameters (inout):	None	
Parameters (out):	OUT1	List of data references to be filled with the received data elements. The specified order of the parameter shall match to the specified order in the corresponding send function.
	numberOfBytesOUT1	(optional) data reference to be filled with the length of the received data element (OUT1) in bytes.
	OUT2	--
	numberOfBytesOUT2	--
	--	--
Return value:	Std_ReturnType	IOC_E_OK: Data was received successfully
Description:	Performs an "explicit" sender-receiver transmission of data elements with a "data" semantic for a unidirectional 1:1 communication between OS-Applications located on the same or on different cores. This API involves a group of data elements which values are specified in parameter. <IocId> is a unique identifier that references a unidirectional 1:1 communication	

	involving many data elements.
Available via:	`Ioc.h`

⌋ ()

General:

[SWS_Os_00747] ⌈A successful call to the `IocReceiveGroup/IocReadGroup` function indicates that data has been received successfully in the given parameters.

The `IocReceiveGroup/IocReadGroup` function has to be generated in case of intra-core and inter-core communication. ⌋ (SRS_Os_80020)

[SWS_Os_00748] ⌈If the `OsIocReceiverPullCB` attribute is defined with a callback function name, the IOC shall call this function on the receiving core for each data transmission. ⌋ (SRS_Os_80020)

Parameters:

[SWS_Os_00749] ⌈In case of “data” semantic the `IocReadGroup` function shall always be able to deliver the last available datum. ⌋ (SRS_Os_80020)

[SWS_Os_00750] ⌈The `IocReceiveGroup/IocReadGroup` function shall guarantee upon returning from execution that the references given in parameters are safe for use. ⌋ (SRS_Os_80020)

[SWS_Os_00804]⌈ The OUT <DataN> parameters of the `IocReceiveGroup/IocReadGroup` function shall be passed as pointer to the array base type for arrays and by references for all other types.⌋ ()

Returned values:

[SWS_Os_00751] ⌈The `IocReceiveGroup/IocReadGroup` function shall return `IOC_E_OK` if the data was received successfully in the list of references given in parameter. ⌋ (SRS_Os_80020)

[SWS_Os_00752] ⌈In case of “event” semantic and if no data is available the function `IocReceiveGroup` shall return `IOC_E_NO_DATA`. ⌋ (SRS_Os_80020)

[SWS_Os_00753] ⌈In case of “event” semantic an `IOC_E_LOST_DATA` Overlayed Error shall be returned by the `IocReceiveGroup` function if the IOC communication service refused an `IocSendGroup` request from sender due to an internal buffer overflow. There is no error in the data returned in parameter. ⌋ (SRS_Os_80020)

8.5.4.5 IocEmptyQueue

[SWS_Os_00754] ⌈

Service name:	IocEmptyQueue_<IocId>

Syntax:	`Std_ReturnType IocEmptyQueue_<IocId>( void )`	
Service ID[hex]:	0x26	
Sync/Async:	Synchronous	
Reentrancy:	Non reentrant	
Parameters (in):	None	
Parameters (inout):	None	
Parameters (out):	None	
Return value:	Std_ReturnType	IOC_E_OK: Content of the queue was successfully deleted
Description:	In case of queued communication identified by the <IocId> in the function name, the content of the IOC internal communication queue shall be deleted.	
Available via:	`Ioc.h`	

⌋ (SRS_Os_80020)

General:

[SWS_Os_00755] ⌈The function `IocEmptyQueue_<IocId>` shall be present for all IOC elements with queued semantics. ⌋ (SRS_Os_80020)

[SWS_Os_00756] ⌈The function `IocEmptyQueue_<IocId>` shall delete all contents from the associated data queue.

The `IocEmptyQueue` should be generated in a more efficient way than an iterative call to an `IocReceive` function. ⌋ (SRS_Os_80020)

8.6 Expected Interfaces

In this chapter all interfaces required from other modules are listed.

8.6.1 Mandatory Interfaces

There are no mandatory interfaces for the IOC.

8.6.2 Optional Interfaces

8.6.2.1 ReceiverPullCB

[SWS_Os_00757] ⌈

Service name:	<ReceiverPullCB>
Syntax:	`void <ReceiverPullCB>( void )`
Service ID[hex]:	--
Sync/Async:	Synchronous
Reentrancy:	Reentrant
Parameters (in):	None

Parameters (inout):	None
Parameters (out):	None
Return value:	None
Description:	This callback function can be configured for the receiver of a communication. If configured, IOC calls this callback on the receiving core for each data reception. <ReceiverPullCB> is the callback function name configured by the receiver in the OsIocReceiverPullCB attribute to be called on data reception."
Available via:	`Os.h`

⌋ (SRS_Os_80020)

[SWS_Os_00758] ⌈The `<ReceiverPullCB>` function name shall be defined within a configuration file for each IOC communication in the `OsIocReceiverPullCB` attribute. ⌋ (SRS_Os_80020)

[SWS_Os_00759] ⌈The name of the callback shall be unique over the micro controller. For this purpose the following example can be considered as orientation for the IOC user:

Example: `Rte_IocReceiveCB_<IocId>`⌋ (SRS_Os_80020)

[SWS_Os_00760] ⌈The `<ReceiverPullCB>` function on the receiver side is using the access rights of the receiving OsApplication. ⌋ (SRS_Os_80020)

Note: This means that such a callback cannot be reused by another OsApplication.

[SWS_Os_00761] ⌈This notification mechanism shall be supported for both queued and unqueued communication semantic. ⌋ (SRS_Os_80020)

The owner of the `<ReceiverPullCB>` function shall pay attention that the execution time of the function shall not last too long. It shall be possible to call this function from an IOC-ISR.

8.7 Hook functions

Hook functions are called by the operating system if specific conditions are met. They are provided by the user. Besides the ProtectionHook below, the hooks from [16] and/or extensions from 7.12 may be called by the OS.

8.7.1 Protection Hook

[SWS_Os_00538] ⌈

Service name:	ProtectionHook	

Syntax:	`ProtectionReturnType ProtectionHook(`
	`    StatusType Fatalerror`
	`)`	

Service ID[hex]:	--	
Sync/Async:	Synchronous	
Reentrancy:	Reentrant	
Parameters (in):	Fatalerror	The error which caused the call to the protection hook
Parameters (inout):	None	
Parameters (out):	None	
Return value:	ProtectionReturnType	PRO_IGNORE PRO_TERMINATETASKISR PRO_TERMINATEAPPL PRO_TERMINATEAPPL_RESTART PRO_SHUTDOWN The return value defines the action the OS shall take after the protection hook.
Description:	The protection hook is always called if a serious error occurs. E.g. exceeding the worst case execution time or violating against the memory protection.	
Available via:	`Os_Externals.h`	

⌋ ()

Depending on the return value the Operating System module will either:

- forcibly terminate the Task/Category 2 ISR which causes the problem OR
- forcibly terminate the OS-Application the Task/Category 2 ISR belong (optional with restart) OR
- shutdown the system OR
- do nothing

(see 7.8.2)

[SWS_Os_00308] ⌈If `ProtectionHook()` returns an invalid value, the Operating System module shall take the same action as if no protection hook is configured. ⌋ ( )

[SWS_Os_00542] ⌈Availability of `ProtectionHook()`: Available in Scalability Classes 2, 3 and 4. ⌋ ( )

8.7.2 Application specific StartupHook

[SWS_Os_00539] ⌈

Service name:	StartupHook_<App>
Syntax:	`void StartupHook_<App>( void )`
Service ID[hex]:	--
Sync/Async:	Synchronous
Reentrancy:	Reentrant
Parameters (in):	None
Parameters (inout):	None
Parameters (out):	None
Return value:	None
Description:	The application specific startup hook is called during the start of the OS (after the user has started the OS via StartOS()).
Available via:	`Os_Externals.h`

⌋ ()

The application specific StartupHook is always called after the standard StartupHook() (see SWS_Os_00236) . If more than one OS-Application is configured which use startup hooks, the order of calls to the startup hooks of the different OS-Applications is not defined.

[SWS_Os_00543] ⌈Availability of `StartupHook_<App>()`: Available in Scalability Classes 3 and 4. ⌋ ( )

8.7.3 Application specific ErrorHook

[SWS_Os_00540] ⌈

Service name:	ErrorHook_<App>	
Syntax:	`void ErrorHook_<App>( StatusType Error )`	
Service ID[hex]:	--	
Sync/Async:	Synchronous	
Reentrancy:	Reentrant	
Parameters (in):	Error	The error which caused the call to the error hook
Parameters (inout):	None	
Parameters (out):	None	
Return value:	None	
Description:	The application specific error hook is called whenever a Task or Category 2 ISR which belongs to the OS-Application causes an error.	
Available via:	`Os_Externals.h`	

⌋ ()

If the general `ErrorHook()` is configured, the general `ErrorHook()` is called before the application specific error hook is called (see SWS_Os_00246).

[SWS_Os_00544] ⌈Availability of `ErrorHook_<App>()`: Available in Scalability Classes 3 and 4. ⌋ ( )

8.7.4 Application specific ShutdownHook

[SWS_Os_00541] ⌈

Service name:	ShutdownHook_<App>	
Syntax:	`void ShutdownHook_<App>( StatusType Fatalerror )`	
Service ID[hex]:	--	
Sync/Async:	Synchronous	
Reentrancy:	Reentrant	
Parameters (in):	Fatalerror	The error which caused the action to shut down the operating system.
Parameters (inout):	None	
Parameters (out):	None	
Return value:	None	
Description:	The application specific shutdown hook is called whenever the system starts the shut down of itself.	
Available via:	`Os_Externals.h`	

⌋ ()

If the general `ShutdownHook()` is configured, the general `ShutdownHook()` is called after all application specific shutdown hook(s) are called (see SWS_Os_00237). If more OS-Applications with an application specific shutdown hook exist the order of calls to these application specific shutdown hooks is not defined.

[SWS_Os_00545] ⌈Availability of `ShutdownHook_<App>()`: Available in Scalability Classes 3 and 4. ⌋ ( )

8.8 Service Interfaces

8.8.1 Port interface of Os

[] ⌈

Name	OsService		
Kind	ProvidedPort	Interface	OsService_{Counter}
Description	--		

Port Defined Argument Value(s)	Type	CounterType
	Value	{ecuc(Os/OsCounter)}
Variation	--	

⌋ ()

8.8.2 Client-Server-Interfaces

8.8.2.1 Os_Service

[SWS_Os_00560] ⌈

Name	OsService_{Counter}	
Comment	--	
IsService	true	
Variation	({ecuc(Os/OsCounter/OsSecondsPerTick)} != NULL) Counter = {ecuc(Os/OsCounter.SHORT-NAME)}	
Possible Errors	0	E_OK
	1	E_OS_ACCESS
	3	E_OS_ID
	7	E_OS_STATE
	8	E_OS_VALUE

Operations

GetCounterValue			
Comments	This service reads the current count value of a counter (returning either the hardware timer ticks if counter is driven by hardware or the software ticks when user drives counter).		
Variation	--		
Parameters	Value	Comment	Contains the current tick value of the counter
		Type	TimeInMicrosecondsType
		Variation	--
		Direction	OUT
Possible Errors	E_OK	Operation successful	
	E_OS_ID	The <CounterID> was not valid The usage of this error value depends on the decision whether standard (for production)or extended error checking (development phase) shall be used.	

		This can be configured by means of the ECUC parameter OsStatus.

GetElapsedValue			
Comments	This service gets the number of ticks between the current tick value and a previously read tick value.		
Variation	--		
Parameters	Value	Comment	in: the previously read tick value of the counter out: the current tick value of the counter
		Type	TimeInMicrosecondsType
		Variation	--
		Direction	INOUT
	ElapsedValue	Comment	The difference to the previous read value
		Type	TimeInMicrosecondsType
		Variation	--
		Direction	OUT
Possible Errors	E_OK	Operation successful	
	E_OS_ID	The CounterID was not valid	
	E_OS_VALUE	The given Value was not valid	

⌋ ()

8.8.2.2 Implementation Data Types

[SWS_Os_00794] ⌈

Name	TimeInMicrosecondsType
Kind	Type
Derived from	uint64
Description	--
Variation	--
Available via	Rte_Os_Type.h

⌋ ()

[SWS_Os_00786] ⌈

Name	CounterType
Kind	Type
Derived from	uint32
Description	This data type identifies a counter.
Variation	--
Available via	Rte_Os_Type.h

⌋ ()

9 Sequence diagrams

9.1 Sequence chart for calling trusted functions

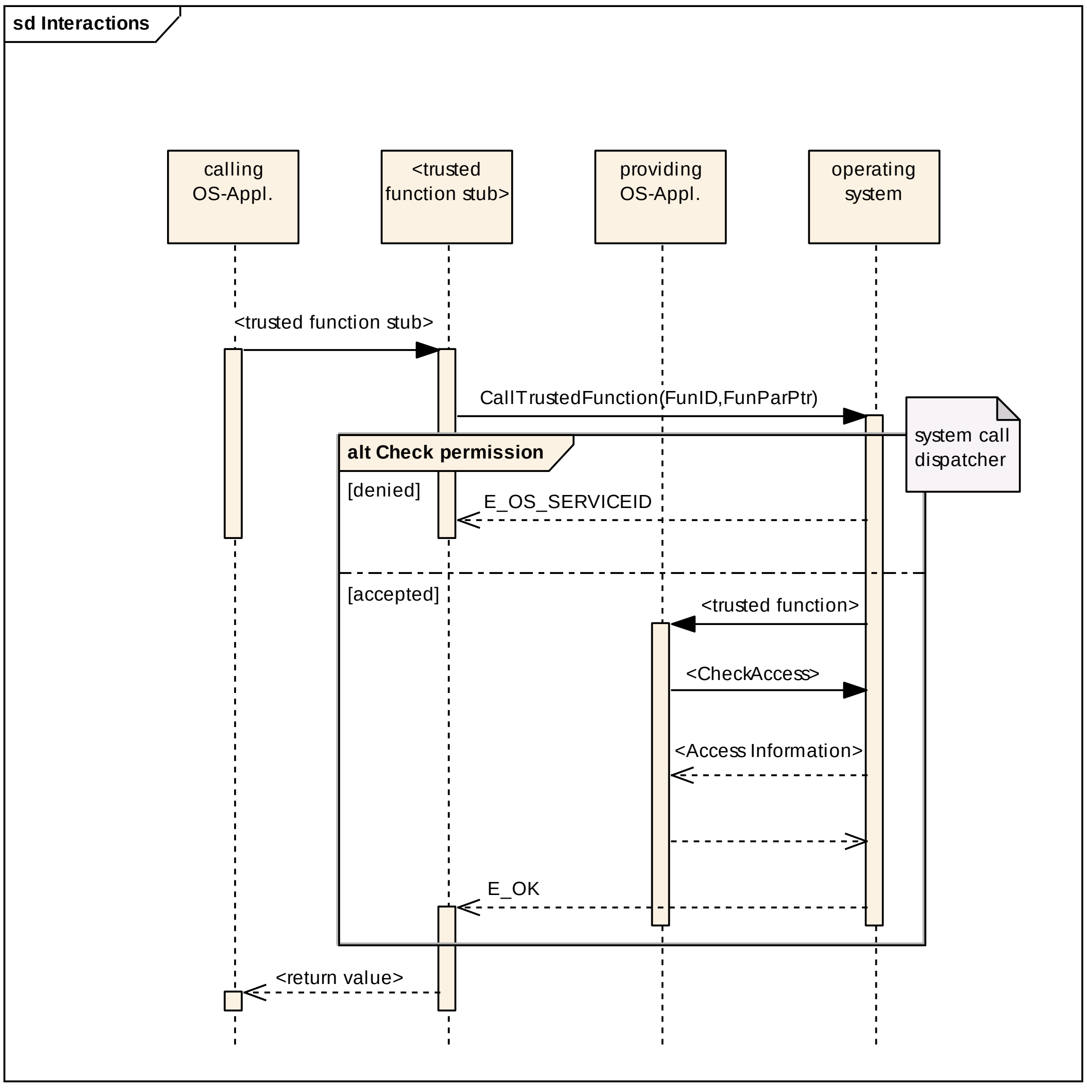

Figure 9.1: System Call sequence chart

The above sequence describes a call to the CallTrustedFunction service. It starts with a user who calls a service which requires itself a call to a trusted function. The service then packs the argument for the trusted function into a structure and calls CallTrustedFunction with the ID and the pointer as arguments. Afterwards the OS checks if the access to the requested service is valid. If no access is granted `E_OS_SERVICEID` is returned. Otherwise the trusted service itself is called and the function checks the arguments for access right, etc.

9.2 Sequence chart for usage of ErrorHook

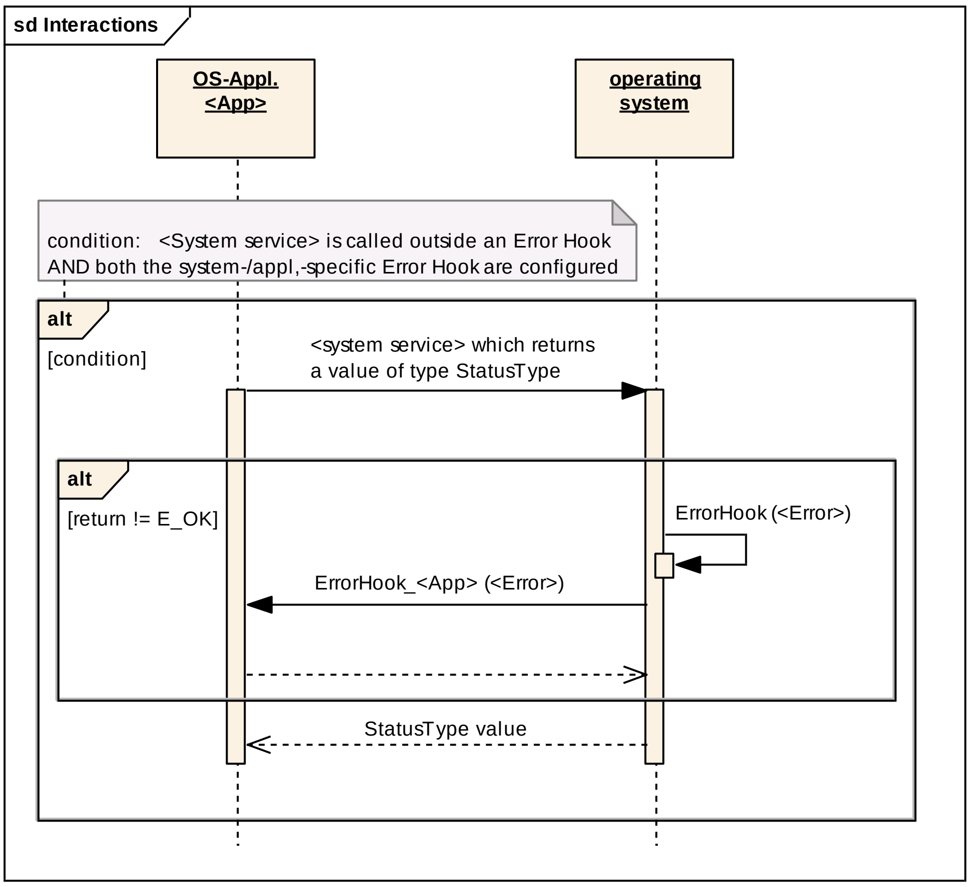
sd Interactions
OS-Appl. <App>
operating system
condition: <System service> is called outside an Error Hook AND both the system-/appl,-specific Error Hook are configured
alt
[condition]
<system service> which returns a value of type StatusType
alt
[return != E_OK]
ErrorHook (<Error>)
ErrorHook_<App> (<Error>)
StatusType value

Figure 9.2: Error Hook sequence chart

The above sequence chart shows the sequence of error hook calls in case a service does not return with `E_OK`. Note that in this case the general error hook and the OS-Application specific error hook are called.

9.3 Sequence chart for ProtectionHook

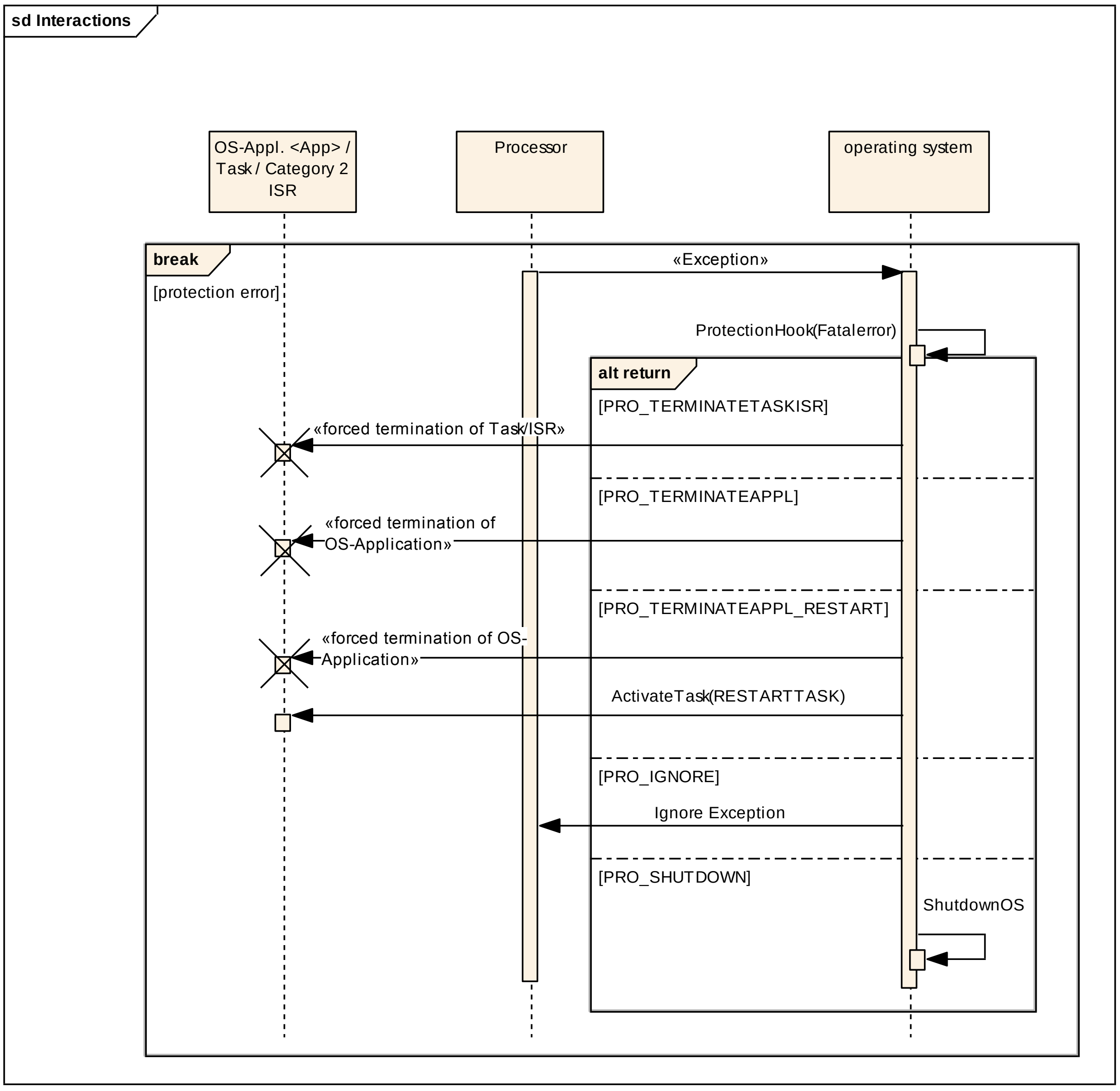
sd Interactions
OS-Appl. <App> / Task / Category 2 ISR
Processor
operating system
break
[protection error]
«Exception»
ProtectionHook(Fatalerror)
alt return
[PRO_TERMINATETASKISR]
«forced termination of Task/ISR»
[PRO_TERMINATEAPPL]
«forced termination of OS-Application»
[PRO_TERMINATEAPPL_RESTART]
«forced termination of OS-Application»
ActivateTask(RESTARTTASK)
[PRO_IGNORE]
Ignore Exception
[PRO_SHUTDOWN]
ShutdownOS

Figure 9.3: Protection Hook sequence chart

The sequence shows the flow of control if a protection error occurs. Depending on the return values of the ProtectionHook, either the faulty Task/ISR is forcibly terminated or the OS-Application is forcibly terminated or the system is shut down. If the action is to terminate the faulty OS-Application an option is to start afterwards the restart task, which can do a cleanup, etc.

9.4 Sequence chart for StartupHook

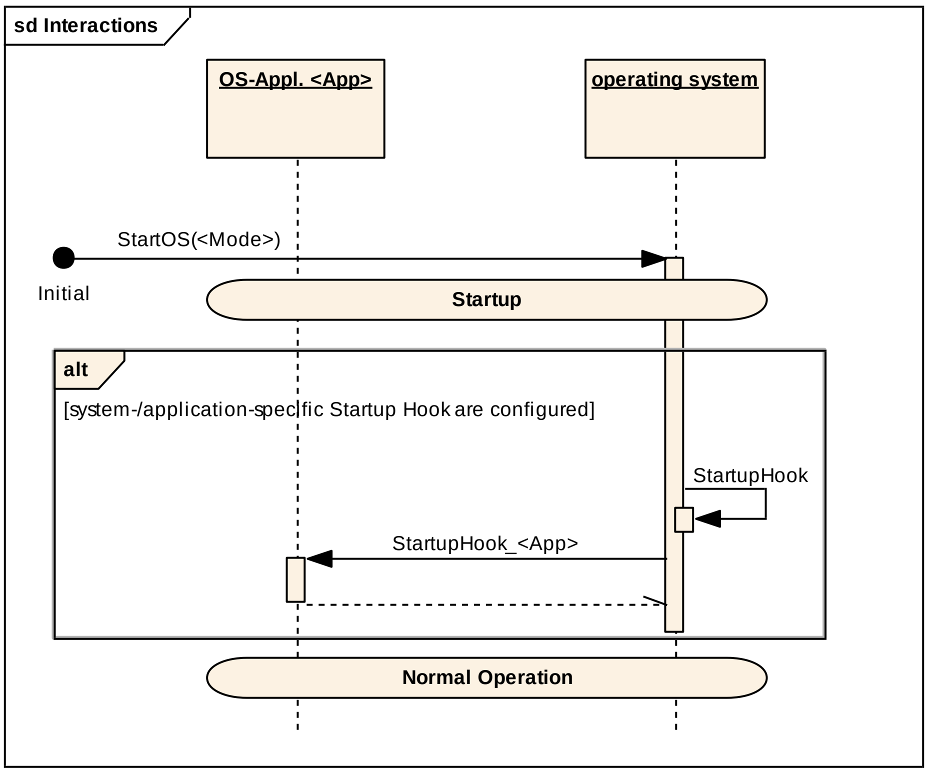

Figure 9.4: StartupHook sequence chart

The above sequence shows the flow of control during the startup of the OS. Like in OSEK OS the user calls the `StartOS()` service to start the OS. During the startup the startup hooks are called in the above order. The rest of the startup sequence is identical to the defined behaviour of OSEK OS.

9.5 Sequence chart for ShutdownHook

The next sequence shows the behaviour in case of a shut down. The flow is the same as in OSEK OS with the exception that the shut down hooks of the OS-Applications are called before the general ShutdownHook is called. Note that the specific shutdown hooks of the application are not allowed to block, they must return to the caller.

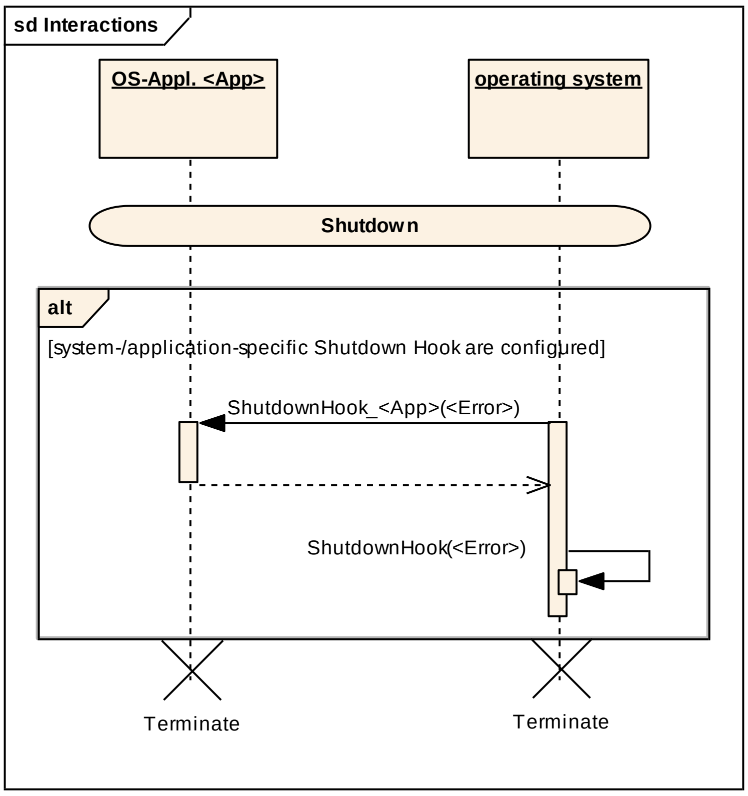

Figure 9.5: ShutdownHook sequence chart

9.6 Sequence diagrams of Sender Receiver communication over the IOC

9.6.1 LastIsBest communication

The figure 11 shows a sequence of successful and failure cases in the interaction between the IOC and the RTE in case of LastIstBest communication (“data” semantic).

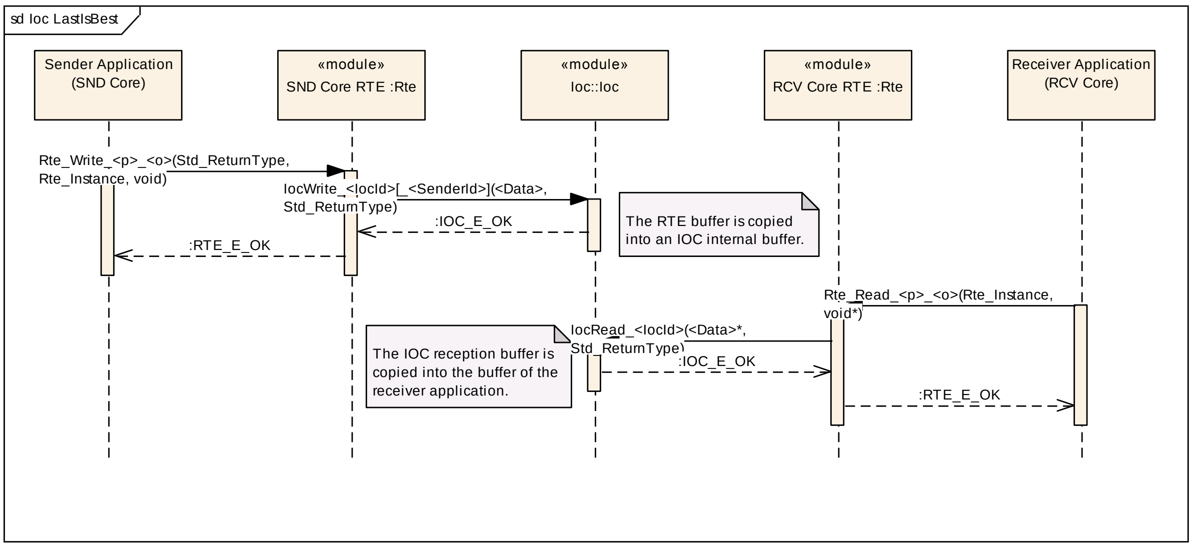
sd Ioc LastIsBest
Sender Application (SND Core)
«module» SND Core RTE :Rte
«module» Ioc::Ioc
«module» RCV Core RTE :Rte
Receiver Application (RCV Core)
Rte_Write_<p>_<o>(Std_ReturnType, Rte_Instance, void)
IocWrite_<IocId>[_<SenderId>](<Data>, Std_ReturnType)
:IOC_E_OK
:RTE_E_OK
The RTE buffer is copied into an IOC internal buffer.
Rte_Read_<p>_<o>(Rte_Instance, void*)
IocRead_<IocId>(<Data>*, Std_ReturnType)
:IOC_E_OK
:RTE_E_OK
The IOC reception buffer is copied into the buffer of the receiver application.

Figure 12: IOC - LastIsBest communication

9.6.2 Queued communication without pull callback

The figure 12 shows the interaction between IOC and RTE with a focus on the congestion control for a queued communication.

The defined communication has no callback functionality for data reception, has an internal buffer size of 2 data elements, no waitpoints are defined and the implicated OS-Applications are located on different cores.

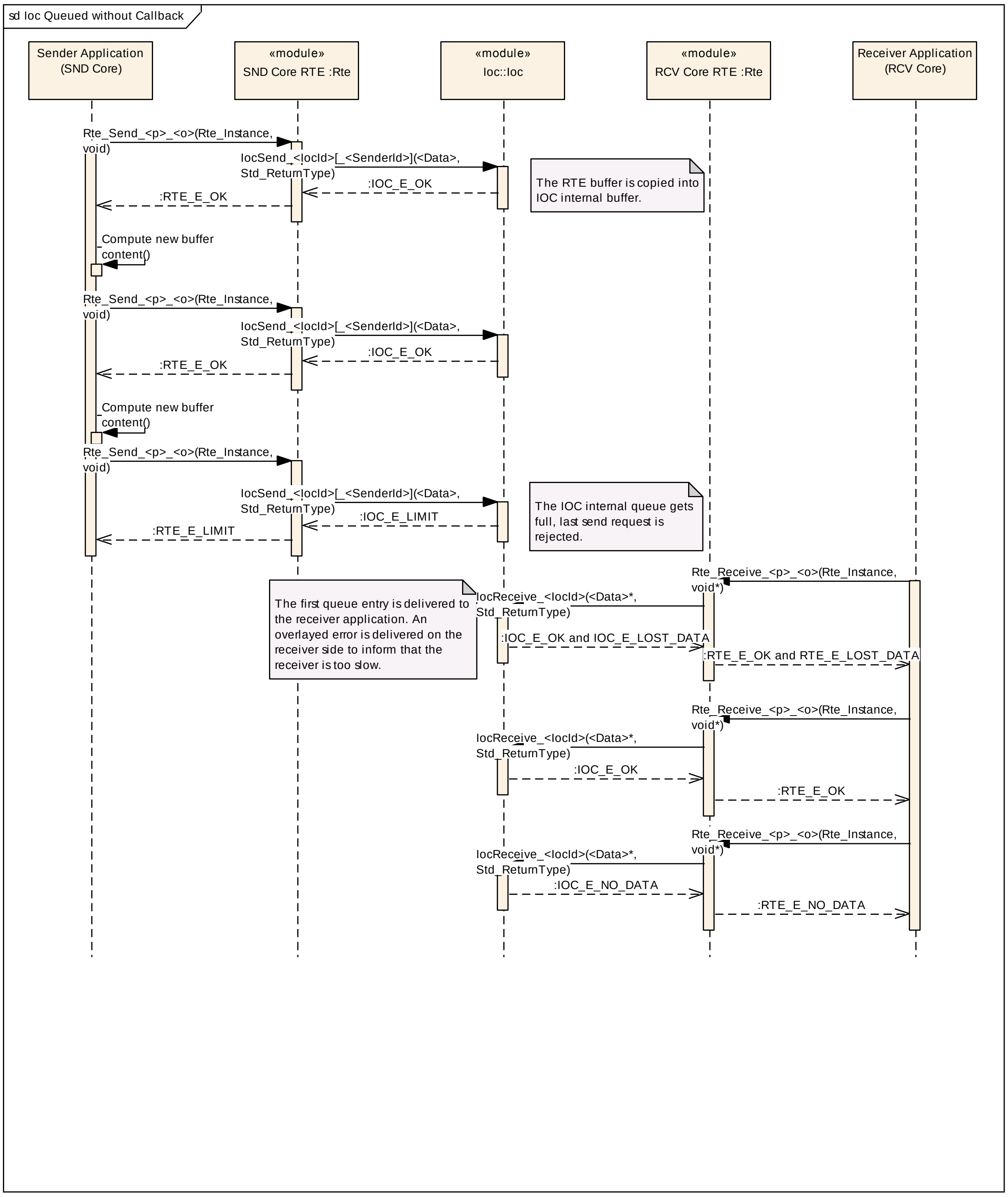
sd Ioc Queued without Callback
Sender Application (SND Core)
«module» SND Core RTE :Rte
«module» Ioc::Ioc
«module» RCV Core RTE :Rte
Receiver Application (RCV Core)
Rte_Send_<p>_<o>(Rte_Instance, void)
IocSend_<IocId>[_<SenderId>](<Data>, Std_ReturnType)
:IOC_E_OK
:RTE_E_OK
The RTE buffer is copied into IOC internal buffer.
Compute new buffer content()
The IOC internal queue gets full, last send request is rejected.
:IOC_E_LIMIT
:RTE_E_LIMIT
Rte_Receive_<p>_<o>(Rte_Instance, void*)
IocReceive_<IocId>(<Data>*, Std_ReturnType)
The first queue entry is delivered to the receiver application. An overlayed error is delivered on the receiver side to inform that the receiver is too slow.
:IOC_E_OK and IOC_E_LOST_DATA
:RTE_E_OK and RTE_E_LOST_DATA
:IOC_E_NO_DATA
:RTE_E_NO_DATA

Figure 13: IOC - Queued communication without callback

9.6.3 Queued communication with pull callback

The figure 13 shows the interaction between IOC and RTE in case of a queued communication with an activated callback functionality. The RTE might handle notification internally and might therefore not provide any callback functions, but a similar scenario will occur in case of communication between CDDs on different cores. The receiving CDD will provide the callback function in this case.

The defined communication has no waitpoints and describes a communication implicating two OS-Applications located on different cores.

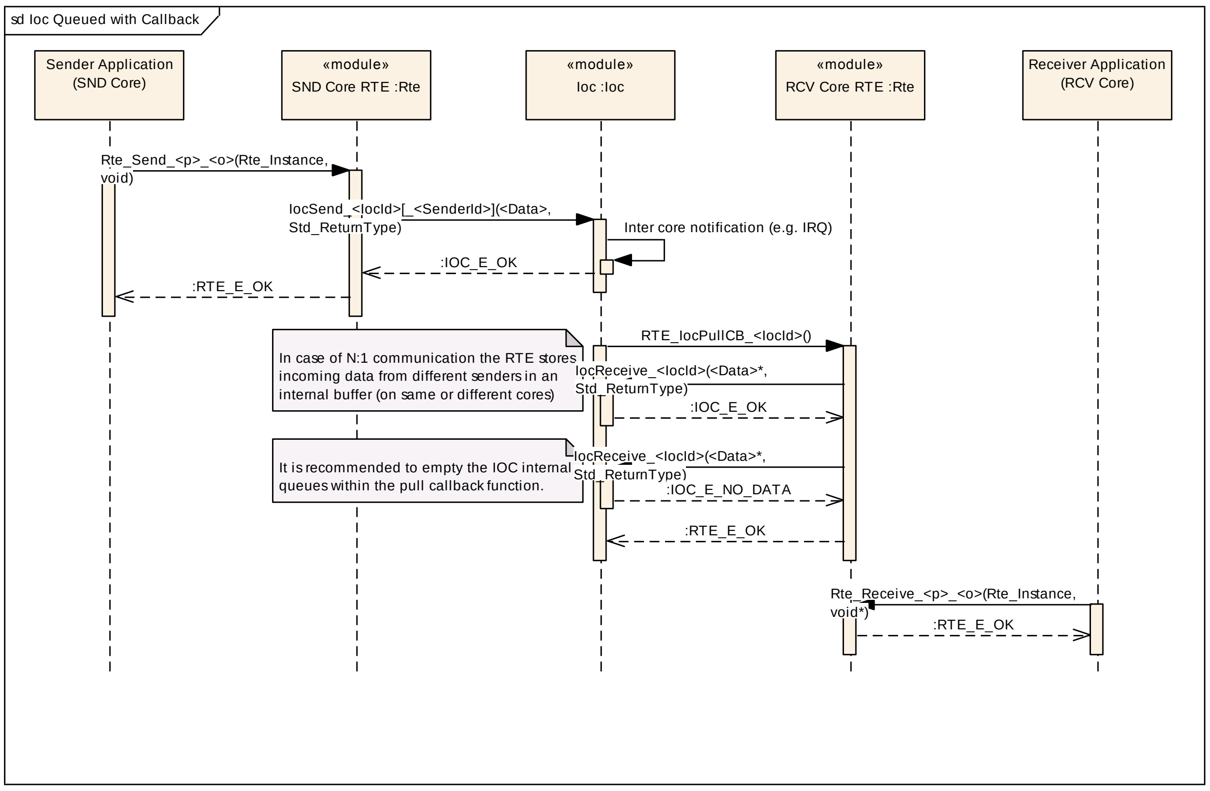
sd Ioc Queued with Callback
Sender Application (SND Core)
«module» SND Core RTE :Rte
«module» Ioc :Ioc
«module» RCV Core RTE :Rte
Receiver Application (RCV Core)
Rte_Send_<p>_<o>(Rte_Instance, void)
IocSend_<IocId>[_<SenderId>](<Data>, Std_ReturnType)
Inter core notification (e.g. IRQ)
:IOC_E_OK
:RTE_E_OK
RTE_IocPullCB_<IocId>()
In case of N:1 communication the RTE stores incoming data from different senders in an internal buffer (on same or different cores)
IocReceive_<IocId>(<Data>*, Std_ReturnType)
:IOC_E_OK
It is recommended to empty the IOC internal queues within the pull callback function.
IocReceive_<IocId>(<Data>*, Std_ReturnType)
:IOC_E_NO_DATA
:RTE_E_OK
Rte_Receive_<p>_<o>(Rte_Instance, void*)
:RTE_E_OK

Figure 14: IOC Queued Communication with callback

10 Configuration Specification

In general, this chapter defines configuration parameters and their clustering into containers. In order to support the specification Chapter 10.1 describes fundamentals. It also specifies a template (table) you shall use for the parameter specification.

Chapter 10.2 specifies the structure (containers) and the parameters of the module Os.

Chapter 10.4 specifies published information of the module Os.

10.1 How to read this chapter

For details refer to the chapter 10.1 “Introduction to configuration specification” in *SWS_BSWGeneral.*

10.1.1 Rules for paramters

Some configuration parameters are configured as floating point values and sometimes these values must be rounded in order to be used. The following rules define the rounding of specific parameters:

- Execution times (for the timing protection) are “round down”
- Timeframes are “round down”

10.2 Containers and configuration parameters

The following chapters summarize all configuration parameters and their containers. Background information about the detailed meaning of the parameters can be found in chapters 7 and 8.

For better readability OIL names of the 2.1 OS specification are given in curly braces in the namefield of configuration parameters.

10.2.1 Os

SWS Item	**ECUC_Os_00396 :**
Module Name	*Os*
Module Description	Configuration of the Os (Operating System) module.
Post-Build Variant Support	false
Supported Config Variants	VARIANT-PRE-COMPILE

Included Containers		
Container Name	***Multiplicity***	***Scope / Dependency***
OsAlarm	0..*	An OsAlarm may be used to asynchronously inform or activate a specific task. It is possible to start alarms automatically at

		system start-up depending on the application mode.
OsAppMode	1..*	OsAppMode is the object used to define ISO 17356-3 properties for an ISO 17356-3 application mode. No standard attributes are defined for AppMode. In a CPU, at least one AppMode object has to be defined. [source: ISO 17356-6] An OsAppMode called OSDEFAULTAPPMODE must always be there for ISO 17356 compatibility.
OsApplication	0..*	An AUTOSAR OS must be capable of supporting a collection of OS objects (tasks, interrupts, alarms, hooks etc.) that form a cohesive functional unit. This collection of objects is termed an OS-Application. All objects which belong to the same OS-Application have access to each other. Access means to allow to use these objects within API services. Access by other applications can be granted separately.
OsCounter	0..*	Configuration information for the counters that belong to the OsApplication.
OsEvent	0..*	Representation of OS events in the configuration context. Adopted from the ISO 17356-6 specification.
OsIoc	0..1	Configuration of the IOC (Inter OS Application Communicator).
OsIsr	0..*	The OsIsr container represents an ISO 17356 interrupt service routine.
OsOS	1	OS is the object used to define ISO 17356-3 properties for an ISO 17356 application. Per CPU exactly one OS object has to be defined.
OsPeripheralArea	0..65534	Container to structure the configuration parameters of one peripheral area. The container short name can be used to access this area.
OsResource	0..*	An OsResource object is used to co-ordinate the concurrent access by tasks and ISRs to a shared resource, e.g. the scheduler, any program sequence, memory or any hardware area.
OsScheduleTable	0..*	An OsScheduleTable addresses the synchronization issue by providing an encapsulation of a statically defined set of alarms that cannot be modified at runtime.
OsSpinlock	0..*	An OsSpinlock object is used to co-ordinate concurrent access by TASKs/ISR2s on different cores to a shared resource.
OsTask	0..*	This container represents an ISO 17356 task.

10.2.2 OsAlarmSetEvent

SWS Item	**ECUC_Os_00016 :**
Container Name	OsAlarmSetEvent
Description	This container specifies the parameters to set an event
Configuration Parameters	

SWS Item	**ECUC_Os_00017 :**
Name	OsAlarmSetEventRef
Parent Container	OsAlarmSetEvent

Description	Reference to the event that will be set by that alarm action		
Multiplicity	1		
Type	Reference to [OsEvent]		
Post-Build Variant Value	false		
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: local		

SWS Item	**ECUC_Os_00018 :**		
Name	OsAlarmSetEventTaskRef		
Parent Container	OsAlarmSetEvent		
Description	Reference to the task that will be activated by that event		
Multiplicity	1		
Type	Reference to [OsTask]		
Post-Build Variant Value	false		
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: local		

No Included Containers

10.2.3 OsAlarm

SWS Item	**ECUC_Os_00003 :**
Container Name	OsAlarm
Description	An OsAlarm may be used to asynchronously inform or activate a specific task. It is possible to start alarms automatically at system start-up depending on the application mode.
Configuration Parameters	

SWS Item	**ECUC_Os_00004 :**		
Name	OsAlarmAccessingApplication		
Parent Container	OsAlarm		
Description	Reference to applications which have an access to this object.		
Multiplicity	0..*		
Type	Reference to [OsApplication]		
Post-Build Variant Multiplicity	false		
Post-Build Variant Value	false		
Multiplicity Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency			

SWS Item	**ECUC_Os_00005 :**
Name	OsAlarmCounterRef

Parent Container	OsAlarm		
Description	Reference to the assigned counter for that alarm		
Multiplicity	1		
Type	Reference to [OsCounter]		
Post-Build Variant Value	false		
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: local		

Included Containers		
Container Name	***Multiplicity***	***Scope / Dependency***
OsAlarmAction	1	This container defines which type of notification is used when the alarm expires.
OsAlarmAutostart	0..1	If present this container defines if an alarm is started automatically at system start-up depending on the application mode.

10.2.4 OsAlarmAction

SWS Item	**ECUC_Os_00006 :**
Choice container Name	OsAlarmAction
Description	This container defines which type of notification is used when the alarm expires.

Container Choices		
Container Name	***Multiplicity***	***Scope / Dependency***
OsAlarmActivateTask	0..1	This container specifies the parameters to activate a task.
OsAlarmCallback	0..1	This container specifies the parameters to call a callback OS alarm action.
OsAlarmIncrementCounter	0..1	This container specifies the parameters to increment a counter.
OsAlarmSetEvent	0..1	This container specifies the parameters to set an event

10.2.5 OsAlarmActivateTask

SWS Item	**ECUC_Os_00007 :**
Container Name	OsAlarmActivateTask
Description	This container specifies the parameters to activate a task.
Configuration Parameters	

SWS Item	**ECUC_Os_00008 :**		
Name	OsAlarmActivateTaskRef		
Parent Container	OsAlarmActivateTask		
Description	Reference to the task that will be activated by that alarm action		
Multiplicity	1		
Type	Reference to [OsTask]		
Post-Build Variant Value	false		
Value Configuration Class	***Pre-compile time***	X	All Variants

	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: local		

No Included Containers

10.2.6 OsAlarmAutostart

SWS Item	**ECUC_Os_00009 :**
Container Name	OsAlarmAutostart
Description	If present this container defines if an alarm is started automatically at system start-up depending on the application mode.
Configuration Parameters	

SWS Item	**ECUC_Os_00010 :**		
Name	OsAlarmAlarmTime		
Parent Container	OsAlarmAutostart		
Description	The relative or absolute tick value when the alarm expires for the first time. Note that for an alarm which is RELATIVE the value must be at bigger than 0.		
Multiplicity	1		
Type	EcucIntegerParamDef		
Range	0 .. 18446744073709551615		
Default value	--		
Post-Build Variant Value	false		
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: local		

SWS Item	**ECUC_Os_00011 :**		
Name	OsAlarmAutostartType		
Parent Container	OsAlarmAutostart		
Description	This specifies the type of autostart for the alarm..		
Multiplicity	1		
Type	EcucEnumerationParamDef		
Range	ABSOLUTE	The alarm is started on startup via SetAbsAlarm().	
	RELATIVE	The alarm is started on startup via SetRelAlarm().	
Post-Build Variant Value	false		
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: local		

SWS Item	**ECUC_Os_00012 :**
Name	OsAlarmCycleTime
Parent Container	OsAlarmAutostart

Description	Cycle time of a cyclic alarm in ticks. If the value is 0 than the alarm is not cyclic.		
Multiplicity	1		
Type	EcucIntegerParamDef		
Range	0 .. 18446744073709551615		
Default value	--		
Post-Build Variant Value	false		
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: local		

SWS Item	**ECUC_Os_00013 :**		
Name	OsAlarmAppModeRef		
Parent Container	OsAlarmAutostart		
Description	Reference to the application modes for which the AUTOSTART shall be performed		
Multiplicity	1..*		
Type	Reference to [OsAppMode]		
Post-Build Variant Multiplicity	false		
Post-Build Variant Value	false		
Multiplicity Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: local		

No Included Containers

10.2.7 OsAlarmCallback

SWS Item	**ECUC_Os_00014 :**
Container Name	OsAlarmCallback
Description	This container specifies the parameters to call a callback OS alarm action.
Configuration Parameters	

SWS Item	**ECUC_Os_00087 :**		
Name	OsAlarmCallbackName		
Parent Container	OsAlarmCallback		
Description	Name of the function that is called when this alarm callback is triggered.		
Multiplicity	1		
Type	EcucFunctionNameDef		
Default value	--		
maxLength	--		
minLength	--		
regularExpression	--		
Post-Build Variant Value	false		
Value Configuration Class	***Pre-compile time***	X	All Variants

	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: local		

No Included Containers

10.2.8 OsAlarmIncrementCounter

SWS Item	**ECUC_Os_00302 :**
Container Name	OsAlarmIncrementCounter
Description	This container specifies the parameters to increment a counter.
Configuration Parameters	

SWS Item	**ECUC_Os_00015 :**		
Name	OsAlarmIncrementCounterRef		
Parent Container	OsAlarmIncrementCounter		
Description	Reference to the counter that will be incremented by that alarm action		
Multiplicity	1		
Type	Reference to [OsCounter]		
Post-Build Variant Value	false		
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: ECU		

No Included Containers

10.2.9 OsApplication

SWS Item	**ECUC_Os_00114 :**
Container Name	OsApplication
Description	An AUTOSAR OS must be capable of supporting a collection of OS objects (tasks, interrupts, alarms, hooks etc.) that form a cohesive functional unit. This collection of objects is termed an OS-Application. All objects which belong to the same OS-Application have access to each other. Access means to allow to use these objects within API services. Access by other applications can be granted separately.
Configuration Parameters	

SWS Item	**ECUC_Os_00115 :**
Name	OsTrusted
Parent Container	OsApplication
Description	Parameter to specify if an OS-Application is trusted or not. true: OS-Application is trusted false: OS-Application is not trusted (default)
Multiplicity	1
Type	EcucBooleanParamDef

Default value	false		
Post-Build Variant Value	false		
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: ECU dependency: Required for scalability class 3 and 4.		

SWS Item	**ECUC_Os_00395 :**		
Name	OsTrustedApplicationDelayTimingViolationCall		
Parent Container	OsApplication		
Description	Parameter to specify if a timing violation which occurs within an trusted OS-Application is raised immediately of if it is delayed until the current task returns to the calling OS-Application (return of CallTrustedFunction) true: violation / call to ProtectionHook() is delayed false: timing violation cause an immediate call to the ProtectionHook().		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default value	true		
Post-Build Variant Value	false		
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: ECU		

SWS Item	**ECUC_Os_00394 :**		
Name	OsTrustedApplicationWithProtection		
Parent Container	OsApplication		
Description	Parameter to specify if a trusted OS-Application is executed with memory protection or not. true: OS-Application runs within a protected environment. This means that write access is limited. false: OS-Application has full write access (default)		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default value	false		
Post-Build Variant Value	false		
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: ECU		

SWS Item	**ECUC_Os_00231 :**		
Name	OsAppAlarmRef		
Parent Container	OsApplication		
Description	Specifies the OsAlarms that belong to the OsApplication.		
Multiplicity	0..*		
Type	Reference to [OsAlarm]		
Post-Build Variant Multiplicity	false		
Post-Build Variant Value	false		
Multiplicity Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Value Configuration Class	***Pre-compile time***	X	All Variants

	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: ECU		

SWS Item	**ECUC_Os_00234 :**		
Name	OsAppCounterRef		
Parent Container	OsApplication		
Description	References the OsCounters that belong to the OsApplication.		
Multiplicity	0..*		
Type	Reference to [OsCounter]		
Post-Build Variant Multiplicity	false		
Post-Build Variant Value	false		
Multiplicity Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: ECU		

SWS Item	**ECUC_Os_00392 :**		
Name	OsAppEcucPartitionRef		
Parent Container	OsApplication		
Description	Denotes which "EcucPartition" is implemented by this "OSApplication".		
Multiplicity	0..1		
Type	Reference to [EcucPartition]		
Post-Build Variant Multiplicity	false		
Post-Build Variant Value	false		
Multiplicity Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency			

SWS Item	**ECUC_Os_00221 :**		
Name	OsAppIsrRef		
Parent Container	OsApplication		
Description	references which OsIsrs belong to the OsApplication		
Multiplicity	0..*		
Type	Reference to [OsIsr]		
Post-Build Variant Multiplicity	false		
Post-Build Variant Value	false		
Multiplicity Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: ECU		

SWS Item	**ECUC_Os_00393 :**		
Name	OsApplicationCoreRef		
Parent Container	OsApplication		
Description	Reference to the Core Definition in the Ecuc Module where the CoreId is defined. This reference is used to describe to which Core the OsApplication is bound.		
Multiplicity	0..1		
Type	Reference to [EcucCoreDefinition]		
Post-Build Variant Multiplicity	false		
Post-Build Variant Value	false		
Multiplicity Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: local		

SWS Item	**ECUC_Os_00230 :**		
Name	OsAppScheduleTableRef		
Parent Container	OsApplication		
Description	References the OsScheduleTables that belong to the OsApplication.		
Multiplicity	0..*		
Type	Reference to [OsScheduleTable]		
Post-Build Variant Multiplicity	false		
Post-Build Variant Value	false		
Multiplicity Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: ECU		

SWS Item	**ECUC_Os_00116 :**		
Name	OsAppTaskRef		
Parent Container	OsApplication		
Description	references which OsTasks belong to the OsApplication		
Multiplicity	0..*		
Type	Reference to [OsTask]		
Post-Build Variant Multiplicity	false		
Post-Build Variant Value	false		
Multiplicity Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: ECU		

SWS Item	**ECUC_Os_00120 :**
Name	OsRestartTask

Parent Container	OsApplication		
Description	Optionally one task of an OS-Application may be defined as Restart Task. Multiplicity = 1: Restart Task is activated by the Operating System if the protection hook requests it. Multiplicity = 0: No task is automatically started after a protection error happened.		
Multiplicity	0..1		
Type	Reference to [OsTask]		
Post-Build Variant Multiplicity	false		
Post-Build Variant Value	false		
Multiplicity Configuration Class	**Pre-compile time**	X	All Variants
	Link time	--	
	Post-build time	--	
Value Configuration Class	**Pre-compile time**	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: ECU dependency: Required for scalability class 3 and 4.		

Included Containers		
Container Name	**Multiplicity**	**Scope / Dependency**
OsApplicationHooks	1	Container to structure the OS-Application-specific hooks
OsApplicationTrustedFunction	0..*	Container to structure the configuration parameters of trusted functions

10.2.10 OsApplicationHooks

SWS Item	**ECUC_Os_00020 :**
Container Name	OsApplicationHooks
Description	Container to structure the OS-Application-specific hooks
Configuration Parameters	

SWS Item	**ECUC_Os_00213 :**		
Name	OsAppErrorHook		
Parent Container	OsApplicationHooks		
Description	Select the OS-Application error hook. true: Hook is called false: Hook is not called		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default value	--		
Post-Build Variant Value	false		
Value Configuration Class	**Pre-compile time**	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: ECU dependency: Required for scalability class 3 and 4.		

SWS Item	**ECUC_Os_00125 :**
Name	OsAppShutdownHook

Parent Container	OsApplicationHooks		
Description	Select the OS-Application specific shutdown hook for the OS-Application. true: Hook is called false: Hook is not called		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default value	--		
Post-Build Variant Value	false		
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: ECU dependency: Required for scalability class 3 and 4.		

SWS Item	**ECUC_Os_00124 :**		
Name	OsAppStartupHook		
Parent Container	OsApplicationHooks		
Description	Select the OS-Application specific startup hook for the OS-Application. true: Hook is called false: Hook is not called		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default value	--		
Post-Build Variant Value	false		
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: ECU dependency: Required for scalability class 3 and 4.		

SWS Item	**ECUC_Os_00402 :**		
Name	OsMemoryMappingCodeLocationRef		
Parent Container	OsApplicationHooks		
Description	Reference to the memory mapping containing details about the section where the code is placed.		
Multiplicity	0..1		
Type	Foreign reference to [SW-ADDR-METHOD]		
Post-Build Variant Value	false		
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: ECU		

No Included Containers

10.2.11 OsApplicationTrustedFunction

SWS Item	**ECUC_Os_00021 :**
Container Name	OsApplicationTrustedFunction
Description	Container to structure the configuration parameters of trusted functions
Configuration Parameters	

SWS Item	**ECUC_Os_00254 :**		
Name	OsTrustedFunctionName		
Parent Container	OsApplicationTrustedFunction		
Description	Trusted function (as part of a trusted OS-Application) available to other OS-Applications. This also supersedes the ISO 17356-6 attribute TRUSTED in APPLICATION because the optionality of this parameter is describing that already.		
Multiplicity	1		
Type	EcucFunctionNameDef		
Default value	--		
maxLength	--		
minLength	--		
regularExpression	--		
Post-Build Variant Value	false		
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: ECU dependency: Required for scalability class 3 and 4 and in trusted OS-Applications.		

No Included Containers

10.2.12 OsAppMode

SWS Item	**ECUC_Os_00022 :**
Container Name	OsAppMode
Description	OsAppMode is the object used to define ISO 17356-3 properties for an ISO 17356-3 application mode. No standard attributes are defined for AppMode. In a CPU, at least one AppMode object has to be defined. [source: ISO 17356-6] An OsAppMode called OSDEFAULTAPPMODE must always be there for ISO 17356 compatibility.
Configuration Parameters	

No Included Containers

10.2.13 OsCounter

SWS Item	**ECUC_Os_00026 :**
Container Name	OsCounter
Description	Configuration information for the counters that belong to the OsApplication.
Configuration Parameters	

SWS Item	**ECUC_Os_00027 :**

Name	OsCounterMaxAllowedValue		
Parent Container	OsCounter		
Description	Maximum possible allowed value of the system counter in ticks.		
Multiplicity	1		
Type	EcucIntegerParamDef		
Range	1 .. 18446744073709551615		
Default value	--		
Post-Build Variant Value	false		
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: local		

SWS Item	**ECUC_Os_00028 :**		
Name	OsCounterMinCycle		
Parent Container	OsCounter		
Description	The MINCYCLE attribute specifies the minimum allowed number of counter ticks for a cyclic alarm linked to the counter.		
Multiplicity	1		
Type	EcucIntegerParamDef		
Range	1 .. 18446744073709551615		
Default value	--		
Post-Build Variant Value	false		
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: local		

SWS Item	**ECUC_Os_00029 :**		
Name	OsCounterTicksPerBase		
Parent Container	OsCounter		
Description	The TICKSPERBASE attribute specifies the number of ticks required to reach a counterspecific unit. The interpretation is implementation-specific.		
Multiplicity	1		
Type	EcucIntegerParamDef		
Range	1 .. 4294967295		
Default value	--		
Post-Build Variant Value	false		
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: local		

SWS Item	**ECUC_Os_00255 :**	
Name	OsCounterType	
Parent Container	OsCounter	
Description	This parameter contains the natural type or unit of the counter.	
Multiplicity	1	
Type	EcucEnumerationParamDef	
Range	HARDWARE	This counter is driven by some hardware e.g. a hardware timer unit.
	SOFTWARE	The counter is driven by some software which calls the IncrementCounter service.

Post-Build Variant Value	false		
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: ECU		

SWS Item	**ECUC_Os_00030 :**		
Name	OsSecondsPerTick		
Parent Container	OsCounter		
Description	Time of one counter tick in seconds.		
Multiplicity	0..1		
Type	EcucFloatParamDef		
Range	[0 .. INF]		
Default value	--		
Post-Build Variant Multiplicity	false		
Post-Build Variant Value	false		
Multiplicity Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: ECU		

SWS Item	**ECUC_Os_00031 :**		
Name	OsCounterAccessingApplication		
Parent Container	OsCounter		
Description	Reference to applications which have an access to this object.		
Multiplicity	0..*		
Type	Reference to [OsApplication]		
Post-Build Variant Multiplicity	false		
Post-Build Variant Value	false		
Multiplicity Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: local		

Included Containers		
Container Name	***Multiplicity***	***Scope / Dependency***
OsDriver	0..1	This Container contains the information who will drive the counter. This configuration is only valid if the counter has OsCounterType set to HARDWARE. If the container does not exist (multiplicity=0) the timer is managed by the OS internally (OSINTERNAL). If the container exists the OS can use the GPT interface to

		manage the timer. The user have to supply the GPT channel. If the counter is driven by some other (external to the OS) source (like a TPU for example) this must be described as a vendor specific extension.
OsTimeConstant	0..*	Allows the user to define constants which can be e.g. used to compare time values with timer tick values. A time value will be converted to a timer tick value during generation and can later on accessed via the OsConstName. The conversation is done by rounding time values to the nearest fitting tick value.

10.2.14 OsEvent

SWS Item	**ECUC_Os_00033 :**
Container Name	OsEvent
Description	Representation of OS events in the configuration context. Adopted from the ISO 17356-6 specification.
Configuration Parameters	

SWS Item	**ECUC_Os_00034 :**		
Name	OsEventMask		
Parent Container	OsEvent		
Description	If event mask would be set to AUTO in OIL, this parameter should be omitted here.		
Multiplicity	0..1		
Type	EcucIntegerParamDef		
Range	0 .. 18446744073709551615		
Default value	--		
Post-Build Variant Multiplicity	false		
Post-Build Variant Value	false		
Multiplicity Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: local		

No Included Containers

10.2.15 OsDriver

SWS Item	**ECUC_Os_00371 :**
Container Name	OsDriver
Description	This Container contains the information who will drive the counter. This configuration is only valid if the counter has OsCounterType set to HARDWARE.

	If the container does not exist (multiplicity=0) the timer is managed by the OS internally (OSINTERNAL). If the container exists the OS can use the GPT interface to manage the timer. The user have to supply the GPT channel. If the counter is driven by some other (external to the OS) source (like a TPU for example) this must be described as a vendor specific extension.
Configuration Parameters	

SWS Item	**ECUC_Os_00032 :**		
Name	OsGptChannelRef		
Parent Container	OsDriver		
Description	Reference to the GPT channel.		
Multiplicity	0..1		
Type	Symbolic name reference to [GptChannelConfiguration]		
Post-Build Variant Multiplicity	false		
Post-Build Variant Value	false		
Multiplicity Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: ECU		

No Included Containers

10.2.16 OsHooks

SWS Item	**ECUC_Os_00035 :**
Container Name	OsHooks
Description	Container to structure all hooks belonging to the OS
Configuration Parameters	

SWS Item	**ECUC_Os_00036 :**		
Name	OsErrorHook		
Parent Container	OsHooks		
Description	Error hook as defined by ISO 17356 true: Hook is called false: Hook is not called		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default value	--		
Post-Build Variant Value	false		
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: local		

SWS Item	**ECUC_Os_00037 :**
Name	OsPostTaskHook
Parent Container	OsHooks

Description	Post-task hook as defined by ISO 17356 true: Hook is called false: Hook is not called		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default value	--		
Post-Build Variant Value	false		
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: local		

SWS Item	**ECUC_Os_00038 :**		
Name	OsPreTaskHook		
Parent Container	OsHooks		
Description	Pre-task hook as defined by ISO 17356 true: Hook is called false: Hook is not called		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default value	--		
Post-Build Variant Value	false		
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: local		

SWS Item	**ECUC_Os_00214 :**		
Name	OsProtectionHook		
Parent Container	OsHooks		
Description	Switch to enable/disable the call to the (user supplied) protection hook. true: Protection hook is called on protection error false: Protection hook is not called		
Multiplicity	0..1		
Type	EcucBooleanParamDef		
Default value	--		
Post-Build Variant Multiplicity	false		
Post-Build Variant Value	false		
Multiplicity Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: ECU dependency: Required for scalability class 2,3 and 4		

SWS Item	**ECUC_Os_00039 :**
Name	OsShutdownHook
Parent Container	OsHooks
Description	Shutdown hook as defined by ISO 17356 true: Hook is called false: Hook is not called
Multiplicity	1

Type	EcucBooleanParamDef		
Default value	--		
Post-Build Variant Value	false		
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: local		

SWS Item	**ECUC_Os_00040 :**		
Name	OsStartupHook		
Parent Container	OsHooks		
Description	Startup hook as defined by ISO 17356 true: Hook is called false: Hook is not called		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default value	--		
Post-Build Variant Value	false		
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: local		

SWS Item	**ECUC_Os_00402 :**		
Name	OsMemoryMappingCodeLocationRef		
Parent Container	OsHooks		
Description	Reference to the memory mapping containing details about the section where the code is placed.		
Multiplicity	0..1		
Type	Foreign reference to [SW-ADDR-METHOD]		
Post-Build Variant Value	false		
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: ECU		

No Included Containers

10.2.17 OsIsr

SWS Item	**ECUC_Os_00041 :**
Container Name	OsIsr
Description	The OsIsr container represents an ISO 17356 interrupt service routine.
Configuration Parameters	

SWS Item	**ECUC_Os_00042 :**	
Name	OsIsrCategory	
Parent Container	OsIsr	
Description	This attribute specifies the category of this ISR.	
Multiplicity	1	
Type	EcucEnumerationParamDef	
Range	CATEGORY_1	Interrupt is of category 1

	CATEGORY_2	Interrupt is of category 2	
Post-Build Variant Value	false		
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: local		

SWS Item	**ECUC_Os_00043 :**		
Name	OsIsrResourceRef		
Parent Container	OsIsr		
Description	This reference defines the resources accessed by this ISR.		
Multiplicity	0..*		
Type	Reference to [OsResource]		
Post-Build Variant Multiplicity	false		
Post-Build Variant Value	false		
Multiplicity Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: local		

SWS Item	**ECUC_Os_00402 :**		
Name	OsMemoryMappingCodeLocationRef		
Parent Container	OsIsr		
Description	Reference to the memory mapping containing details about the section where the code is placed.		
Multiplicity	0..1		
Type	Foreign reference to [SW-ADDR-METHOD]		
Post-Build Variant Value	false		
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: ECU		

Included Containers		
Container Name	***Multiplicity***	***Scope / Dependency***
OsIsrTimingProtection	0..1	This container contains all parameters which are related to timing protection If the container exists, the timing protection is used for this interrupt. If the container does not exist, the interrupt is not supervised regarding timing violations.

10.2.18 OsIsrResourceLock

SWS Item	**ECUC_Os_00388 :**
Container Name	OsIsrResourceLock
Description	This container contains a list of times the interrupt uses resources.

Configuration Parameters

SWS Item	**ECUC_Os_00389 :**	
Name	OsIsrResourceLockBudget	
Parent Container	OsIsrResourceLock	
Description	This parameter contains the maximum time the interrupt is allowed to hold the given resource (in seconds).	
Multiplicity	1	
Type	EcucFloatParamDef	
Range	[0 .. INF]	
Default value	--	
Post-Build Variant Value	false	
Value Configuration Class	***Pre-compile time***	X All Variants
	Link time	--
	Post-build time	--
Scope / Dependency	scope: ECU dependency: Required for scalability class 2 and 4	

SWS Item	**ECUC_Os_00390 :**	
Name	OsIsrResourceLockResourceRef	
Parent Container	OsIsrResourceLock	
Description	Reference to the resource the locking time is depending on	
Multiplicity	1	
Type	Reference to [OsResource]	
Post-Build Variant Value	false	
Value Configuration Class	***Pre-compile time***	X All Variants
	Link time	--
	Post-build time	--
Scope / Dependency	scope: ECU dependency: Required for scalability class 2 and 4	

No Included Containers

10.2.19 OsIsrTimingProtection

SWS Item	**ECUC_Os_00326 :**
Container Name	OsIsrTimingProtection
Description	This container contains all parameters which are related to timing protection If the container exists, the timing protection is used for this interrupt. If the container does not exist, the interrupt is not supervised regarding timing violations.
Configuration Parameters	

SWS Item	**ECUC_Os_00229 :**
Name	OsIsrAllInterruptLockBudget
Parent Container	OsIsrTimingProtection
Description	This parameter contains the maximum time for which the ISR is allowed to lock all interrupts (via SuspendAllInterrupts() or DisableAllInterrupts()) (in seconds).
Multiplicity	0..1

Type	EcucFloatParamDef		
Range	[0 .. INF]		
Default value	--		
Post-Build Variant Multiplicity	false		
Post-Build Variant Value	false		
Multiplicity Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: ECU dependency: Required for scalability class 2 and 4		

SWS Item	**ECUC_Os_00222 :**		
Name	OsIsrExecutionBudget		
Parent Container	OsIsrTimingProtection		
Description	The parameter contains the maximum allowed execution time of the interrupt (in seconds).		
Multiplicity	0..1		
Type	EcucFloatParamDef		
Range	[0 .. INF]		
Default value	--		
Post-Build Variant Multiplicity	false		
Post-Build Variant Value	false		
Multiplicity Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: ECU dependency: Required for scalability class 2 and 4		

SWS Item	**ECUC_Os_00387 :**		
Name	OsIsrOsInterruptLockBudget		
Parent Container	OsIsrTimingProtection		
Description	This parameter contains the maximum time for which the ISR is allowed to lock all Category 2 interrupts (via SuspendOSInterrupts()) (in seconds).		
Multiplicity	0..1		
Type	EcucFloatParamDef		
Range	[0 .. INF]		
Default value	--		
Post-Build Variant Multiplicity	false		
Post-Build Variant Value	false		
Multiplicity Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: ECU		

	dependency: Required for scalability class 2 and 4

SWS Item	**ECUC_Os_00223 :**		
Name	OsIsrTimeFrame		
Parent Container	OsIsrTimingProtection		
Description	This parameter contains the minimum inter-arrival time between successive interrupts (in seconds).		
Multiplicity	0..1		
Type	EcucFloatParamDef		
Range	[0 .. INF]		
Default value	--		
Post-Build Variant Multiplicity	false		
Post-Build Variant Value	false		
Multiplicity Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: ECU dependency: Required for scalability class 2 and 4		

Included Containers		
Container Name	***Multiplicity***	***Scope / Dependency***
OsIsrResourceLock	0..*	This container contains a list of times the interrupt uses resources.

10.2.20 OsOS

SWS Item	**ECUC_Os_00044 :**
Container Name	OsOS
Description	OS is the object used to define ISO 17356-3 properties for an ISO 17356 application. Per CPU exactly one OS object has to be defined.
Configuration Parameters	

SWS Item	**ECUC_Os_01019 :**		
Name	OsNumberOfCores		
Parent Container	OsOS		
Description	Maximum number of cores that are controlled by the OS. The OS uses the value internally. It depends on the ECU HW.		
Multiplicity	0..1		
Type	EcucIntegerParamDef		
Range	1 .. 65535		
Default value	--		
Post-Build Variant Multiplicity	false		
Post-Build Variant Value	false		
Multiplicity Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	

	Post-build time	--	
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: local		

SWS Item	**ECUC_Os_00259 :**		
Name	OsScalabilityClass		
Parent Container	OsOS		
Description	A scalability class for each System Object "OS" has to be selected. In order to customize the operating system to the needs of the user and to take full advantage of the processor features the operating system can be scaled according to the scalability classes. If the scalability class is omitted this translates to the OIL AUTO mechanism.		
Multiplicity	0..1		
Type	EcucEnumerationParamDef		
Range	SC1	--	
	SC2	--	
	SC3	--	
	SC4	--	
Post-Build Variant Value	false		
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: ECU		

SWS Item	**ECUC_Os_00307 :**		
Name	OsStackMonitoring		
Parent Container	OsOS		
Description	Select stack monitoring of Tasks/Category 2 ISRs true: Stacks are monitored false: Stacks are not monitored		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default value	--		
Post-Build Variant Value	false		
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: ECU		

SWS Item	**ECUC_Os_00046 :**		
Name	OsStatus		
Parent Container	OsOS		
Description	The Status attribute specifies whether a system with standard or extended status has to be used. Automatic assignment is not supported for this attribute.		
Multiplicity	1		
Type	EcucEnumerationParamDef		
Range	EXTENDED	--	
	STANDARD	--	
Post-Build Variant Value	false		
Value	***Pre-compile time***	X	All Variants

Configuration Class	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: local		

SWS Item	ECUC_Os_00047 :		
Name	OsUseGetServiceId		
Parent Container	OsOS		
Description	As defined by ISO 17356		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default value	--		
Post-Build Variant Value	false		
Value Configuration Class	Pre-compile time	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: local		

SWS Item	ECUC_Os_00048 :		
Name	OsUseParameterAccess		
Parent Container	OsOS		
Description	As defined by ISO 17356		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default value	--		
Post-Build Variant Value	false		
Value Configuration Class	Pre-compile time	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: local		

SWS Item	ECUC_Os_00049 :		
Name	OsUseResScheduler		
Parent Container	OsOS		
Description	The OsUseResScheduler attribute defines whether the resource RES_SCHEDULER is used within the application.		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default value	true		
Post-Build Variant Value	false		
Value Configuration Class	Pre-compile time	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: local		

Included Containers		
Container Name	Multiplicity	Scope / Dependency
OsHooks	1	Container to structure all hooks belonging to the OS

10.2.21 OsPeripheralArea

SWS Item	ECUC_Os_00397 :

Container Name	OsPeripheralArea		
Description	Container to structure the configuration parameters of one peripheral area. The container short name can be used to access this area.		
Post-Build Variant Multiplicity	false		
Multiplicity Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Configuration Parameters			

SWS Item	**ECUC_Os_00400 :**		
Name	OsPeripheralAreaEndAddress		
Parent Container	OsPeripheralArea		
Description	Last valid address of a peripheral area.		
Multiplicity	1		
Type	EcucIntegerParamDef		
Range	0 .. 18446744073709551615		
Default value	--		
Post-Build Variant Value	false		
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: local		

SWS Item	**ECUC_Os_00398 :**	
Name	OsPeripheralAreaId	
Parent Container	OsPeripheralArea	
Description	Id of peripheral area.	
Multiplicity	1	
Type	EcucIntegerParamDef (Symbolic Name generated for this parameter)	
Range	0 .. 18446744073709551615	
Default value	--	
Post-Build Variant Value	false	
Scope / Dependency	scope: local	

SWS Item	**ECUC_Os_00399 :**		
Name	OsPeripheralAreaStartAddress		
Parent Container	OsPeripheralArea		
Description	First valid address of a peripheral area.		
Multiplicity	1		
Type	EcucIntegerParamDef		
Range	0 .. 18446744073709551615		
Default value	--		
Post-Build Variant Multiplicity	false		
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: local		

SWS Item	**ECUC_Os_00401 :**
Name	OsPeripheralAreaAccessingApplication

Parent Container	OsPeripheralArea
Description	Reference to application which have access to this object.
Multiplicity	0..*
Type	Reference to [OsApplication]
Post-Build Variant Multiplicity	false
Post-Build Variant Value	false
Scope / Dependency	scope: local

No Included Containers

10.2.22 OsResource

SWS Item	**ECUC_Os_00252 :**
Container Name	OsResource
Description	An OsResource object is used to co-ordinate the concurrent access by tasks and ISRs to a shared resource, e.g. the scheduler, any program sequence, memory or any hardware area.
Configuration Parameters	

SWS Item	**ECUC_Os_00050 :**		
Name	OsResourceProperty		
Parent Container	OsResource		
Description	This specifies the type of the resource.		
Multiplicity	1		
Type	EcucEnumerationParamDef		
Range	INTERNAL	The resource is an internal resource.	
	LINKED	The resource is a linked resource (a second name for a existing resource).	
	STANDARD	The resource is a standard resource.	
Post-Build Variant Value	false		
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: local		

SWS Item	**ECUC_Os_00051 :**		
Name	OsResourceAccessingApplication		
Parent Container	OsResource		
Description	Reference to applications which have an access to this object.		
Multiplicity	0..*		
Type	Reference to [OsApplication]		
Post-Build Variant Multiplicity	false		
Post-Build Variant Value	false		
Multiplicity Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	

	Post-build time	--	
Scope / Dependency	scope: local		

SWS Item	**ECUC_Os_00052 :**		
Name	OsResourceLinkedResourceRef		
Parent Container	OsResource		
Description	The link to the resource. Must be valid if OsResourceProperty is LINKED. If OsResourceProperty is not LINKED the value is ignored.		
Multiplicity	0..1		
Type	Reference to [OsResource]		
Post-Build Variant Multiplicity	false		
Post-Build Variant Value	false		
Multiplicity Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: local		

No Included Containers

10.2.23 OsScheduleTable

SWS Item	**ECUC_Os_00141 :**
Container Name	OsScheduleTable
Description	An OsScheduleTable addresses the synchronization issue by providing an encapsulation of a statically defined set of alarms that cannot be modified at runtime.
Configuration Parameters	

SWS Item	**ECUC_Os_00053 :**		
Name	OsScheduleTableDuration		
Parent Container	OsScheduleTable		
Description	This parameter defines the modulus of the schedule table (in ticks).		
Multiplicity	1		
Type	EcucIntegerParamDef		
Range	0 .. 18446744073709551615		
Default value	--		
Post-Build Variant Value	false		
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: local		

SWS Item	**ECUC_Os_00144 :**
Name	OsScheduleTableRepeating
Parent Container	OsScheduleTable
Description	true: first expiry point on the schedule table shall be processed at final expiry point delay ticks after the final expiry point is processed.

	false: the schedule table processing stops when the final expiry point is processed.		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default value	--		
Post-Build Variant Value	false		
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: ECU		

SWS Item	**ECUC_Os_00145 :**		
Name	OsScheduleTableCounterRef		
Parent Container	OsScheduleTable		
Description	This parameter contains a reference to the counter which drives the schedule table.		
Multiplicity	1		
Type	Reference to [OsCounter]		
Post-Build Variant Value	false		
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: ECU		

SWS Item	**ECUC_Os_00054 :**		
Name	OsSchTblAccessingApplication		
Parent Container	OsScheduleTable		
Description	Reference to applications which have an access to this object.		
Multiplicity	0..*		
Type	Reference to [OsApplication]		
Post-Build Variant Multiplicity	false		
Post-Build Variant Value	false		
Multiplicity Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: local		

Included Containers		
Container Name	***Multiplicity***	***Scope / Dependency***
OsScheduleTableAutostart	0..1	This container specifies if and how the schedule table is started on startup of the Operating System. The options to start a schedule table correspond to the API calls to start schedule tables during runtime.
OsScheduleTableExpiryPoint	1..*	The point on a Schedule Table at which the OS activates tasks and/or sets events
OsScheduleTableSync	0..1	This container specifies the synchronization parameters of the schedule table.

10.2.24 OsScheduleTableAutostart

SWS Item	**ECUC_Os_00335 :**
Container Name	OsScheduleTableAutostart
Description	This container specifies if and how the schedule table is started on startup of the Operating System. The options to start a schedule table correspond to the API calls to start schedule tables during runtime.
Configuration Parameters	

SWS Item	**ECUC_Os_00056 :**		
Name	OsScheduleTableAutostartType		
Parent Container	OsScheduleTableAutostart		
Description	This specifies the type of the autostart for the schedule table.		
Multiplicity	1		
Type	EcucEnumerationParamDef		
Range	ABSOLUTE	The schedule table is started during startup with the StartScheduleTableAbs() service.	
	RELATIVE	The schedule table is started during startup with the StartScheduleTableRel() service.	
	SYNCHRON	The schedule table is started during startup with the StartScheduleTableSynchron() service.	
Post-Build Variant Value	false		
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: local		

SWS Item	**ECUC_Os_00057 :**		
Name	OsScheduleTableStartValue		
Parent Container	OsScheduleTableAutostart		
Description	Absolute autostart tick value when the schedule table starts. Only used if the OsScheduleTableAutostartType is ABSOLUTE. Relative offset in ticks when the schedule table starts. Only used if the OsScheduleTableAutostartType is RELATIVE.		
Multiplicity	0..1		
Type	EcucIntegerParamDef		
Range	0 .. 18446744073709551615		
Default value	--		
Post-Build Variant Multiplicity	false		
Post-Build Variant Value	false		
Multiplicity Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: ECU		

SWS Item	**ECUC_Os_00058 :**
Name	OsScheduleTableAppModeRef

Parent Container	OsScheduleTableAutostart		
Description	Reference in which application modes the schedule table should be started during startup		
Multiplicity	1..*		
Type	Reference to [OsAppMode]		
Post-Build Variant Multiplicity	false		
Post-Build Variant Value	false		
Multiplicity Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: ECU		

No Included Containers

10.2.25 OsScheduleTableEventSetting

SWS Item	**ECUC_Os_00059 :**
Container Name	OsScheduleTableEventSetting
Description	Event that is triggered by that schedule table.
Configuration Parameters	

SWS Item	**ECUC_Os_00060 :**		
Name	OsScheduleTableSetEventRef		
Parent Container	OsScheduleTableEventSetting		
Description	Reference to event that will be set by action		
Multiplicity	1		
Type	Reference to [OsEvent]		
Post-Build Variant Value	false		
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: local		

SWS Item	**ECUC_Os_00061 :**		
Name	OsScheduleTableSetEventTaskRef		
Parent Container	OsScheduleTableEventSetting		
Description	--		
Multiplicity	1		
Type	Reference to [OsTask]		
Post-Build Variant Value	false		
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: local		

No Included Containers

10.2.26 OsScheduleTableExpiryPoint

SWS Item	**ECUC_Os_00143 :**
Container Name	OsScheduleTableExpiryPoint
Description	The point on a Schedule Table at which the OS activates tasks and/or sets events
Configuration Parameters	

SWS Item	**ECUC_Os_00062 :**		
Name	OsScheduleTblExpPointOffset		
Parent Container	OsScheduleTableExpiryPoint		
Description	The offset from zero (in ticks) at which the expiry point is to be processed.		
Multiplicity	1		
Type	EcucIntegerParamDef		
Range	0 .. 18446744073709551615		
Default value	--		
Post-Build Variant Value	false		
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency			

Included Containers		
Container Name	***Multiplicity***	***Scope / Dependency***
OsScheduleTableEventSetting	0..*	Event that is triggered by that schedule table.
OsScheduleTableTaskActivation	0..*	Task that is triggered by that schedule table.
OsScheduleTblAdjustableExpPoint	0..1	Adjustable expiry point

10.2.27 OsScheduleTableTaskActivation

SWS Item	**ECUC_Os_00066 :**
Container Name	OsScheduleTableTaskActivation
Description	Task that is triggered by that schedule table.
Configuration Parameters	

SWS Item	**ECUC_Os_00067 :**		
Name	OsScheduleTableActivateTaskRef		
Parent Container	OsScheduleTableTaskActivation		
Description	Reference to task that will be activated by action		
Multiplicity	1		
Type	Reference to [OsTask]		
Post-Build Variant Value	false		
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: ECU		

No Included Containers

10.2.28 OsScheduleTblAdjustableExpPoint

SWS Item	**ECUC_Os_00068 :**
Container Name	OsScheduleTblAdjustableExpPoint
Description	Adjustable expiry point
Configuration Parameters	

SWS Item	**ECUC_Os_00069 :**		
Name	OsScheduleTableMaxLengthen		
Parent Container	OsScheduleTblAdjustableExpPoint		
Description	The maximum positive adjustment that can be made to the expiry point offset (in ticks).		
Multiplicity	1		
Type	EcucIntegerParamDef		
Range	0 .. 18446744073709551615		
Default value	--		
Post-Build Variant Value	false		
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: local		

SWS Item	**ECUC_Os_00070 :**		
Name	OsScheduleTableMaxShorten		
Parent Container	OsScheduleTblAdjustableExpPoint		
Description	The maximum negative adjustment that can be made to the expiry point offset (in ticks).		
Multiplicity	1		
Type	EcucIntegerParamDef		
Range	0 .. 18446744073709551615		
Default value	--		
Post-Build Variant Value	false		
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: local		

No Included Containers

10.2.29 OsScheduleTableSync

SWS Item	**ECUC_Os_00063 :**
Container Name	OsScheduleTableSync
Description	This container specifies the synchronization parameters of the schedule table.
Configuration Parameters	

SWS Item	**ECUC_Os_00064 :**
Name	OsScheduleTblExplicitPrecision
Parent Container	OsScheduleTableSync
Description	This configuration is only valid if the explicit synchronization is used.

Multiplicity	0..1		
Type	EcucIntegerParamDef		
Range	0 .. 18446744073709551615		
Default value	--		
Post-Build Variant Multiplicity	false		
Post-Build Variant Value	false		
Multiplicity Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: ECU		

SWS Item	**ECUC_Os_00065 :**		
Name	OsScheduleTblSyncStrategy		
Parent Container	OsScheduleTableSync		
Description	AUTOSAR OS provides support for synchronization in two ways: explicit and implicit.		
Multiplicity	1		
Type	EcucEnumerationParamDef		
Range	EXPLICIT	The schedule table is driven by an OS counter but processing needs to be synchronized with a different counter which is not an OS counter object.	
	IMPLICIT	The counter driving the schedule table is the counter with which synchronisation is required.	
	NONE	No support for synchronisation.	
Default value	NONE		
Post-Build Variant Value	false		
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: ECU		

No Included Containers

10.2.30 OsSpinlock

SWS Item	**ECUC_Os_00258 :**
Container Name	OsSpinlock
Description	An OsSpinlock object is used to co-ordinate concurrent access by TASKs/ISR2s on different cores to a shared resource.
Configuration Parameters	

SWS Item	**ECUC_Os_01038 :**
Name	OsSpinlockLockMethod

Parent Container	OsSpinlock		
Description	Lock method which is used when a spinlock is taken. Note that it is possible that a user (e.g. a Task) might hold more than one spinlock. In this case the last lock taken is forced to use at least a lock methode which locks as strong as the current one.		
Multiplicity	1		
Type	EcucEnumerationParamDef		
Range	LOCK_ALL_INTERRUPTS	--	
	LOCK_CAT2_INTERRUPTS	--	
	LOCK_NOTHING	--	
	LOCK_WITH_RES_SCHEDULER	--	
Default value	LOCK_NOTHING		
Post-Build Variant Value	false		
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: local		

SWS Item	**ECUC_Os_01021 :**		
Name	OsSpinlockAccessingApplication		
Parent Container	OsSpinlock		
Description	Reference to OsApplications that have an access to this object.		
Multiplicity	1..*		
Type	Reference to [OsApplication]		
Post-Build Variant Multiplicity	false		
Post-Build Variant Value	false		
Multiplicity Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: local		

SWS Item	**ECUC_Os_01022 :**		
Name	OsSpinlockSuccessor		
Parent Container	OsSpinlock		
Description	Reference to OsApplications that have an access to this object. To check whether a spinlock can be occupied (in a nested way) without any danger of deadlock, a linked list of spinlocks can be defined. A spinlock can only be occupied in the order of the linked list. It is allowed to skip a spinlock. If no linked list is specified, spinlocks cannot be nested.		
Multiplicity	0..1		
Type	Reference to [OsSpinlock]		
Post-Build Variant Multiplicity	false		
Post-Build Variant Value	false		
Multiplicity Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Value Configuration Class	***Pre-compile time***	X	All Variants

	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: local		

No Included Containers

10.2.31 OsTask

SWS Item	**ECUC_Os_00073 :**
Container Name	OsTask
Description	This container represents an ISO 17356 task.
Configuration Parameters	

SWS Item	**ECUC_Os_00074 :**		
Name	OsTaskActivation		
Parent Container	OsTask		
Description	This attribute defines the maximum number of queued activation requests for the task. A value equal to "1" means that at any time only a single activation is permitted for this task. Note that the value must be a natural number starting at 1.		
Multiplicity	1		
Type	EcucIntegerParamDef		
Range	1 .. 4294967295		
Default value	--		
Post-Build Variant Value	false		
Value Configuration Class	*Pre-compile time*	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: local		

SWS Item	**ECUC_Os_00075 :**		
Name	OsTaskPriority		
Parent Container	OsTask		
Description	The priority of a task is defined by the value of this attribute. This value has to be understood as a relative value, i.e. the values show only the relative ordering of the tasks. ISO 17356-3 defines the lowest priority as zero (0); larger values correspond to higher priorities.		
Multiplicity	1		
Type	EcucIntegerParamDef		
Range	0 .. 4294967295		
Default value	--		
Post-Build Variant Value	false		
Value Configuration Class	*Pre-compile time*	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: local		

SWS Item	**ECUC_Os_00076 :**
Name	OsTaskSchedule
Parent Container	OsTask
Description	The OsTaskSchedule attribute defines the preemptability of the task.

	If this attribute is set to NON, no internal resources may be assigned to this task.		
Multiplicity	1		
Type	EcucEnumerationParamDef		
Range	FULL	Task is preemptable.	
	NON	Task is not preemptable.	
Post-Build Variant Value	false		
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: local		

SWS Item	**ECUC_Os_00402 :**		
Name	OsMemoryMappingCodeLocationRef		
Parent Container	OsTask		
Description	Reference to the memory mapping containing details about the section where the code is placed.		
Multiplicity	0..1		
Type	Foreign reference to [SW-ADDR-METHOD]		
Post-Build Variant Value	false		
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: ECU		

SWS Item	**ECUC_Os_00077 :**		
Name	OsTaskAccessingApplication		
Parent Container	OsTask		
Description	Reference to applications which have an access to this object.		
Multiplicity	0..*		
Type	Reference to [OsApplication]		
Post-Build Variant Multiplicity	false		
Post-Build Variant Value	false		
Multiplicity Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: local		

SWS Item	**ECUC_Os_00078 :**		
Name	OsTaskEventRef		
Parent Container	OsTask		
Description	This reference defines the list of events the extended task may react on.		
Multiplicity	0..*		
Type	Reference to [OsEvent]		
Post-Build Variant Multiplicity	false		
Post-Build Variant Value	false		
Multiplicity Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	

Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: local		

SWS Item	**ECUC_Os_00079 :**		
Name	OsTaskResourceRef		
Parent Container	OsTask		
Description	This reference defines a list of resources accessed by this task.		
Multiplicity	0..*		
Type	Reference to [OsResource]		
Post-Build Variant Multiplicity	false		
Post-Build Variant Value	false		
Multiplicity Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: local		

Included Containers		
Container Name	***Multiplicity***	***Scope / Dependency***
OsTaskAutostart	0..1	This container determines whether the task is activated during the system start-up procedure or not for some specific application modes. If the task shall be activated during the system start-up, this container is present and holds the references to the application modes in which the task is auto-started.
OsTaskTimingProtection	0..1	This container contains all parameters regarding timing protection of the task.

10.2.32 OsTaskAutostart

SWS Item	**ECUC_Os_00080 :**
Container Name	OsTaskAutostart
Description	This container determines whether the task is activated during the system start-up procedure or not for some specific application modes. If the task shall be activated during the system start-up, this container is present and holds the references to the application modes in which the task is auto-started.
Configuration Parameters	

SWS Item	**ECUC_Os_00081 :**
Name	OsTaskAppModeRef
Parent Container	OsTaskAutostart
Description	Reference to application modes in which that task is activated on startup of the OS
Multiplicity	1..*
Type	Reference to [OsAppMode]
Post-Build Variant	false

Multiplicity			
Post-Build Variant Value	false		
Multiplicity Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency			

No Included Containers

10.2.33 OsTaskResourceLock

SWS Item	**ECUC_Os_00082 :**
Container Name	OsTaskResourceLock
Description	This container contains the worst case time between getting and releasing a given resource (in seconds).
Configuration Parameters	

SWS Item	**ECUC_Os_00083 :**		
Name	OsTaskResourceLockBudget		
Parent Container	OsTaskResourceLock		
Description	This parameter contains the maximum time the task is allowed to lock the resource (in seconds)		
Multiplicity	1		
Type	EcucFloatParamDef		
Range	[0 .. INF]		
Default value	--		
Post-Build Variant Value	false		
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: ECU dependency: Required for scalability class 2 and 4		

SWS Item	**ECUC_Os_00084 :**		
Name	OsTaskResourceLockResourceRef		
Parent Container	OsTaskResourceLock		
Description	Reference to the resource used by the task		
Multiplicity	1		
Type	Reference to [OsResource]		
Post-Build Variant Value	false		
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: ECU dependency: Required for scalability class 2 and 4		

No Included Containers

10.2.34 OsTaskTimingProtection

SWS Item	**ECUC_Os_00325 :**		
Container Name	OsTaskTimingProtection		
Description	This container contains all parameters regarding timing protection of the task.		
Configuration Parameters			

SWS Item	**ECUC_Os_00085 :**		
Name	OsTaskAllInterruptLockBudget		
Parent Container	OsTaskTimingProtection		
Description	This parameter contains the maximum time for which the task is allowed to lock all interrupts (via SuspendAllInterrupts() or DisableAllInterrupts()) (in seconds).		
Multiplicity	0..1		
Type	EcucFloatParamDef		
Range	[0 .. INF]		
Default value	--		
Post-Build Variant Multiplicity	false		
Post-Build Variant Value	false		
Multiplicity Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: ECU dependency: Required for scalability class 2 and 4		

SWS Item	**ECUC_Os_00185 :**		
Name	OsTaskExecutionBudget		
Parent Container	OsTaskTimingProtection		
Description	This parameter contains the maximum allowed execution time of the task (in seconds).		
Multiplicity	0..1		
Type	EcucFloatParamDef		
Range	[0 .. INF]		
Default value	--		
Post-Build Variant Multiplicity	false		
Post-Build Variant Value	false		
Multiplicity Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: ECU dependency: Required for scalability class 2 and 4		

SWS Item	**ECUC_Os_00086 :**
Name	OsTaskOsInterruptLockBudget
Parent Container	OsTaskTimingProtection
Description	This parameter contains the maximum time for which the task is allowed to

	lock all Category 2 interrupts (via SuspendOSInterrupts()) (in seconds).		
Multiplicity	0..1		
Type	EcucFloatParamDef		
Range	[0 .. INF]		
Default value	--		
Post-Build Variant Multiplicity	false		
Post-Build Variant Value	false		
Multiplicity Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: ECU dependency: Required for scalability class 2 and 4		

SWS Item	**ECUC_Os_00391 :**		
Name	OsTaskTimeFrame		
Parent Container	OsTaskTimingProtection		
Description	The minimum inter-arrival time between activations and/or releases of a task (in seconds).		
Multiplicity	0..1		
Type	EcucFloatParamDef		
Range	[0 .. INF]		
Default value	--		
Post-Build Variant Multiplicity	false		
Post-Build Variant Value	false		
Multiplicity Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: ECU dependency: Only available in scalability class 2 and 4		

Included Containers		
Container Name	***Multiplicity***	***Scope / Dependency***
OsTaskResourceLock	0..*	This container contains the worst case time between getting and releasing a given resource (in seconds).

10.2.35 OsTimeConstant

SWS Item	**ECUC_Os_00386 :**
Container Name	OsTimeConstant
Description	Allows the user to define constants which can be e.g. used to compare time values with timer tick values. A time value will be converted to a timer tick value during generation and can later on accessed via the OsConstName. The conversation is done by rounding time values to the nearest fitting tick value.
Configuration Parameters	

SWS Item	**ECUC_Os_00002 :**		
Name	OsTimeValue		
Parent Container	OsTimeConstant		
Description	This parameter contains the value of the constant in seconds.		
Multiplicity	1		
Type	EcucFloatParamDef		
Range	[0 .. INF]		
Default value	--		
Post-Build Variant Value	false		
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: ECU		

No Included Containers

10.3 Containers and configuration parameter extensions of the IOC

This section describes the content of the IOC Configuration Description that is needed for the generation of the IOC API.

10.3.1 OsIoc

SWS Item	**ECUC_Os_01000 :**
Container Name	OsIoc
Description	Configuration of the IOC (Inter OS Application Communicator).
Configuration Parameters	

Included Containers		
Container Name	***Multiplicity***	***Scope / Dependency***
OsIocCommunication	0..*	Representation of a 1:1 or N:1 or N:M (unqueued only) communication between software parts located in different OS-Applications that are bound to the same or to different cores. The name shall begin with the name of the sending software service and be followed by a unique identifier delivered by the sending software service. In the case of RTE as user attention shall be paid on the fact that uniqueness for identifier names has to be reached over ports, data elements, object instances and maybe additional identification properties (E.g. Case 1:N mapping to 1:1). Example: ▪ <NameSpace>_UniqueID

10.3.2 OsIocCommunication

SWS Item	**ECUC_Os_01003 :**
Container Name	OsIocCommunication
Description	Representation of a 1:1 or N:1 or N:M (unqueued only) communication between software parts located in different OS-Applications that are bound to the same or to different cores. The name shall begin with the name of the sending software service and be followed by a unique identifier delivered by the sending software service. In the case of RTE as user attention shall be paid on the fact that uniqueness for identifier names has to be reached over ports, data elements, object instances and maybe additional identification properties (E.g. Case 1:N mapping to 1:1). Example: ▪ <NameSpace>_UniqueID
Configuration Parameters	

SWS Item	**ECUC_Os_01001 :**		
Name	OsIocBufferLength		
Parent Container	OsIocCommunication		
Description	This attribute defines the size of the IOC internal queue to be allocated for a queued communication. This configuration information shall allow the optimization of the needed memory for communications requiring buffers within the RTE and within the IOC.		
Multiplicity	0..1		
Type	EcucIntegerParamDef		
Range	0 .. 4294967295		
Default value	--		
Post-Build Variant Multiplicity	false		
Post-Build Variant Value	false		
Multiplicity Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: local		

Included Containers		
Container Name	***Multiplicity***	***Scope / Dependency***
OsIocDataProperties	1..*	Data properties of the data to be transferred on the IOC communication channel.
OsIocReceiverProperties	1..*	Representation of receiver properties for one communication. For each OsIocCommunication one (1:1) or many receivers (N:M) have to be defined. This container should be instantiated within an OsIocCommunication.
OsIocSenderProperties	1..*	Representation of sender properties for one communication. For each OsIocCommunication one (1:1) or many senders (N:1 or N:M) have to be defined. Multiplicity > 1 (N:1 or N:M

		communication) is only allowed for Multiplicity of OsIocDataTypeRef = 1. This container should be instantiated within an OsIocCommunication.

10.3.3 OsIocSenderProperties

SWS Item	**ECUC_Os_01015 :**
Container Name	OsIocSenderProperties
Description	Representation of sender properties for one communication. For each OsIocCommunication one (1:1) or many senders (N:1 or N:M) have to be defined. Multiplicity > 1 (N:1 or N:M communication) is only allowed for Multiplicity of OsIocDataTypeRef = 1. This container should be instantiated within an OsIocCommunication.
Configuration Parameters	

SWS Item	**ECUC_Os_01036 :**		
Name	OsIocFunctionImplementationKind		
Parent Container	OsIocSenderProperties		
Description	This parameter is used to select whether this communication is implemented as a macro or as a function.		
Multiplicity	0..1		
Type	EcucEnumerationParamDef		
Range	DO_NOT_CARE	It is not defined whether a macro or a function is used.	
	FUNCTION	Communication is implemented as a function	
	MACRO	Communication is implemented as a macro	
Default value	DO_NOT_CARE		
Post-Build Variant Multiplicity	false		
Post-Build Variant Value	false		
Multiplicity Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: local		

SWS Item	**ECUC_Os_01016 :**	
Name	OsIocSenderId	
Parent Container	OsIocSenderProperties	
Description	Representation of a sender in a N:1 or N:M communication to distinguish between senders. This parameter does not exist in 1:1 communication.	
Multiplicity	0..1	
Type	EcucIntegerParamDef	
Range	0 .. 255	
Default value	--	

Post-Build Variant Multiplicity	false		
Post-Build Variant Value	false		
Multiplicity Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: ECU		

SWS Item	**ECUC_Os_01014 :**		
Name	OsIocSendingOsApplicationRef		
Parent Container	OsIocSenderProperties		
Description	This attribute is a reference to the sending OS-Application instance defined in the configuration file of the OS. This information shall allows the generator to get additional information necessary for the code generation like: • The protection properties of the communicating OS-Applications to find out which protection boundaries have to be crossed. • The core identifiers to find out if an intra or an inter core communication has to be realized • Interrupt details in case of cross core notification to realize over IRQs		
Multiplicity	1		
Type	Reference to [OsApplication]		
Post-Build Variant Value	false		
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: local		

No Included Containers

10.3.4 OsIocReceiverProperties

SWS Item	**ECUC_Os_01017 :**
Container Name	OsIocReceiverProperties
Description	Representation of receiver properties for one communication. For each OsIocCommunication one (1:1) or many receivers (N:M) have to be defined. This container should be instantiated within an OsIocCommunication.
Configuration Parameters	

SWS Item	**ECUC_Os_01037 :**
Name	OsIocFunctionImplementationKind
Parent Container	OsIocReceiverProperties
Description	This parameter is used to select whether this communication is implemented as a macro or as a function.
Multiplicity	0..1
Type	EcucEnumerationParamDef

Range	DO_NOT_CARE		It is not defined whether a macro or a function is used.
	FUNCTION		Communication is implemented as a function
	MACRO		Communication is implemented as a macro
Default value	DO_NOT_CARE		
Post-Build Variant Multiplicity	false		
Post-Build Variant Value	false		
Multiplicity Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: local		

SWS Item	**ECUC_Os_01010 :**		
Name	OsIocReceiverPullCB		
Parent Container	OsIocReceiverProperties		
Description	This attribute defines the name of a callback function that the IOC shall call on the receiving core for each data reception. In case of non existence of this attribute no ReceiverPullCB notification shall be applied by the IOC. The name of the function shall begin with the name of the receiving module, followed with a callback name and followed by the IocId. Example: void RTE_ReceiverPullCB_RTE25 (void). If this attribute does not exist, it means that no ReceiverPullCB shall be called (No notification from IOC is required). If this attribute exists the IOC shall call the callback function on the receiving core.		
Multiplicity	0..1		
Type	EcucFunctionNameDef		
Default value	--		
maxLength	--		
minLength	--		
regularExpression	--		
Post-Build Variant Multiplicity	false		
Post-Build Variant Value	false		
Multiplicity Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: local		

SWS Item	**ECUC_Os_01012 :**
Name	OsIocReceivingOsApplicationRef
Parent Container	OsIocReceiverProperties
Description	This attribute is a reference to the receiving OsApplication instance defined

	in the configuration file of the OS. This information allows for the generator to get additional information necessary for the code generation like: • The protection properties of the communicating OsApplications to find out which protections have to be crossed • The core identifiers to find out if an intra or an inter core communication has to be realized • Interrupt details in case of cross core notification to realize over IRQs		
Multiplicity	1		
Type	Reference to [OsApplication]		
Post-Build Variant Value	false		
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: local		

No Included Containers

10.3.5 OsIocDataProperties

SWS Item	**ECUC_Os_01023 :**
Container Name	OsIocDataProperties
Description	Data properties of the data to be transferred on the IOC communication channel.
Configuration Parameters	

SWS Item	**ECUC_Os_01035 :**		
Name	OsIocDataPropertyIndex		
Parent Container	OsIocDataProperties		
Description	This parameter is used to define in which order the data is send, e.g. whether IocSendGroup(A,B) or IocSendGroup(B,A) shall be used.		
Multiplicity	0..1		
Type	EcucIntegerParamDef		
Range	0 .. 255		
Default value	--		
Post-Build Variant Multiplicity	false		
Post-Build Variant Value	false		
Multiplicity Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: local		

SWS Item	**ECUC_Os_01024 :**
Name	OsIocInitValue
Parent Container	OsIocDataProperties
Description	Initial Value for the data to be transferred on the IOC communication

	channel.		
Multiplicity	0..1		
Type	EcucStringParamDef		
Default value	--		
maxLength	--		
minLength	--		
regularExpression	--		
Post-Build Variant Multiplicity	false		
Post-Build Variant Value	false		
Multiplicity Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: local		

SWS Item	**ECUC_Os_01005 :**		
Name	OsIocDataTypeRef		
Parent Container	OsIocDataProperties		
Description	This is the type of the data to be transferred on the IOC communication channel. This attribute is necessary to generate the parameter type of the Ioc functions. Additionally this information should be used to compute the data size for necessary data copy operations within the Ioc module. If more than one attribute is defined, the IOC generator should generate an IocXxxGroup function (Xxx= CHOICE [Send, Receive, Write, Read]). N:1 or N:M communication (Multiplicity of OsIocSenderProperties > 1) is only allowed for multiplicity of OsIocDataTypeRef = 1		
Multiplicity	1		
Type	Foreign reference to [IMPLEMENTATION-DATA-TYPE]		
Post-Build Variant Value	false		
Value Configuration Class	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: local		

No Included Containers

10.4 Containers and configuration parameters for ARTI (Draft)

This section describes the structure (containers) and the parameters of ARTI objects related to the OS configuration. ARTI objects are defined by the MOD_ARTI model. For a detailed description of the referenced ARTI parameters, please see chapter 10 of SWS_ClassicPlatformARTI.

10.4.1 ArtiHardware

SWS Item	**ARTI_ArtiHardware_xxxx**
Container Name:	ArtiHardware

Description:	The “ArtiHardware” container contains ARTI extensions to the EcucHardware module.
Post-Build Variant:	False
Multiplicity:	
Included Containers	

Included Containers		
Container Name	***Multiplicity***	***Scope/Dependency***
ArtiHwCoreClass	0..1	Contains the layout of an ARTI "Core" object, extending the EcucCoreDefinition.
ArtiHwCoreInstance	1..*	Represents an instance of an ARTI "Core" object, extending the EcucCoreDefinition.

Example:
Exemplary values of the ArtiHardware container:
```
<ECUC-MODULE-CONFIGURATION-VALUES>
  <SHORT-NAME>Vendor1ArtiHardware</SHORT-NAME>
  <DEFINITION-REF DEST="ECUC-MODULE-DEF">
      /AUTOSAR/ArtiDefs/ArtiHardware</DEFINITION-REF>
  <CONTAINERS>
    <ECUC-CONTAINER-VALUE>
      <SHORT-NAME>ArtiCoreClass</SHORT-NAME>
      <DEFINITION-REF DEST="ECUC-PARAM-CONF-CONTAINER-DEF">
          /AUTOSAR/ArtiDefs/ArtiHardware/ArtiHwCoreClass</DEFINITION-REF>
      <...>
    </ECUC-CONTAINER-VALUE>
    <ECUC-CONTAINER-VALUE>
      <SHORT-NAME>ArtiCore0</SHORT-NAME>
      <DEFINITION-REF DEST="ECUC-PARAM-CONF-CONTAINER-DEF">
       /AUTOSAR/ArtiDefs/ArtiHardware/ArtiHwCoreInstance</DEFINITION-REF>
      <...>
    </ECUC-CONTAINER-VALUE>
    <ECUC-CONTAINER-VALUE>
      <SHORT-NAME>ArtiCore1</SHORT-NAME>
      <DEFINITION-REF DEST="ECUC-PARAM-CONF-CONTAINER-DEF">
       /AUTOSAR/ArtiDefs/ArtiHardware/ArtiHwCoreInstance</DEFINITION-REF>
      <...>
    </ECUC-CONTAINER-VALUE>
  </CONTAINERS>
</ECUC-MODULE-CONFIGURATION-VALUES>
```

10.4.2 ArtiHwCoreClass

SWS Item	**ARTI_ArtiHardware_xxxx**
Container Name:	ArtiHwCoreClass
Description:	Contains the layout of an ARTI "Core" object, extending the EcucCoreDefinition.
Post-Build Variant:	False
Multiplicity:	
Configuration Parameters	

Name:	ArtiHwCoreClassDescription
Description:	Description of the "Core" class.
Type	EcucStringParamDef
Post-Build Variant Multiplicity:	False
Post-Build Variant Value:	False
Multiplicity Configuration Class:	Pre-Compile
Value Configuration Class:	Pre-Compile
Scope	ECU

Name:	ArtiCurrentApplicationClassRef
Description:	Refers to the ArtiObjectClassParameter that defines the ArtiCurrentApplicationInstance parameter.
Type	Reference to ArtiObjectClassParameter
Post-Build Variant Multiplicity:	False
Post-Build Variant Value:	False
Multiplicity Configuration Class:	Pre-Compile
Value Configuration Class:	Pre-Compile
Scope	ECU

Name:	ArtiCurrentTaskClassRef
Description:	Refers to the ArtiObjectClassParameter that defines the ArtiCurrentTaskInstance parameter.
Type	Reference to ArtiObjectClassParameter
Post-Build Variant Multiplicity:	False
Post-Build Variant Value:	False
Multiplicity Configuration Class:	Pre-Compile
Value Configuration Class:	Pre-Compile
Scope	ECU

Name:	ArtiLastErrorClassRef
Description:	Refers to the ArtiObjectClassParameter that defines the ArtiLastErrorInstance parameter.
Type	Reference to ArtiObjectClassParameter
Post-Build Variant Multiplicity:	False
Post-Build Variant Value:	False
Multiplicity Configuration Class:	Pre-Compile
Value Configuration Class:	Pre-Compile

Scope	ECU

Example:
Exemplary value of an ArtiHwCoreClass container:

```
<ECUC-CONTAINER-VALUE>
  <SHORT-NAME>ArtiCoreClass</SHORT-NAME>
  <DEFINITION-REF DEST="ECUC-PARAM-CONF-CONTAINER-DEF">/AUTOSAR/ArtiDefs/
      ArtiHardware/ArtiHwCoreClass</DEFINITION-REF>
  <REFERENCE-VALUES>
    <ECUC-REFERENCE-VALUE>
      <DEFINITION-REF DEST="ECUC-REFERENCE-DEF">/AUTOSAR/ArtiDefs/
          ArtiHardware/ArtiHwCoreClass/ArtiCurrentApplicationClassRef
          </DEFINITION-REF>
      <VALUE-REF DEST="ECUC-CONTAINER-VALUE">/Vendor1/Vendor1Arti/
          ArtiObjectClassParameter_ArtiHwCore_CurrentApplication
          </VALUE-REF>
    </ECUC-REFERENCE-VALUE>
    <ECUC-REFERENCE-VALUE>
      <DEFINITION-REF DEST="ECUC-REFERENCE-DEF">/AUTOSAR/ArtiDefs/
          ArtiHardware/ArtiHwCoreClass/ArtiCurrentTaskClassRef
          </DEFINITION-REF>
      <VALUE-REF DEST="ECUC-CONTAINER-VALUE">/Vendor1/Vendor1Arti/
          ArtiObjectClassParameter_ArtiHwCore_CurrentTask</VALUE-REF>
    </ECUC-REFERENCE-VALUE>
  </REFERENCE-VALUES>
</ECUC-CONTAINER-VALUE>
```

10.4.3 ArtiHwCoreInstance

SWS Item	**ARTI_ArtiHardware_xxxx**
Container Name:	ArtiHwCoreInstance
Description:	Represents an instance of an ARTI "Core" object, extending the EcucCoreDefinition.
Post-Build Variant:	False
Multiplicity:	
Configuration Parameters	

Name:	ArtiHwCoreInstanceDescription
Description:	Description of the "Core" instance.
Type	EcucStringParamDef
Post-Build Variant Multiplicity:	False
Post-Build Variant Value:	False
Multiplicity Configuration Class:	Pre-Compile
Value Configuration Class:	Pre-Compile
Scope	ECU

Name:	ArtiCurrentApplicationInstanceRef
Description:	Refers to the ArtiObjectInstanceParameter that

	contains the evaluation for the "current application" that is running on this core.
Type	Reference to ArtiObjectInstanceParameter
Post-Build Variant Multiplicity:	False
Post-Build Variant Value:	False
Multiplicity Configuration Class:	Pre-Compile
Value Configuration Class:	Pre-Compile
Scope	ECU

Name:	ArtiCurrentTaskInstanceRef
Description:	Refers to the ArtiObjectInstanceParameter that contains the evaluation for the "current task" that is running on this core.
Type	Reference to ArtiObjectInstanceParameter
Post-Build Variant Multiplicity:	False
Post-Build Variant Value:	False
Multiplicity Configuration Class:	Pre-Compile
Value Configuration Class:	Pre-Compile
Scope	ECU

Name:	ArtiLastErrorInstanceRef
Description:	Refers to the ArtiObjectInstanceParameter that contains the evaluation for the "last error" that happened on this core.
Type	Reference to ArtiObjectInstanceParameter
Post-Build Variant Multiplicity:	False
Post-Build Variant Value:	False
Multiplicity Configuration Class:	Pre-Compile
Value Configuration Class:	Pre-Compile
Scope	ECU

Name:	ArtiEcucCoreRef
Description:	Refers to the EcucCoreDefinition of this core.
Type	Reference to EcucCoreDefinition
Post-Build Variant Multiplicity:	False
Post-Build Variant Value:	False
Multiplicity Configuration Class:	Pre-Compile
Value Configuration Class:	Pre-Compile
Scope	ECU

Example:
Exemplary value of an ArtiHwCoreInstance container:

```
<ECUC-CONTAINER-VALUE>
  <SHORT-NAME>ArtiCore0</SHORT-NAME>
  <DEFINITION-REF DEST="ECUC-PARAM-CONF-CONTAINER-DEF">/AUTOSAR/ArtiDefs/
      ArtiHardware/ArtiHwCoreInstance</DEFINITION-REF>
  <REFERENCE-VALUES>
    <ECUC-REFERENCE-VALUE>
      <DEFINITION-REF DEST="ECUC-REFERENCE-DEF">/AUTOSAR/ArtiDefs/
          ArtiHardware/ArtiHwCoreInstance/
          ArtiCurrentApplicationInstanceRef</DEFINITION-REF>
      <VALUE-REF DEST="ECUC-CONTAINER-VALUE">/Vendor1/Vendor1Arti/
          ArtiObjectInstanceParameter_CurrentApplicationOnCore0
          </VALUE-REF>
    </ECUC-REFERENCE-VALUE>
    <ECUC-REFERENCE-VALUE>
      <DEFINITION-REF DEST="ECUC-REFERENCE-DEF">/AUTOSAR/ArtiDefs/
          ArtiHardware/ArtiHwCoreInstance/
          ArtiCurrentTaskInstanceRef</DEFINITION-REF>
      <VALUE-REF DEST="ECUC-CONTAINER-VALUE">/Vendor1/Vendor1Arti/
          ArtiObjectInstanceParameter_CurrentTaskOnCore0</VALUE-REF>
    </ECUC-REFERENCE-VALUE>
    <ECUC-REFERENCE-VALUE>
      <DEFINITION-REF DEST="ECUC-REFERENCE-DEF">/AUTOSAR/ArtiDefs/
          ArtiHardware/ArtiHwCoreInstance/ArtiEcucCoreRef
          </DEFINITION-REF>
      <VALUE-REF DEST="ECUC-CONTAINER-VALUE">/Vendor1/Vendor1EcucEcuC/
          Hardware/Core0</VALUE-REF>
    </ECUC-REFERENCE-VALUE>
  </REFERENCE-VALUES>
</ECUC-CONTAINER-VALUE>
```

10.4.4 ArtiOs

SWS Item	**ARTI_ArtiOs_xxxx**
Container Name:	ArtiOs
Description:	The “ArtiOs” container contains ARTI extensions to the EcucDefs/Os module.
Post-Build Variant:	False
Multiplicity:	
Included Containers	

Included Containers		
Container Name	***Multiplicity***	***Scope/Dependency***
ArtiOsClass	0..1	Contains the layout of an ARTI "Os" object, extending the EcuC OsOS.
ArtiOsInstance	0..*	Represents an instance of an ARTI "Os" object, extending the EcuC OsOS.
ArtiOsTaskClass	0..1	Contains the layout of an ARTI "OsTask" object, extending the EcuC OsTask.

ArtiOsTaskInstance	0..*	Represents an instance of an ARTI "OsTask" object, extending the EcuC OsTask.

Example:
Exemplary values of the ArtiOs container:
```
<ECUC-MODULE-CONFIGURATION-VALUES>
  <SHORT-NAME>Vendor1ArtiOs</SHORT-NAME>
  <DEFINITION-REF DEST="ECUC-MODULE-DEF">
      /AUTOSAR/ArtiDefs/ArtiOs</DEFINITION-REF>
  <CONTAINERS>
    <ECUC-CONTAINER-VALUE>
      <SHORT-NAME>ArtiOsClass_Conf</SHORT-NAME>
      <DEFINITION-REF DEST="ECUC-PARAM-CONF-CONTAINER-DEF">
          /AUTOSAR/ArtiDefs/ArtiOs/ArtiOsClass</DEFINITION-REF>
      <...>
    </ECUC-CONTAINER-VALUE>
    <ECUC-CONTAINER-VALUE>
      <SHORT-NAME>ArtiOsInstance_Conf</SHORT-NAME>
      <DEFINITION-REF DEST="ECUC-PARAM-CONF-CONTAINER-DEF">
          /AUTOSAR/ArtiDefs/ArtiOs/ArtiOsInstance</DEFINITION-REF>
      <...>
    </ECUC-CONTAINER-VALUE>
    <ECUC-CONTAINER-VALUE>
      <SHORT-NAME>ArtiOsTaskClass_Conf</SHORT-NAME>
      <DEFINITION-REF DEST="ECUC-PARAM-CONF-CONTAINER-DEF">
          /AUTOSAR/ArtiDefs/ArtiOs/ArtiOsTaskClass</DEFINITION-REF>
    </ECUC-CONTAINER-VALUE>
    <ECUC-CONTAINER-VALUE>
      <SHORT-NAME>ArtiOsTaskInstance_TaskHighPriority</SHORT-NAME>
      <DEFINITION-REF DEST="ECUC-PARAM-CONF-CONTAINER-DEF">
          /AUTOSAR/ArtiDefs/ArtiOs/ArtiOsTaskInstance</DEFINITION-REF>
      <...>
    </ECUC-CONTAINER-VALUE>
    <ECUC-CONTAINER-VALUE>
      <SHORT-NAME>ArtiOsTaskInstance_TaskLowPriority</SHORT-NAME>
      <DEFINITION-REF DEST="ECUC-PARAM-CONF-CONTAINER-DEF">
          /AUTOSAR/ArtiDefs/ArtiOs/ArtiOsTaskInstance</DEFINITION-REF>
      <...>
    </ECUC-CONTAINER-VALUE>
  </CONTAINERS>
</ECUC-MODULE-CONFIGURATION-VALUES>
```

10.4.5 ArtiOsClass

SWS Item	**ARTI_ArtiOs_xxxx**
Container Name:	ArtiOsClass
Description:	Contains the layout of an ARTI "Os" object, extending the EcuC OsOS.
Post-Build Variant:	False
Multiplicity:	
Configuration Parameters	

Name:	ArtiOsClassDescription

Description:	Description of the "Os" class.
Type	EcucStringParamDef
Post-Build Variant Multiplicity:	False
Post-Build Variant Value:	False
Multiplicity Configuration Class:	Pre-Compile
Value Configuration Class:	Pre-Compile
Scope	ECU

Name:	ArtiOsAppModeClassRef
Description:	Refers to the ArtiObjectClassParameter that defines the ArtiOsAppModeInstance parameter.
Type	Reference to ArtiObjectClassParameter
Post-Build Variant Multiplicity:	False
Post-Build Variant Value:	False
Multiplicity Configuration Class:	Pre-Compile
Value Configuration Class:	Pre-Compile
Scope	ECU

Example:
Exemplary value of an ArtiOsClass container:

```
<ECUC-CONTAINER-VALUE>
  <SHORT-NAME>ArtiOsClass_Conf</SHORT-NAME>
  <DEFINITION-REF DEST="ECUC-PARAM-CONF-CONTAINER-DEF">/AUTOSAR/ArtiDefs/
      ArtiOs/ArtiOsClass</DEFINITION-REF>
  <REFERENCE-VALUES>
    <ECUC-REFERENCE-VALUE>
      <DEFINITION-REF DEST="ECUC-REFERENCE-DEF">/AUTOSAR/ArtiDefs/
          ArtiOs/ArtiOsClass/ArtiOsAppModeClassRef</DEFINITION-REF>
      <VALUE-REF DEST="ECUC-CONTAINER-VALUE">/Vendor1/Vendor1Arti/
          ArtiObjectClassParameter_ArtiOs_OsAppMode</VALUE-REF>
    </ECUC-REFERENCE-VALUE>
  </REFERENCE-VALUES>
</ECUC-CONTAINER-VALUE>
```

10.4.6 ArtiOsInstance

SWS Item	**ARTI_ArtiOs_xxxx**
Container Name:	ArtiOsInstance
Description:	Represents an instance of an ARTI "Os" object, extending the EcuC OsOS.
Post-Build Variant:	False
Multiplicity:	
Configuration Parameters	

Name:	ArtiOsInstanceDescription
Description:	Description of the "Os" instance.
Type	EcucStringParamDef
Post-Build Variant Multiplicity:	False
Post-Build Variant Value:	False
Multiplicity Configuration Class:	Pre-Compile
Value Configuration Class:	Pre-Compile
Scope	ECU

Name:	ArtiOsAppModeInstanceRef
Description:	Refers to the ArtiObjectInstanceParameter that contains the evaluation for the "application mode" of this OS.
Type	Reference to ArtiObjectInstanceParameter
Post-Build Variant Multiplicity:	False
Post-Build Variant Value:	False
Multiplicity Configuration Class:	Pre-Compile
Value Configuration Class:	Pre-Compile
Scope	ECU

Name:	ArtiOsAppModeEcucRef
Description:	Refers to the EcucDefs/Os/OsAppMode of this OS.
Type	Reference to OsAppMode
Post-Build Variant Multiplicity:	False
Post-Build Variant Value:	False
Multiplicity Configuration Class:	Pre-Compile
Value Configuration Class:	Pre-Compile
Scope	ECU

Name:	ArtiOsEcucRef
Description:	Refers to the EcucDefs/Os/OsOS of this OS.
Type	Reference to OsOS
Post-Build Variant Multiplicity:	False
Post-Build Variant Value:	False
Multiplicity Configuration Class:	Pre-Compile
Value Configuration Class:	Pre-Compile
Scope	ECU

Name:	ArtiOsTaskHookRef
Description:	Refers to a hook for a specific task.
Type	Reference to ArtiHook
Post-Build Variant Multiplicity:	False
Post-Build Variant Value:	False
Multiplicity Configuration Class:	Pre-Compile
Value Configuration Class:	Pre-Compile
Scope	ECU

Example:
Exemplary value of an ArtiOsInstance container:

```
<ECUC-CONTAINER-VALUE>
  <SHORT-NAME>ArtiOsInstance_Conf</SHORT-NAME>
  <DEFINITION-REF DEST="ECUC-PARAM-CONF-CONTAINER-DEF">/AUTOSAR/ArtiDefs/
      ArtiOs/ArtiOsInstance</DEFINITION-REF>
  <REFERENCE-VALUES>
    <ECUC-REFERENCE-VALUE>
      <DEFINITION-REF DEST="ECUC-REFERENCE-DEF">/AUTOSAR/ArtiDefs/
          ArtiOs/ArtiOsInstance/ArtiOsAppModeEcucRef</DEFINITION-REF>
      <VALUE-REF DEST="ECUC-CONTAINER-VALUE">/Vendor1/Vendor1EcucOs/
          AppMode</VALUE-REF>
    </ECUC-REFERENCE-VALUE>
    <ECUC-REFERENCE-VALUE>
      <DEFINITION-REF DEST="ECUC-REFERENCE-DEF">/AUTOSAR/ArtiDefs/
          ArtiOs/ArtiOsInstance/ArtiOsAppModeInstanceRef</DEFINITION-REF>
      <VALUE-REF DEST="ECUC-CONTAINER-VALUE">/Vendor1/Vendor1Arti/
          ArtiObjectInstanceParameter_OsAppMode</VALUE-REF>
    </ECUC-REFERENCE-VALUE>
    <ECUC-REFERENCE-VALUE>
      <DEFINITION-REF DEST="ECUC-REFERENCE-DEF">/AUTOSAR/ArtiDefs/
          ArtiOs/ArtiOsInstance/ArtiOsEcucRef</DEFINITION-REF>
      <VALUE-REF DEST="ECUC-CONTAINER-VALUE">/Vendor1/Vendor1EcucOs/
          Vendor1Os</VALUE-REF>
    </ECUC-REFERENCE-VALUE>
    <ECUC-REFERENCE-VALUE>
      <DEFINITION-REF DEST="ECUC-REFERENCE-DEF">/AUTOSAR/ArtiDefs/
          ArtiOs/ArtiOsInstance/ArtiOsTaskHookRef</DEFINITION-REF>
      <VALUE-REF DEST="ECUC-CONTAINER-VALUE">/Vendor1/Vendor1Arti/
          ArtiHook_ArtiOs_TaskStart</VALUE-REF>
    </ECUC-REFERENCE-VALUE>
    <ECUC-REFERENCE-VALUE>
      <DEFINITION-REF DEST="ECUC-REFERENCE-DEF">/AUTOSAR/ArtiDefs/
          ArtiOs/ArtiOsInstance/ArtiOsTaskHookRef</DEFINITION-REF>
      <VALUE-REF DEST="ECUC-CONTAINER-VALUE">/Vendor1/Vendor1Arti/
          ArtiHook_ArtiOs_TaskStop</VALUE-REF>
    </ECUC-REFERENCE-VALUE>
  </REFERENCE-VALUES>
</ECUC-CONTAINER-VALUE>
```

10.4.7 ArtiOsTaskClass

SWS Item	**ARTI_ArtiOs_xxxx**
Container Name:	ArtiOsTaskClass
Description:	Contains the layout of an ARTI "OsTask" object, extending the EcuC OsTask.
Post-Build Variant:	False
Multiplicity:	
Configuration Parameters	

Name:	ArtiOsTaskClassDescription
Description:	Description of the "OsTask" class.
Type	EcucStringParamDef
Post-Build Variant Multiplicity:	False
Post-Build Variant Value:	False
Multiplicity Configuration Class:	Pre-Compile
Value Configuration Class:	Pre-Compile
Scope	ECU

Example:
Exemplary value of an ArtiOsTaskClass container:
```
<ECUC-CONTAINER-VALUE>
  <SHORT-NAME>ArtiOsTaskClass_Conf</SHORT-NAME>
  <DEFINITION-REF DEST="ECUC-PARAM-CONF-CONTAINER-DEF">/AUTOSAR/ArtiDefs/
      ArtiOs/ArtiOsTaskClass</DEFINITION-REF>
</ECUC-CONTAINER-VALUE>
```

10.4.8 ArtiOsTaskInstance

SWS Item	**ARTI_ArtiOs_xxxx**
Container Name:	ArtiOsTaskInstance
Description:	Represents an instance of an ARTI "OsTask" object, extending the EcuC OsTask.
Post-Build Variant:	False
Multiplicity:	
Configuration Parameters	

Name:	ArtiOsTaskInstanceDescription
Description:	Description of the "OsTask" instance.
Type	EcucStringParamDef
Post-Build Variant Multiplicity:	False
Post-Build Variant Value:	False

Multiplicity Configuration Class:	Pre-Compile
Value Configuration Class:	Pre-Compile
Scope	ECU

Name:	ArtiOsGenericComponentInstanceRef
Description:	Refers to an ArtiGenericComponentInstance that extends the OsTask.
Type	Reference to ArtiGenericComponentInstance
Post-Build Variant Multiplicity:	False
Post-Build Variant Value:	False
Multiplicity Configuration Class:	Pre-Compile
Value Configuration Class:	Pre-Compile
Scope	ECU

Name:	ArtiOsTaskEcucRef
Description:	Refers to the EcucDefs/Os/OsTask of this task.
Type	Reference to OsTask
Post-Build Variant Multiplicity:	False
Post-Build Variant Value:	False
Multiplicity Configuration Class:	Pre-Compile
Value Configuration Class:	Pre-Compile
Scope	ECU

Example:
Exemplary value of an ArtiOsTaskInstance container:

```
  <ECUC-CONTAINER-VALUE>
    <SHORT-NAME>ArtiOsTaskInstance_TaskHighPriority</SHORT-NAME>
    <DEFINITION-REF DEST="ECUC-PARAM-CONF-CONTAINER-DEF">/AUTOSAR/ArtiDefs/
        ArtiOs/ArtiOsTaskInstance</DEFINITION-REF>
    <REFERENCE-VALUES>
      <ECUC-REFERENCE-VALUE>
        <DEFINITION-REF DEST="ECUC-REFERENCE-DEF">/AUTOSAR/ArtiDefs/
            ArtiOs/ArtiOsTaskInstance/
            ArtiGenericComponentInstanceRef</DEFINITION-REF>
        <VALUE-REF DEST="ECUC-CONTAINER-VALUE">/Vendor1/Vendor1ArtiGeneric/
            ArtiGenericComponentInstance_TaskHighPriority</VALUE-REF>
      </ECUC-REFERENCE-VALUE>
      <ECUC-REFERENCE-VALUE>
        <DEFINITION-REF DEST="ECUC-REFERENCE-DEF">/AUTOSAR/ArtiDefs/
            ArtiOs/ArtiOsTaskInstance/ArtiOsTaskEcucRef</DEFINITION-REF>
        <VALUE-REF DEST="ECUC-CONTAINER-VALUE">/Vendor1/Vendor1EcucOs/
            TaskHighPriority</VALUE-REF>
      </ECUC-REFERENCE-VALUE>
    </REFERENCE-VALUES>
  </ECUC-CONTAINER-VALUE>
```

10.5 Published Information

For details refer to the chapter 10.3 “Published Information” in *SWS_BSWGeneral.*

11 Generation of the OS

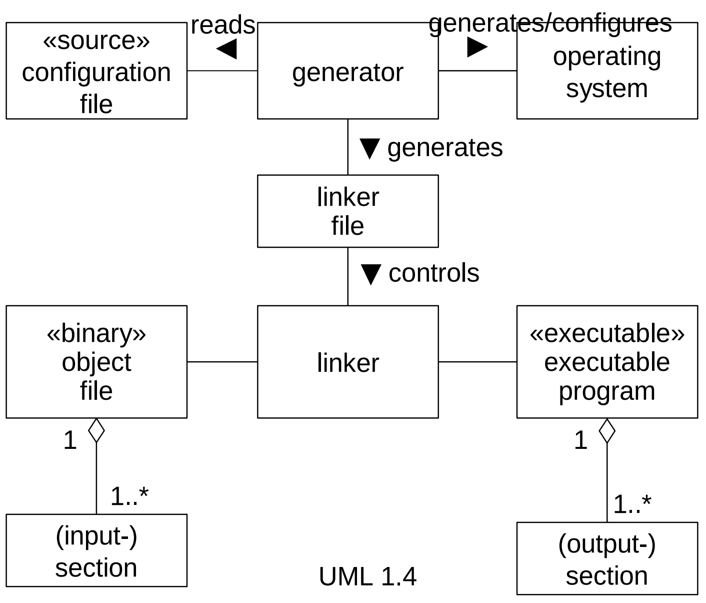

Figure 11.1: Generation Activities

11.1 Read in configuration

[SWS_Os_00172] ⌈The generator shall provide the user the ability of reading the information of a selectable configuration file. ⌋ ( )

11.2 Consistency check

The conistency check can issue warnings or errors. Warnings mean that the generation is completed successfully, only indicating a not advisable configuration. Errors mean that the generation is not performed.

[SWS_Os_00173] ⌈The generator shall provide the user the ability of performing a consistency check of the current configuration. ⌋ ( )

[SWS_Os_00050] ⌈If service protection is required and `OsStatus` is not equal to `EXTENDED` (all the associated error handling is provided), the consistency check shall issue an error. ⌋ ( )

[SWS_Os_00045] ⌈If timing protection is configured together with OSEK OS Category 1 interrupts, the consistency check shall issue a warning. ⌋ ( )

[SWS_Os_00562] ⌈If timing protection is configured together with Pre- or PostTaskHook the consistency check shall issue a warning. ⌋ ( )

[SWS_Os_00320] ⌈If configured attributes do not match the configured scalability class (e.g. defining an execution time budget in Tasks or Category 2 ISRs and selected scalability class is 1) the consistency check shall issue a warning. ⌋ ( )

[SWS_Os_00311] ⌈If `OsScalabilityClass` is `SC3` or `SC4` AND a Task OR Category 2 ISR OR Counters OR Alarms OR Schedule tables does not belong to exactly one OS-Application the consistency check shall issue an error. ⌋ ( )

[SWS_Os_00361] ⌈If `OsScalabilityClass` is `SC3` or `SC4` AND a Category 1 ISR does not belong to exactly one trusted OS-Application the consistency check shall issue an error⌋ ( )

[SWS_Os_00177] ⌈If `OsScalabilityClass` is `SC3` or `SC4` AND an interrupt source that is used by the OS is assigned to an OS-Application, the consistency check shall issue an error. ⌋ ( )

[SWS_Os_00303] ⌈If `OsAlarmIncrementCounter` is configured as action on alarm expiry AND the alarm is driven directly or indirectly (a cyclic chain of alarm actions with `OsAlarmIncrementCounter`) by that counter, the consistency check shall issue a warning.. ⌋ ( )

[SWS_Os_00328] ⌈If `OsStatus` is `STANDARD` and `OsScalabilityClass` is `SC3` or `SC4` the consistency check shall issue an error. ⌋ ( )

[SWS_Os_00343] ⌈If `OsScalabilityClass` is `SC3` or `SC4` AND a task is referenced within a schedule table object AND the OS-Application of the schedule table has no access to the task, the consistency check shall issue an error. ⌋ ( )

[SWS_Os_00344] ⌈If `OsScalabilityClass` is `SC3` or `SC4` AND a task is referenced within an alarm object AND the OS-Application of the alarm has no access to the task, the consistency check shall issue an error. ⌋ ( )

[SWS_Os_00440] ⌈If a schedule table has `OsScheduleTblSyncStrategy = IMPLICIT` and the `OsCounterMaxAllowedValue+1` of the associated counter is not equal to the duration of the schedule table then the consitency check shall issue an error. ⌋ ( )

[SWS_Os_00461] ⌈If `OsScalabilityClass` is `SC2`, `SC3` or `SC4` `AND` Alarm Callbacks are configured the conistency check shall isuue an error. ⌋ ( )

11.3 Generating operating system

[SWS_Os_00179] ⌈If the consistency check of the read-in configuration file has not run free of errors, the generator shall not generate/configure the operating system. ⌋ ( )

[SWS_Os_00336] ⌈The generator shall generate a relocatable memory section containing the interrupt vector table. ⌋ (SRS_Os_11019)

[SWS_Os_00370] ⌈The generator shall print out information about timers used internally by the OS during generation (e.g. on console, list file). ⌋ (SRS_Frt_00022)

[SWS_Os_00393] ⌈The generator shall create conversation macros to convert counter ticks (given as argument) into real time. The format of the macro is `OS_TICKS2<Unit>_<Counter>(ticks)` whereas `<Unit>` is one of `NS` (nanoseconds), `US` (microseconds), `MS` (milliseconds) or `SEC` (seconds) and `<Counter>` is the name of the counter; E.g. `OS_TICKS2MS_MyCounter()`)⌋ (SRS_Frt_00047)

[SWS_Os_00815]⌈ The OS code shall wrap each declaration of Task, ISR and hook functions with the Memory Mapping Allocation Keywords macros.

```
#define OS_START_SEC_<sadm>
#include "Os_MemMap.h"

<Task, ISR or hook functions declaration>

#define OS_STOP_SEC_<sadm>
#include "Os_MemMap.h"
```

where <sadm> is the shortName of the SwAddrMethod if configured in OsMemoryMappingCodeLocationRef.⌋ (SRS_BSW_00351)

12 Application Notes

12.1 Hooks

In OSEK OS, PreTask & PostTask Hooks run at the level of the OS with unrestricted access rights and therefore must be trusted. It is strongly recommended that these hook routines are only used during debugging and are not used in a final product.

When an OS-Application is killed the shutdown and startup hooks of the OS-Application are not called. Cleanup of OS-Application specific data can be done in the restart task.

All application-specific hook functions (startup, shutdown and error) must return (blocking or endless loops are not acceptable).

12.2 Providing Trusted Functions

Address checking shall be done before data is accessed. Special care must be taken if parameters passed by reference point to the stack space of a task or interrupt, because this address space might no longer belong to the task or interrupt when the address is used.

The following code fragment shows an example how a trusted function is called and how the checks should be done.

```
struct parameter_struct {type1 name1, type2 name2, StatusType
return_value};

/* This service is called by the user and uses a trusted function */

StatusType system_service(
               type1 parameter1,
               type2 parameter2)
{
    /* store parameters in a structure (parameter1 and parameter2) */
    struct  parameter_struct  local_struct;
    local_struct.name1 = parameter1;
    local_struct.name2 = parameter2;

    /* call CallTrustedFunction with appropriate index and
     * pointer to structure */
    if(CallTrustedFunction(SYSTEM_SERVICE_INDEX, &local_struct) !=
       E_OK)
         return(FUNCTION_DOES_NOT_EXIST);
    return(local_struct.return_value);
}

/* The CallTrustedFunction() service switches to the privileged
 * mode. Note that the example is only a fragment! */

StatusType CallTrustedFunction(
                TrustedFunctionIndexType        ix,
                TrustedFunctionParameterRefType ref)
{
    /* check for legal service index and return error if necessary */
    if(ix > MAX_SYSTEM_SERVICE)
        return(E_OS_SERVICEID);

    /* some implementation specific magic happens: the processor is
     * set to privileged mode */
    ….

    /*  indirectly call target function based on the index */
    (*(system-service_list[ix]))(ix, ref);

    /* some implementation specific magic happens: the processor is
     * set to non-privileged mode */
    ….

    return(E_OK);
}
```

```
/* This part of the system service is called by
 * CallTrustedFunction() */

void TRUSTED_system_service_part2 (TrustedFunctionIndexType a,
parameter_struct *local_struct)
{
    TaskRefType task;
    type1 parameter1;
    type2 parameter2;

    if (GetTaskID(&task) != E_OK)
        task = INVALID_TASK;

    /* get parameters out of the structure (parameter1 and
     * parameter2) */
    parameter1 = local_struct.name1;
    parameter2 = local_struct.name2;

    /* check the parameters if necessary */
    /* example is for parameter1 being an address and parameter2
     * being a size */
    /* example only for system_service called from tasks */
    if(GetISRID()!=INVALID_ISR)
    {
        /* error: not callable from ISR  */
        local_struct.return_value = E_OS_ACCESS;
    }
    else if(OSMEMORY_IS_WRITEABLE(CheckTaskMemoryAccess(
                 task,parameter1,parameter2)))
    {
        /* system_service_part3() is now the function as it
         * would be if directly called in a non-protected
         * environment */
        local_struct.return_value =
              system_service_part3(parameter1,parameter2);
    }
    else
    {
        /* error handling */
        local_struct.return_value = E_OS_ACCESS;
    }
}
```

Note: Since the service of `CallTrustedFunction()` is very generic, it is needed to define a stub-interface which does the packing and unpacking of the arguments (as the example show). Depending on the implementation the stub interface may be (partly) generated by the generation tool.

12.3 Software Components and OS-Applications

Trusted OS-Applications can be permitted access to IO space. As software components can not be allowed direct access to the hardware, software components can not be trusted OS-Applications because this would violate this protection feature. The configuration process must ensure that this is the case.

The AUTOSAR Virtual Function Bus (VFB) specification places no restrictions on how runnables from software components are mapped to OS tasks. However, the protection mechanisms in AUTOSAR OS apply only to OS managed objects. This means that all runnables in a task:

- Are not protected from each other at runtime
- Share the same protection boundary

If runnables need to be protected they must therefore be allocated to different tasks and those tasks protected accordingly.

A simple rule can suffice:

"When allocating runnables to tasks, only allocate runnables from the same software component into the same task."

If multiple software components from the same application are to reside on the same processor, then, assuming protection is required between applications (or parts thereof) on the same processor, this rule could be modified to relax the scope of protection to the application:

"When allocating runnables to tasks, only allocate runnables from the same application into the same task."

If an OS-Application is killed and the restart task is activated it can not assume that the startup of the OS-Application has finished. Maybe the fault happened in the application startup hook and no task of the application was started so far.

12.4 Global Time Synchronization

The OS currently assumes that the global time synchronization is done by the user (unless implicit synchronization is used). This allows maximum flexibility regarding the time source. For synchronization with e.g. FlexRay some glue code may be necessary which transfer the information from the time source to the OS.

12.5 Working with FlexRay

Schedule tables in the AUTOSAR OS may be synchronized with a global (network) time provided by FlexRay in essentially two ways:

1. Using the FlexRay interface's services for controlling timer interrupts related to global time to provide a "hardware" counter tick source to drive the processing of a schedule table (implicit synchronization)

2. Using the FlexRay interface's service for accessing the current global time and passing this into the OS through the SyncScheduleTable() OS service call

This section looks at the second option only.

In FlexRay time is presented as a tuple of a Cycle and a MacrotickOffset within the cycle. Cycle is an 8-bit value and MacrotickOffset is a 16-bit value.
In AUTOSAR OS a schedule table is associated with an underlying counter that has a notion of ticks. It is therefore possible to synchronize with either the Cycle or the tuple of Cycle/MacrotickOffset to give the resolution of synchronization required by the application.
If Cycle only resolution is required then an OS COUNTER object should be configured to have a OsCounterMaxAllowedValue equal to the maximum number of Cycles. If Cycle/MacrotickOffset is required then an OS COUNTER object should be configured with a OsCounterMaxAllowedValue of the maximum number of Cycles multiplied by the MacrotickOffset. This provides the OS with a time base against which a ScheduleTable can be synchronized.

Synchronization between the OS and an external global time source is provided by telling the OS the global time through the SyncScheduleTable() service call. This call takes a scalar parameter of TickType so to interface this to FlexRay's representation of time a small conversion needs to be done. The following example assumes a Cycle of 255 with 65535 Macroticks per Cycle. TickType is at least 24-bits wide.

```
#define OSTIME(x) (TickType)(x);
FrIf_GetGlobalTime(Controller, &Cycle, &Macrotick);
SyncScheduleTable(Tbl, ((OSTIME(Cycle) << 16)+(OSTIME(Macrotick))));
```

Telling the ScheduleTable that GlobalTime can be done when the application detects that the FlexRay controller has lost synchronization with the network (by polling the controller sync status). The following code indicates how this can be used to force an associated ScheduleTable into the `SCHEDULETABLE_RUNNING` state from the `SCHEDULETABLE_RUNNING_AND_SYNCHRONOUS` state.

```
Fr_SyncStateType CurrentSyncStatus;
if (FrIf_GetSyncState(Controller, &CurrentSyncStatus) == E_OK) {
  if (CurrentSyncStatus == FR_ASYNC ) {
    SetScheduleTableAsync(Table);
  }
}
```

Of course, other actions are possible here, like stopping the ScheduleTable, as best fits user requirements.

12.6 Migration from OIL to XML

This version of the AUTOSAR OS specification does not directly support the configuration via OIL. The support for OIL was dropped in favour of XML because XML is the standard configuration language in AUTOSAR and is essential if configuration data has to be imported / exported from / to other AUTOSAR modules or between different tools during development.

Since OIL and XML are both ASCII formats a tool vendor may offer a possibility to import (old) OIL files and to store them as (AUTOSAR OS) XML files. Currently all known vendors support at least the import of existing OIL configurations.

Note that for showing conformance to the OSEK OS specification, each OSEK OS vendor must support OIL. This means that practically each AUTOSAR OS vendor will offer some sort of import of OIL configurations – at least to show the OSEK OS conformance.

12.7 Migrating RES_SCHEDULER in AUTOSAR OS

As stated in 7.1.2.1 AUTOSAR OS treats `RES_SCHEDULER` as a normal resource. If you have legacy code which is migrated to AUTOSAR OS please take care of the following aspects:

- In OSEK OS there is no need to configure the `RES_SCHEDULER` in the OIL file. If you migrate to AUTOSAR OS the configuration is done in XML and each resource must be properly configured. The easiest way to do this is to configure a resource `RES_SCHEDULER` in XML (OsResource) and allow any Task in your system to use this resource[7].
- Avoid that ISRs are using the `RES_SCHEDULER`. In OSEK OS this is also not possible.
- Make the `RES_SCHEDULER` a `STANDARD` resource (at least not an `INTERNAL` resource). The symbol `RES_SCHEDULER` must be present which is not the case if the resource is an `INTERNAL` resource.
- If you are using OS-Applications, the `RES_SCHEDULER` should belong to a trusted OS-Application. Tasks of other OS-Applications should be configured to have the right to access the resource.

12.8 Debug support

For the AUTOSAR OS the following information may be useful for users and should be considert for debug support (and may be published, e.g. in the BSWMD):

- General information about how to retrieve the current (active) Task or ISR and their (current) priority and (current) stack.
- For ISRs: Information about the name of interrupts, their mapping to the ISR identifier, the associated hardware and the used stack(s).
- For Tasks: Information about the name of the Task, its identifier, the task state, the possible priorities, the event mask (if its an extended task), the OS-Application to whom the Task belongs (if existant) and the used stack.
- For Resources: Information about the name of the Resource, its mapping to the identifier, its priority and the current owner (the Task/ISR which currently holds the Resource)

[7] This work can be done automatically by a configuration tool duirng importing an OIL file

- For Alarms: Information about the name of the Alarm, its mapping to the identifier, the counter to whom it belong, the action which is executed on expiry and the current state (running or stopped). In running state the next expiry in ticks and the possible cycle time shall be also published.
- For Counters: Information about the name of the Counter, its mapping to the identifier, its associated alarms and the current counter value.
- For Schdule Tables: Information about the name of the Schedule Table, its mapping to the identifier, its current state and the next expiry point (if the table is running).
- For OS-Applications: Information about the name of the OS-Application, its mapping to the identifier, its current state and the memory sections assigned to it (if memory protection is used).

User documentation should contain information about the implemeted debug features.

12.9 Integration hints for peripheral protection

Peripheral protection requires configuration on the core level usually conditioned by a supervisor access. For this reason the task of the peripheral protection is assigned to the OS module.

Peripheral protection may be implemented in two ways
 - using MPU
 - using dedicated peripheral protection units of the target MCU.

When using the memory protection unit, it is reasonable if two or more protected region descriptors are available for peripheral protection mechanism. The region descriptors shall be programmed to allow access to those peripherals the current OS-Application shall work with. The defined regions shall cover all memory mapped configuration registers for the periphiherals to be protected. The advantage of using the MPU is that the configuration is the same as for memory protection. One of the disadvantages of this method is that it could be impossilbe to cover all peripheral control registers with available MPU region descriptors. The number of such descriptors is typically low.

Beware that using this method may have implication to the linker file of the project software configuration.

Second method is using a dedicated register protection schema. This method shall allow to precisely select peripherals for every OS Application. However the number of peripherals may make the register protection implementation rather bulky. Therefore it is advisable to reduce the number of protected peripherals to a reasonable value.

For both methods the configuration shall be placed into custom OS Application properties. The configuration shall be active when a task (or ISR) of a particular OS Application is running.

12.10 Termination of OSApplications

Inconsistencies may occur when an OsApplication is terminated and restarted, depending on its state at the termination.

A notification from an asynchronous job started before the termination of OsApplication can occur after the restart of OsApplication.

An asynchronous memory read or write started before the termination of OsApplication can occur after restart, and cause data inconsistency.

A requested mode or state to another OsApplication (e.g. from a SW-C to A BSW) can lead to unsynchronized state machines after an OsApplication restart.

Therefore some measures shall be taken to avoid these inconsistencies and guaranty a correct behavior.

Integration code shall stop all signals and signalgroups during its OsApplication restart. This ensures that no late asynchronous notification will occur after the OsApplication restart. These signals and signalgroups can be then safely restarted if needed.

A SW-C shall cancel jobs on all its memory blocks with a call to `NvM_CancelJobs` during the restart of its OsApplication. As the job might have already been started, the call to `NvM_CancelJobs` can return an error; in that case, the OsApplication shall wait until end of the job to continue. After all jobs are ensured to be cancelled, then all memory blocks shall be reset to their initial value, in order to avoid inconsistency of data which might have been written before the cancellation.

Any SW-C having responsible for requesting mode or state to BSW mode managers shall always request a default mode upon a restart of its OsApplication. Thus the BSW mode manager would not be stuck into a mode previously requested by the OsApplication before its termination. To support this task, note that RTE offers mechanisms to handle partition stop and restart wrt. mode machines. For mode managers an "error mode" to be set by RTE can be identified. For mode user partition the behaviour can also be selected. Furthermore an interaction to BswM to trigger an action list in case of partition restart can be initiated. Refer to RTE specification for details.

As a global hint, in any non-trusted OsApplication, which could be terminated, there shall always be a restart task which does the following actions:

Cancel all jobs which can result in an asynchronous notification or shared memory, I/O access.

Reset all shared memory with a default value.

Reset any mode or state residing in another OsApplication and controlled by this given OsApplication to a default value.

Please note that some of these actions need to be performed even if an OSApplication is merely terminated and not restarted. For example, it may still be necessary to stop all signals and signal groups used by the OSApplication. Otherwise, it may happen that a bus never goes to sleep.

Consequently, in such a case it is necessary to activate the restart task to perform the necessary cleanup even if the OSApplication is only terminated and not restarted. Calling `TerminateApplication(<ownappid>,NO_RESTART)` in the restart task will finally set the OSApplication to `APPLICATION_TERMINATED`.

13 AUTOSAR Service implemented by the OS

13.1 Scope of this Chapter

This chapter is an addition to the specification of the Operating System. Whereas the other parts of the specification define the behavior and the C-interfaces of the OS module, this chapter formally specifies the corresponding AUTOSAR Service in terms of the SWC Template. The interfaces described here will be visible on the VFB and are used by the RTE generator to create the glue code between the application software (SWC) and the OS.

13.1.1 Package

The following definitions are interpreted to be in
`ARPackage AUTOSAR/Services/Os`

13.2 Overview

The AUTOSAR Operating System is normally not used directly by SWCs. Even the other BSW modules which are below the RTE are using the BSW Scheduler to have access to OS services. The BSW Scheduler of course uses the OS to implement its features, e.g. critical sections.

Nevertheless there is one case where it makes sense to allow SWCs access to services of the OS:

- Timer services
  Since the number of timers in an ECU is limited it make sense to share these units across several SWCs. The functionality of the timer services of the OS which are offered to the SWCs are:
  - A service to get the current value of a – hardware or software – counter
  - A service which calculates the time difference between the current timer value and a given (previouls read) timer value
  - Both services will return real time values instead of ticks. This limits the access to the services to those counters which are counting time. Other counters e.g. counting errors or angles are not accessible.

13.3 Specification of the Ports and Port Interfaces

The detailed port interface can be found in chapter 8.8.

The notation of possible error codes resulting from server calls follows the approach in the meta-model. It is a matter of the RTE specification [9], how those error codes will be passed via the actual API.

14 Outlook on Memory Protection Configuration

As stated before, memory protection configuration is not standardized yet. Nevertheless it seems helpful to contribute a recommendation in this chapter, how the configuration might work.

14.1 Configuration Approach

Both, SW-Components and BSW modules, map code and variables to dedicated, disjoined memory sections (see meta-class»ObjectFileSection« in chapter 7.3 of »Software Component Template«, Version 2.0.1, and »module specific sections« in chapter 8.2 of »Specification of Memory Mapping«, Version 1.0.1).

This essential precondition (avoid an inseparable conglomeration of variables in the default section) can be used to support configuration of memory protection domains:

1. The generator can save for each OS-Application a (processor-specific) maximum number of output sections for data in a file (to be used in the linker file).

2. The generator can uniquely identify the address spaces of the data output sections with symbols using the naming convention (see »memory allocation keywords« `_STOP_SEC_VAR` and `_START_SEC_VAR` for start and stop symbols) in the specification mentioned above.

The input data sections in the object files of an OS-Application can then be assigned to the output sections (with potential tool support). Usually, this is one segment for global data, and one segment for code.

To archieve portability, the user shall group all variables belonging to a private data section (Task/ISR or OS-Application) in separate files.

15 Not applicable requirements

[SWS_Os_00767] ⌈These requirements are not applicable to this specification.⌋ (SRS_BSW_00344, SRS_BSW_00404, SRS_BSW_00405, SRS_BSW_00170, SRS_BSW_00419, SRS_BSW_00383, SRS_BSW_00384, SRS_BSW_00375, SRS_BSW_00406, SRS_BSW_00168, SRS_BSW_00407, SRS_BSW_00423, SRS_BSW_00337, SRS_BSW_00369, SRS_BSW_00339, SRS_BSW_00422, SRS_BSW_00417, SRS_BSW_00409, SRS_BSW_00385, SRS_BSW_00386, SRS_BSW_00437, SRS_BSW_00161, SRS_BSW_00162, SRS_BSW_00415, SRS_BSW_00325, SRS_BSW_00342, SRS_BSW_00007, SRS_BSW_00413, SRS_BSW_00347, SRS_BSW_00441, SRS_BSW_00305, SRS_BSW_00307, SRS_BSW_00310, SRS_BSW_00373, SRS_BSW_00327, SRS_BSW_00335, SRS_BSW_00350, SRS_BSW_00410, SRS_BSW_00411, SRS_BSW_00314, SRS_BSW_00361, SRS_BSW_00301, SRS_BSW_00302, SRS_BSW_00328, SRS_BSW_00312, SRS_BSW_00006, SRS_BSW_00439, SRS_BSW_00357, SRS_BSW_00377, SRS_BSW_00378, SRS_BSW_00306, SRS_BSW_00308, SRS_BSW_00309, SRS_BSW_00358, SRS_BSW_00414, SRS_BSW_00440, SRS_BSW_00330, SRS_BSW_00009, SRS_BSW_00401, SRS_BSW_00172, SRS_BSW_00010, SRS_BSW_00333, SRS_BSW_00374, SRS_BSW_00379, SRS_BSW_00003, SRS_BSW_00318, SRS_BSW_00321, SRS_BSW_00334, SRS_Frt_00032)